

Dmitry Gizlyk

NEURAL NETWORKS FOR ALGO TRADING WITH **MQL5**



 MetaQuotes

© 2000 — 2024, MetaQuotes Ltd.

Contents

Neural Networks for Algorithmic Trading with MQL5	5
Introduction.....	6
1. Basic principles of artificial intelligence construction.....	7
1.1 Neuron and principles of building neural networks.....	8
1.2 Activation functions.....	12
1.3 Weight initialization methods in neural networks.....	23
1.4 Neural network training.....	26
1.4.1 Loss functions.....	28
1.4.2 Error gradient backpropagation method.....	33
1.4.3 Methods for optimizing neural networks.....	38
1.5 Techniques for improving the convergence of neural networks.....	48
1.5.1 Regularization.....	48
1.5.2 Dropout.....	50
1.5.3 Normalization.....	52
1.6 Artificial intelligence in trading.....	56
2. MetaTrader 5 features for algorithmic trading.....	57
2.1 Program types and their construction features.....	60
2.2 Statistical analysis and fuzzy logic tools.....	67
2.3 OpenCL: Parallel computations in MQL5.....	72
2.4 Integration with Python.....	82
3. Building the first neural network model in MQL5.....	84
3.1 Problem statement.....	85
3.2 File arrangement structure.....	87
3.3 Choosing the input data.....	87
3.4 Creating the framework for the future MQL5 program.....	100
3.4.1 Defining constants.....	101
3.4.2 Mechanism for describing the structure of the future neural network.....	104
3.4.3 Neural network base class and organization of forward and backward pass processes.....	107
3.4.4 Dynamic storage array of neural layers.....	126
3.5 Description of a Python script structure.....	130
3.6 Fully connected neural layer.....	138
3.6.1 Architecture and implementation principles.....	139
3.6.2 Creating a neural layer using MQL5 tools.....	142
3.6.3 Activation function class.....	161
3.7 Organizing parallel computing using OpenCL.....	169
3.7.1 Creating an OpenCL program.....	170
3.7.2 Implementing functionality on the main program side.....	186
3.8 Implementing the perceptron model in Python.....	218
3.9 Creating training and testing samples.....	222
3.10 Gradient distribution verification.....	230
3.11 Comparative testing of implementations.....	240
4. Basic types of neural layers.....	262
4.1 Convolutional neural networks.....	263
4.1.1 Description of architecture and implementation principles.....	264
4.1.2 Construction using MQL5.....	266
4.1.3 Organizing parallel computing in convolutional networks using OpenCL.....	287
4.1.4 Implementing a convolutional model in Python.....	307
4.1.5 Practical testing of convolutional models.....	314
4.2 Recurrent neural networks.....	334
4.2.1 Description of architecture and implementation principles.....	335

4.2.2. Building an LSTM block in MQL5.....	339
4.2.2.1 Feed-forward method.....	350
4.2.2.2 Backpropagation methods.....	359
4.2.2.3 Saving and restoring the LSTM block.....	368
4.2.3 Organizing parallel computing in the LSTM block.....	372
4.2.4 Implementing recurrent models in Python.....	381
4.2.4.1 Building a test recurrent model in Python.....	384
4.2.5 Comparative testing of recurrent models.....	389
5. Attention mechanisms.....	401
5.1 Self-Attention.....	403
5.1.1 Description of architecture and implementation principles.....	403
5.1.2 Building Self-Attention with MQL5 tools.....	406
5.1.2.1 Self-Attention feed-forward method.....	416
5.1.2.2 Self-Attention backpropagation methods.....	423
5.1.2.3 File operations.....	431
5.1.3 Organizing parallel computing in the attention block.....	434
5.1.4 Testing the attention mechanism.....	456
5.2 Multi-Head attention.....	459
5.2.1 Description of the Multi-Head Self-Attention architecture.....	461
5.2.2 Building Multi-Head Self-Attention in MQL5.....	464
5.2.2.1 Multi-Head Self-Attention feed-forward method.....	474
5.2.2.2 Multi-Head Self-Attention backpropagation methods.....	479
5.2.2.3 File operations.....	486
5.2.3 Organizing parallel computing for Multi-Head Self-Attention.....	488
5.2.4 Building Multi-Head Self-Attention in Python.....	499
5.2.4.1 Creating a new neural layer class.....	502
5.2.4.2 Creating a script to test Multi-Head Self-Attention.....	507
5.2.5 Comparative testing of Attention models.....	511
5.3 GPT architecture.....	516
5.3.1 Description of the architecture.....	517
5.3.2 Building a GPT model in MQL5.....	518
5.3.2.1 GPT feed-forward method.....	529
5.3.2.2 GPT backpropagation methods.....	534
5.3.2.3 File operations.....	543
5.3.3 Organizing parallel computing in the GPT model.....	547
5.3.4 Comparative testing of implementations.....	559
6. Architectural solutions for improving model convergence.....	570
6.1 Batch normalization.....	571
6.1.1 Principles of batch normalization implementation.....	572
6.1.2 Building a batch normalization class in MQL5.....	575
6.1.2.1 Batch normalization feed-forward methods.....	578
6.1.2.2 Batch normalization backpropagation methods.....	581
6.1.2.3 File operations.....	586
6.1.3 Organizing multi-threaded computations in the batch normalization class.....	588
6.1.4 Implementing batch normalization in Python.....	598
6.1.4.1 Creating a script to test batch normalization.....	601
6.1.5 Comparative testing of models using batch normalization.....	608
6.2 Dropout.....	623
6.2.1 Building Dropout in MQL5.....	625
6.2.1.1 Feed-forward method.....	629
6.2.1.2 Backpropagation methods for Dropout.....	631
6.2.1.3 File operations.....	633
6.2.2 Organizing multi-threaded operations in Dropout.....	635
6.2.3 Implementing Dropout in Python.....	641
6.2.4 Comparative testing of models with Dropout.....	646
7. Testing trading capabilities of the model.....	651

7.1 Introduction to MetaTrader 5 Strategy Tester.....	652
7.2 Developing an Expert Advisor template using MQL5.....	654
7.3 Creating a model for testing.....	661
7.4 Determining Expert Advisor parameters.....	675
7.5 Testing the model on new data.....	686
Conclusion.....	690

Neural Networks for Algorithmic Trading with MQL5

In the era of digital technology and artificial intelligence, algorithmic trading is transforming financial markets, offering innovative strategies. The book "Neural Networks for Algorithmic Trading with MQL5" serves as a unique guide that combines advanced technological knowledge with practical guidance on creating trading algorithms. This book is tailored for traders, developers, and financial analysts who wish to understand the principles of neural networks and their application in algorithmic trading on the MetaTrader 5 platform.

The book has 7 chapters that cover everything you need to know to get started with neural networks and integrate them into your trading robots in MQL5. Beginning with basic principles of neural networks and advancing to more complex architectural solutions and attention mechanisms, this book provides all the necessary information for the successful implementation of machine learning in your algorithmic trading solutions.

You will discover how to use different types of neural networks, including convolutional and recurrent models, and how to integrate them into the MQL5 environment. Additionally, the book explores architectural solutions to improve model convergence, such as Batch Normalization and Dropout.

Furthermore, the author provides practical guidance on how to train neural networks and embed them into your trading strategies. You will learn how to create trading Expert Advisors to test the performance of trained models on new data, enabling you to evaluate their potential in real-world financial markets.

- [Chapter 1](#) introduces you to the world of artificial intelligence, laying the foundation with essential neural network building blocks, such as activation functions and weight initialization methods.
- [Chapter 2](#) explores MetaTrader 5 capabilities in detail, describing how to utilize the platform tools to create powerful algorithmic trading strategies.
- [Chapter 3](#) guides you through the step-by-step development of your first neural network model in MQL5, covering everything from data preparation to model implementation and testing.
- [Chapter 4](#) delves deep into understanding fundamental neural layer types, including convolutional and recurrent neural networks, their practical implementation, and comprehensive testing.
- [Chapter 5](#) introduces attention mechanisms like Self-Attention and Multi-Head Self-Attention, presenting advanced data analysis methodologies.
- [Chapter 6](#) explains architectural solutions to improve model convergence, such as Batch Normalization and Dropout.
- [Chapter 7](#) concludes the book and offers methods for testing trading strategies using the developed neural network models under real trading conditions through MetaTrader 5.

With "Neural Networks for Algorithmic Trading with MQL5", you will gain comprehensive knowledge and practical skills for creating your own trading robots capable of analyzing markets and making decisions using advanced machine learning technologies. This book will be an invaluable resource for anyone who wants to use artificial intelligence in algorithmic trading and explore new horizons in financial analytics and trading.



Examples from the book "Neural networks for algorithmic trading with MQL5"



Examples from the book are also available in the [public project](#) \MQL5\Shared Projects\NeuroBook

Introduction

The whole history of mankind is the creation and improvement of tools. From the moment the ancient man took the first stick in his hands, the tools of physical labor have been constantly improved. Alongside constant improvements in tools for physical labor, humans also develop tools for intellectual work.

From the first digital mechanical calculating machine, built by Wilhelm Schickard in 1623, the world of computing machines has evolved to modern computers, which, thanks to developed algorithms, allow us to move from simple calculations to solving other more intellectual tasks. Next, artificial intelligence algorithms emerged and started covering various aspects of our everyday lives. More and more often, news feeds are filled with messages like "The neural network has been taught...".

The term "artificial intelligence" was first coined by John McCarthy, who gave the following definition:

"Artificial intelligence is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable."

At the same time, the world saw the extensive development of the art of stock trading. In an attempt to predict the future movement of exchange-traded instruments, traders meticulously started charts looking for price movement patterns, developed trading rules, and created trading strategies.

The advent of computing machines has also influenced this area of human activity. The use of computers has allowed more information to be processed in less time. As a result, the analysis of price movements of exchange-traded instruments has become more detailed and in-depth.

Further development has resulted in programs capable of operating according to predefined trading strategies and executing trades 24 hours a day without human intervention.

In this book, we will attempt to combine the above two areas of activity. In this context, another definition of artificial intelligence, as formulated by Andreas Kaplan and Michael Heintz, would be most appropriate:

"Artificial intelligence is a system's ability to correctly interpret external data, to learn from such data, and to use those learnings to achieve specific goals and tasks through flexible adaptation."

That's the property we're going to exploit. We will look at the basic principles and foundations of artificial intelligence, and then use the widely used *MetaTrader 5* terminal and demonstrate its capabilities in building different algorithms for intelligent programs.

On real data, we will test the ability of implemented algorithms to identify patterns, because understanding patterns allows us to determine the most likely vector of development of upcoming events.

We do not consider this book to be a teaching tool on artificial intelligence algorithms. The book gives only basic concepts and principles without delving deeply into the mathematical features of computation and the construction of algorithms.

This work will be more interesting for practitioners. The book provides examples of using different algorithms to solve real-life cases and presents the results of training neural networks built on different architectures using different algorithms.

I would like to draw the attention of all readers to the fact that stock trading is associated with high risks. The responsibility for any trading operation lies with the reader. The book looks at tools, not ready-made trading solutions. To use the provided tools in real trade, additional work is required to build a trading robot and/or indicators for decision-making by the trader, as well as thorough testing.

1. Basic principles of artificial intelligence construction

Knowledge of the world and oneself in it is an integral part of human existence. Reflections on the nature of consciousness have long been raised by philosophers. Neurophysiologists and psychologists have developed theories about the principles and mechanisms of the human brain operation. As in several other sciences, processes observed in nature laid the foundation for the creation of intelligent machines.

The main structural unit in the human brain is the neuron. The exact number of neurons in the human nervous system is not definitively known, while estimates suggest approximately 100 billion. Neurons, each consisting of a **cell body**, **dendrites** and **axon**, connect with each other forming a complex network. The points at which they connect are called synapses.

The described processes and structures served as the basis for the creation of artificial neural networks. In 1943, Warren McCulloch and Walter Pitts published the article [A logical calculus of the ideas immanent in nervous activity](#), in which they proposed and described two theories of neural networks: with loops and without them. These theories represented a significant step in understanding the interaction of neurons and later formed the basis for the principles of constructing neuron interactions in artificial neural networks. Donald Hebb's book [The organization of behavior: A neuropsychological theory](#) released in 1949 laid the foundation for neural learning.

The works mentioned above explored the processes in the human brain and were further developed in the works of Frank Rosenblatt. His mathematical model of the perceptron developed in 1957 formed the basis of the world's first neurocomputer "Mark-1", which he created in 1960. It should be noted that various versions of the perceptron are successfully used today to solve various tasks.

But let's proceed systematically. In this chapter, we will examine the mathematical models of the neuron and the perceptron:

- [Neuron and principles of neural network construction](#). This section elaborates on the structure of the neuron and the fundamental concepts underlying artificial neural networks, as well as their importance in understanding intelligent systems.
- [Activation functions](#) are an integral part of neural networks, determining how a neuron should respond to incoming signals. This section focuses on the different types of activation functions and their role in the neural network learning process.
- [Weight initialization methods in neural networks](#). Weight initialization is a critical step in preparing the network for training, influencing its ability to learn and converge.
- [Neural network training](#) is considered through the key components: loss functions, gradient backpropagation, and optimization methods which together form the basis for efficient network training.
- [Techniques for improving the convergence of neural networks](#), such as Dropout and normalization, detail strategies for improving neural network performance and stability during training.
- [Artificial intelligence in trading](#) covers the practical application of the technologies discussed, exploring how artificial intelligence and machine learning can be used to analyze financial markets and make trading decisions.

Thus, the chapter provides a comprehensive overview of artificial intelligence and neural networks, covering their structure, mechanisms, and real-world applications, particularly in algorithmic trading.

1.1 Neuron and principles of building neural networks

In their paper entitled "[A logical calculus of the ideas immanent in nervous activity](#)", Warren McCulloch and Walter Pitts proposed a mathematical model of a neuron and described the basic principles of neural network organization. The mathematical model of an artificial neuron involves two computation stages. Similar to a human neuron, in the mathematical model of an artificial neuron, the dendrites are represented by a vector of numerical values X , which is input into the artificial neuron. The dependence of the neuron value on each specific input is determined by the vector of weights, denoted as W . The first computation stage of the artificial neuron model is implemented as the product of the vector of initial signals by the vector of weights, which gives a weighted sum of initial data from the mathematical point of view.

$$S = \sum_{i=1}^n w_i x_i,$$

where:

- n = number of elements in the input sequence
- w_i = weight of the i th element of the sequence
- x_i = the i th element of the input sequence

The weights determine the sensitivity of the neuron to changes in a particular input value and can be either positive or negative. This way, the operation of excitatory and inhibitory signals is simulated. The values of weights satisfying the solution of a particular problem are selected in the process of training the neural network.

As mentioned before, a signal appears on the axon of a neuron only after a critical value has accumulated in the cell body. In the mathematical model of an artificial neuron, this step is implemented by introducing an activation function.

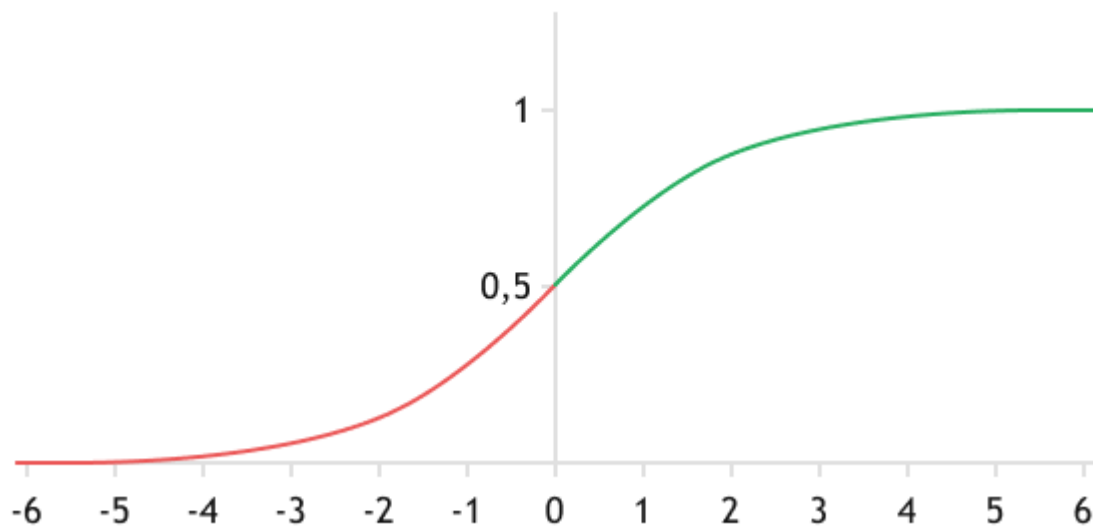
$$OUT = f(W, X) = f\left(\sum_{i=1}^n w_i x_i\right)$$

Variations are possible here. The first models used a simple function to compare the weighted sum of input values with a certain threshold value. Such an approach simulated the nature of a biological neuron, which can be excited or at rest. The graph of such a neuron activation function will have a sharp drop in value at the threshold point.



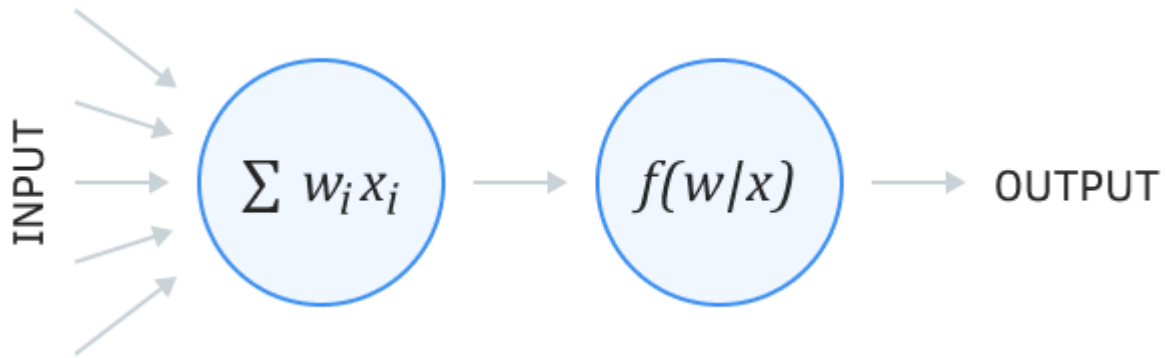
Graph of the threshold function of neuron activation.

In 1960, Bernard Widrow and Marcian Hoff published their work "[Adaptive switching circuits](#)", in which they presented the Adaline adaptive linear classification machine. This work has shown that using continuous neuron activation functions allows solving a wider range of problems with less error. Since then and up to our time, various sigmoid functions have been widely used as neuron activation functions. In this version, a smoother graph of the mathematical model of the neuron is obtained.



Graph of the logistic function (Sigmoid)

We will discuss different versions of activation functions and their advantages and disadvantages in the next chapter of the book. In a general form, the mathematical model of an artificial neuron can be schematically represented as follows.



Scheme of the mathematical model of a neuron

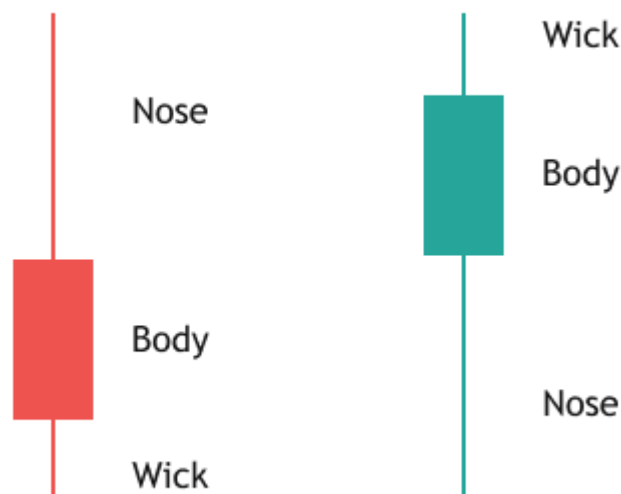
This mathematical model of a neuron allows generating a logical *True* or *False* answer based on the analysis of input data. Let's consider the model's operation using the example of searching for the candlestick pattern "Pin Bar".

According to the classic Pin Bar model, the size of the candlestick "Nose" should be at least 2.5 times the size of the body and the second tail. Mathematically, it can be visualized as follows:

$$Nose > 2.5(Body + Wick)$$

or

$$1 * Nose - 2.5 * Body - 2.5 * Wick > 0$$



Pin Bar

According to the mathematical model of the neuron, we will input three values into the neuron: the size of the candlestick nose, body, and tail. The weights will be 1, -2.5, and -2.5, respectively. It should be noted that we will not consider weights when constructing the neural network. They will be selected during the training process.

The activation function will be a logical comparison of the weighted sum with zero. If the weighted sum of input values is greater than zero, the candlestick pattern is found and the neuron is activated. The output of the neuron is 1. If the weighted sum is less than zero, then the pattern is not found. The neuron remains deactivated and the output of the neuron is 0.

1. Basic principles of artificial intelligence construction

Now we have a neuron that will respond to the Pin Bar candlestick pattern. However, note that in a bullish pattern, the nose will be the lower tail, and in a bearish one, it will be the upper tail. That is, if we input a vector of values containing the upper tail, body, and lower tail of a candlestick, we need two neurons to define the pattern: one will define a bullish pattern and the other a bearish pattern.

Does this mean we will need to create a program for each pattern separately? No. We will combine them into a single neural network model.

In a neural network, all neurons are grouped into sequential layers. According to their location and purpose, neural layers are categorized into input, hidden, and output layers. There is always one input and one output layer, but the number of hidden layers can vary, depending on the complexity of the task at hand.

The number of neurons in the input layer corresponds to the number of inputs, which is three in our example: upper tail, body, and lower tail.

The hidden layer in our case consists of two neurons that define a bullish and a bearish pattern. The number of hidden layers and neurons in them is set during the design of the neural network and is determined by its architect depending on the complexity of the problem to be solved.

The number of hidden layers determines how the input data space is divided into subclasses. A neural network with a single hidden layer divides the input data space by a hyperplane. The presence of two hidden layers allows the formation of a convex region in the input data space. The third hidden layer allows the formation of almost any region in space.

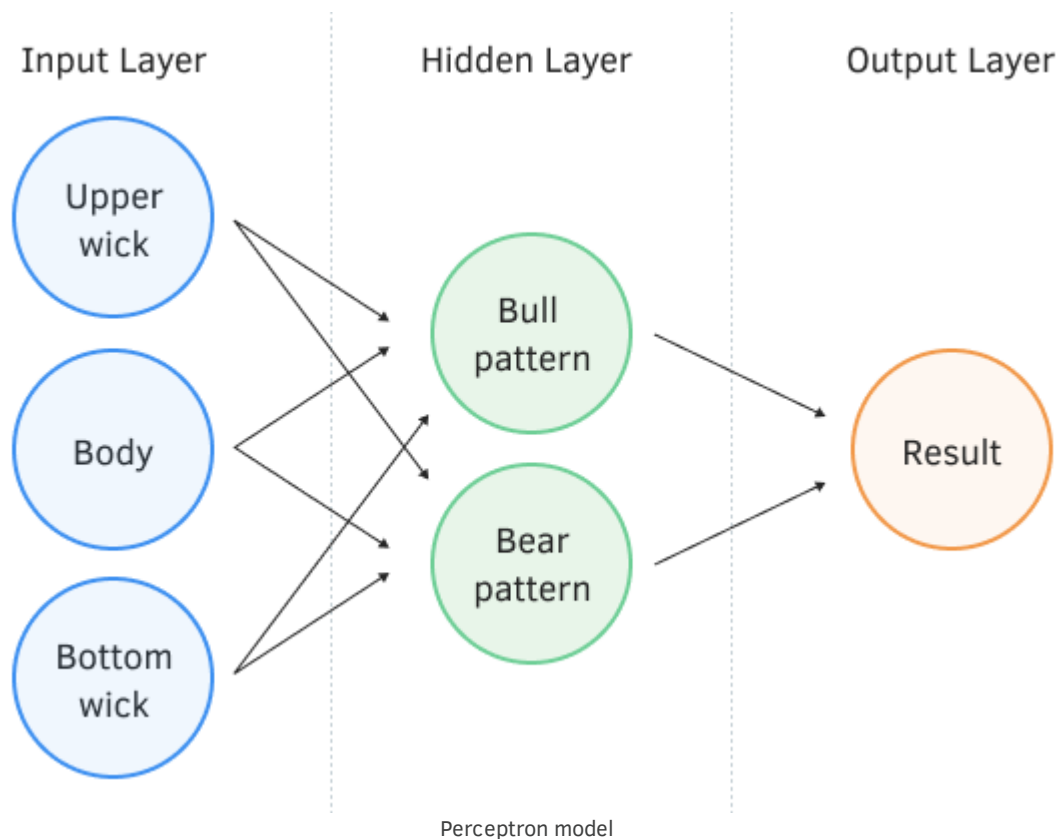
The number of neurons in the hidden layer is determined by the number of sought-after features at each level.

The number of neurons in the output layer is determined by the neural network architect depending on the possible solution variants for the given task. For binary and regression problems, a single neuron may be sufficient. For classification tasks, the number of neurons will correspond to a finite number of classes.

An exception is a binary classification, where all objects are divided into two classes. In this case, one neuron is sufficient, since the probability of assigning an object to the second class P_2 is equal to the difference between one and the probability of assigning an object to the first class P_1 .

$$P_2 = 1 - P_1$$

In our example, the output layer will contain only one neuron, which will give the result: whether to open a trade or not, and in which direction. For this, we will assign a weight of 1 to the bullish pattern, and a weight of -1 to the bearish pattern. As a result, the buy signal will be 1, while the sell signal will be -1. Zero will mean there is no trading signal.



Such a neural network model was proposed by Frank Rosenblatt in 1957 and was named *Perceptron*. This model is one of the first artificial neural network models. It is capable of establishing associative connections between input data and the resulting action. In real life, it can be compared to a person's reaction to a traffic light signal.

Of course, the perceptron is not without flaws; there are a number of limitations in its use. However, over the years of research, good results have been achieved in using the perceptron for classification and approximation tasks. Moreover, mechanisms for training the perceptron have been developed, which we will discuss shortly.

1.2 Activation functions

Perhaps one of the most challenging tasks faced by a neural network architect is the choice of the neuron activation function. After all, it is the activation function that creates the nonlinearity in the neural network. To a large extent, the neural network training process and the final result as a whole depend on the choice of the activation function.

There is a whole range of activation functions, and each of them has its advantages and disadvantages. I suggest that we review and discuss some of them in order to learn how to properly utilize their merits and address or accept their shortcomings.

Threshold (Step) activation function

The step activation function was probably one of the first to be applied. This is not surprising, as it mimics the action of a biological neuron:

- Only two states are possible (activated or not).

1. Basic principles of artificial intelligence construction

- The neuron is activated when the threshold value θ is reached.

Mathematically, this activation function can be expressed by with the following formula:

$$f(x) = \begin{cases} 1, & x \geq \theta \\ 0, & x < \theta \end{cases}$$

If $\theta=0$, the function has the following graph.



Threshold (step) activation function

This activation function is easy to understand, but its main drawback is the complexity or even the impossibility of training the neural network. The fact is that neural network training algorithms use the first-order derivative. However, the derivative of the function under consideration is always zero, except for $x=\theta$ (it is not defined at this point).

It is quite easy to implement this function in the form of *MQL5* program code. The ***theta*** constant defines the level at which the neuron will be activated. When calling the activation function, we pass the pre-calculated weighted sum of the initial data in the parameters. Inside the function, we compare the value obtained in the parameters with *theta* activation level and return the activation value of the neuron.

```
const double theta = 0;
//—
double ActStep(double x)
{
    return (x >= theta ? 1 : 0);
}
```

The *Python* implementation is also quite straightforward.

```
theta = 0
def ActStep (x):
    return 1 if x >= theta else 0
```

Linear activation function

The linear activation function is defined by a linear function:

$$f(x) = ax + b,$$

Where:

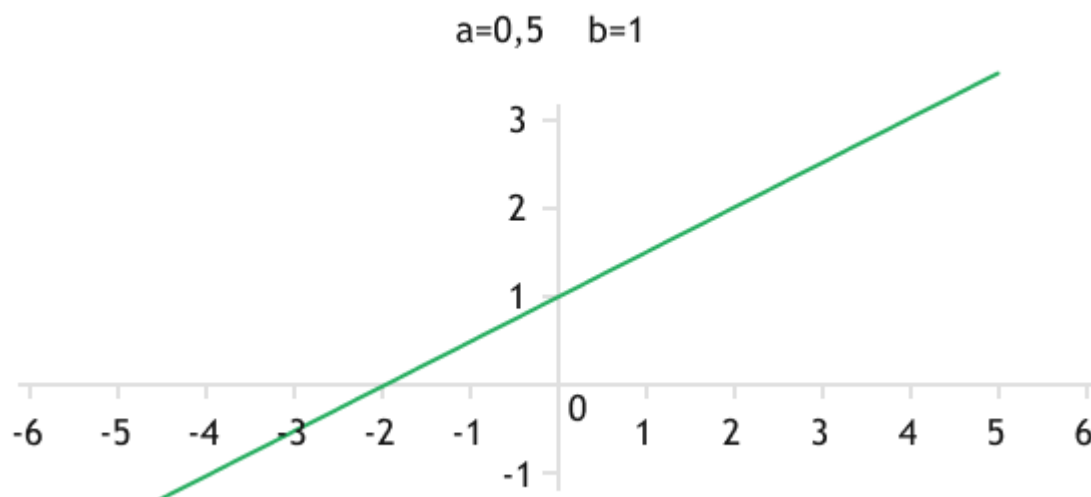
- a defines the angle of inclination of the line.
- b is the vertical displacement of the line.

As a special case of the linear activation function, if $a=1$ and $b=0$, the function has the form $f(x) = x$.

The function can generate values in the range from $-\infty$ to $+\infty$ and is differentiable. The derivative of the function is constant and equal to a , which facilitates the process of training the neural network. The mentioned properties allow for the widespread use of this activation function when solving regression problems.

It should be noted that computing the weighted sum of neuron inputs is a linear function. The application of a linear activation function gives a linear function of the entire neuron and neural network. This property prevents the use of the linear activation function for solving nonlinear problems.

At the same time, by creating nonlinearity in the hidden layers of the neural network by using other activation functions, we can use the linear activation function in the output layer neurons of our model. Such a technique can be used to solve nonlinear regression problems.



Graph of a linear function

The implementation of the linear activation function in the *MQL5* program code requires the creation of two constants: a and b . Similarly to the implementation of the previous activation function, when calling the function, we will pass the pre-calculated weighted sum of inputs in the parameters. Inside the function, the implementation of the calculation part fits into one line.

```
const double a = 1.0;
const double b = 0.0;
//—
double ActLinear(double x)
{
    return (a * x + b);
}
```

In *Python*, the implementation is similar.

```
a = 1.0
b = 0.0
def ActLinear (x):
    return a * x + b
```

Logistic activation function (Sigmoid)

The logistic activation function is probably the most common S-shaped function. The values of the function range from 0 to 1, and they are asymmetrical relative to the point [0, 0.5]. The graph of the function resembles a threshold function, but with a smooth transition between states.

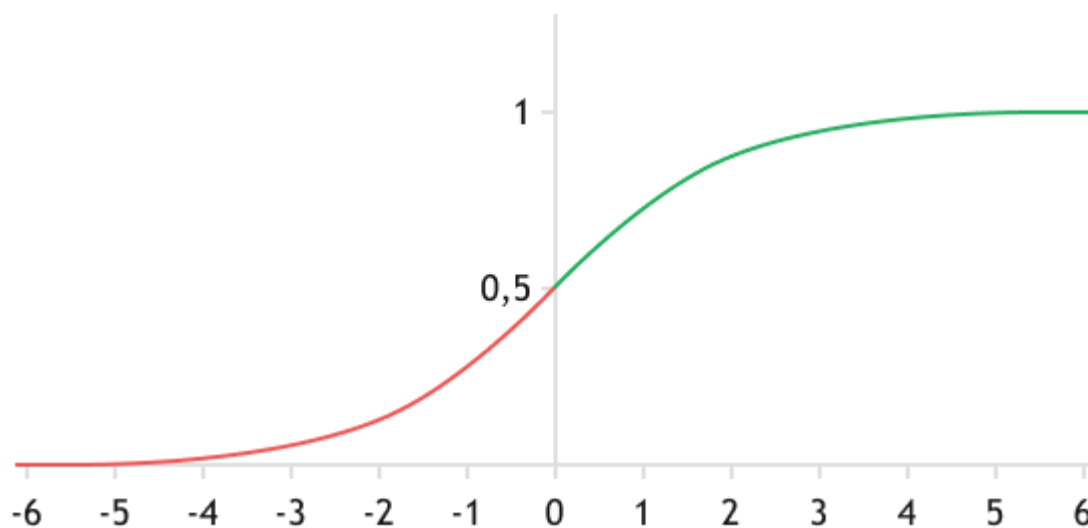
The mathematical formula of the function is as follows:

$$f(x) = \frac{1}{1 + e^{-x}}$$

This function allows the normalization of the output values of the function in the range [0, 1]. Due to this property, the use of the logistic function introduces the concept of probability into the practice of neural networks. This property is widely used in the output layer neurons when solving classification problems, where the number of output layer neurons equals the number of classes, and an object is assigned to a particular class based on the highest probability (maximum value of the output neuron).

The function is differentiable over the entire interval of permitted values. The value of the derivative can be easily calculated through the function value using the formula:

$$\frac{df(x)}{dx} = f(x) * (1 - f(x))$$



Graph of the logistic function (Sigmoid)

Sometimes a slightly modified logistic function can be used in neural networks:

$$f(x) = \frac{a}{1 + e^{-x}} - b,$$

Where:

- **a** stretches the range of function values from 0 to a.
- **b**, similarly to a linear function, shifts the resulting value.

The derivative of such a function is also calculated through the value of the function using the formula:

$$\frac{df(x)}{dx} = f(x) * \left(1 - \frac{f(x)}{a}\right)$$

In practice, the most common applications are $b = \frac{1}{2}a$, so that the graph of the function is asymmetric with respect to the origin.

All of the above properties add to the popularity to using the logistic function as a neuron activation function.

However, this function also has its flaws. For input values less than -6 and greater than 6 , the function value is pressed to the limits of the range of function values, and the derivative tends to zero. As a consequence, the error gradient also tends to zero. This leads to a decrease in the training rate of the neural network, and sometimes even makes the network nearly untrainable.

Below I propose to consider the implementation of the most general version of the logistic function with two constants **a** and **b**. Let's calculate the exponent using `exp()`.

```
const double a = 1.0;
const double b = 0.0;
//——
double ActSigmoid(double x)
{
    return (a / (1 + exp(-x)) - b);
}
```

When implementing in Python, before using the exponent function, you must import the *math* library, which contains the basic math functions. The rest of the algorithm and the function implementation are similar to the implementation in *SQL5*.

```
import math
a = 1.0
b = 0.0
def ActSigmoid (x):
    return a / (1 + math.exp(-x)) - b
```

Hyperbolic tangent (tanh)

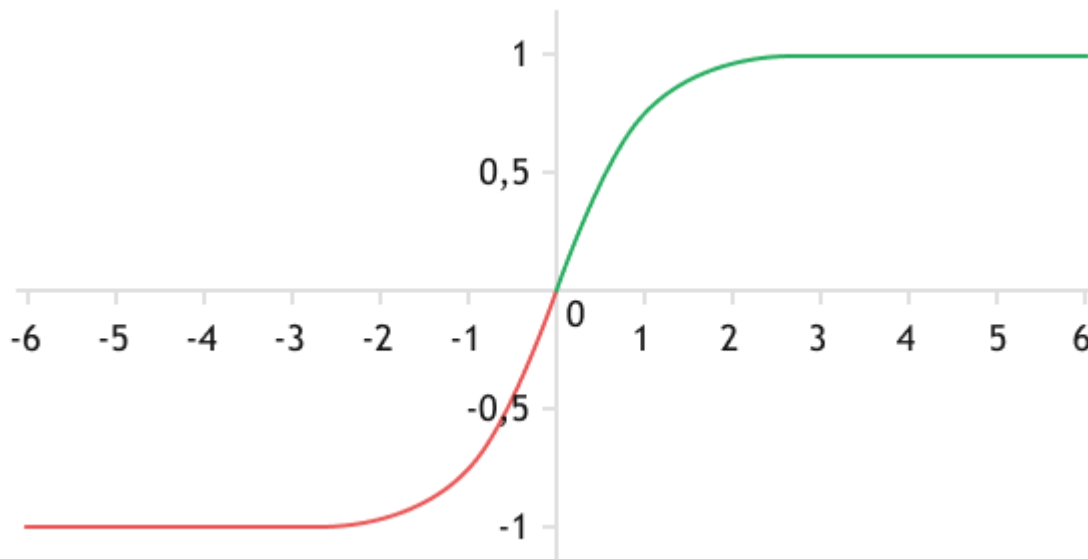
An alternative to the logistic activation function is the hyperbolic tangent (*Tanh*). Just like the logistic function, it has an S-shaped graph, and the function values are normalized. But they belong to the range from -1 to 1 , and the the neuron state is changed out 2 times faster. The graph of the function is also asymmetric, but unlike the logistic function, the center of asymmetry is at the center of coordinates.

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

The function is differentiable on the entire interval of permitted values. The derivative value can be easily calculated through the function value using the formula:

$$\frac{df(x)}{dx} = (1 + f(x))(1 - f(x)) = 1 - (f(x))^2$$

The hyperbolic tangent function is an alternative to the logistic function, which quite often converges faster.



Graph of the hyperbolic tangent function (TANH)

But it also has the main drawback of the logistic function: during saturation of the function, when the function values approach the boundaries of the value range, the derivative of the function approaches zero. As a result, the gradient of the error tends to zero.

The hyperbolic tangent function is already implemented in the programming languages we use, and it can be called by simply calling the `tanh()` function.

```
double ActTanh(double x)
{
    return tanh(x);
}
```

The implementation in Python is similar.

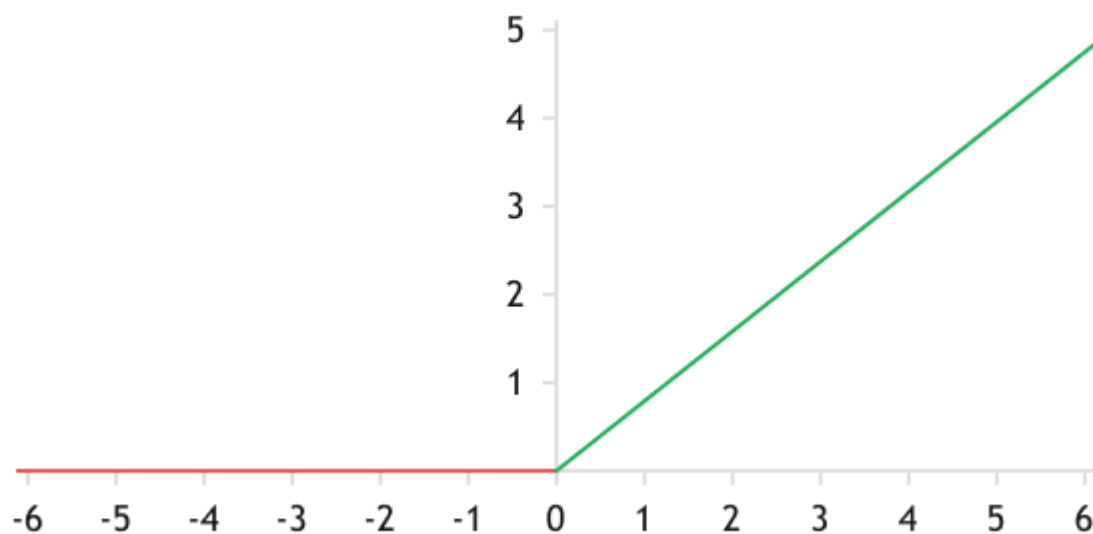
```
import math
def ActTanh (x):
    return math.tanh(x)
```

Rectified line unit (ReLU)

Another widely used activation function for neurons is *ReLU* (rectified linear unit). When the input values are greater than zero, the function returns the same value, similar to a linear activation function. For values less than or equal to zero, the function always returns 0. Mathematically, this function is expressed by the following formula:

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} = \max(0, x)$$

The graph of the function is something between a threshold function and a linear function.



Graph of the ReLU function

ReLU, probably, is one of the most common activation functions at the moment. It has become so popular due to its properties:

- Like the threshold function, it operates based on the principle of a biological neuron, activating only after reaching a threshold value (0). Unlike the threshold function, when activated, the neuron returns a variable value rather than a constant.
- The range of values is from 0 to $+\infty$, which allows us to use the function in solving regression problems.
- When the function value is greater than zero, its derivative is equal to one.
- The function calculation does not require complex computations, which accelerates the training process.

The literature provides examples where neural networks with *ReLU* are trained up to 6 times faster than networks using *TANH*.

However, the use of *ReLU* also has its drawbacks. When the weighted sum of the inputs is less than zero, the derivative of the function is zero. In such a case, the neuron is not trained and does not transmit the error gradient to the preceding layers of the neural network. In the process of training there is a probability to get such a set of weights that the neuron will be deactivated during the whole training cycle. This effect has been called "dead neurons".

Subjectively, the presence of dead neurons can be detected by observing the increase in the learning rate: the more the learning rate accelerates with each iteration, the more dead neurons the network contains.

Several variations of this function have been proposed to minimize the effect of dead neurons when using *ReLU*, but they all boil down to one thing: applying a certain coefficient a for a weighted sum less than zero.

$$f(x) = \begin{cases} x, & x > 0 \\ ax, & x \leq 0 \end{cases}$$

LReLU

Leaky ReLU

$a = 0.01$

PReLU

Parametric ReLU

The parameter a is selected in the process of training the neural network

RReLU

Randomized ReLU

The parameter a is set randomly when the neural network is created

The classical version of *ReLU* is conveniently realized using the `max()` function. Implementing its variations will require the creation of a constant or variable a . The initialization approach will depend on the chosen function (*LReLU* / *PReLU* / *RReLU*). Inside our activation function, we will create logical branching, depending on the value of the received parameter.

```
const double a = 0.01;
//—
double ActPReLU(double x)
{
    return (x >= 0 ? x : a * x);
}
```

In Python, the implementation is similar.

```
a = 0.01
def ActPReLU (x):
    return x if x >= 0 else a * x
```

Softmax

While the previously mentioned functions were calculated solely for individual neurons, the *Softmax* function is applied to all neurons of a specific layer in a network, typically the last layer. Similar to sigmoid, this function uses the concept of probability in neural networks. The range of values of the function lies between 0 and 1, and the sum of all output values of neurons of the taken layer is equal to 1.

The mathematical formula of the function is as follows:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

The function is differentiable over the entire interval of values, and its derivative can be easily calculated through the value of the function:

$$\frac{df(x_i)}{dx_i} = \begin{cases} i = j & f(x_i) * (1 - f(x_i)) \\ i \neq j & -f(x_j)f(x_i) \end{cases}$$

The function is widely used in the last layer of the neural network in classification tasks. The output value of the neuron normalized by the *Softmax* function is said to indicate the probability of assigning the object to the corresponding class of the classifier.

It is worth noting that *Softmax* is computationally intensive, which is why its application is justified in the last layer of neural networks used for multi-class classification.

1. Basic principles of artificial intelligence construction

The implementation of the *Softmax* function in *MQL5* will be slightly more complicated than the examples discussed above. This is due to the processing of the neurons of the entire layer. Consequently, the function will receive a whole array of data in its parameters rather than a single value.

It should be noted that arrays in *MQL5*, unlike variables, are passed to function parameters by pointers to memory elements rather than by values.

Our function will take pointers to two data arrays *X* and *Y* as parameters and return a logical result at the end of the operations. The actual results of the operations will be in the array *Y*.

In the function body, we first check the size of the source data array *X*. The resulting array must be of non-zero length. We then resize the array to record the *Y* results. If any of the operations fails, we exit the function with the result *false*.

```
bool SoftMax(double& X[], double& Y[])
{
    uint total = X.Size();
    if(total == 0)
        return false;
    if(ArrayResize(Y, total) <= 0)
        return false;
```

Next, we organize two loops. In the first one, we calculate exponents for each element of the obtained data set and summarize the obtained values.

```
//--- Calculation of exponent for each element of the array
    double sum = 0;
    for(uint i = 0; i < total; i++)
        sum += Y[i] = exp(X[i]);
```

In the second loop, we normalize the values of the array created in the first loop. Before exiting the function, we return the obtained values.

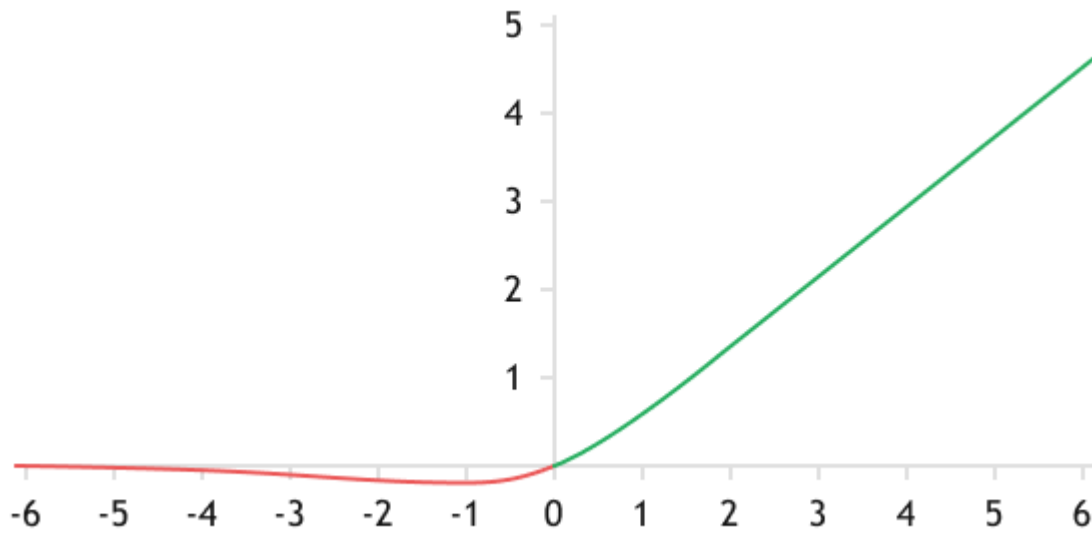
```
//--- Normalization of data in an array
    for(uint i = 0; i < total; i++)
        Y[i] /= sum;
//---
    return true;
}
```

In *Python*, the implementation looks much simpler since the *Softmax* function is already implemented in the *Scipy* library.

```
from scipy.special import softmax
def ActSoftMax (X):
    return softmax(X)
```

Swish

In October 2017, a team of researchers from Google Brain worked on the automatic search for activation features. They presented the results in the article "[Searching for Activation Functions](#)". The article summarizes the results of testing a range of features against ReLU. The best performance was achieved in neural networks with the Swish activation feature. The replacing of ReLU with Swish (without retraining) improved the performance of the neural networks.



Graph of the Swish function

The mathematical formula of the function is as follows:

$$f(x) = x * \text{sigmoid}(\beta x) = \frac{x}{1 + e^{-\beta x}}$$

The parameter β affects the nonlinearity of the function and can be taken as a constant during network design, or can be selected during training. When $\beta=0$, the function reduces to a linearly scaled function.

$$f(x) = \frac{x}{1 + e^{-x*0}} = \frac{x}{1 + e^0} = \frac{x}{2}$$

When $\beta=1$, the graph of the Swish function approaches ReLU. But unlike the latter, the function is differentiable over the entire range of values.

A function is differentiable and its derivative is calculated through the value of the function. But unlike the sigmoid, it also requires an input value to calculate the derivative. The mathematical formula for the derivative is of the form:

$$\frac{df(x)}{dx} = \beta f(x) + \frac{f(x)(1 - \beta f(x))}{x}$$

The implementation of the function in the *MQL5* program code is similar to the Sigmoid presented above. The parameter a is replaced by the obtained value of the weighted sum and the nonlinearity parameter β is added.

```
const double b=1.0;
//—
double ActSwish(double x)
{
    return (x / (1 + exp(-b * x)));
}
```

The implementation in *Python* is similar.

```
import math
```

1. Basic principles of artificial intelligence construction

```
b=1.0
def ActSwish (x):
    return x / (1 + math.exp(-b * x))
```

It is worth noting that this is by no means a complete list of possible activation functions. There are different variations to the functions mentioned above, as well as different functions can be utilized altogether. Activation functions and threshold values should be selected by the architect of the neural network. It is not always the case that all neurons in a network have the same activation function. Neural networks in which the activation function varies from layer to layer are widely used in practice. Such networks are called **heterogeneous** networks.

Later we will see that, for implementing neural networks, it is much more convenient to utilize vector and matrix operations, which are provided in *MQL5*. This in particular concerns activation functions, because matrix and vector operations provide the **Activation** function which calculate activation functions for the whole data array with one line of code. In our case, the function would be calculated for the entire neural layer. The implementation of the function is as follows.

```
bool vector::Activation(
    vector&          vect_out,      // vector to get values
    ENUM_ACTIVATION_FUNCTION activation // function type
);

bool matrix::Activation(
    matrix&          matrix_out,    // matrix to get values
    ENUM_ACTIVATION_FUNCTION activation // function type
);
```

In its parameters, the function receives a pointer to a vector or matrix (depending on the data source) for storing results, along with the type of activation function used. It should be noted that the range of activation functions in *MQL5* vector/matrix operations is much wider than described above. Their complete list is given in the table.

Identifier	Description
AF_ELU	Exponential linear unit
AF_EXP	Exponential
AF_GELU	Linear unit of Gauss error
AF_HARD_SIGMOID	Rigid sigmoid
AF_LINEAR	Linear
AF_LRELU	Leaky linear rectifier (Leaky ReLU)
AF_RELU	Truncated linear transformation ReLU
AF_SELU	Scaled exponential linear function (Scaled ELU)
AF_SIGMOID	Sigmoid
AF_SOFTMAX	Softmax
AF_SOFTPLUS	Softplus

1.2 Activation functions

Identifier	Description
AF_SOFTSIGN	Softsign
AF_SWISH	Swish function
AF_TANH	Hyperbolic tangent
AF_TRELU	Linear rectifier with threshold

1.3 Weight initialization methods in neural networks

When creating a neural network, before its first training run, we need to somehow set the initial weight. This seemingly simple task is of great importance for the subsequent training of the neural network and, in general, has a significant impact on the result of the entire work.

The fact is that the gradient descent method, which is most often used for training neural networks, cannot distinguish the local minima of a function from its global minimum. In practice, various solutions are applied to minimize this problem, and we will talk about them a bit later. However, the question remains open.

The second point is that the gradient descent method is an iterative process. Therefore, the total training time for a neural network directly depends on how far from the endpoint we are at the beginning.

Moreover, let's not forget about the laws of mathematics and the peculiarities of the activation functions that we discussed in the previous section of this book.

Initializing weights with a single value

Probably the first thing that comes to mind is to take a certain constant (0 or 1) and initialize all weights with a single value. Unfortunately, this is far from the best option, which is related to the laws of mathematics.

Using zero as a synaptic coefficient is often fatal to neural networks. In this case, the weighted sum of the input data would be zero. As we know from the previous section, many versions of the activation function in such a case return 0, and the neuron remains deactivated. Consequently, no signal goes further down the neural network.

$$S_0 = \sum_{i=1}^n 0 * x_i = 0$$

The derivative of such a function with respect to x_i will be zero. Consequently, during the training of the neural network, the error gradient through such a neuron will also not be passed to the preceding layers, paralyzing the training process.

$$\frac{dS_0}{dx_i} = 0$$

1. Basic principles of artificial intelligence construction

Using 0 for the initialization of synaptic (weight) coefficients results in an untrainable neural network, which in most cases will generate 0 (depending on the activation function) regardless of the input data received.

Using a constant other than zero as a weighting factor also has disadvantages. The input layer of the neural network is supplied with a set of initial data. All neurons of the subsequent layer work with this dataset in the same way. Within the framework of a single neuron, according to the laws of mathematics, the constant can be factored out in the formula for calculating the weighted sum. As a result, in the first stage, we get a scaling of the sum of the initial values. Changes in weights are possible during training. However, this only applies to the first layer of neurons receiving the initial data.

$$S_{const} = \sum_{i=1}^n const * x_i = const \sum_{i=1}^n x_i$$

If you look at the neural layer as a whole. Then all neurons in the same layer receive the same dataset. By using the same coefficient, all neurons generate the same signal. As a consequence, all neurons of one layer work synchronously as one neuron. This, in turn, leads to the same value being present at all inputs of all neurons of the subsequent layer. This happens from layer to layer throughout the neural network.

The applied learning algorithms do not allow the isolation of an individual neuron among a large number of identical values. Therefore, all weights will be changed synchronously during the training process. Each layer, except for the first one after the input, will receive its weights, uniform for the entire layer. This results in the linear scaling of the results obtained on the same neuron.

Initializing the synaptic coefficients with a single number other than zero causes the neural network to degenerate down to one neuron.

Initializing weights with random values

Since we cannot initialize a neural network with a single number, let's try initializing with random values. For maximum efficiency, let's not forget about what was mentioned above. We need to make sure that no two synaptic coefficients are the same. This will be facilitated by a continuous uniform distribution.

As practice has shown, such an approach yields results. Unfortunately, this is not always the case. Due to the random selection of weights, it is sometimes necessary to initialize the neural network several times before the desired result is achieved. The range of variation in the weights has a significant impact. If the gap between the minimum and maximum is large enough, some neurons will be isolated and others completely ignored.

Moreover, in deep neural networks, there is a risk of the so-called "gradient explosion" and "gradient vanishing".

The gradient explosion manifests itself when using weights greater than one. In this case, when the initial data is multiplied by factors greater than one, the weighted sum increases continuously and exponentially with each layer. At the same time, generating a large number at the output often leads to a large error.

During the training process, we will use an error gradient to adjust the weights. In order to pass the error gradient from the output layer to each neuron of our network, we need to multiply the obtained error by the weights. As a result, the error gradient, just like the weighted sum, will grow exponentially as it progresses through the layers of the neural network.

As a consequence, at some point, we will get a number that exceeds our technical capabilities for recording values, and we won't be able to further train and use the network.

The opposite situation occurs if we choose weight values close to zero. Constantly multiplying the initial data by weights less than one reduces the weighted sum of weight values. This process progresses exponentially with the increase in the number of layers of the neural network.

As a consequence, during the training process, we may encounter a situation where the gradient of a small error, when passing through layers, becomes smaller than the technically feasible precision. For our neurons, the error gradient will become zero, and they will not learn.

At the time of writing the book, the common practice is to initialize neurons using the Xavier method, proposed in 2010. Xavier Glorot and Yoshua Bengio proposed initializing the neural network with random numbers from a continuous normal distribution centered at point 0 and with a variance (δ^2) equal to $1/n$.

This approach enables the generating of synaptic coefficients such that the average of the neuron activations will be zero, and their variance will be the same for all layers of the neural network. Xavier initialization is most relevant when using hyperbolic tangent (*tanh*) as an activation function.

The theoretical justification for this approach was given in the article "[Understanding the difficulty of training deep feedforward neural networks](#)".

Xavier initialization gives good results when using sigmoid activation functions. But when ReLU is used as an activation function, it is not as efficient. This is due to the characteristics of the ReLU itself.

$$ReLU(x_i) = \max(0, x_i)$$

Since ReLU only misses positive weighted sum values, and negative ones are zeroed, the probability theory states that half the neurons will be deactivated most of the time. Consequently, the neurons of the subsequent layer will receive only half of the information, and the weighted sum of their inputs will be less. As the number of layers in the neural network increases, this effect will intensify: fewer and fewer neurons will reach the threshold value, and more and more information will be lost as it passes through the neural network.

A solution was proposed by Kaiming He in February 2015 in the article "[Surpassing Human-Level Performance on ImageNet Classification](#)". In the article, it's suggested to initialize the weights for neurons with ReLU activation from a continuous normal distribution with a variance (δ^2) equal to $2/n$. And when using PReLU as activation, the distribution variance should be $2/((1+a^2) * n)$. This method of initializing synaptic scales is called "He-initialization".

Initializing with a random orthogonal matrix

In December 2013, Andrew M. Saxe presented a three-layer neural network in the form of matrix multiplication in the article "[Exact solutions to the nonlinear dynamics of learning in deep linear neural networks](#)", thereby showing the correspondence between the neural network and singular decomposition. The synaptic weight matrix of the first layer is represented by an orthogonal matrix, the vectors of which are the coordinates of the initial data in some n -dimensional space.

Since the vectors of an orthogonal matrix are orthonormalized, the initial data projections they generate are completely independent. This approach allows for the neural network to be pre-prepared in such a way that each neuron will learn to recognize its feature in the input data independently of the training of other neurons located in the same layer.

However, the method is not used widely, primarily due to the complexity of generating orthogonal matrices. The advantages of the method are demonstrated with the growth of the number of layers of the neural network. Therefore, in practice, initialization with orthogonal matrices can be found in deep neural networks when initialization with random values does not yield results.

Using pre-trained neural networks

This method can hardly be referred to as initialization, but its practical application is becoming increasingly popular. The essence of the method is as follows: to solve the problem, use a neural network that was trained on the same or similar data but solves different tasks. A series of lower layers are taken from a pre-trained neural network. These layers have already been trained to extract features from the initial data. Then, a few new layers of neurons are added, which will solve the given task based on the already extracted features.

In the first step, pre-trained layers are blocked and new layers are trained. If the training fails to produce the desired result, the learning block is removed from the borrowed neural layers and the neural network is retrained.

A variation of this method is the approach of first creating a multilayer neural network and training it to extract different features from the initial data. These can be unsupervised learning algorithms for dividing data into classes or autoencoder algorithms. In the latter, the neural network first extracts features from the initial data and then tries to return the original data based on the selected features.

After pre-training, the layers of neurons responsible for feature extraction are taken, and additional layers of neurons for solving the given task are added to them.

When constructing deep networks, this approach can help train the neural network faster compared to training a large neural network directly. This is because, during one training pass, a smaller neural network requires fewer operations to be performed compared to training a deep neural network. In addition, smaller neural networks are less prone to the risk of gradient explosion or vanishing.

In the practical part of the book, we will return to the process of initializing neural networks and in practice evaluate the advantages and disadvantages of each method.

1.4 Neural network training

We have already learned about the structure of an artificial neuron, the organization of data exchange between neurons, and the principles of neural network construction. We also learned how to initialize synaptic coefficients. The next step is to train the neural network.

Tom Mitchell proposed the following definition of machine learning:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks T , as measured by P , improves with experience E ."

Usually, three main approaches to neural network training are distinguished:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

Algorithms of *supervised learning* in practice produce the best results, but they require a lot of preparatory work. The very principle of supervised learning implies that there are correct answers. As a

1. Basic principles of artificial intelligence construction

supervisor, at each iteration of learning, we will guide the neural network by showing it the correct results, thereby encouraging the neural network to memorize what is true and what is false.

In this approach, correlation links between the raw data and the correct answers are set up within the neural network. Ideally, a neural network should learn to extract essential features from the set of initial data. By generalizing the set of extracted features, it should determine the object's affiliation to a particular class (classification tasks) or indicate the most probable course of events (regression tasks).

The complexity of this approach lies in the need for extensive preparatory work. This approach requires mapping the correct answers to each set of initial data from the training sample. It is not always possible to automate this work, and human resources have to be involved. At the same time, using manpower to prepare the training set and correct answers increases the risk of errors in the sample and, as a result, improper configuration of the neural network.

Another risk of this approach is the overfitting of the neural network. This phenomenon usually occurs when training deep networks on a small set of input data. In such a case, the neural network is able to "memorize" all pairs of initial data sets with correct answers. In doing so, it will lose any ability to generalize the data. As a result, we will get a neural network with excellent results on the training data set and completely random answers on the test sample and when using it on real data.

The risk of neural network overfitting can be reduced by using various regularization, normalization, and dropout methods which we will discuss later.

Also, note that we can encounter a task where there is no clear correct answer for the presented data sets. In such cases, other approaches to training neural networks are used.

Unsupervised learning is used when there are no correct answers for the training sample. Unsupervised learning algorithms allow for the extraction of individual features of the original data objects. Comparing the extracted features, the algorithms cluster the original data, grouping the most similar objects into certain classes. The number of such classes is specified in the hyperparameters of the neural network.

Cost saving during the preparation stage comes at the expense of object recognition quality and a more limited range of solvable tasks.

With the growth of the volume of initial data for training, unsupervised learning algorithms are widely used for the preliminary training of neural networks. Initially, a neural network is created and trained unsupervised on a large dataset. This allows us to train the neural network to extract individual features from the initial data set and to partition a large amount of data into separate classes of objects.

Then, decision-making neural layers (most often fully connected perceptron neural layers) are added to the pre-trained network, and the neural network is further trained using supervised learning algorithms.

This approach allows training the neural network on a large volume of initial data, which helps minimize the risk of overfitting the neural network. At the same time, since training on the primary dataset occurs unsupervised, we can further train the deep neural network on a relatively small set of paired original data with correct answers. This reduces the resources required for preparatory work during supervised training.

A separate approach to neural network training can be called *reinforcement learning*. This approach is used to solve optimization problems that require constructing a strategy. The best results are demonstrated when training neural networks with computer and logic games, for which this method was developed. It is applicable for long finite processes, when throughout the process the neural network needs to make a series of decisions based on the state of the environment, and the cumulative result of the decisions taken will only be clear at the end of the process. For example, winning or losing a game.

The essence of the method is to assign some sort of reward or penalty for each action. During the training process, the strategy with the maximum reward is determined.

In this book, more attention will be given to supervised learning, which in practice shows the best training results and is applicable for solving regression tasks, including time series forecasting.

1.4.1 Loss functions

When starting training, it is necessary to choose methods for determining the quality of network training. Training a neural network is an iterative process. At each iteration, we need to determine how accurate the neural network calculations are. In the case of supervised learning, it refers to how much they differ from the reference. By knowing the deviation only can we understand how much and which way we need to adjust the synaptic coefficients.

Therefore, we need a certain metric that will impartially and mathematically accurately indicate the error of the neural network's performance.

At first glance, it is quite a trivial task to compare two numbers (the calculated value of the neural network and the target). But as a rule, at the output of a neural network, we get not one value, but an entire vector. To solve this problem, let's turn to mathematical statistics. Let us introduce a *loss function* that depends on the calculated value (y') and the reference value (y).

$$L(y, y')$$

This function should determine the deviation of the calculated value from the reference value (error). If we consider the computed and target values as points in space, then the error can be seen as the distance between these points. Therefore, the loss function should be continuous and non-negative for all permitted values.

In an ideal state the calculated and reference values are the same, and the distance between the points is zero. Therefore, the function must be convex downwards with a minimum at $L(y, y')=0$.

The book "[Robust and Non-Robust Models in Statistics](#)" by L.B.Klebanov describes four properties that a loss function should have:

- Completeness of information
- The absence of a randomization condition
- Symmetry condition
- Rao-Blackwell state (statistical estimates of parameters can be improved)

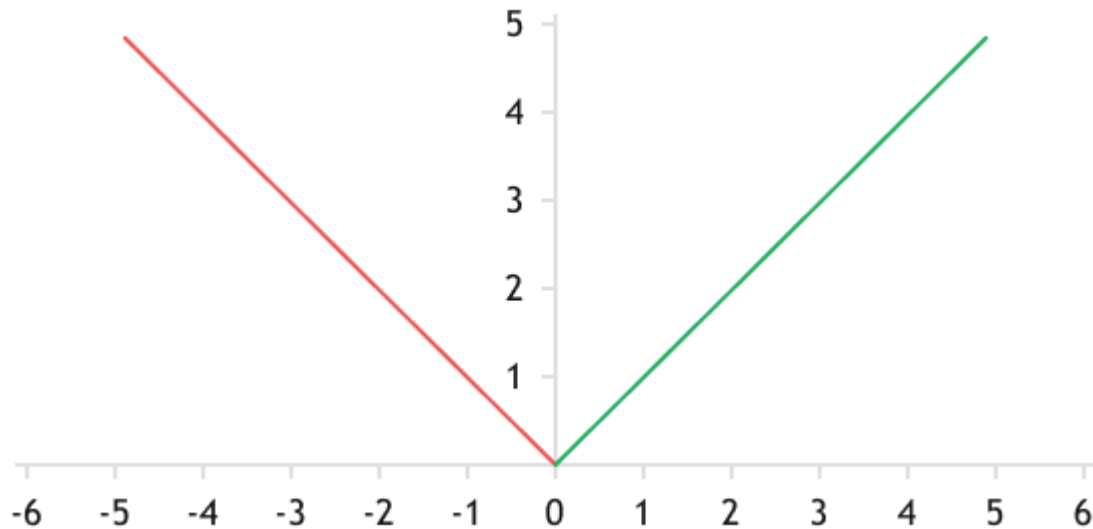
The book presents quite a few mathematical theorems and their proofs. It demonstrates the relationship between the choice of loss function and a statistical estimate. As a consequence, certain statistical issues can be resolved through the proper choice of the loss function.

Mean Absolute Error (MAE)

One of the earliest loss functions, the Mean Absolute Error (MAE), was introduced by the 18th-century French mathematician Pierre-Simon Laplace. He proposed using the absolute difference between the reference and computed values as a measure of deviation.

$$L(y, y') = \frac{1}{n} \sum_{i=1}^n |y_i - y_i'|$$

The function has a graph that is symmetric about zero, and linear before and after zero.



Graph of the Mean Absolute Deviation function

The use of Mean Absolute Error provides a linear approximation of the analytical function to the training dataset across the entire range of error.

Let's look at the implementation of this function in *MQL5* program code. To calculate deviations, the function must receive two data vectors: calculated and reference values. This data will be passed as parameters to the function.

At the beginning of the method, we compare the size of the resulting arrays. Ideally, array sizes should be at least zero. If the check fails, we exit the function with a result of the maximum possible error, *DBL_MAX*.

```
double MAE(double &calculated[], double &target[])
{
    double result = DBL_MAX;
    //---
    if(calculated.Size() < target.Size() || target.Size() <= 0)
        return result;
```

After successfully passing the checks, we create a loop to accumulate the absolute values of deviations. In conclusion, we divide the accumulated sum by the number of reference values.

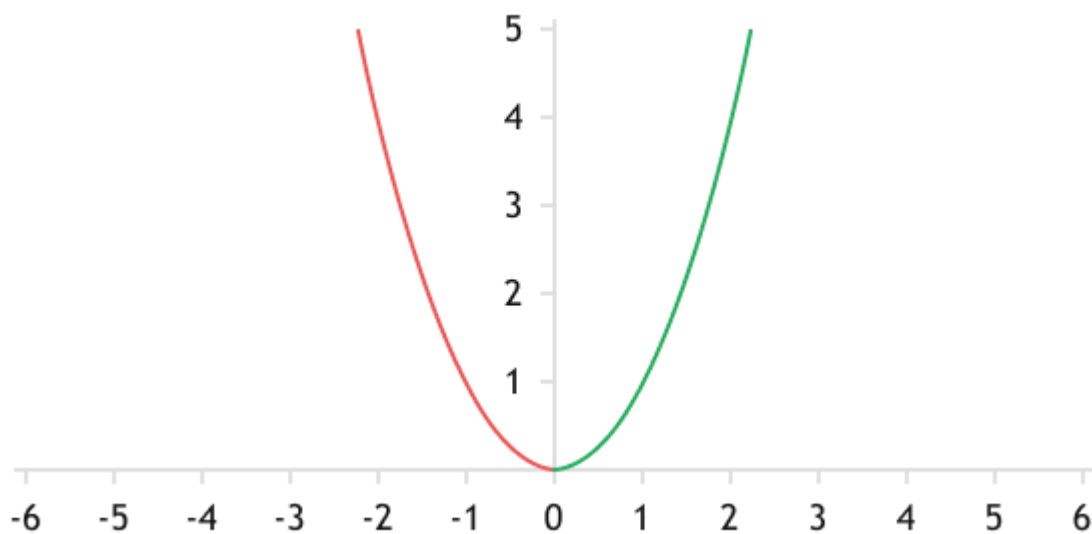
```
//---
    result = 0;
    int total = target.Size();
    for(int i = 0; i < total; i++)
        result += MathAbs(calculated[i] - target[i]);
    result /= total;
    //---
    return result;
}
```

Mean Squared Error (MSE)

The 19th-century German mathematician Carl Friedrich Gauss proposed using the square of the deviation instead of the absolute value in the formula for mean absolute deviation. The function is called the standard deviation.

$$L(y, y') = \frac{1}{n} \sum_{i=1}^n (y_i - y'_i)^2$$

Thanks to squaring the deviation, the error function takes the form of a parabola.



Graph of the mean squared deviation function

When using mean squared deviation, the speed of error compensation is higher when the error itself is larger. When the error decreases, the speed of its compensation also decreases. In the case of neural networks, this allows for faster convergence of the neural network with large errors and finer tuning with small errors.

But there is a flip side to the coin: the property mentioned above makes the function sensitive to noisy phenomena, as rare, large deviations can lead to a bias in the function.

Currently, the use of mean squared error as a loss function is widely employed in solving regression problems.

The algorithm for implementing *MSE* in *ML5* is similar to implementing *MAE*. The only difference is in the body of the loop, where the sum of the squares of the deviations is calculated instead of their absolute values.

```
double MSE(double &calculated[], double &target[])
{
    double result = DBL_MAX;
    //---
    if(calculated.Size() < target.Size() || target.Size() <= 0)
        return result;
```

```
//---
    result = 0;
    int total = target.Size();
    for(int i = 0; i < total; i++)
        result += Math.Pow(calculated[i] - target[i], 2);
    result /= total;
//---
    return result;
}
```

Cross-entropy

For solving classification tasks, the cross-entropy function is most commonly used as the loss function.

Entropy is a measure of uncertainty in distribution.

Applying entropy shifts calculations from the realm of absolute values into the realm of probabilities. Cross-entropy defines the similarity of probabilities of events occurring in two distributions and is calculated using the formula:

$$L(y, y') = - \sum_{i=1}^n p(y_i) * \log(p(y'_i)),$$

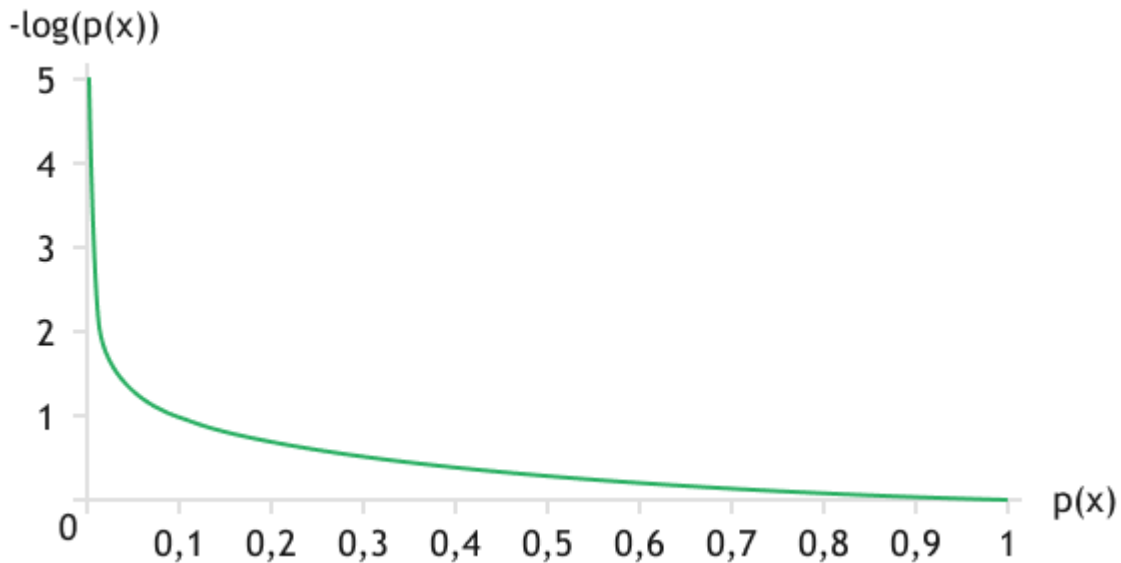
where:

- $p(y_i)$ = the probability of the i th event occurring in the reference distribution
- $p(y'_i)$ = the probability of the i th event occurring in the calculated distribution

Since we are examining probabilities of events occurring, the probability values of an event always lie within the range of 0 to 1. The value of the logarithm in this range is negative, so adding a minus sign before the function shifts its value into the positive range and makes the function strictly decreasing. For clarity, the logarithmic function graph is shown below.

During training, for events in the reference distribution, when an event occurs, its probability is equal to one. The probability of a missing event occurring is zero. Based on the graph of the function, the event that occurred in the reference distribution but was not predicted by the analytical function will generate the highest error. Thus, we will stimulate the neural network to predict expected events.

It is the application of the probabilistic model that makes this function most attractive for classification purposes.



Graph of the logarithmic function

An implementation of this feature is presented below. The implementation algorithm is similar to the previous two functions.

```
double LogLoss(double &calculated[], double &target[])
{
    double result = DBL_MAX;
//---
    if(calculated.Size() < target.Size() || target.Size() <= 0)
        return result;
//---
    result = 0;
    int total = target.Size();
    for(int i = 0; i < total; i++)
        result -= target[i] * MathLog(calculated[i]);
//---
    return result;
}
```

Only three of the most commonly used loss functions are described above. But in fact, their number is much higher. And here, as in the case of activation functions, we will be assisted by vector and matrix operations implemented in *MQL5*, among which the *Loss* function is implemented. This function allows to compute the loss function between two vectors/matrices of the same size in just one line of code. The function is called for a vector or matrix of calculated values. The parameters of the function include a vector/matrix of reference values and the type of loss function.

```
double vector::Loss(
    const vector& vect_true, // true value vector
    ENUM_LOSS_FUNCTION loss // loss function type
);

double matrix::Loss(
    const matrix& matrix_true, // true value matrix
    ENUM_LOSS_FUNCTION loss // loss function type
);
```

MetaQuotes provides 14 readily implemented loss functions. These are listed in the table below.

Identifier	Description
LOSS_MSE	Mean squared error
LOSS_MAE	Average absolute error
LOSS_CCE	Categorical cross-entropy
LOSS_BCE	Binary cross-entropy
LOSS_MAPE	Average absolute error in percentages
LOSS_MSLE	Mean-squared logarithmic error
LOSS_KLD	Kulback-Leibler divergence
LOSS_COSINE	Cosine similarity/proximity
LOSS_POISSON	Poisson loss function
LOSS_HINGE	Hinge loss function
LOSS_SQ_HINGE	Quadratic piecewise linear loss function
LOSS_CAT_HINGE	Categorical piecewise linear loss function
LOSS_LOG_COSH	The logarithm of the hyperbolic cosine
LOSS_HUBER	Huber loss function

1.4.2 Error gradient backpropagation method

Once we have defined the loss function, we can move on to training the neural network. The actual learning process involves iteratively adjusting the neural network parameters (synaptic weights) at which the value of the neural network [loss function](#) will be minimized.

From the previous section, we learned that the loss function is concave downward. Therefore, when starting the training from any point on the loss function graph, we should move in the direction of minimizing the error. For complex functions like a neural network, the most convenient method is the gradient descent algorithm.

The gradient of a multi-variable function (which a neural network is) is defined as a vector composed of the partial derivatives of the function with respect to its arguments. From our mathematics course, we know that the derivative of a function characterizes the rate of change of the function at a given point.

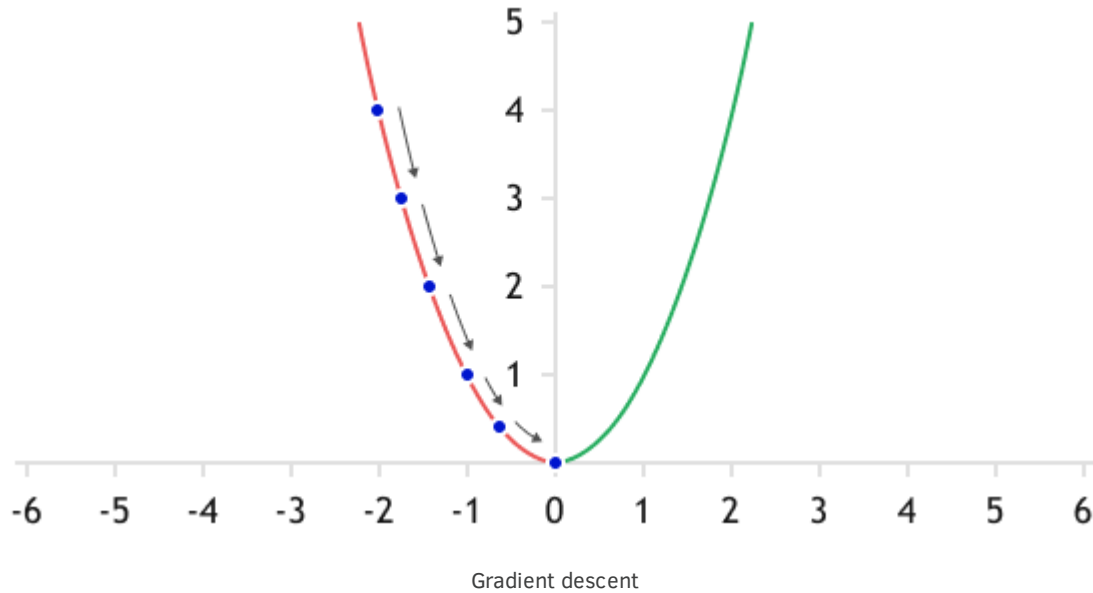
Hence, the gradient indicates the direction of the fastest growth of the function. Moving in the direction of the negative gradient (opposite to the gradient), we will descend at the maximum speed towards the minimum of the function.

The algorithm of action will be as follows:

1. Initialize the weights of the neural network using one of the ways described [earlier](#).
2. Compute the predicted data on the training sample.

1. Basic principles of artificial intelligence construction

3. Using the [loss function](#), calculate the computational error of the neural network.
4. Determine the gradient of the loss function at the obtained point.
5. Adjust the synaptic coefficients of the neural network towards the negative gradient.

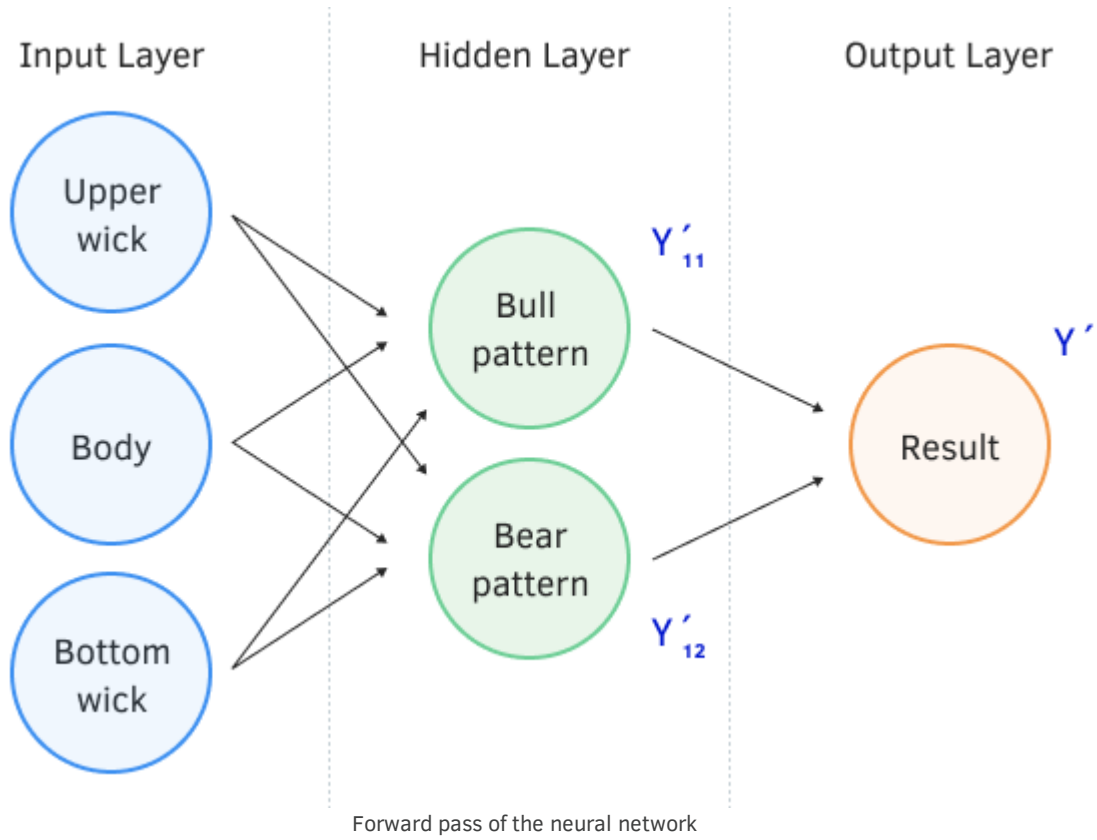


Since a nonlinear loss function is used often, the direction of the anti-gradient vector will change at each point on the loss function graph. Therefore, we will reduce the loss function gradually, getting closer and closer to the minimum with each iteration.

At first glance, the algorithm is quite simple and logical. But how do we technically implement point 5 of our algorithm in the case of a multilayer neural network?

This issue is addressed using the backpropagation algorithm, which consists of two main components:

1. *Forward pass*. Point 2 from our algorithm above. During the forward pass, a set of data from the training sample is fed to the input of the neural network and processed in the neural network sequentially from the input layer to the output layer. The intermediate values on each neuron are preserved.



2. The *backward pass* includes steps 3-5 of our algorithm.

At this point, it's worth recalling some mathematics. We talk about partial derivatives of a function, but we also want to train a neural network that consists of a large number of neurons. At the same time, each neuron represents a complex function, and to update the weights of the neural network, we need to calculate the partial derivatives of the composite function of our neural network with respect to each weight.

According to the rules of mathematics, the derivative of a composite function is equal to the product of the derivative of the outer function and the derivative of the inner function.

$$\frac{dF(G(x))}{dx} = \frac{dF(G(x))}{dG(x)} \frac{dG(x)}{dx}$$

Let us use this rule and find the partial derivatives of the loss function L by the weight of the output neuron w_i and by the i th input value x_i .

$$\frac{dL(A(S(X, W)))}{dw_i} = \frac{dL(A(S(X, W)))}{dA(S(X, W))} \frac{dA(S(X, W))}{dS(X, W)} \frac{dS(X, W)}{dw_i},$$

$$\frac{dL(A(S(X, W)))}{dx_i} = \frac{dL(A(S(X, W)))}{dA(S(X, W))} \frac{dA(S(X, W))}{dS(X, W)} \frac{dS(X, W)}{dx_i},$$

Where:

- L = loss function
- A = activation function of the neuron

1. Basic principles of artificial intelligence construction

- S = weighted sum of the raw data
- X = vector of initial data
- W = vector of weights
- w_i = i th weighting factor for which the derivative is calculated
- x_i = i th element of the initial data vector

The first thing to notice in the formulas presented above is the complete coincidence of the first two multipliers. I.e., when calculating partial derivatives on weights and initial data, we only need to calculate the error gradient in front of the activation function once, and using this value, calculate partial derivatives for all elements of the vectors of weights and initial data.

Using a similar method, we can determine the partial derivative with respect to the weight of one of the neurons in the hidden layer that precedes the output neuron layer. For this purpose, in the previous formula we replace the vector of initial data with the function of the hidden layer neuron. The vector of weights will be transformed into a scalar value of the corresponding weight.

$$\frac{dL(A(S(A_h(S_h(X_h, W_h))), w_i))}{dw_h} = \frac{dL(A(S(A_h(S_h(X_h, W_h))), w_i))}{dA(S(A_h(S_h(X_h, W_h))), w_i)} \frac{dA(S(A_h(S_h(X_h, W_h))), w_i)}{dS(A_h(S_h(X_h, W_h))), w_i)} \dots \frac{dS_h(X_h, W_h)}{dw_h},$$

Where:

- A_h = activation function of the hidden layer neuron
- S_h = weighted sum of the original data of the hidden layer neuron
- X_h = vector of initial data for the hidden layer neuron
- W_h = vector of weights of the hidden layer neuron
- w_h = weight of the hidden layer for which the derivative is calculated

Note that if in the last formula, we return X instead of the function of the hidden layer neuron, we see in the first function multipliers the function of the private derivative of the i th input value presented above.

$$X = A_h(S_h(X_h, W_h)),$$

$$\frac{dL(A(S(A_h(S_h(X_h, W_h))), w_i))}{dw_h} = \frac{dL(A(S(X, W)))}{dA(S(X, W))} \frac{dA(S(X, W))}{dS(X, W)} \frac{dS(X, W)}{dx_i} \dots \frac{dS_h(X_h, W_h)}{dw_h}.$$

Hence,

$$\frac{dL(A(S(A_h(S_h(X_h, W_h))), w_i))}{dw_h} = \frac{dL(A(S(X, W)))}{dx_i} \frac{dA_h(S_h(X_h, W_h))}{dS_h(X_h, W_h)} \frac{dS_h(X_h, W_h)}{dw_h}.$$

Similar formulas can be provided for each neuron in our network. Thus, we can calculate the derivative and error gradient of the neuron output once, and then propagate the error gradient to all the connected neurons in the previous layer.

Following this logic, we first determine the deviations from the reference value using the loss function. The loss function can be anything that satisfies the requirements described in the previous section.

$$L(Y, Y'),$$

Where:

- Y = vector of reference values
- Y' = vector of values at the output of the neural network

Next, we determine how the states of the neurons in the output layer should change in order for our loss function to reach its minimum value. From a mathematical perspective, we determine the error gradient on each neuron in the output layer by calculating the partial derivative of the loss function with respect to each parameter.

$$grad_i^{out} = \frac{\delta L(Y, Y')}{\delta y_i'} * L(Y, Y')$$

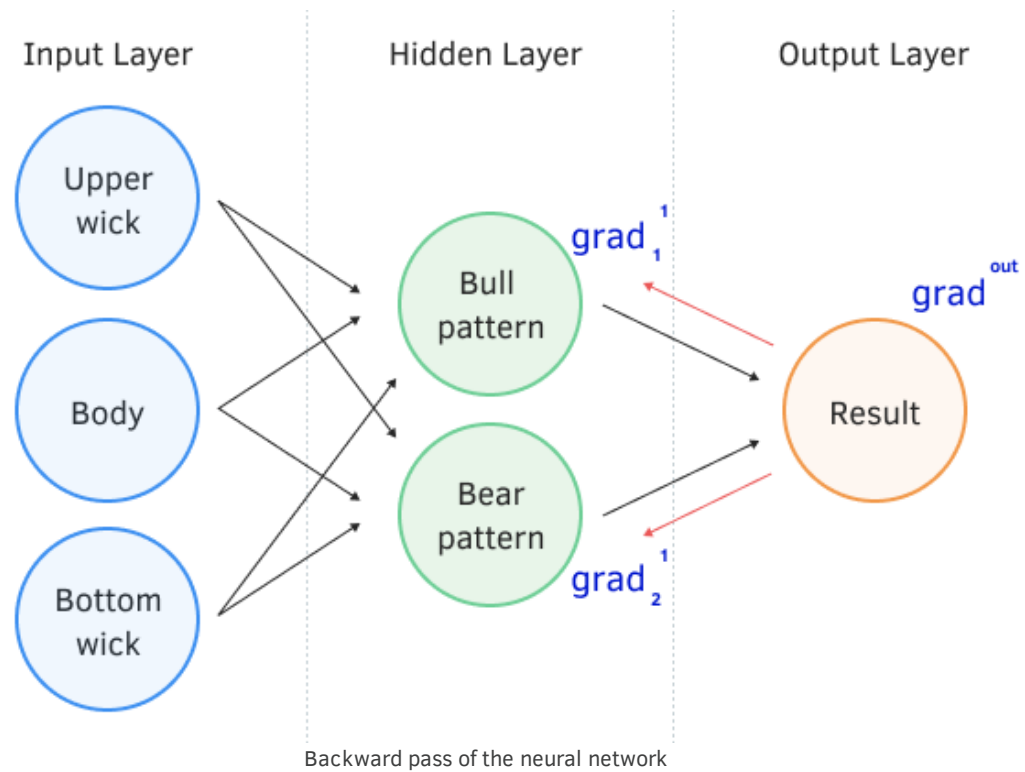
We then "descend" the error gradient from the output neural layer to the input layer by running it sequentially through all the hidden neural layers of our network. In this way, we are effectively bringing the reference value to each neuron at this stage of training.

$$grad_i^{j-1} = \sum_{k=1}^n \frac{\delta A_{kj}(S_{kj}(W_{kj}Y'_{j-1}))}{\delta y'_{ij-1}} grad_k^j,$$

Where:

- $grad_i^{j-1}$ = gradient at the output of the i th neuron of the $j-1$ layer
- A_{kj} = the activation function of the k th neuron on the j th layer
- S_{kj} = weighted sum of incoming data of the k th neuron on the j th layer
- W_{kj} = vector of synaptic coefficients of the k th neuron on the j th layer

After obtaining the error gradients at the output of each neuron, we can proceed to adjust the synaptic coefficients of the neurons. For this purpose, we will go through all layers of the neural network one more time. At each layer, we will search all neurons and for each neuron, we will update all synaptic connections.



We will talk about ways to update the weights in the next chapter.

After updating the weights, we return to step 2 of our algorithm. The cycle is repeated until the minimum of the function is found. Determining the achievement of the minimum can be done by observing zero partial derivatives. In general, it will be noticeable by the absence of change of the error at the output of the neural network after the next cycle of updating the weights, because at zero derivatives the process of training of the neural network stops.

1.4.3 Methods for optimizing neural networks

We continue to move forward in studying the basic principles of neural network training. In previous chapters, we have already discussed various [loss functions](#) and the [error gradient backpropagation algorithm](#), which allows us to determine the influence of each neuron on the overall result and the direction of change in the output value of each neuron to minimize the overall error at the output.

Below is the formula of the mathematical model of a neuron.

$$OUT = f(W, X) = f\left(\sum_{i=1}^n w_i x_i\right),$$

Where:

- f = activation function
- n = number of elements in the input sequence
- w_i = weight of the i th element of the sequence
- x_i = i th element of the input sequence

Gradient descent and stochastic gradient descent

In the above formula, you can see that the output value of the neuron depends on the activation function, the input sequence, and the weight vector. The activation formula is set during the construction of the neural network and remains unchanged. The neuron does not affect the input sequence. Therefore, in the learning process, we can change the value at the output of the neuron only by choosing the optimal values of the weights.

The rate of change of the neuron's output value when changing a particular weight is equal to the partial derivative of the neuron's mathematical model function with respect to that weight. From this, to get the delta change of a particular weight, you need to multiply the error at the neuron's output in the current situation by the partial derivative of the neuron's mathematical model with respect to that weight.

It should be noted that the function of the mathematical model of a neuron most often is non-linear. Therefore, the partial derivative is not constant over the entire permitted range of values. Therefore, the "learning coefficient" parameter is introduced into the formula for updating the weights, which determines the learning rate.

The approach described above is called **gradient descent**. In general, it can be expressed by the formula:

$$w_{il}^j = w_{il}^j - \alpha \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_i^j,$$

where:

- w_{il}^j = l th weight on the i th neuron of the j th layer
- α = learning coefficient that determines the learning rate
- $grad_i^j$ = gradient at the output of the i th neuron of the j th layer

The training coefficient α is a hyperparameter that is selected during the neural network validation process. It is selected from the range 0 to 1. In this case, the training coefficient cannot be equal to the extreme values.

When $\alpha=0$, there will be no learning, because by multiplying any gradient by zero, we always get zero for updating the weights.

When $\alpha=1$ or close to this value, there is another problem. If, when moving towards the error minimum, the value of the partial derivative decreases, then using a sufficiently large step will throw us past the minimum point. In the worst case, the error at the output of the neural network will even increase. Moreover, a large learning coefficient promotes maximum adaptation of the network to the current situation. In this case, the ability to generalize is lost. Such training will not be able to identify key features and adequately work "in the field."

The method works well with small neural networks and datasets. But neural networks are getting bigger, and training sets are growing. To make one training iteration, you need to perform a forward pass through the entire dataset and save information about the state of all neurons for all samples as this information will be needed for the backward pass. Consequently, we will need additional memory allocation in the amount of the number of neurons * the number of data sets.

The solution was found in the use of ***stochastic gradient descent***. The method retains the algorithm of standard gradient descent, but the weights are updated for each randomly selected data set.

On each iteration, we randomly select one data set from the training sample. Then we perform a forward and backward pass and update the weights. After that, we "shuffle" the training sample and select the next data set randomly.

We repeat the iterations until we achieve an acceptable error in the neural network output.

Stochastic gradient descent has lower convergence compared to standard gradient descent. Furthermore, more iterations are usually required to achieve an acceptable error. But in stochastic gradient descent, a single iteration takes less time, since it is carried out on a randomly chosen data set, rather than on the entire sample, as in standard gradient descent. In general, the process of training a neural network is carried out with less time and resource costs.

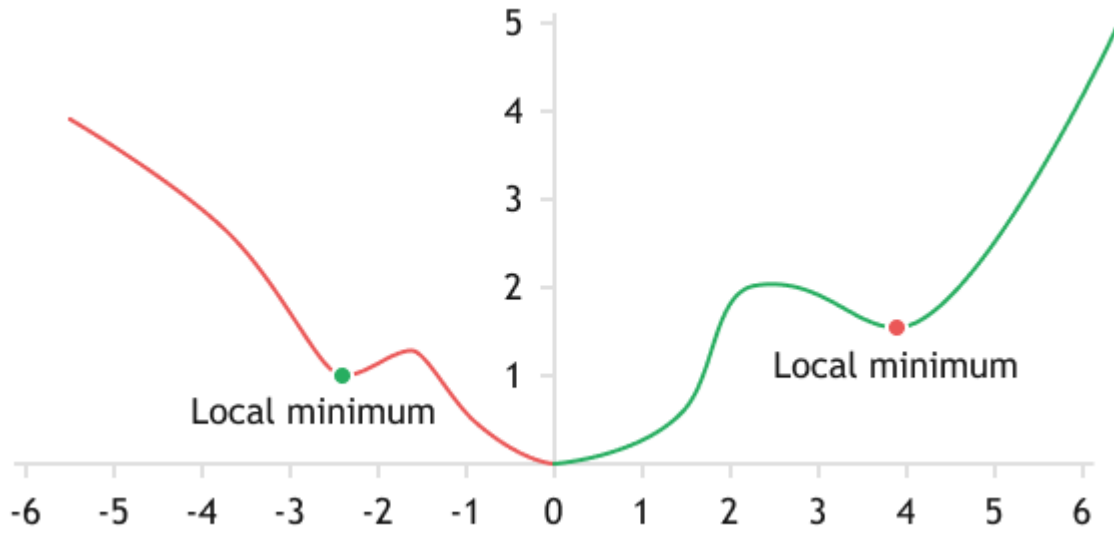
Below is an example of implementing a function to update weights using the gradient descent method. In parameters, the function receives two pointers to data arrays (current weights and gradients to them) and a training coefficient. First, we check the correspondence of the sizes of the arrays. Then, we organize a loop where for each element of the weight array, we calculate a new value using the formula mentioned above. The obtained value is saved in the corresponding cell of the array.

```
bool SGDUpdate(double &m_cWeights[],
               double &m_cGradients[],
               double learningRate)
{
    if(m_cWeights.Size() > m_cGradients.Size() ||
       m_cWeights.Size() <= 0)
        return false;
    //---
    for(int i = 0; i < m_cWeights.Size(); i++)
        m_cWeights[i] -= learningRate * m_cGradients[i];
    return true;
}
```

Momentum accumulation method

Probably, the main drawback of gradient descent methods is the inability to distinguish between local and global minima. During the training process, there is always a high risk of stopping at a local minimum without reaching the desired accuracy level of the neural network.

Careful selection of the learning coefficient, experiments with different weight initialization options, and several training iterations do not always yield the desired results.



Local minima on the graph of a complex function

One of the solutions was borrowed from the physics of natural phenomena. If you put the ball in a small depression or hole, then it will lie motionless at its bottom. But as soon as we send the same ball along some inclined surface into this hole, it will easily jump out of it and roll further. This behavior of the ball is explained by the momentum accumulated during the descent along the inclined surface.

Similar to the ball, it was suggested to accumulate momentum during the training of the weights, and then add this momentum to the weight update formula using the gradient descent method.

Momentum accumulates for each individual weight. When updating a specific weight for a prolonged period in one direction, its momentum will accumulate, and as a result, it will move towards the desired goal at a faster pace. Thanks to the accumulated energy, we can overcome local minima, similar to a ball rolled down an inclined surface.

Unfortunately, there is also a flip side to the coin. By accumulating momentum, we will skip not only the local minimum but also the global one. With unlimited momentum accumulation, the value of our error will move like a pendulum on the loss function graph. Let's mimic the force of friction in nature and add the β coefficient in the range from 0 to 1 (excluding the boundary points), which will serve the role of frictional force. This coefficient characterizes the rate of momentum attenuation. The closer β is to 1, the longer the momentum is maintained.

All of the above can be written in the form of two mathematical formulas:

$$\Delta_t = \alpha \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_i^j + \beta \Delta_{t-1}$$

$$w_{il}^j = w_{il}^j - \Delta_t$$

where:

- Δ_t = change in the weight at the current step
- Δ_{t-1} = change in the weight at the previous training iteration
- β = pulse damping factor

As a result, we got a decent algorithm for dealing with local minima. Unfortunately, it is not a panacea. To overcome the local minimum, you need to accumulate enough momentum. To do this, the initialization must be at a sufficient distance from the local minimum. When solving practical problems, we do not know where the local and global minima actually are. And the initialization of weights in a random way can throw us anywhere.

Moreover, the application of this method requires additional memory to store the last momentum of each neuron and extra computational effort for the added calculations.

Still, the momentum accumulation method is used in practice and demonstrates better convergence compared to stochastic gradient descent.

When implementing the momentum accumulation method, we will need to add a decay coefficient to the function parameters and another array to store the accumulated momentum for each weight. The logic of building the function remains the same: first, we check the correctness of the input data, and then in a loop, we update the weights. When updating the weights, as in the provided formulas, we first calculate the change in the synaptic coefficient taking into account the accumulated momentum, and then its new value. The obtained values are stored in the corresponding cells of the arrays for weights and momentum values.

```
bool MomentumUpdate(double &m_cWeights[],
                    double &m_cGradients[],
                    double &m_cMomentum[],
                    double learningRate,
                    double beta)
{
    if(m_cWeights.Size() > m_cGradients.Size() ||
       m_cWeights.Size() > m_cMomentum.Size() ||
       m_cWeights.Size() <= 0)
        return false;

    //---
    for(int i = 0; i < m_cWeights.Size(); i++)
    {
        m_cMomentum[i] = learningRate * m_cGradients[i] +
                        beta * m_cMomentum[i];
        m_cWeights[i] -= m_cMomentum[i];
    }
    return true;
}
```

Adaptive gradient method (AdaGrad)

Both methods discussed above have a learning rate hyperparameter. It is important to understand that the entire process of neural network training largely depends on the choice of this parameter. Setting the learning rate too high can cause the error to continually increase instead of decreasing. Using a low learning rate will lead to an extended duration of the training process and increase the likelihood of getting stuck in a local minimum, even when using the momentum accumulation method.

Therefore, when validating the architecture of a neural network, a lot of time is devoted specifically to selecting the correct learning rate coefficient. Furthermore, it is always difficult to select the right learning rate. Moreover, one always wants to train a neural network with minimal time and resource expenditures.

There is a practice of gradually decreasing the learning rate during the training process. The network training process starts with a relatively high rate, which allows for a rapid descent to a certain error level. After reducing the learning rate, a more refined adjustment of the neural network weights is carried out to reduce the overall error. There can be several iterations with a reduced coefficient, but with each reduction of the weight, the effectiveness of that iteration decreases.

Note that we use one learning rate for all neural layers and neurons. However, not all features and neurons contribute equally to the final result of the neural network, so our learning rate should be quite versatile.

I think it goes without saying that universality is the enemy of the best: to create any universal product or select a value (as in our case), we need to compromise to meet all the requirements as best as possible, while these requirements are often contradictory.

One might think, in such a case, it would be advisable to offer individual learning rates for neurons. But to solve this problem manually is virtually impossible. In 2011, the AdaGrad adaptive gradient method was proposed. The proposed method is a variation of the gradient descent discussed above and does not exclude the use of the learning rate coefficient. At the same time, the authors of the method suggest accumulating the sum of the squares of the gradients for all previous iterations and, when updating the weights, dividing the learning coefficient by the square root of the accumulated sum of the squared gradients.

$$G_{ilt}^j = G_{ilt-1}^j + (grad_{ilt}^j)^2$$

$$w_{il}^j = w_{il}^j - \frac{\alpha}{\sqrt{G_{ilt}^j + \varepsilon}} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_{ilt}^j$$

Where:

- G_t, G_{t-1} = sum of the squares of the gradients at the current and previous steps, respectively
- ε = a small positive number to avoid division by zero

In this way, we obtain an individual and constantly decreasing learning rate coefficient for each neuron. However, this requires additional computational resources and extra memory to store the sums of squared gradients.

The implementation function for the AdaGrad method is very similar to the function of updating the weights using the cumulative momentum method. In it, we abandon the use of the decay coefficient but still use the momentum array, in which we will accumulate the sum of squared gradients. The changes also affected the calculation of the new weight value. The complete function code is shown below.

```
bool AdaGradUpdate(double &m_cWeights[],
                  double &m_cGradients[],
                  double &m_cMomentum[],
                  double learningRate)
{
    if(m_cWeights.Size() > m_cGradients.Size() ||
        m_cWeights.Size() > m_cMomentum.Size() ||
        m_cWeights.Size() <= 0)
        return false;
```

```
//---
for(int i = 0; i < m_cWeights.Size(); i++)
{
    double G = m_cMomenum[i] + MathPow(m_cGradients[i], 2);
    m_cWeights[i] -= learningRate / (MathSqrt(G) + 1.0e-10) *
        m_cGradients[i];
    m_cMomentum[i] = G;
}
return true;
}
```

In the above formula, you can notice the main problem of this method. We continuously accumulate the sum of squared gradients. As a consequence, on a sufficiently long training sample, our learning rates will quickly tend to zero. This will lead to the paralysis of the neural network and the impossibility of further training.

A solution was proposed in the RMSProp method.

RMSProp method

The RMSProp weight update method is a logical extension of the AdaGrad method. It retains the idea of automatically adjusting the learning rate based on the frequency of updates and the magnitude of gradients coming to the neuron. However, it addresses the main issue of the previously discussed method – the paralysis of training on large training datasets.

Like AdaGrad, the RMSProp method exploits the sum of squared gradients, but in RMSProp, an exponentially smoothed average of squared gradients is used.

$$REMS(G_{il}^j)_t = \gamma (grad_{il(t-k)}^j)^2 + (1 - \gamma) REMS(G_{il}^j)_{t-1}$$

$$w_{il}^j = w_{il}^j - \frac{\alpha}{\sqrt{REMS(G_{il}^j) + \varepsilon}} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_{il}^j$$

Where:

- $REMS(G)_t$ and $REMS(G)_{t-1}$ = exponential average of the squares of the gradients at the current and previous iteration
- γ = exponential smoothing factor

The use of an exponentially smoothed average of squared gradients prevents the learning rate of neurons from decreasing to zero. At the same time, each weight will receive an individual learning rate, depending on the incoming gradients. As the gradients increase, the learning rate will gradually decrease, and as the gradients decrease, the learning rate coefficient will increase. This will allow in the first case to limit the maximum learning rate, and in the second case, to update the coefficients even with small error gradients.

It should be noted that the use of the squared gradient allows this method to work even when the neuron receives gradients of different directions. If we skip over the minimum because of the high learning rate during the training process and move in the opposite direction on the next iteration, the accumulated square of gradients will gradually decrease the learning rate, thereby allowing us to descend closer to the minimum error.

The implementation of this approach almost completely repeats the implementation of the adaptive gradient method. We will simply replace the calculation of the sum of squared gradients with their exponential average. To do this, we need an additional parameter γ .

```
bool RMSPropUpdate(double &m_cWeights[],
                  double &m_cGradients[],
                  double &m_cMomentum[],
                  double learningRate,
                  double gamma)
{
    if(m_cWeights.Size() > m_cGradients.Size() ||
       m_cWeights.Size() > m_cMomentum.Size() ||
       m_cWeights.Size() <= 0)
        return false;

    //---
    for(int i = 0; i < m_cWeights.Size(); i++)
    {
        double R = (1-gamma) * m_cMomentum[i] +
                   gamma * MathPow(m_cGradients[i], 2);
        m_cWeights[i] -= learningRate / (MathSqrt(R) + 1.0e-10) *
                        m_cGradients[i];
        m_cMomentum[i] = R;
    }
    return true;
}
```

Adadelta method

In the AdaGrad and RMSProp methods, we gave an individual learning rate to each neuron, but still left the learning rate hyperparameter in the numerator of the formula. The creators of the Adadelta method went a little further and proposed to completely abandon this hyperparameter. In the mathematical formula of the Adadelta method, it is replaced by the exponential average of changes in weights over the previous iterations.

$$REMS(\delta w_{il}^j)_t = \gamma_w (\delta w_{il(t-k)}^j)^2 + (1 - \gamma_w) REMS(\delta w_{il}^j)_{t-1}$$

$$REMS(G_{il}^j)_t = \gamma_G (grad_{il(t-k)}^j)^2 + (1 - \gamma_G) REMS(G_{il}^j)_{t-1}$$

$$w_{il}^j = w_{il}^j - \frac{REMS(\delta w_{il}^j)}{REMS(G_{il}^j) + \varepsilon} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_{il}^j$$

Where:

- $REMS(\delta w)_t, REMS(\delta w)_{t-1}$ = exponential average of squared changes in weights for the current and previous iterations

In the practical application of this method, you may encounter cases where both the coefficients for the exponential smoothing of squared weight deltas and gradients are the same, as well as cases where they are individual. The decision is made by the neural network architect.

Below is an example of the implementation of the method using *MQL5* tools. The logic behind constructing the algorithm fully replicates the functions presented above. The changes only affected the calculations that are peculiar to the method: the abandonment of the learning rate and the introduction of an additional averaging coefficient, along with another array of data.

```
bool AdadeltaUpdate(double &m_cWeights[],
                    double &m_cGradients[],
                    double &m_cMomentumW[],
                    double &m_cMomentumG[],
                    double gammaW, double gammaG)
{
    if(m_cWeights.Size() > m_cGradients.Size() ||
        m_cWeights.Size() > m_cMomentumW.Size() ||
        m_cWeights.Size() > m_cMomentumG.Size() ||
        m_cWeights.Size() <= 0)
        return false;

    //---
    for(int i = 0; i < m_cWeights.Size(); i++)
    {
        double W = (1-gammaW) * m_cMomentumW[i] +
            gammaW * MathPow(m_cWeights[i], 2);
        double G = (1-gammaG) * m_cMomentumG[i] +
            gammaG * MathPow(m_cGradients[i], 2);
        m_cWeights.At(i) -= MathSqrt(W) / (MathSqrt(G) + 1.0e-10) *
            m_cGradients[i];
        m_cMomentumW[i] = W;
        m_cMomentumG[i] = G;
    }
    return true;
}
```

Adaptive moment estimation method

In 2014, Diederik P. Kingma and Jimmy Lei Ba proposed Adam's adaptive moment assessment method. According to the authors, the method combines the advantages of the AdaGrad and RMSProp methods and works well in online learning. This method consistently demonstrates good results on various datasets and is currently recommended as the default choice in various packages.

The method is based on the calculation of the exponential average of the gradient m and the exponential average of the squares of the gradient v . Each exponential average has its own hyperparameter β , which determines the averaging period.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) grad_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) grad_t^2$$

The authors suggest using the default β_1 at the level of 0.9, and β_2 at the level of 0.999. In this case, m_0 and v_0 take zero values. With the parameters of the formulas presented above, at the beginning of training, they return values close to zero. As a consequence, we get a low learning rate at the initial stage. To speed up learning, the authors proposed to correct the obtained moments.

$$\hat{m} = \frac{m}{1 - \beta_1}$$

$$\hat{v} = \frac{v}{1 - \beta_2}$$

The updating of parameters is carried out by adjusting them based on the ratio of the corrected gradient moment m to the square root of the corrected gradient moment v . To eliminate division by zero, the ε constant close to zero is added to the denominator. The resulting ratio is corrected by the learning factor α , which in this case is the upper limit of the learning step. By default, the authors suggest using α at the level of 0.001.

$$w_t = w_{t-1} - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \varepsilon}$$

The implementation of the *Adam* method is a little more complicated than the ones presented above, but in general it follows the same logic. Changes are visible only in the body of the weight update loop.

```
bool AdamUpdate(double &m_Weights[],
                double &m_cGradients[],
                double &m_cMomentumM[],
                double &m_cMomentumV[],
                double learningRate,
                double beta1, double beta2)
{
    //---
    if(m_Weights.Size() > m_cGradients.Size() ||
        m_Weights.Size() > m_cMomentumM.Size() ||
        m_Weights.Size() > m_cMomentumV.Size() ||
        m_Weights.Size() <= 0)
        return false;
    //---
    for(int i = 0; i < m_Weights.Size(); i++)
    {
        double w = m_Weights[i];
        double delta = m_cGradients[i];
        double M = beta1 * m_cMomentumM[i] + (1 - beta1) * delta;
        double V = beta2 * m_cMomentumV[i] + (1 - beta2) * MathPow(delta, 2);
        double m = M / (1 - beta1);
        double v = V / (1 - beta2);

        w -= learningRate * m / (MathSqrt(v) + 1.0e-10);
        m_Weights[i] = w;
        m_cMomentumM[i] = M;
        m_cMomentumV[i] = V;
    }
    //---
    return true;
}
```

1.5 Techniques for improving the convergence of neural networks

In the previous chapters of the book, we have learned the basic principles of building and training neural networks. However, we also have identified certain challenges that arise during the training process of neural networks. We have encountered local minimums that can stop training earlier than we achieve the desired results. We also discussed issues of vanishing and exploding gradients and touched upon the problems of co-adaptation of neurons, retraining, and many others which we'll discuss later.

On the path of human progress, we continually strive to refine tools and technologies. This applies to the algorithms of training neural networks as well. Let's discuss methods that, if not completely solve certain issues in neural network training, at least aim to minimize their impact on the final learning outcome.

1.5.1 Regularization

In the pursuit of minimized neural network error, we often complicate our model. What a disappointment it can be when, after prolonged and meticulous work, we achieve an acceptable training set error, only to find the model's error soaring during testing. Such a situation is quite common and is known as 'model overfitting'.

The reasons for this phenomenon are quite mundane and are related to the imperfections, or more precisely, the complexity of our world. Both the raw data and the benchmark results for the training and test datasets were obtained not under controlled laboratory conditions but were taken from real life. Hence, in addition to the analyzed features, they include a number of unaccounted factors, which we attributed to the so-called noise at the design stage for various reasons.

During the training process, we expect that the model will extract significant features from the given volume of raw data and establish relationships between these features and the expected outcome. However, due to the excessive complexity of the model, it can discover relationships between random variables that don't actually exist. It ends up "memorizing" the training dataset. As a result, we get an error close to zero on the training sample. In this process, the test dataset contains its own random noise deviations that don't fit into the concept learned from the training dataset. This confuses our model. As a result, we get a striking difference in the error of the neural network on the training and test samples.

The regularization methods discussed in this section are designed to exclude or minimize the influence of random noise and emphasize the regular features during the model training process. In the practice of training neural networks, you most commonly encounter the use of two methods: L1 and L2 regularizations. Both of them are built on the addition of the sum of weight norms to the loss function.

L1-regularization

L1-regularization is often referred to as lasso regression or Manhattan regression. The essence of this method lies in adding the sum of absolute values of weights to the loss function.

$$L_{L1}(Y, Y', W) = L(Y, Y') + \lambda \sum_{i=1}^n |w_i|,$$

Where:

- $L_{L1}(Y, Y', W)$ = loss function with L1-regularization

1. Basic principles of artificial intelligence construction

- $L(Y, Y')$ = one of the **loss functions** discussed earlier
- λ = regularization coefficient (penalty)
- w_i = i th weighting coefficient

In the process of training the neural network, we will minimize our loss function. In this case, the minimization of the loss function depends directly on the sum of the absolute weight values. Thus, in our model training, we introduce an additional constraint of selecting weights as close to zero as possible.

The partial derivative of such a loss function will take the form:

$$\frac{dL_{L1}(Y, Y', W)}{dw_i} = \frac{dL(Y, Y')}{dw_i} + \lambda \text{sign}(w_i)$$

Here, we don't explicitly calculate the derivative of the loss function itself to isolate the influence of regularization directly.

The function $\text{sign}(w_i)$ returns the sign of the weight when it is non-zero and 0 when the weight is zero. Since λ is a constant, and we consistently subtract the value of the derivative multiplied by the learning rate and the error gradient when updating the weights, then, when training the neural network, the model will set features that do not have a direct impact on the outcome to zero. This will completely eliminate the influence of random noise on the result.

L1 regularization introduces a penalty for large weights, thus enabling the selection of important features and mitigating the influence of random noise on the final outcome.

L2-regularization

L2, or ridge, regularization, like L1 regularization, introduces a large weighting penalty into the loss function. However, in this case, the L2 norm is used, which is the sum of the squares of the weights. As a result, the loss function will have the following form.

$$L_{L2}(Y, Y', W) = L(Y, Y') + \lambda \sum_{i=1}^n w_i^2$$

Similar to L1-regularization, we add a constraint to the model training process to use weighting coefficients as close to zero as possible. Let's look at the derivative of our loss function.

$$\frac{dL_{L2}(Y, Y', W)}{dw_i} = \frac{dL(Y, Y')}{dw_i} + 2\lambda w_i$$

In the L2-regularization derivative formula, the penalty λ is multiplied by the weight. This implies that during training, the penalty is not constant but dynamic. It decreases proportionally as the weight decreases. In this process, each weight receives an individual penalty based on its magnitude. Hence, unlike L1 regularization, during the training of the neural network, the weights of the features that do not have a direct impact on the outcome will decrease. However, they will never reach zero, unless calculation precision limits allow for it.

L2 regularization introduces a penalty for large weights, thus enhancing the influence of important features and reducing, though not eliminating, the impact of random noise on the final outcome.

Elastic Net

As mentioned above, L1-regularization simplifies the model by zeroing out the weights for parameters that do not directly affect the expected outcome of the model. Applying such an approach is justified when we are reasonably confident about the presence of a small number of redundant features, the exclusion of which can only improve the model performance.

If, however, we understand that the overall result is a combination of small contributions from all the features used and the exclusion of any feature would worsen the model performance, then in such a scenario, using L2 regularization is justified.

But which of the methods to use when our model receives an obviously excessive number of features? Moreover, we do not understand the individual impact of features on the outcome. Perhaps excluding certain features could simplify our model and improve its performance. At the same time, excluding other features would have a negative impact on the model's performance.

At such times, Elastic Net regularization is applied. This model adds penalties based on both the L1 and L2 norms of weights to the loss function, combining the advantages of L1 and L2 regularization.

$$L_{Elastic}(Y, Y', W) = L(Y, Y') + \lambda_1 \sum_{i=1}^n |w_i| + \lambda_2 \sum_{i=1}^n w_i^2$$

Please note that in the Elastic Net formula, L1 and L2 regularization each have their own regularization coefficients. Thus, by changing the regularization coefficients λ_1 and λ_2 , the regularization model can be controlled. By setting them both to zero, we achieve model optimization without regularization. When $\lambda_1 > 0$ and $\lambda_2 = 0$ we have pure L1 regularization, and when $\lambda_1 = 0$ and $\lambda_2 > 0$ we get L2-regularization.

1.5.2 Dropout

We continue studying methods for improving the convergence of neural networks. Let's consider the dropout technology.

When training a neural network, a large number of features are fed into each neuron, the influence of each of which is difficult to assess. As a result, errors of some neurons are smoothed out by the correct values of others, and errors accumulate at the output of the neural network. Training stops at a certain local minimum with a sufficiently large error that does not meet our requirements. This effect was called co-adaptation of features, in which the influence of each feature seemingly adjusts to the surrounding environment. It would be better for us to get the opposite effect when the environment is decomposed into individual features and evaluate separately the impact of each of them.

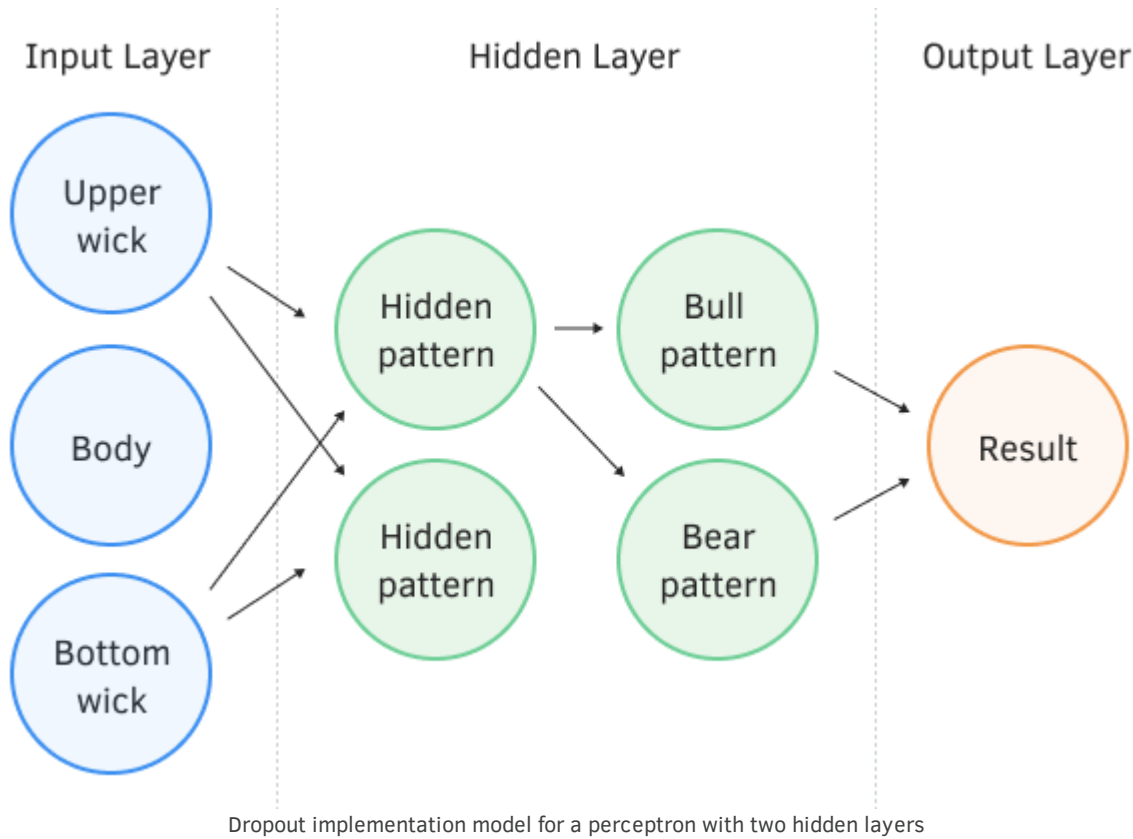
To combat complex co-adaptation of features, in July 2012, a group of scientists from the University of Toronto, in a paper "[Improving neural networks by preventing co-adaptation of feature detectors](#)", proposed randomly excluding some neurons during the training process. Reducing the number of features during training increases the significance of each one, and the constant change in the quantitative and qualitative composition of features reduces the risk of their co-adaptation. Such a method is called Dropout.

Applying this method can be compared to decision trees because by excluding some neurons at random, we get a new neural network with its own weights at each training iteration. According to the rules of combinatorics, the variability of such networks is quite high.

1. Basic principles of artificial intelligence construction

At the same time, all the features and neurons are evaluated during the operation of the neural network. Thereby, we obtain the most accurate and independent assessment of the current state of the studied environment.

The authors of the solution in their paper point out that the method can also be used to improve the quality of pre-trained models.



Describing the proposed solution from a mathematical point of view, we can say that each individual neuron is excluded from the process with a certain given probability P . Thus, the neuron will participate in the neural network training process with a probability of $q=1-P$.

To determine the list of excluded neurons, the method uses a pseudorandom number generator with a normal distribution. This approach allows for the most uniform possible exclusion of neurons. In practice, we will generate a vector of binary features of size equal to the input sequence. In the vector, we will set 1 for the features that are used and 0 for the excluded elements.

However, the exclusion of the analyzed features undoubtedly leads to a decrease in the sum at the input of the neuron activation function. To compensate for this effect, we multiply the value of each feature by a factor of $1/q$. It's easy to notice that this coefficient will increase the values, as the probability q is always in the range from 0 to 1.

$$D_i = \frac{1}{q} m_i x_i,$$

Where:

- D_i = elements of the Dropout results vector
- q = probability of using a neuron during the learning process

- m_i = the element of the masking vector
- x_i = the elements of the input sequence vector

During the backward pass in the training process, the error gradient is multiplied by the derivative of the aforementioned function. As can be easily seen, in the case of Dropout, the backward pass will be similar to the forward pass which uses the masking vector from the forward pass.

$$\frac{dD_i}{dx_i} = \frac{1}{q} m_i$$

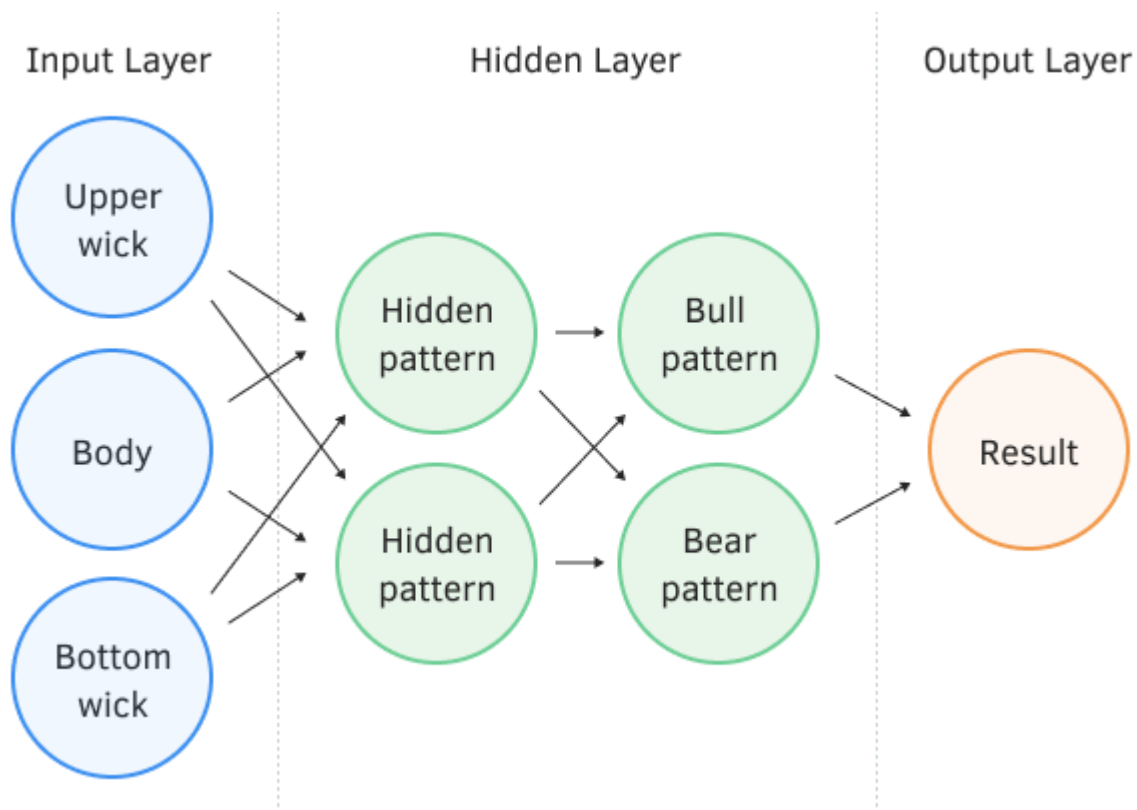
During the operation of the neural network, the masking vector is filled with units, allowing values to be transmitted seamlessly in both directions.

In practice, the coefficient $1/q$ is constant throughout training, so we can easily count this coefficient once and write it instead of units in the masking tensor. In this way, we eliminate the operations of recalculating the coefficient and multiplying it by 1 of the mask in each training iteration.

1.5.3 Batch normalization

The practical application of neural networks implies the use of various approaches to data normalization. All of these approaches aim to maintain the training data and the output of hidden layers of the neural network within a specified range and with certain statistical characteristics of the dataset, such as variance and median. Why is this so important? We remember that network neurons apply linear transformations which shift the sample towards the anti-gradient in the learning process.

Consider a fully connected perceptron with two hidden layers. In the forward pass, each layer generates a set of data that serves as a training sample for the next layer. The output layer's results are compared with the reference data, and during the backward pass, the error gradient is propagated from the output layer through hidden layers to the input data.



Perceptron with two hidden layers

Having obtained the error gradient for each neuron, we update the weights, tuning our neural network to the training samples from the last forward pass. Here arises a conflict: we are adjusting the second hidden layer (labeled as 'Bull pattern' and 'Bear pattern' in the above diagram) to the output data of the first hidden layer (labeled as 'Hidden pattern' in the diagram). However, by changing the parameters of the first hidden layer, we have already altered the data array. In other words, we are adjusting the second hidden layer to a data sample that no longer exists.

The situation is similar to the output layer, which adjusts to the already modified output of the second hidden layer. If we also consider the distortion between the first and second hidden layers, the scales of error amplification increase. The deeper the neural network, the stronger the manifestation of this effect. This phenomenon is referred to as internal covariate shift.

In classical neural networks, this problem was partially solved by reducing the learning rate. Small changes in the weights do not significantly alter the distribution of the output of the neural layer. However, this approach does not solve the problem of scaling with an increase in the number of layers of the neural network and reduces the learning rate. Another issue with a low learning rate is the potential stop at local minima (we already discussed this issue in the section about [neural network optimization methods](#)).

Here, it's also worth mentioning the necessity of normalizing the input data. Quite often, when solving various tasks, diverse input data is fed into the input layer of a neural network, which might belong to samples with different distributions. Some inputs can have values that significantly exceed the magnitudes of the others. Such values will have a greater impact on the final result of the neural network. At the same time, the actual impact of the described factor might be significantly lower, while the absolute values of the sample are determined by the nature of the metric.

The below chart shows an example illustrating the reflection of a single price movement using two oscillators (MACD and RSI). When considering the indicator charts, you can notice the correlation of

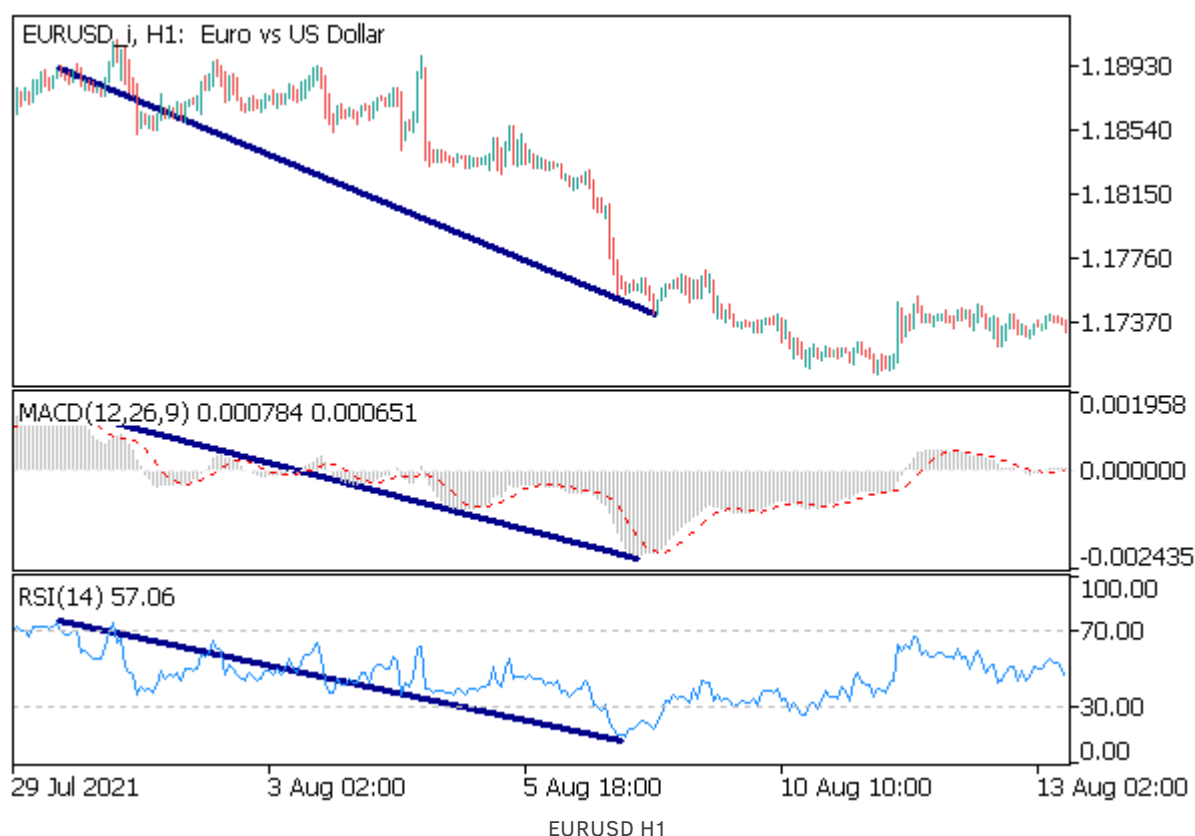
1. Basic principles of artificial intelligence construction

the curves. At the same time, the numerical values of the indicators differ by hundreds of thousands of times. This is because RSI values are normalized on a scale from 0 to 100, while MACD values depend on the accuracy of price representation on the graph, as MACD shows the distance between two moving averages.

When building a trading strategy, we can utilize either of these indicators individually or consider the values of both indicators and execute trading operations only when the signals from the indicators align. In practice, this approach enables the exclusion of some false signals, which eventually can reduce the drawdown of the trading strategy. However, before we input such diverse signals into the neural network, it's advisable to normalize them to a comparable form. This is what the normalization of the initial data will help us to achieve.

Of course, we can perform the normalization of the input data while preparing the training and testing datasets outside the neural network. But this approach increases the preparatory work. Moreover, during practical usage of such a neural network, we will need to consistently prepare the input data using a similar algorithm. It is much more convenient to assign this work to the neural network itself.

In February 2015, Sergey Ioffe and Christian Szegedy proposed the [Batch Normalization](#) method in their work "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". The method was proposed to address the issue of internal covariate shift. This algorithm can also be applied for normalizing input data.



The essence of the method was to normalize each individual neuron over a certain time interval by shifting the median of the sample to zero and scaling the sample's variance to one.

The normalization algorithm is as follows. First, the average value is calculated from the data sample.

1. Basic principles of artificial intelligence construction

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i,$$

Where :

- μ_B = the arithmetic mean of the feature over the sample
- m = the sample size (batch)

Then we calculate the variance of the original sample.

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

We normalize the sample data by reducing the sample to zero mean and unit variance.

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

Note that a small positive constant ε is added to the denominator of the sample variance to prevent division by zero.

As it turned out, such normalization can distort the influence of the input data. This is why the authors of the method added one more step: scaling and offset. They introduced the variables γ and β , which are trained together with the neural network using the gradient descent method.

$$y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma\beta}(x_i)$$

Applying this method allows obtaining a dataset with a consistent distribution at each training step, which in practice makes the training of the neural network more stable and enables an increase in the learning rate. Overall, this enhances the training quality while reducing the time required for neural network training.

However, at the same time, the cost of storing additional coefficients increases. Furthermore, calculating the moving average and variance requires storing in memory the historical data of each neuron for the entire batch size. An alternative here could be the use of Exponential Moving Average (EMA): calculating the EMA only requires the previous value of the function and the current element of the sequence.

Experiments conducted by the authors of the method demonstrate that the application of Batch Normalization also serves as a form of regularization. This allows for the elimination of other regularization methods, including the previously discussed Dropout. Moreover, there are more recent works showing that the combined use of Dropout and Batch Normalization has a negative effect on the training results of a neural network.

In modern architectures of neural networks, the proposed normalization algorithm can be found in various shapes. The authors suggest using Batch Normalization directly before the non-linearity (activation function). One of the variations of this algorithm is [Layer Normalization](#), introduced by Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton in July 2016 in their work "Layer Normalization."

1.6 Artificial intelligence in trading

Previous sections introduced the basic principles and algorithms for building neural networks. However, our primary interest lies in the practical application of the presented technologies, and, I am certainly not the first to consider it.

Computer technologies have long been integrated and successfully applied in trading. It's difficult to imagine trading without the use of computer technologies nowadays. Primarily, thanks to the internet and computers, traders no longer need to be physically present on the trading floor. The trading terminal software can be easily installed on any computer and even on mobile devices (smartphones, tablets). This enables traders to analyze the market and execute trading operations from virtually any location on our planet.

The aforementioned trading terminals not only facilitate trade execution but also provide all the necessary tools for detailed real-time market analysis. They include features for constructing graphical objects on price charts and a variety of indicators that can dynamically update values and display them on the chart according to the current market situation.

Another direction of applying computer technologies in trading is algorithmic trading. Algorithmic trading involves creating computer programs (robots) that execute trading operations without human intervention, following a predefined trading strategy. This method has its own advantages and disadvantages compared to manual human trading.

A created program can work tirelessly 24 hours a day, 7 days a week, which is impossible for a human. Accordingly, the program will not miss any signal to enter or exit a position. The robot will strictly follow the specified algorithm. In contrast, a human, while evaluating the market situation, may consider personal past experiences and subjective feelings, which can vary.

First and foremost, deviating from the trading strategy disrupts the balance between profitable and losing trades, and over a long time frame, it's likely to have a negative impact on the trading account balance.

On the other hand, it can be quite challenging to precisely describe all aspects of a trading strategy in mathematical terms. In this case, the trader's personal experience and their personal feeling of the market will play a significant role. The program does not have these features, and the tolerances built in by the programmer may not be ideal.

Among the benefits of algorithmic trading, we can also include the absence of psychological factors in programs. Meanwhile, the psychological barrier often causes traders, especially newcomers, to deviate from their trading strategies.

On the other hand, time series are variable. Therefore, any trading strategy has a limited lifespan. As a consequence, over time, there's a need to adapt trading systems to current market conditions, and a classical robot can't evaluate its performance or make changes to its trading algorithm or parameters without human assistance.

So what do we expect from the application of artificial intelligence and neural networks in particular?

When building a mathematical model using a neural network, we do not prescribe the entire trading algorithm, as in classical algorithmic trading. We simply provide a training dataset and let the neural network itself discover patterns and correlations between the input data and the final outcome. In doing so, we expect the neural network to capture not only the obvious patterns but also the subtle fluctuations that can enhance the effectiveness of the trading system.

1. Basic principles of artificial intelligence construction

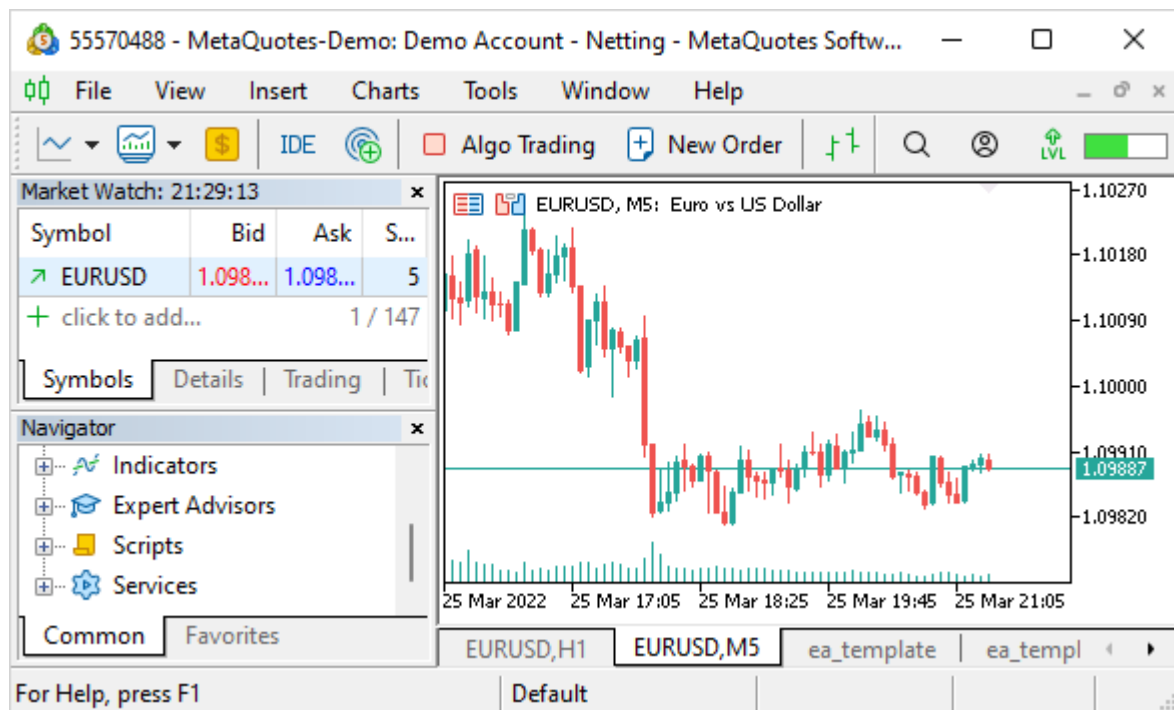
When creating a training dataset for the neural network, we should not limit ourselves to the input data of a single strategy. There may be much more input data than a human is capable of processing. However, the final mathematical model might produce signals that don't fit neatly into any of the expected strategies. As a result, we expect to obtain performance higher than that of robots built according to the classical algorithmic trading scheme.

And, of course, the learning ability of neural networks enables the creation of methods for assessing the performance of a strategy and initiating the training process of the neural network in a timely manner for adaptation to current market conditions.

Thus, we anticipate a reduction in the negative aspects of algorithmic trading while retaining its positive aspects.

2. MetaTrader 5 features for algorithmic trading

For the practical part of this book, we will use the [MetaTrader 5](#) trading terminal. It is a modern, constantly evolving platform developed by [MetaQuotes Ltd.](#)



MetaTrader 5

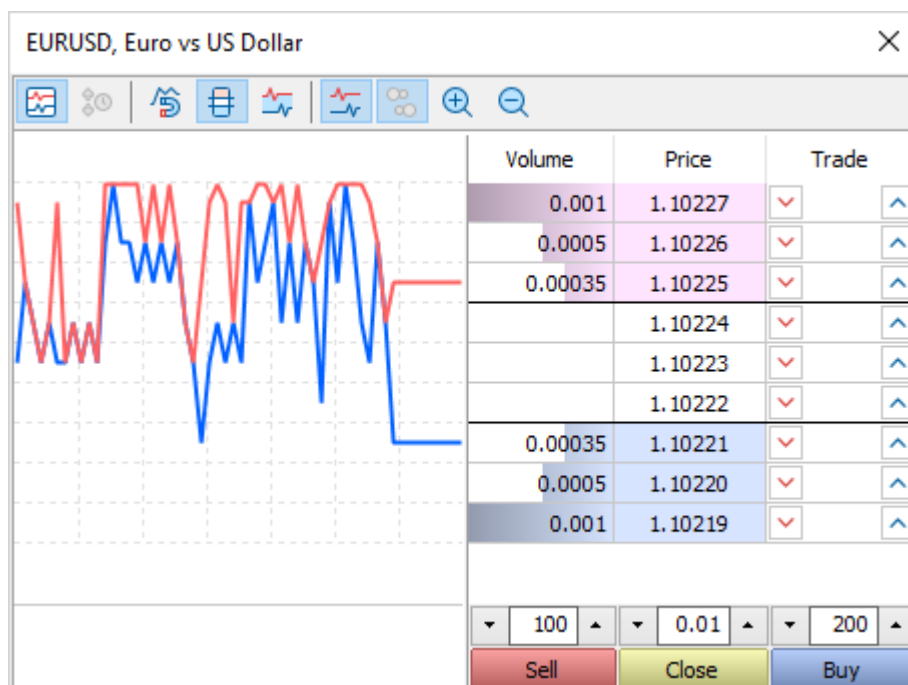
MetaTrader 5 is a multi-asset trading platform. It is widely used by traders all around the world to execute trading operations in the Forex market, stock exchanges, and futures markets. MetaTrader 5 is a comprehensive program that enables trading operations and provides extensive capabilities for conducting detailed technical and fundamental analysis of the market situation.

The platform features an extended Depth of Market with a tick chart and the Time and Sales window. This tool enables the analysis of the current state and the one-click execution of trading operations. Moreover, there is the option to set stop loss and take profit levels for placed orders, which proves quite beneficial for implementing scalping trading strategies.

The provided Depth of Market allows placing market and pending orders, as well as modifying them. For more information on the capabilities of this tool, please visit https://www.metatrader5.com/en/terminal/help/trading/depth_of_market.

2. MetaTrader 5 features for algorithmic trading

Additionally, for conducting technical analysis, there are extensive capabilities for adding various graphical objects directly onto the price chart. The range of applied objects is quite impressive. Among these are simple lines (vertical, horizontal, and various diagonal trend lines), as well as various channels, Fibonacci levels, and more complex shapes. There's the possibility to assign different colors and visual styles to objects, as well as adding custom names and descriptions to objects.



Depth of Market



Objects of technical analysis on the chart

The platform features a comprehensive list of oscillators, volume indicators, and trend indicators, capable of meeting the requirements of any user. At the same time, if the available range of indicators is not enough for you, it is possible to create a custom indicator based on your own formula. You can create it yourself or order it from experienced programmers via the [Freelance](https://www.mql5.com) service at [mql5.com](https://www.mql5.com).

2. MetaTrader 5 features for algorithmic trading

Furthermore, in the [Market](#) section of the platform, you can purchase or download indicators from various third-party developers, and the list of these indicators is constantly updated and expanded. It's unlikely that any other platform could offer such a broad spectrum of technical analysis tools.



Indicators on the chart

The capability to analyze each instrument across 21 timeframes, ranging from 1 minute to 1 month, provides a comprehensive and detailed analysis.

The indicators and graphical objects applied to the price chart can be saved as templates, which can then be easily reloaded onto the chart with just a couple of mouse clicks.

For enthusiasts of fundamental analysis, the platform provides a news feed and a calendar of financial events, allowing the display of markers for past and upcoming events directly on the instrument's chart. This enables tracking changes and analyzing trading situations rapidly in the future.

The one-click trading feature from the chart of the trading instrument helps traders execute operations swiftly and at the best price.

In addition, the platform gives almost unlimited possibilities for algorithmic trading, that is, for automated trading by using robots. *MetaQuotes* specialists have developed the *MQL5 IDE* (Integrated Development Environment) specifically for the platform. This environment allows users to create their own indicators and trading strategies, as well as test and optimize them using the built-in strategy tester with historical data gathered from real ticks.

The *MetaTrader 5* platform is widely adopted and offered for use by the majority of brokers around the world, enabling traders to choose a trading provider according to their preferences.



In the context of the book's theme, you will undoubtedly be interested in exploring the potential for implementing neural network technologies and algorithms through the tools provided by the *MetaTrader 5* platform. Let's take a closer look at the proposed tool.

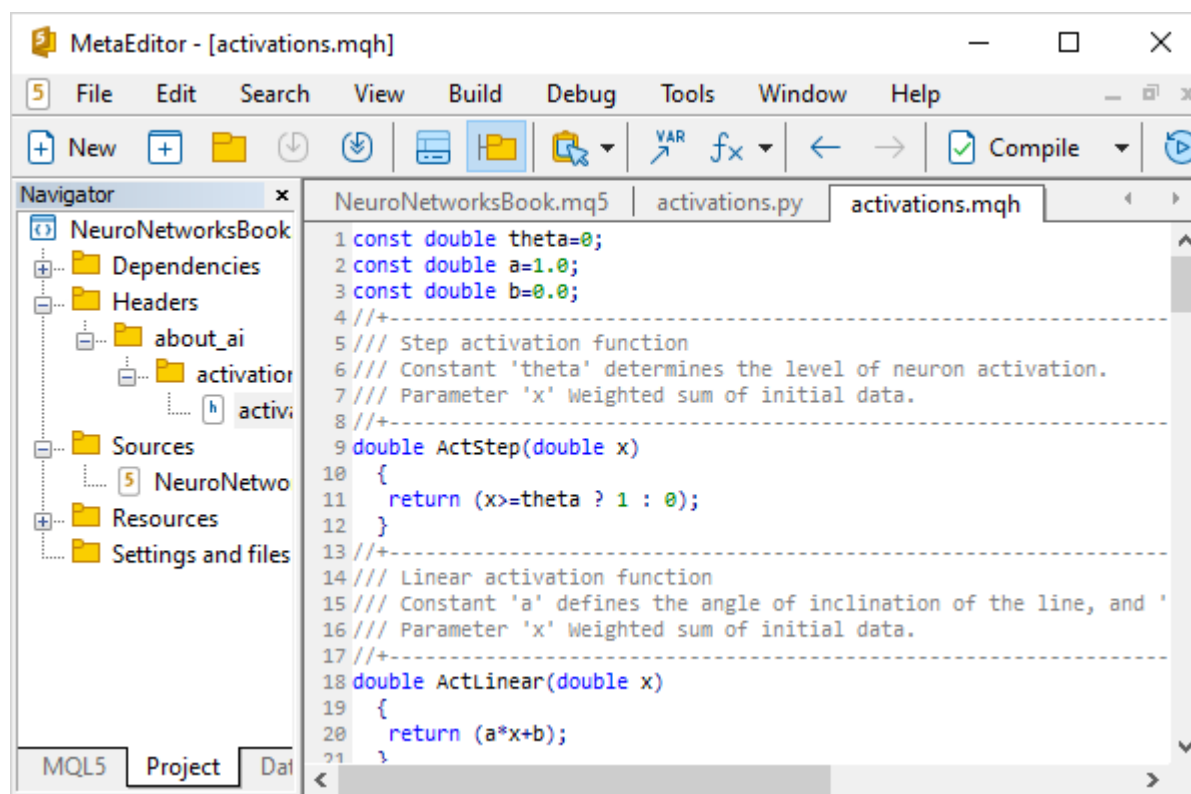
2.1 Program types and their construction features

The [MetaTrader 5 platform](#) package includes a modern development environment [MetaEditor](#) which enables the creation of various programs for algorithmic trading. Programs can be written in the specially designed programming language called MetaQuotes Language 5 (MQL5). The language syntax is closely aligned with C++, enabling programming in an object-oriented style. This facilitates the transition to using MQL5 for a large community of programmers.

The interaction of the MetaTrader 5 platform with programs is organized in such a way that the price movements of instruments and changes in the trading account are tracked by the platform. When predetermined changes occur, the platform generates [events](#) in the instrument chart open in the platform. When an event occurs, the user programs attached to the chart are checked. These can be software Expert Advisors, indicators, and scripts. [Event handlers](#) are defined in the platform for each event and program type.

An event handler is a special function defined by the *MQL5* programming language. Such a function has a strictly specified name, return value type, and a list and type of parameters. Based on the return value type and the parameter types, the event handler of the client terminal identifies functions for processing the occurred event. If a function has parameters that do not match the predetermined ones or if a different return value type is specified, then such a function will not be used to process the event.

2. MetaTrader 5 features for algorithmic trading



Built-in development environment MetaEditor

Each type of program can only handle certain events. Thus, if the event handler does not correspond to the program type, such a function will not be called by the terminal.

The MQL5 language includes a series of [trading functions](#) and predefined event handlers, which are used for Expert Advisors to allow them to execute the trading strategies embedded in them. It also offers an opportunity to write your own technical analysis indicators, scripts, services, and libraries of included functions.

Each program type is designed to perform its specific tasks and has special features of construction.

Expert Advisors (EAs)

Probably, at the forefront of algorithmic trading are Expert Advisors (trading robots) which are programs capable of independently analyzing the market and conducting trading operations based on a programmed strategy and its trading rules.

Technically, in MetaTrader 5, an Expert Advisor is tied to a specific chart on which it runs. In doing so, it only handles predefined [events](#) from this specific chart. The occurrence of each event triggers the corresponding functionality of the trading strategy. Among such events can be program launch and deinitialization, timer triggering, arrival of a new tick, scheduled events, and user events.

At the same time, the trading strategy of the Expert Advisor can include the analysis of other timeframes of the current instrument, as well as the analysis of any instrument in the terminal on any timeframe. This allows you to build multi-currency and multi-timeframe strategies.

In addition, Expert Advisors have the technical ability to receive data from any technical indicator installed in the terminal. This greatly expands the possibilities for building different trading strategies.

Each predefined event calls the corresponding EA function, in which the program code for event processing is written.

Immediately after launching the Expert Advisor, the terminal generates an **Init** event, which triggers the **OnInit** function. Global variables and objects are initialized in the body of this function. If necessary, a timer is started. The function has no input parameters but returns an integer value of the return code as a result of its execution. A non-zero return code indicates a failed initialization. In this case, the terminal generates a program termination event called **Deinit**.

```
//+-----+
//| Expert initialization function |
//+-----+
int OnInit()
{
    //---

    //--- create timer
    EventSetTimer(60);
    //---
    return(INIT_SUCCEEDED);
}
```

When the program is completed, the *MetaTrader 5* terminal generates a **Deinit** event that triggers the execution of the **OnDeinit** function. The function has one input integer parameter which receives the code of the reason for the program termination. Inside the function body, if necessary, global variables, classes, and graphical objects are removed, data is saved in file resources, the timer initiated during program initialization is closed, and other operations required for the proper termination of the program and the cleanup of its traces in the terminal are performed.

```
//+-----+
//| Expert deinitialization function |
//+-----+
void OnDeinit(const int reason)
{
    //---

    //--- destroy timer
    EventKillTimer();
}
```

When a new tick arrives for the symbol chart on which the Expert Advisor is running, the **NewTick** event is generated. This triggers the **OnTick** function. This event is generated only for Expert Advisors, so the **OnTick** function will not be launched in other programs. Of course, the specified function can always be called forcibly from any place in the program, but it will no longer be the **NewTick** event processing.

The **OnTick** function has no input parameters and does not return any code. The main purpose of the function is to execute the price fluctuations handler in the advisor, which evaluates changes in the market situation and checks the rules of the embedded strategy for the need to perform any trading operations. Sometimes, according to the trading strategy rules, the Expert Advisor should perform operations not at every price movement, but, for example, at the opening of a new candlestick. In such cases, checking for the occurrence of the expected event is added to the **OnTick** function.

2. MetaTrader 5 features for algorithmic trading

```
//+-----+
//| Expert tick function                               |
//+-----+
void OnTick()
{
//---

}
```

If the algorithm of an Expert Advisor does not require the processing of each price movement but is based on the execution of cyclic operations with a certain time interval, even if there are no price movements observed during this time, the use of a timer can be very beneficial.

For this, when initializing the program in the *OnInit* function, it is necessary to initialize the timer using the *EventSetTimer* function. The function parameters specify the timer delay period in seconds. After that, the terminal will generate a **Timer** event for the chart, and the **OnTimer** function of the Expert Advisor will be launched for execution.

```
//+-----+
//| Timer function                                     |
//+-----+
void OnTimer()
{
//---

}
```

When using a timer in the program's code, it is necessary to unload it from the terminal's memory upon program termination within the *OnDeinit* function. The *EventKillTimer* function is used for this purpose. This function has no parameters. It should be noted that the platform provides for the use of only one timer on the chart.

Within one EA, you can use both the *OnTick* and *OnTimer* functions, if necessary.

Among Expert Advisors, there is a subclass of semi-automatic Expert Advisors and trading panels. Such programs are not capable of independent trading without human intervention. Programs of this type perform trading operations at the trader's command. The programs themselves are designed to facilitate the trader's work and to take over some routine operations. This can be money management, setting stop loss and take profit levels, position maintenance, and much more.

The interaction between the program and the user is implemented through **ChartEvent** group events in Expert Advisors. These events trigger the execution of the *OnChartEvent* function, which accepts four parameters from the terminal:

- *id*: event identifier,
- *lparam*: event parameter of type *long*,
- *dparam*: event parameter of type *double*,
- *sparam*: event parameter of type *string*.

```
//+-----+
//| ChartEvent function |
//+-----+
void OnChartEvent(const int id,
                  const long &lparam,
                  const double &dparam,
                  const string &sparam)
{
    //---
}
```

The event can be generated for Expert Advisors and technical indicators. In this case, for each type of event, the input parameters of the function have certain values required to process the event.

Technical indicators

Another program type is a custom indicator. Indicators are programs that perform analytical functions to assist traders in conducting technical analysis of the market situation. During their operation, indicators process each price movement on the chart of their trading instrument. They can display various graphical objects, thus generating signals for subsequent analysis by the trader.

Like Expert Advisors, custom indicators can use data from other indicators, instruments, and timeframes in their calculations. But at the same time, indicators cannot perform trading operations. Thus, the indicator application scope is limited to the framework of technical analysis.

Similar to Expert Advisors, technical indicators have *Init*, *Timer*, and *ChartEvent* event handlers. The construction of functions for processing these events is similar to the corresponding functions of electronic Expert Advisors, but instead of the *NewTick* event, the *Calculate* event is generated for indicators. This event is handled by the *OnCalculate* function. There are two types of *OnCalculate* function depending on the scope of the indicator:

- shorthand

```
//+-----+
//| Custom indicator iteration function |
//+-----+
int OnCalculate (const int rates_total,
                 const int prev_calculated,
                 const int begin,
                 const double& price[])
{
    //---

    //--- return value of prev_calculated for the next call
    return(rates_total);
}
```

- full

2. MetaTrader 5 features for algorithmic trading

```
//+-----+
//| Custom indicator iteration function |
//+-----+
int OnCalculate(const int rates_total,
               const int prev_calculated,
               const datetime &time[],
               const double &open[],
               const double &high[],
               const double &low[],
               const double &close[],
               const long &tick_volume[],
               const long &volume[],
               const int &spread[])
{
    //---

    //--- return value of prev_calculated for the next call
    return(rates_total);
}
```

Within a single indicator, you can only use one of the versions of the function.

Both versions of the *OnCalculate* function have parameters:

- *rates_total*: number of items in the timeseries,
- *prev_calculated*: number of recalculated elements of the time series at the previous run of the function.

The use of the *prev_calculated* parameter allows you to implement algorithms in which the indicator does not recalculate previously calculated historical values. This reduces the number of iterations in processing each new price fluctuation.

The work of indicators in the *MetaTrader 5* terminal is organized as follows. The *prev_calculated* parameter receives the value that the function returned at the previous run. Therefore, in the general case, at the end of a successful function completion, it's sufficient to return the value of the *rates_total* parameter. If errors occur while the function is running, you can return the current value of *prev_calculated*. In this case, the function will start recalculating from the current location the next time it is run. If you return 0, upon the next launch, the indicator will be recalculated for the entire history, as it was during the initial launch.

When defined briefly, the function has only one input array of time series (*price*) and a parameter for shifting significant values relative to the beginning of the time series (*begin*). In this version, the calculation of indicator values is based on the data of one time series. Which time series will be used is set by the trader when launching the technical indicator. This can be either any of the price time series or the buffer values of another indicator.

When using the full version of the *OnCalculate* function, the function gets all price time series in its parameters. In such a case, the user does not have the option to choose a time series when launching the indicator. If it's necessary to use a buffer of data from another indicator, it needs to be explicitly written in the program code of the indicator.

In *MetaTrader 5*, there is a limitation in the ability to run only one Expert Advisor for each chart. If you need to run two or more Expert Advisors in one terminal, you should open an individual chart for each Expert Advisor. There is no such limitation for indicators as *MetaTrader 5* allows you to use built-in and

custom indicators in different versions in parallel on one chart of a trading instrument. The indicator can display data both on the price chart of the instrument itself and in sub-windows.

Scripts

Following launching, Expert Advisors and custom indicators remain in the terminal memory until they are forcibly closed by the trader. As certain events occur, the terminal launches the relevant functionality of Expert Advisors and indicators. Scripts are provided to perform any one-time operations. This is a separate type of program that does not handle any events other than its startup event.

Immediately after being launched, they perform the designated functionality and are unloaded from the terminal's memory. Along with Expert Advisors, scripts are able to perform trading operations, but it is impossible to run more than one script on a symbol chart at the same time.

There is only one **OnStart** event handler in the body of the script, which is launched immediately after the program starts. The *OnStart* function does not receive any parameters and does not return any codes.

```
//+-----+
//| Script program start function |
//+-----+
void OnStart()
{
//---

}
```

A separate program type is Services. Unlike the aforementioned types of programs, a service does not require binding to a specific price chart of a trading instrument. Just like scripts, services do not process any events other than their own launch. However, they are capable of generating custom events themselves and sending them to charts for further processing in Expert Advisors.

The *MetaEditor* development environment provides the possibility to create libraries and include files. These files are designed to store and distribute frequently used program blocks. Libraries are compiled files and provide individual functions for export to other running programs. The code of the executed functions itself is hidden. Plug-in files, unlike libraries, are open-source files. In terms of performance, it's preferable to use include files, but they do not ensure code secrecy during distribution.

In addition to event handlers, all programs can contain other functions and classes that will need to be called from the event handler functions. They can also have external input parameters set by the user when the program is launched.

Another technical aspect should also be taken into consideration. *MetaTrader 5* is positioned as a platform with multi-threaded computing. In this case, three threads are allocated for each trading instrument's chart: one for the Expert Advisor, one for the script, and one for indicators. All indicators loaded onto one trading instrument's chart operate within the same thread as the chart itself. Therefore, it's not recommended to perform complex calculations within indicators.

Hence, we can use EAs or scripts to build our neural network.

2.2 Statistical analysis and fuzzy logic tools

The *MetaEditor* development environment provides a separate type of include files with the *mqh* extension, which enable the exchange of frequently used code blocks. *MetaQuotes* delivers *MetaTrader 5* with a vast Standard Library, which includes classes and methods for implementing a wide variety of tasks. This includes classes available for analyzing data using mathematical statistics and fuzzy logic.

The library of mathematical statistics offers functionality for working with basic statistical distributions. It has more than twenty distributions and five features are presented for each:

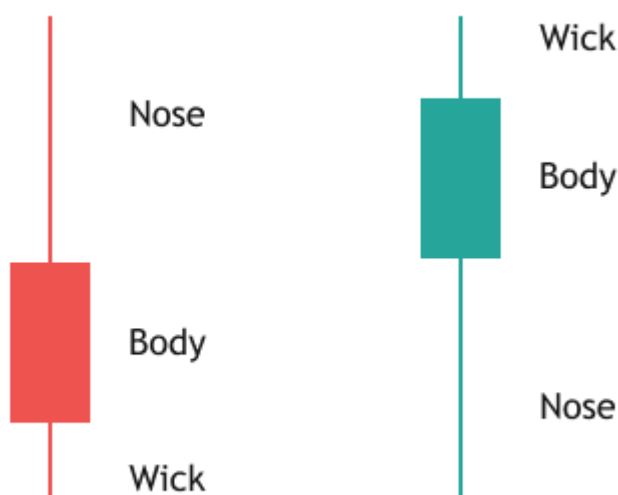
1. Calculation of distribution density.
2. Calculation of probabilities.
3. Calculation of distribution quantiles.
4. Generation of random numbers with a given distribution.
5. Calculation of theoretical distribution moments.

The library also allows you to calculate the statistical characteristics of a given data set. With the help of this library, one can easily perform statistical analysis of a sample from the historical data of the analyzed instrument. You can also compare the statistical indicators of several instruments and observe the dynamics of the statistical indicators of one instrument based on historical data from different time intervals.

Additionally, one can conduct a multifaceted and comprehensive analysis, and use its results as the foundation for building one's trading system.

Before discussing the capabilities of the fuzzy logic library, let us consider the concept itself. The concept of fuzzy logic was proposed by American scientist Lotfi Zadeh in 1965. This innovation allows the addition of a certain share of subjectivity inherent in real life to calculations. After all, you'll agree that when describing certain objects and processes, we often use vague and approximate reasoning.

We often hear the phrase "Words are used out of context." This suggests that the interpretation of words and their use in speech is highly dependent on the context. It is also difficult to describe a single candlestick on a chart. We can tell what color it is and mention the presence of shadows. But you'll agree, with such a description, we can divide all candlesticks into two classes based on their color. In fuzzy logic theory, these would be two sets.



Candlesticks

For further description, we will need to introduce additional concepts and measurements. We can compare a candlestick with neighboring candlesticks, calculate some kind of average, or take some kind of benchmark and compare to it. In doing so, we again get an inaccurate description. The deviation from our benchmark or average can vary, just as the influence of a factor can change significantly depending on the size of this deviation. The application of fuzzy logic allows us to solve this problem by introducing "fuzzy" set boundaries.

Three stages are distinguished in the history of fuzzy systems development:

1. 1960-70s: development of theoretical aspects of fuzzy logic and fuzzy sets;
2. 1970s-80s: the first practical results in the field of fuzzy systems control;
3. From the 1980s to the present day: the creation of various software packages for constructing fuzzy systems significantly broadens the application scope of fuzzy logic.

Let us review the basic concepts of fuzzy set theory.

First, it is **a fuzzy set**, that is, a set of values unified by some rules.

The mathematical description of these rules is combined into a **membership function** which is a characteristic of a fuzzy set and is denoted by $MF_C(x)$ – the degree of the x value membership in the fuzzy set C .

The set of values of initial data satisfying the membership function is called the **term set**.

A collection of fuzzy sets and their rules are combined into a **fuzzy model (system)**.

The results of the fuzzy model operation are determined from a combination of fuzzy sets using a system of fuzzy logical inferences. The MQL5 fuzzy logic library implements the Mamdani and Sugeno fuzzy logic inference systems.

To understand the differences between the usual mathematical description and the fuzzy logic membership function, let us consider an example of the description of a Doji candlestick (a candlestick without a body). Such candlesticks often act as harbingers of a trend change, as they appear in the area of supply and demand equilibrium.

In practice, it's rare to encounter a candlestick with a zero body size, where the opening price equals the closing price with mathematical precision. Therefore, some sort of tolerance is used when specifying a Doji. For example, let's assume that a Doji candlestick is any candlestick with a body of no more than 5 pips.

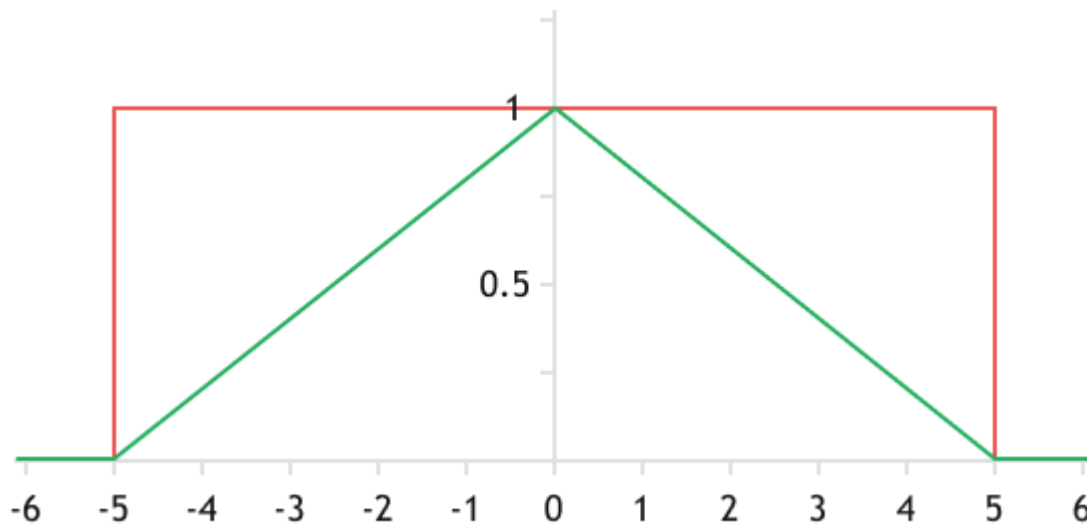
With such an assumption, using conventional logic, candlesticks with a body size of 1 point and 4 points will be classified as Doji and will have the same value for the strategy being used. At the same time, a candlestick with a body of 6 pips will no longer fall into the Doji category and will be ignored by the strategy. Why is it that a deviation of 3 points in the first case ($4 - 1 = 3$) doesn't matter, but a smaller deviation of 2 points ($6 - 4 = 2$) in the second case makes a fundamental difference? The application of fuzzy logic can smooth out these angles and account for deviations in both cases.

The figure below shows the chart of assigning a candlestick to the Doji class (set) depending on the length of the candlestick body. The red line represents the classical mathematical logic with the previously accepted allowance, and the green line reflects the rule of fuzzy logic. As we can see from the graph, the use of fuzzy logic rules will allow us to make decisions depending on the strength level of the incoming signal. For instance, if the candlestick body is larger and approaches the boundaries of the fuzzy set, we can reduce the risk for the operation or even ignore such a signal.

Mathematically, the function showing the membership of a candlestick in the Doji fuzzy set can be represented as:

$$MF_{Doji}(Body) = \begin{cases} \frac{a-Body}{a}, & ABS(Body) \leq a \\ 0, & ABS(Body) > a \end{cases}$$

$$Body = Close - Open$$



Graph of assigning a candle to the Doji class (red - mathematical logic, green - fuzzy logic)

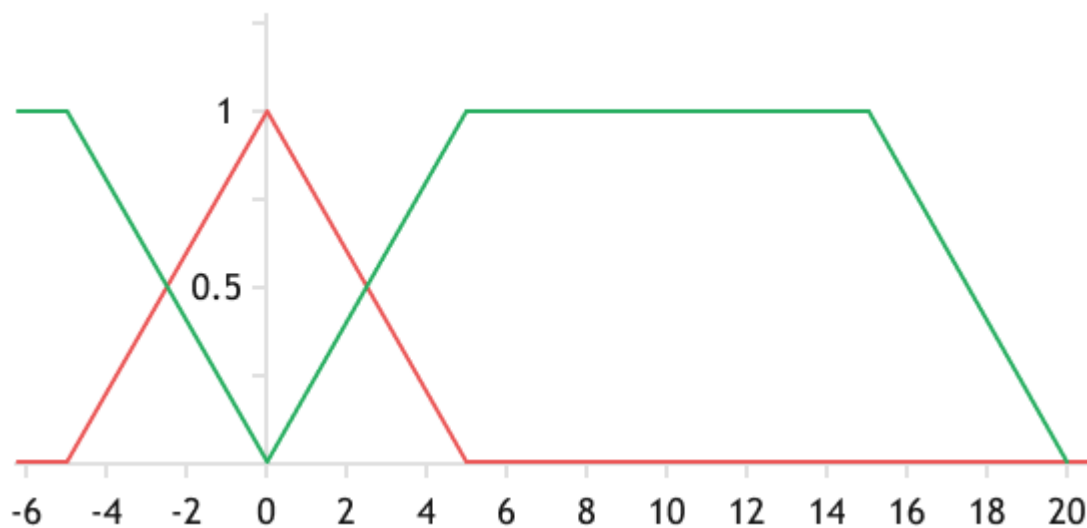
In this case, we have obtained a special case of the symmetric triangular activation function. To define it, in fact, we needed only one parameter a which stands for boundary of the the range of 5 points. The center of the distribution is at point 0. In the general case, to define a triangular membership function, three parameters are required: the lower bound, the center, and the upper bound of the fuzzy set.

There are other membership functions, but the most widespread are the aforementioned triangular, trapezoidal, and Gaussian membership functions. Meanwhile, the triangular and trapezoidal functions can be symmetric (when the left and right zones of boundary fuzziness are equal) and asymmetric.

The graph of a trapezoidal function differs from the graph of a triangular function by the presence of a plateau in the upper part. To define such a function, four points are required, indicating the upper and lower boundaries of the left and right zones of fuzziness. Between the blur zones, the function takes the value 1, and to the left and right of these zones, it takes the value 0. For instance, let's introduce a rule to determine the size of the body of an average candlestick with a body size ranging from 5 to 15 points and a fuzziness boundary zone of 5 points. The mathematical notation of such a rule will take the form:

$$MF_{Midl}(Body) = \begin{cases} 1 - \frac{b-ABS(Body)}{b-a}, & a \leq ABS(Body) \leq b & (0 \leq ABS(Body) \leq 5) \\ 1, & b \leq ABS(Body) \leq c & (5 \leq ABS(Body) \leq 15) \\ 1 - \frac{ABS(Body)-c}{d-c}, & c \leq ABS(Body) \leq d & (15 \leq ABS(Body) \leq 20) \\ 0, & ABS(Body) \notin [a; d] & (ABS(Body) \notin [0; 20]) \end{cases}$$

Thus, we have already defined the second rule for the candlestick body. It is common to show the set of rules for a single variable on a single graph. In the chart below, the red triangular term represents the Doji candlestick, and the green trapezoidal term represents the average candlestick.



The aggregate term sets of Doji (red) and the average statistical candlestick (green).

Please note that on the graph, there is no sharp division between the Doji and the average candlestick at the 5-point mark, as it would be with threshold classification. Instead, we have a line crossing at about 2.5 points. In this case, the membership function will take a value of about 0.5. This means that a candlestick with a body of 2.5 points is equally applicable to the fuzzy Dodgy and medium candlestick sets. In such a case, secondary factors should be looked at to determine the controlling influence.

Continuing such iterations, we can describe the rules for a candlestick with a large body, as well as add rules for the candlestick shadows. Once we have done the work of defining the rules for describing candlesticks and their components, we will be able to describe various candlestick patterns with ease. For example, we can use fuzzy logic tools to describe a pin bar in quite a simple way by definition: a candlestick with a long one shadow, a small body, and a small or absent second shadow.

Note that using the rules of fuzzy logic allows us to move from clear values to some abstract definitions and approximate reasoning inherent in human logic. Therefore, the concepts of linguistic and fuzzy variables are introduced in fuzzy logic theory.

The linguistic variable has:

- A *name*, in the above examples it is "Candlestick Body";
- A *set of its values* referred to as the base *termset*. In our case, these are the Doji, Medium (regular) and Large candlesticks;
- A *set of permitted values*;
- A *syntactic rule* that describes terms using natural language words;
- A *semantic rule* defining the correspondence between the values of a linguistic variable and a fuzzy set of valid values.

In general practice, the fuzziness of the boundaries of fuzzy sets allows us to consider the natural symbiosis of the influence of different forces in the areas of their intersection. It also allows us to account for the fact that the effect of the force fades as the distance from the source of the impact increases.

The presence of fuzzy rules is an important, but not the sole, part of constructing a model. The process of fuzzy model building can be divided into three conventional stages:

1. Selecting baseline data
2. Defining a knowledge base (set of rules)
3. Defining the fuzzy logic inference method

It is quite natural that the entire construction process depends on the initial stage: the determination of the set of source data influences both the overall possibility of their classification and the number of possible classes (terms). Consequently, the set of rules (as well as their filling) for defining fuzzy sets is also determined based on the set of permitted values and the task at hand. It should be noted that even for the same set of input data, the underlying term set and rule set may vary depending on the task at hand.

Very often the parameters of the rules for defining fuzzy sets are strongly influenced by the subjective knowledge and experience of the model architect. Therefore, the practice of hybrid models has become widespread. In them, the parameter selection of rules is carried out by a neural network during its training on a training dataset.

Based on the created knowledge base, a fuzzy logic inference system is defined in the model. Fuzzy logical inference is the process of obtaining a fuzzy set that corresponds to the current input values, using fuzzy rules and fuzzy operations.

A number of logical operations have been developed for fuzzy sets, just as for regular sets. The main ones are union (fuzzy OR) and intersection (fuzzy AND). There is a general approach to performing fuzzy intersection, union, and complement operations.

To build the process of fuzzy logical inference, the MQL5 library offers the implementation of two main methods: [Mamdani](#) and [Sugeno](#).

When using the Mamdani method, the value of the output variable is defined by a fuzzy term. The fuzzy rule of this method can be described as follows:

$$\text{if } (X_1 \text{ is } a_1) \wedge (X_2 \text{ is } a_2) \wedge \dots \wedge (X_n \text{ is } a_n) \text{ then } (Y \text{ is } d)(W),$$

Where:

- $\mathbf{X}$ = a vector of input variables
- $\mathbf{Y}$ = an output variable
- $\mathbf{a}$ = a vector of initial data
- $\mathbf{d}$ = a value of the output variable
- $\mathbf{W}$ = the rule weight

In the Sugeno method, unlike Mamdani, the value of the output variable is determined not by a fuzzy set but by a linear function of the input data. The rule of this method is of the form:

if (X_1 is a_1) $\wedge$ (X_2 is a_2) $\wedge$... $\wedge$ (X_n is a_n) *then* ($Y = b_0 + X_1b_1 + X_2b_2 + \dots + X_nb_n$)(W),

where $\mathbf{b}$ is the vector of weights at free terms of the output value function.

2.3 OpenCL: Parallel computations in MQL5

A large huge number of calculations are carried out while training and running neural networks. This is a rather resource-intensive process. Solving more complex problems also requires more complex neural networks with a large number of neurons. With an increase in the number of neurons in the network, the amount of computations performed increases, and consequently, the consumption of resources and time also increases. While humanity has learned to create new and more advanced computational machines, managing time is still beyond our capabilities.

It's natural that the next wave of neural network development has come with the advancement of computational power. As a rule, neurons perform fairly simple operations, but in large numbers. At the same time, there are a lot of similar neurons in the neural network. This allows for parallelizing individual blocks of computations on different computational resources and then consolidating the obtained data. As a result, the time for performing operations is significantly reduced.

The development of computing technologies has led to the emergence of video cards (GPU) with a large number of computing cores capable of performing simple mathematical operations. Next, it became possible to transfer part of the calculations from the CPU to the GPU, which made it possible to parallelize the calculation process both between the microprocessor and the video card, and at the video card level between different computing units.

[OpenCL](#) (Open Computing Language) is an open free standard for cross-platform parallel programming of various accelerators used in supercomputers, cloud servers, personal computers, mobile devices, and embedded platforms.

OpenCL is a C-like programming language that allows GPU computing. The support of this language in MQL5 allows us to organize multi-threaded calculations of our neural networks on the GPU directly from the MQL5 program.

To understand the organization of GPU computing, it is necessary to make a short digression into the architecture of video cards and the OpenCL API.

In OpenCL terminology, a computer's microprocessor (CPU) is a *Host*. It manages all the processes of the program that is being executed. All microprocessors with support for OpenCL technology in the CPU and GPU are *Devices*. Each device has its own unique number within the platform.

One Device can have multiple *Computer Units*. Their number is determined by the number of physical and virtual microprocessor cores. For video cards, these will be SIMD cores. Each SIMD core contains several *Stream Cores*. Each thread processor has several processing elements *Processing Elements* (or *ALU*).

The specific number of *Computer Units*, *SIMD cores*, *Stream Cores*, and *Processing Elements* depends on the architecture of a particular device.

An important feature of the GPU is vector computing. Each microprocessor consists of several computing modules. They can all execute the same instruction. At the same time, different executable threads may have different initial data. This allows all threads of the GPU program to process data in parallel. Thus, all computing modules are loaded evenly. A big advantage is that vectorization of computations is done automatically at the hardware level, without the need for additional processing in the program code.

OpenCL was developed as a cross-platform environment for creating programs using mass parallel computing technology. The applications created in it have their own hierarchy and structure. Organization of the program, preparation, and consolidation of data are carried out in the *Host* program.

The *Host*, like a regular application, starts and runs on the CPU. To organize multi-threaded computations, a context is allocated, which is an environment for executing specialized objects of the OpenCL program. A context combines a set of OpenCL devices for running a program, the program objects themselves with their source codes, and a set of memory objects visible to the host and to OpenCL devices. The function responsible for creating a context in *MQL5* is [CLContextCreate](#), in which the device for executing the program is specified as a parameter. The function returns a context handle.

```
int CLContextCreate(  
    int          device      // OpenCL device sequence number or macro  
);
```

Inside the context, an *OpenCL* program is created using the [CLProgramCreate](#) function. The parameters of this function include the context handle and the source code of the program itself. As a result of the function execution, we obtain a program handle.

```
int CLProgramCreate(  
    int          context,    // handle for OpenCL context  
    const string source      // source  
);
```

An OpenCL program is divided into separate kernels which are executable functions. The [CLKernelCreate](#) function is provided for declaring a kernel. In the parameters of the function, you specify the handle of the previously created program and the name of the kernel within it. At the output, we get the handle of the kernel.

```
int CLKernelCreate(  
    int          program,    // handle to an OpenCL object  
    const string kernel_name // kernel name  
);
```

Please note that later, when the kernel is called from the main program on the GPU, several of its instances are launched in different parallel threads. This defines the index space, *NDRange*, which can be one-dimensional, two-dimensional, or three-dimensional. *NDRange* is an array of integers. The size of

the array indicates the dimension of the space, and its elements indicate the dimension in each of the directions.

Each copy of the kernel is executed for each index from this space and is called a *Work-Item*. Each work item is provided with a global index *ID*. In addition, each such unit executes the same code, but the data for execution may be different.

Work items are organized into *Work-Groups*. Work groups represent a larger partition in the index space. Each group is assigned a group index *ID*. The dimensionality of work groups corresponds to the dimensionality used for addressing individual elements. Each element is assigned a unique local index *ID* within the group. Thus, work units can be addressed either by the global index *ID* or by a combination of group and local indices.

This approach allows for reducing the computation time, but at the same time complicates the process of data exchange between different kernel instances. This needs to be taken into account when creating programs.

As mentioned above, the OpenCL program operates within its own context, isolated from the calling program. As a consequence, it does not have access to the variables and arrays of the main program. Therefore, before starting the program, you need to copy all the data necessary for executing the program from RAM to GPU memory. After the kernel has finished its execution, the obtained results need to be loaded back from the GPU memory. At this point, you need to understand that the time spent on copying data from RAM to GPU memory and back is an overhead time when performing calculations on video cards. Therefore, in order to reduce the overall execution time of the entire program, transferring calculations to the GPU is advisable only when the time saved from GPU computations significantly outweighs the costs of data transfer.

Inside the *GPU*, there is also a ranking of memory into global, local, and private. The fastest access is achieved for the kernel to private memory, but access to it is only possible from the current instance of the kernel. The most time-consuming access is required for the global memory, but its capacity is the largest among the three mentioned. All running instances of the kernel have access to it. Global memory is used to exchange information with the main program.

Global memory provides read and write access to elements for all work groups. Each *Work-Item* can write to and read from any part of the global memory.

Local memory is a group-local memory area, in which you can create variables shared by the entire group. It can be implemented as dedicated memory on the OpenCL device or allocated as a region within the global memory.

Private memory is an area visible only to the *Work-Item*. Variables defined in the private memory of one work item are not visible to others.

Sometimes constant memory is also allocated. This is an area of global memory that remains constant during the execution of the kernel. The host allocates and initializes memory objects located in constant memory.

Let's consider two implementations of a single task: one using the OpenCL technology and the other without. As you will see later, one of the main operations we will be using is matrix multiplication. We will be performing matrix multiplication of matrix by matrix and matrix by vector. For the experiment, I propose comparing the multiplication of a matrix by a vector.

For the matrix-vector multiplication function in the classical implementation, we will use a system of two nested loops. Below is an example of such an implementation. The function parameters include a

matrix and two vectors: the matrix and one vector of input data, as well as one vector for storing the results. According to the rules of vector mathematics, multiplication is only possible when the number of columns in the matrix is equal to the size of the vector. The result of such an operation will be a vector with a size equal to the number of rows in the matrix.

```
//+-----+
//| CPU vector multiplication function |
//+-----+
bool MultCPU(matrix<TYPE> &source1, vector<TYPE> &source2, vector<TYPE> &result)
{
//---
    ulong rows = source1.Rows();
    ulong cols = source1.Cols();
    if(cols != source2.Size())
    {
        PrintFormat("Size of vectors not equal: %d != %d", cols, source2.Size());
        return false;
    }
//---
    result = vector<TYPE>::Zeros(rows);
    for(ulong r = 0; r < rows; r++)
    {
        result[r] = 0;
        for(ulong c = 0; c < cols; c++)
            result[r] += source1[r, c] * source2[c];
    }
//---
    return true;
}
//+-----+
```

In the body of the function, we will first determine the dimensions of the matrix, check for compatibility with the vector size, and set the size of the output vector to be equal to the number of rows in the input matrix. After that, we will construct a system of loops. The outer loop will iterate over the rows of the matrix and, accordingly, the elements of the result vector. In the body of this loop, we will start by setting the corresponding element of the result vector to zero. Then, we will create a nested loop and calculate the sum of the products of corresponding elements of the current matrix row and the vector.

The function is not complicated. However, the weak point of such an implementation is the increase in execution time proportional to the growth of the number of elements in the matrix and the vector.

This issue can be solved by using OpenCL. Of course, such an implementation will be a little more complicated. First, let's write an OpenCL program and save it in the *mult_vect_ocl.cl* file. The **.cl* extension is generally accepted for OpenCL programs, but not necessary for implementation in the *MQL5* environment. In this case, we will use the file only to store the program text, while the program will be loaded as text.

We will enable support for the *double* data type in the program code. Please note that not all GPUs support the *double* type. And even if they do, in most cases this functionality is disabled by default.

```
//--- By default some GPUs don't support doubles
//--- cl_khr_fp64 directive is used to enable work with doubles
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
```

And another aspect to consider. MetaTrader 5 allows the use of OpenCL devices for calculations with both double precision support and without. Therefore, when using the double data type in your OpenCL program, it's important to check the compatibility of the used device. Otherwise, we can get an error during the execution of the OpenCL program and while terminating its operation.

At the same time, MetaTrader 5 does not limit the ability to use all available data types. The OpenCL language allows for the use of various scalar data types:

- Boolean: *bool*
- Integers: *char, uchar, short, ushort, int, uint, long, ulong*
- Floating-point: *float, double*

Similar data types are also supported in MQL5. It's important to remember that each data type has its own limitations on the possible range of values, as well as the amount of memory used to store the data. Therefore, if your program doesn't require high precision or the range of possible values isn't too large, it's recommended to use less resource-intensive data types. This will allow more efficient use of device memory and reduce the cost of copying data between the main memory and OpenCL context memory. In particular, the type *double* can be replaced with *float*. It provides lower precision, but it occupies half the memory and is supported by all modern OpenCL devices. This helps reduce the costs of data transfer between devices and expand the application's usability.

OpenCL also allows you to use vector data types. Vectorization allows parallelizing computations at the microprocessor level rather than at the software level. Using a vector of four elements of the type *double* allows you to completely fill the 256-bit vector of *SIMD* instructions and perform calculations on the entire vector in one cycle. In this way, during one clock cycle of the microprocessor, we perform operations on four elements of our data array.

OpenCL supports vector variables of all integer and floating point types of 2, 3, 4, 8, and 16 elements. However, the possibility of using them depends on the specific device. Therefore, before choosing a vector dimension, check the technical characteristics of your equipment.

Now back to our program. Please provide the kernel code for calculating the vector product of a matrix row with a vector. In the kernel parameters, we will specify pointers to the arrays of input data and results. We will also pass the number of columns in the matrix as a parameter. The number of rows in the matrix does not matter for operations in the kernel, since it only performs operations of multiplying one row by a vector. Essentially, it is the multiplication of two vectors.

Note here that instead of the type of data buffers, we specified the abstract type *TYPE*. You will not find such a data type in any documentation. In fact, as mentioned above, not all OpenCL devices support the type *double*. To make our program more versatile, it was decided to replace the data type using a macro substitution. We will specify the actual data type in the main program. This approach allows us to literally change the data type in one place in the main program. After that, the entire program will switch to working with the specified data type without the risk of losing information due to type mismatch.

In the kernel body, the *get_global_id* function will specify the global *ID* index of the running *Work-Item* unit. In this case, the index serves as an equivalent to the iteration counter of the outer loop in the classical implementation. It specifies the sequence number of the matrix array and the element of the result vector. Next, we will calculate the sum of values for the corresponding thread in a similar manner to the calculation inside the nested loop of the classical implementation. But there is a nuance here. For the calculations, we will utilize vector operations with four elements. In turn, to use vector operations, we need to prepare the data. We get an array of scalar elements from the *Host* program, so we will transfer the necessary elements to our private vector variables using the *ToVect* function (we

will consider its code below). Then, using the vector operation *dot*, we obtain the value of the multiplication of two vectors of four elements. In other words, with one operation, we obtain the sum of the products of four pairs of values. The obtained value is added to a local variable where the product of the matrix row and the vector accumulates.

After exiting the loop, we will save the accumulated sum into the corresponding element of the result vector.

```
//+-----+
//| Mult of vectors                                     |
//+-----+
__kernel void MultVectors(__global TYPE *source1,
                          __global TYPE *source2,
                          __global TYPE *result,
                          int cols)
{
    int shift = get_global_id(0) * cols;
    TYPE z = 0;
    for(int i = 0; i < cols; i+=4)
    {
        TYPE4 x = ToVect(source1, i, cols, shift);
        TYPE4 y = ToVect(source2, i, cols, 0);
        z += dot(x,y);
    }
    result[get_global_id(0)] = z;
}
```

As mentioned earlier, to transfer data from the scalar value buffer to a vector variable, we have created the *ToVect* function. In the function parameters, we pass a pointer to the data buffer, the starting element, the total number of elements in the vector (matrix row), and the offset in the buffer before the beginning of the vector. The last parameter, offset, is needed to accurately determine the start of a row in the matrix buffer since OpenCL uses one-dimensional data buffers.

Next, we check the number of elements until the end of the vector to avoid going beyond its bounds and transfer the data from the buffer to the private vector variable. We fill the missing elements with zero values.

```

TYPE4 ToVect(__global TYPE *array, int start, int size, int shift)
{
    TYPE4 result = (TYPE4)0;
    if(start < size)
    {
        switch(size - start)
        {
            case 1:
                result = (TYPE4)(array[shift+start], 0, 0, 0);
                break;
            case 2:
                result = (TYPE4)(array[shift+start], array[shift+start + 1], 0, 0);
                break;
            case 3:
                result = (TYPE4)(array[shift+start], array[shift+start + 1],
                                array[shift+start + 2], 0);
                break;
            default:
                result = (TYPE4)(array[shift+start], array[shift+start + 1],
                                array[shift+start + 2], array[shift+start + 3]);
                break;
        }
    }
    return result;
}

```

As a result, the function returns the created vector variable with the corresponding values.

This completes the *OpenCL* program. Next, we will continue working on the side of the main program (*Host*). *MQL5* provides the *COpenCL* class in the standard library *OpenCL.mqh* for operations with *OpenCL*.

First, let's perform the preparatory work: we will include the standard library, load the previously created *OpenCL* program as a resource, and declare constants for the kernel, buffer, and program parameters indices. We will also specify the data type used in the program. I specified the *float* type because my laptop's integrated GPU does not support *double*.

```

#include <OpenCL/OpenCL.mqh>
#resource "mult_vect_ocl.cl" as string OCLprogram
#define TYPE float
const string ExtType = StringFormat("#define TYPE %s\r\n"
                                     "#define TYPE4 %s4\r\n",
                                     typename(TYPE), typename(TYPE));

//+-----+
//| Defines |
//+-----+
#define cl_program ExtType+OCLprogram
//---
#define k_kernel 0
#define k_source1 0
#define k_source2 1
#define k_result 2
#define k_cols 3

```

Let's declare an instance of the class for working with OpenCL and variables for storing data buffer handles.

```
COpenCL*          cOpenCL;
int               buffer_Source1;
int               buffer_Source2;
int               buffer_Result;
```

In the next step, we will initialize an instance of the class. To do this, we will create the *OpenCL_Init* function. In the function parameters, we will pass the matrix and the vector of input data.

In the function body, we will create an instance of the class for working with OpenCL, initialize the program, specify the number of kernels, and create pointers to the kernel and data buffers. We will also copy the input data into the context memory. At each step, we check the results of the operations, and in case of an error, we exit the method with a result of *false*. The function code is provided below.

```
bool OpenCL_Init(matrix<TYPE> &source1, vector<TYPE> &source2)
{
//--- creation of OpenCL program, kernel and buffers
cOpenCL = new COpenCL();
if(!cOpenCL.Initialize(cl_program, true))
    return false;
if(!cOpenCL.SetKernelsCount(1))
    return false;
if(!cOpenCL.KernelCreate(k_kernel, "MultVectors"))
    return false;
buffer_Source1 = CLBufferCreate(cOpenCL.GetContext(),
                                (uint)(sizeof(TYPE) * source1.Rows() *
                                        source1.Cols()), CL_MEM_READ_ONLY);
buffer_Source2 = CLBufferCreate(cOpenCL.GetContext(),
                                (uint)(sizeof(TYPE) * source2.Size()),
                                CL_MEM_READ_ONLY);

buffer_Result = CLBufferCreate(cOpenCL.GetContext(),
                                (uint)(sizeof(TYPE) * source1.Rows()),
                                CL_MEM_WRITE_ONLY);
if(buffer_Result <= 0 || buffer_Source1 <= 0 || buffer_Source2 <= 0)
    return false;
if(!CLBufferWrite(buffer_Source1,0,source1) ||
    !CLBufferWrite(buffer_Source2,0,source2))
    return false;
//---
return true;
}
```

The actual calculations will be carried out in the kernel. To run it, let's write the *MultOCL* function. In the function parameters, we will pass a pointer to the result vector and the dimensions of the input data matrix.

First, we will pass pointers to data buffers and parameters of buffer sizes to the kernel. These operations are performed by the *CLSetKernelArgMem* and *SetArgument* methods. We define the index space in the *NDRange* array according to the number of rows in the source data matrix. The kernel is launched for execution using the *Execute* method. After executing the entire array of kernel instances, we read the computation results from the device memory using the *CLBufferRead* method.

```

bool MultOCL(int rows, int cols, vector<TYPE> &result)
{
    result=vector<TYPE>::Zeros(rows);
    //--- Set parameters
    if(!CLSetKernelArgMem(cOpenCL.GetKernel(k_kernel), k_source1, buffer_Source1))
        return false;
    if(!CLSetKernelArgMem(cOpenCL.GetKernel(k_kernel), k_source2, buffer_Source2))
        return false;
    if(!CLSetKernelArgMem(cOpenCL.GetKernel(k_kernel), k_result, buffer_Result))
        return false;
    if(!cOpenCL.SetArgument(k_kernel, k_cols, cols))
        return false;
    //--- Run kernel
    int off_set[] = {0};
    int NDRange[] = {rows};
    if(!cOpenCL.Execute(k_kernel, 1, off_set, NDRange))
        return false;
    //--- Get result
    uint data_read = CLBufferRead(buffer_Result, 0, result);
    if(data_read <= 0)
        return false;
    //---
    return true;
}

```

After the program has finished running, it's necessary to release resources and delete the instance of the class for working with OpenCL. This functionality is performed in the *OpenCL_Deinit* function. In it, we will first check the validity of the pointer to the object, then call the *Shutdown* method to release resources, and finally delete the object.

```

void OpenCL_Deinit()
{
    if(!cOpenCL)
        return;
    //---
    cOpenCL.Shutdown();
    delete cOpenCL;
}

```

Obviously, when using OpenCL, the amount of work for the programmer increases. What do we get in return?

To evaluate the performance, let's create a small script *openc1_test.mq5*. In the external parameters of the script, we specify the size of the input data matrix.

```

//+-----+
//| External parameters |
//+-----+
sinput int Rows = 100000; // Rows in a matrix
sinput int Colms = 100;   // Columns in a matrix

```

In the body of the script, let's declare the matrix and data vectors. We will fill the input data with random values.

2. MetaTrader 5 features for algorithmic trading

```
//+-----+
//| Script Program |
//+-----+
void OnStart()
{
    matrix<TYPE> X = matrix<TYPE>::Zeros(Rows, Colms);
    vector<TYPE> Y = vector<TYPE>::Zeros(Colms);
    vector<TYPE> Z;
    for(int i = 0; i < Colms; i++)
    {
        for(int r = 0; r < Rows; r++)
            X[r, i] = MathRand() / (TYPE)32767;
        Y[i] = MathRand() / (TYPE)32767;
    }
}
```

In the next step, we will initialize the OpenCL context by calling the previously discussed *OpenCL_Init* function. At the same time, do not forget to check the results of the operations.

```
if(!OpenCL_Init(X, Y))
    return;
```

Now we can measure the speed of operations in the OpenCL context. Using the *GetTickCount* function, we get the number of milliseconds from the system start before and after the calculations. Calculations are feasible in the previously considered *MultOCL* function.

```
uint start = GetTickCount();
if(!MultOCL(Rows, Colms, Z))
    Print("Error OCL function");
uint end = GetTickCount();
PrintFormat("%.1e OCL duration %0 000d msec, result %.5e",
            Rows * Colms, end - start, Z.Sum());
OpenCL_Deinit();
```

After performing the operations, we clear the OpenCL context.

In a similar manner, we will measure the execution time of operations using the classical method on the CPU.

```
start = GetTickCount();
if(!MultCPU(X, Y, Z))
    Print("Error CPU function");
end = GetTickCount();
PrintFormat("%.1e CPU duration %0 000d msec, result %.5e",
            Rows * Colms, end - start, Z.Sum());
```

In conclusion of the script, we will once again add a timing measurement for the matrix-vector multiplication using matrix operations in MQL5.

```
start = GetTickCount();
Z = X.MatMul(Y);
end = GetTickCount();
PrintFormat("%.1e matrix operation duration %0 000d msec, result %.5e",
            Rows * Colms, end - start, Z.Sum());
}
```


The described script was tested on a laptop with an Intel Core i7-1165G7 CPU and an integrated Intel(R) Iris(R) Xe GPU. Based on the measured execution times, the OpenCL technology emerged as the winner. The slowest was the classical implementation using the nested loops system. Furthermore, the computation results were identical in all three variants.

OpenCL: GPU device 'Intel(R) Iris(R) Xe Graphics' selected
1.0e+07 OCL duration 0 msec, result 2.60176e+06
1.0e+07 CPU duration 125 msec, result 2.60176e+06
1.0e+07 matrix operation duration 16 msec, result 2.60176e+06

The results of the comparative testing of computations using OpenCL and without it are as follows:

It's important to note that when measuring the computation speed using OpenCL technology, we excluded overhead costs such as the initialization and deinitialization of the OpenCL context, program, buffers, and data transfer. Therefore, when performing individual operations, its usage might not be as efficient. However, as will be shown further, during the training and operation of neural networks, there will be many such operations, and the process of initializing the OpenCL context and program will only occur once, during the program launch. At the same time, we will try to minimize the process of data exchange between devices. Therefore, utilizing this technology will be highly beneficial.

2.4 Integration with Python

Python is a high-level programming language with dynamic typing and automatic memory management. It is oriented towards improving developer productivity and code readability, and it belongs to fully object-oriented programming languages.

Python belongs to interpreted programming languages. It is often used to create scripts.

The syntax of the language is minimalistic, which increases the productivity of the programmer. In conjunction with the language interpretability, this allows for quick coding and immediate testing of individual program components. This helps reduce the time spent on finding and fixing errors during debugging of software products, and in some cases, it enables the evaluation of solution effectiveness at the design stage without the need to create a complete product.

At the same time, interpreted programming languages are noticeably inferior to compiled ones in terms of program execution speed. The solution to this problem lies within the Python architecture itself. It is designed so that its small core can be easily extended with a set of libraries, including those written in compiled programming languages.

Thus, Python can be compared to a constructor in which programs are assembled from ready-made blocks that are already written and defined in libraries. This explains the large number of standard libraries. Moreover, in your program, you utilize only the functionality that is necessary to solve a specific task.

An unusual feature of the language is the use of whitespace indentation to denote code blocks. If you're accustomed to the clear delineation of code blocks with curly braces in C-like languages, this might seem inconvenient. On the other hand, structuring the program code makes it visually understandable. One glance at the code is enough to determine the presence of nested blocks and their boundaries.

At the same time, this places a certain responsibility on the programmer. While in languages where the compiler checks for the presence of opening and closing braces and issues an error message if they

don't match, in the case of structuring code with indentation, the responsibility lies entirely on the programmer. In this case, an incorrect structure can change the course of program execution.

Dynamic typing allows the programmer to be less concerned about data compatibility when storing them in variables, as the variables will automatically acquire the type of data being assigned.

The standard library contains a large set of useful functions. There are tools for working with text, and for writing network applications.

Additional functionality can be implemented using a wide range of third-party libraries. Among them, you can find tools for mathematical modeling, and for writing web applications, and for developing games. In addition, there is the possibility of integrating libraries written in C or C++ and other languages.

A specialized software repository has been created for software written in Python, which provides tools for easy installation of packages into the operating system. Among the repository libraries, you can find functions to suit any preference, including those for currency markets and machine learning.

Considering all the above, Python has become one of the most popular programming languages. It is used in data analysis and machine learning. As of July 2021, Python is ranked third in the TIOBE Programming Language Popularity Rankings with a score of 10.95%.

Starting with the Build 2085 version, released in June 2019, MetaTrader 5 received API for requesting data from the terminal to Python applications. Since then, this functionality has been constantly developed. Currently, you can run Python scripts directly on the terminal chart along with MQL5 applications.

At the same time, the functionality of Python applications is also expanding. You can fetch quotes from the terminal for analysis and based on the analysis results, open and close positions, and set pending orders. There's also the capability to retrieve information about the current account status, open positions, and orders. For a complete list of features, see the [Python integration documentation page](#).

To set up a Python connection to MetaTrader 5, you first need to download and install the latest version of the interpreter from <https://www.python.org/downloads/windows/>.

When installing Python, be sure to check the "Add Python 3.9 to PATH%" checkbox (version may vary) to be able to run Python scripts from the command line.

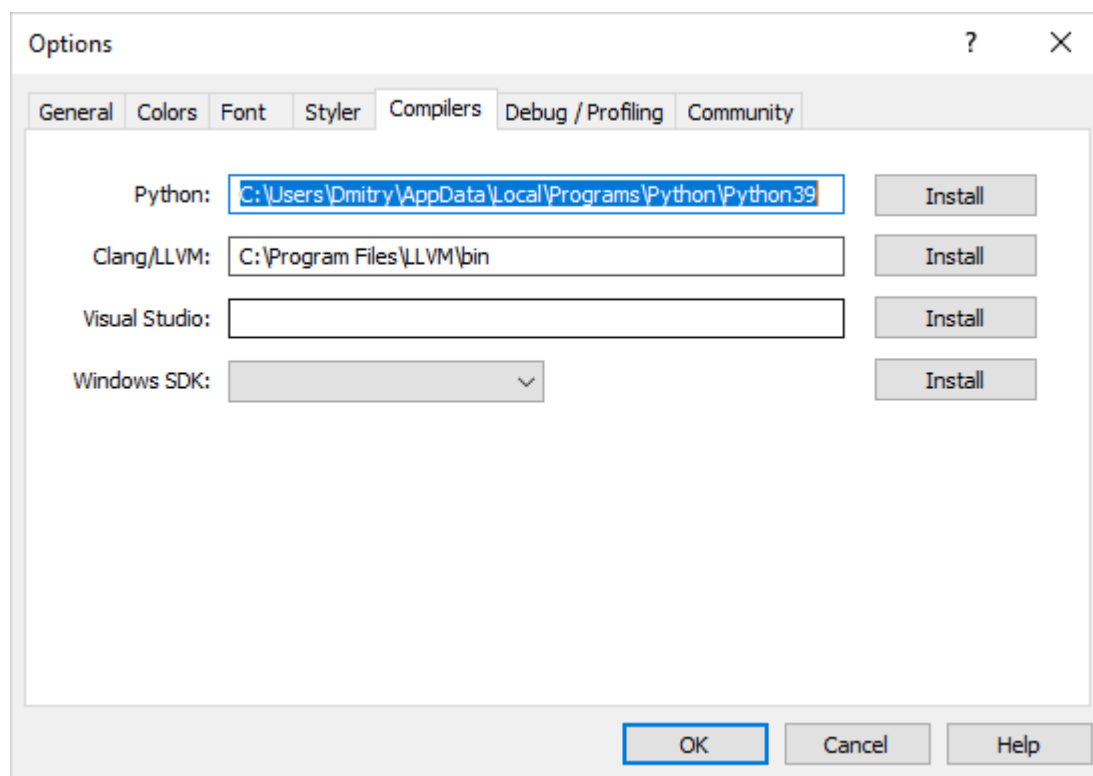
After that, launch and update the *MetaTrader5* module. In this case, we are talking about the *Python library*, not the terminal. To do this, enter the following commands at the command prompt.

```
pip install MetaTrader5
pip install --upgrade MetaTrader5
```

After these iterations, Python scripts will be able to access operations with the MetaTrader 5 terminal.

MetaEditor also has Python support. In the editor settings on the "Compilers" tab, all you need to do is specify the location of the interpreter.

After that, you can create multilingual projects in the MetaEditor integrated environment. Such projects will include programs written in MQL and Python. Similarly, you can add support for the C/C++ language.



Python integration in MetaEditor

3. Building the first neural network model using MQL5

This book should not be regarded as a textbook for studying artificial intelligence and neural networks. Its purpose is not to serve as a comprehensive work that encompasses all aspects of this field. On the contrary, the book provides only the basic concepts without delving into the mathematical explanations of specific points.

It aims to be a practical work. We invite you to explore possible solutions to a practical case and compare the effectiveness of different algorithms in solving a particular problem. We believe this book will be useful in studying the practical implementation of various neural network algorithms, their training, and practical use.

And, of course, our case study will be directly related to financial markets. Artificial intelligence technologies have long been used in the financial sector, but this topic has not yet received wide coverage. This is largely due to the commercial use of such products.

This chapter delves into the topic of algorithmic trading, with an emphasis on demonstrating various methodologies for addressing tasks related to algorithmic trading, as well as analyzing and comparing the performance of different algorithmic approaches. The discussion is based on a clear [statement of the problem](#), which includes defining key objectives and constraints specific to the financial markets context.

A separate section covers the [selection and analysis of raw data](#), including the choice of suitable financial metrics and the analysis of their correlations. It also explains the features of time series, which are critically important for successfully forecasting market movements. Also, the chapter shows how to create the [framework of the future program](#) in MQL5 and how to declare constants to ensure code robustness and portability. You will learn how to describe the structure of the created neural network and how to effectively organize work with complex network architectures.

In the section on [creating the base neural network class](#), you will be introduced to the concepts of feed-forward and backpropagation runs in the context of neural network programming. Special attention is given to dynamic arrays for storing neural layers, greatly simplifying the management of complex data structures during the development process.

The description of methods for integrating a neural network into a Python program provides an understanding of how various components of a neural network can be combined into a single system using the capabilities of this popular programming language. The section on the [fully connected neural layer](#) provides details about its architecture and creation principles, explaining the general structure and operation of such layers in neural networks. In addition, it covers the process of creating an activation function class and selecting appropriate activation functions.

The section on [parallel computing using OpenCL](#) demonstrates how this technology can be used to speed up computational processes in neural networks. This solution can significantly increase the efficiency of data processing by distributing tasks among several computing devices.

3.1 Problem statement

Before embarking on the practical implementation of our first neural network, it's essential to define the objective and the means to achieve it. When developing the architecture of a neural network, we must have a clear understanding of what data should be provided as input and output for the neural network. The number of neurons in the input layer and their type entirely depend on the dataset being used. The architecture of the output layer depends on the expected outcome and on how the results of the developed neural network's work will be represented.

Let's formulate a problem that we would like to solve using artificial intelligence. We work on financial markets, and we need a tool to forecast the future movement of the analyzed instrument. The task seems somewhat familiar and is often on the minds of traders. Everyone tries to solve it in their own way.

But there are no specifics in this task. What do we mean by "future movement"? Does the future arrive in 5 minutes, 1 hour, 1 day, or 1 month? How about something in between? What is the minimum price movement we will react to? What metrics can we use to evaluate the accuracy of our model's performance? Our goal must be specific and measurable.

We realize that price does not move in a straight line. There are always large and small price fluctuations. Small fluctuations are essentially noise. To identify trends and tendencies that can potentially yield profits, we must filter out this noise. The MetaTrader 5 platform provides the ZigZag indicator. It is one of the oldest indicators used in financial markets. The sole purpose of this indicator is to identify the most significant extremes on the instrument's chart, thereby indicating trends and tendencies while excluding minor noisy fluctuations.

3. Building the first neural network model in MQL5



Three parameters are used to customize the indicator:

- **Depth** sets the number of candlesticks to search for extrema. As the parameter increases, the indicator highlights the most significant extremes.
- **Deviation** defines the number of points between two neighboring extrema to be displayed on the chart.
- **Backstep** indicates the minimum distance between neighboring extrema in candlesticks.

In our case, we can use ZigZag to find extrema and specify training targets for our neural network. By applying the indicator to historical data of the training set, we can determine the direction and distance to the nearest extreme for each candlestick and its preceding candlestick combination. By doing so, we will teach the model to determine the potential direction and strength of a future price movement.

Metrics for evaluating the model's performance can include both the proportion of correctly predicted directional movements and the accuracy in determining the strength of such movements.

The task of predicting the upcoming direction of movement is regarded as a binary classification problem. Based on the ZigZag indicator data, at any given point we can have either an upward movement of the price chart (*Buy*) or a downward movement of the price chart (*Sell*). This does not contradict the generally accepted division of trend movements into *BUY*, *SELL*, and *FLAT*, as flat movements are essentially alternating *Buy* and *Sell* oscillations of small amplitude.

At the same time, using mathematical statistics alone, we cannot provide a definitive answer about the direction of the upcoming movement. We can only provide a probabilistic answer based on our past experience. We will "draw" this experience from the training sample.

As for predicting the strength of movement, here we would like to obtain a quantitative assessment. This will help to correctly assess the risk of the trade and determine the point for setting take profit with the highest probability of achievement.

In this way, the task of predicting future movement becomes specific and measurable. Let's formulate it as forecasting the most probable direction of the upcoming price movement and its expected strength.

3.2 File arrangement structure

In this book, we will create many files for various purposes. Before proceeding, I suggest that you decide on the structure of the file arrangement. It should be noted that working in the MQL5 development environment imposes some limitations on the file structure: each type of program has its own directory.

- *terminal_dir\MQL5\Experts* is the directory for storing Expert Advisors;
- *terminal_dir\MQL5\Indicators* stores indicators;
- *terminal_dir\MQL5\Scripts* is the directory for scripts;
- *terminal_dir\MQL5\Include* is the directory for storing various libraries of included files;
- *terminal_dir\MQL5\Libraries* is the directory for storing compiled dynamic libraries.

At the same time, the development environment does not restrict the creation of subdirectories for organizing files. Within the scope of this book, we will be creating three types of files. First and foremost, we have our library of include files, where we will primarily focus on organizing the operation of neural network models. As part of testing the created models, we will be generating and using various scripts. At the end of the book, we will create an Expert Advisor template to demonstrate the approaches of using models in practical trading.

Thus, we will create our files in three subdirectories:

- *terminal_dir\MQL5\Experts* for Expert Advisors;
- *terminal_dir\MQL5\Scripts* for scripts;
- *terminal_dir\MQL5\Include* for various libraries of included files.

To separate our files from all others, we will create a *NeuroNetworksBook* subdirectory in each of the specified directories. We will specify deeper structuring for each file we create.

3.3 Choosing the input data

Having defined the problem statement, let's now turn our attention to the selection of input data. There are specific approaches to this task. At first glance, it might seem like you could load all available information into the neural network and let it learn the correct dependencies during training. This approach will prolong learning indefinitely, without guaranteeing the desired outcome.

The first challenge we face is the volume of information. To convey a large amount of information to the neural network, we would need a considerably large input layer of neurons with a substantial number of connections. Hence, more training time will be required.

Furthermore, we will encounter the issue of data incomparability. Samples of different metrics will have very different statistical characteristics. For example, the price of an instrument will always be positive, while its change can be both positive and negative.

Some indicators have normalized values and others do not. The magnitudes of values for various indicators and the amplitude of their changes can differ by orders of magnitude. However, their impact

on the outcome may be comparable, or an indicator with lower values may have an even greater impact.

Such a situation will significantly complicate the training process, as it will be challenging to discern the impact of small values within an array of large values.

Another problem lies in the use of highly correlated features. The presence of a correlation between features can indicate either a cause-and-effect relationship between them or that both variables are dependent on a common underlying factor. Consequently, the use of correlated variables combines the two mentioned problems and their consequences. Using multiple variables depending on one factor exaggerates its impact on the overall outcome. Unnecessary neural connections complicate the model and delay learning.

Selecting features

Let's undertake preparatory work, taking into account the considerations mentioned above. Of course, we won't manually select and compare data, as we live in the era of computer technology. To calculate the correlation coefficient, let's create a small script in the file *initial_data.mq5*.

As a reminder, according to the previously described [directory structure](#), all scripts are saved in the folder *terminal_dir\MQL5\Scripts*. For the scripts in our book, we will create the *NeuroNetworksBook* subdirectory, and for all the scripts in this chapter, we will create the *initial_data* subdirectory. So the full path of the file to be created will be:

- *terminal_dir\MQL5\Scripts\NeuroNetworksBook\initial_data\initial_data.mq5*

We will directly calculate the correlation coefficient using the Mathematical Statistics Library from the MetaTrader 5 platform. In the script header, include the necessary library, and in the external parameters, specify the period for analysis.

```
#include <Math\Stat\Math.mqh>
//+-----+
//| Script Parameters |
//+-----+
input datetime Start = D'2015.01.01 00:00:00'; // Period Start
input datetime End   = D'2020.12.31 23:59:00'; // Period End
```

When you run a script, MetaTrader 5 generates a **Start** event that is handled by the **OnStart** function in the script body. At the beginning of this function, we get multiple indicator handles for further analysis.

Note that first on my list of indicators is ZigZag, which we will use to get reference values when training the neural network. Here, we will use it to check the correlation of indicator readings with reference values.

Indicator settings are defined by the user at the problem statement stage. The neural network will be trained on the M5 timeframe data, so I set the *Depth* parameter to 48, which corresponds to four hours. In this way, I expect that the indicator will reflect 4-hour extremes.

The selection of the indicators list and parameters is up to the neural network architect. Parameter tuning is also possible when assessing correlation, which we will explore a bit later. At this stage, let us specify the indicators and their parameters from our subjective considerations.

3. Building the first neural network model in MQL5

```
void OnStart(void)
{
    int h_ZZ=iCustom(_Symbol,PERIOD_M5,"Examples\\ZigZag.ex5",48,1,47);
    int h_CCI=iCCI(_Symbol,PERIOD_M5,12,PRICE_TYPICAL);
    int h_RSI=iRSI(_Symbol,PERIOD_M5,12,PRICE_TYPICAL);
    int h_Stoh=iStochastic(_Symbol,PERIOD_M5,12,8,3,MODE_LWMA,STO_LOWHIGH);
    int h_MACD=iMACD(_Symbol,PERIOD_M5,12,48,12,PRICE_TYPICAL);
    int h_ATR=iATR(_Symbol,PERIOD_M5,12);
    int h_BB=iBands(_Symbol,PERIOD_M5,48,0,3,PRICE_TYPICAL);
    int h_SAR=iSAR(_Symbol,PERIOD_M5,0.02,0.2);
    int h_MFI=iMFI(_Symbol,PERIOD_M5,12,VOLUME_TICK);
}
```

The next step is to load historical quotes and indicator data. To obtain historical data, we will create a series of arrays with names corresponding to the names of indicators and quotes. This will help us avoid confusion while working with them.

Price data in MQL5 can be obtained by **CopyOpen**, **CopyHigh**, **CopyLow**, and **CopyClose** functions. The functions are created according to the same template, and it is clear from the function name which quotes it returns. The **CopyBuffer** function is responsible for receiving data from indicator buffers. The function call is similar to the function for obtaining quotes, with the only difference being that the instrument's name and timeframe are replaced with the indicator handle and the buffer number. I'll remind you that we obtained the indicator handles a little earlier.

```
double close[], open[],high[],low[];
if(CopyClose(_Symbol,PERIOD_M5,Start,End,close)<=0 ||
   CopyOpen(_Symbol,PERIOD_M5,Start,End,open)<=0 ||
   CopyHigh(_Symbol,PERIOD_M5,Start,End,high)<=0 ||
   CopyLow(_Symbol,PERIOD_M5,Start,End,low)<=0)
    return;
```

All functions write data to the specified array and return the number of copied values. So, when making the call, we check for the presence of loaded data, and if there is no data, we exit the script. In this case, we'll give the terminal some time to load quotes from the server and recalculate indicator values. After that, we will rerun the script.

3. Building the first neural network model in MQL5

```
double zz[], cci[], macd_main[], macd_signal[], rsi[], atr[], bands_medium[];
double bands_up[], bands_low[], sar[], stoch[], ssig[], mfi[];
datetime end_zz=End+PeriodSeconds(PERIOD_M5)*(12*24*5);
if(CopyBuffer(h_ZZ,0,Start,end_zz,zz)<=0 ||
   CopyBuffer(h_CCI,0,Start,End,cci)<=0 ||
   CopyBuffer(h_RSI,0,Start,End,rsi)<=0 ||
   CopyBuffer(h_MACD,MAIN_LINE,Start,End,macd_main)<=0 ||
   CopyBuffer(h_MACD,SIGNAL_LINE,Start,End,macd_signal)<=0 ||
   CopyBuffer(h_ATR,0,Start,End,atr)<=0 ||
   CopyBuffer(h_BB,BASE_LINE,Start,End,bands_medium)<=0 ||
   CopyBuffer(h_BB,UPPER_BAND,Start,End,bands_up)<=0 ||
   CopyBuffer(h_BB,LOWER_BAND,Start,End,bands_low)<=0 ||
   CopyBuffer(h_SAR,0,Start,End,sar)<=0 ||
   CopyBuffer(h_Stoh,MAIN_LINE,Start,End,stoch)<=0 ||
   CopyBuffer(h_Stoh,SIGNAL_LINE,Start,End,ssig)<=0 ||
   CopyBuffer(h_MFI,0,Start,End,mfi)<=0)
{
    return;
}
```

As mentioned above, not all variables are comparable. Although linear transformations will not significantly affect the correlation coefficient, we still need to preprocess some values.

First, this applies to parameters that directly point to the instrument price. Since our goal is to create a tool capable of projecting accumulated knowledge onto future market situations, where there will be similar price movements but at a new price level, we need to move away from absolute price values and move toward a relative range.

So, instead of using the candlestick opening and closing prices, we can take their difference (the size of the candlestick body) as an indicator of price movement intensity. We will also replace the *High* and *Low* candlestick extremes with their deviation from the opening or closing price. We will treat SAR and Bollinger Bands indicators in the same way.

Remember the classic MACD trading rules. In addition to the actual indicator values, their position relative to the signal line within the histogram is also crucial. To test this relationship, let's add the difference between the indicator lines as another variable.

Now, let's address our reference point for the price movement. The ZigZag indicator gives absolute price values of extremes on a particular candlestick. However, we ideally want to know the price reference point for each market situation. In other words, we need a price guide for the upcoming movement on each candlestick. In doing so, we will consider two options for such a benchmark:

- Direction of movement (vector *target1*).
- Magnitude of movement (vector *target2*).

We can solve this task using a loop. We will iterate over the ZigZag indicator values in reverse order (from the newest values to the oldest values). If the indicator finds an extremum, we will save its value in a local variable **extremum**. If there is no extremum, we will use the last saved value.

Simultaneously, in the same loop, we will calculate and save the target values for our dataset. For this, we will subtract the closing price of the analyzed bar from the price value of the last peak. This way we get the magnitude of the movement to the nearest future extremum (target2 vector). The sign of this value will indicate the direction of movement (vector target1).

3. Building the first neural network model in MQL5

```
int total = ArraySize(close);
double target1[], target2[], oc[], bmc[], buc[], blc[], macd_delta[];
if(ArrayResize(target1, total) <= 0 || ArrayResize(target2, total) <= 0 ||
   ArrayResize(oc, total) <= 0 || ArrayResize(bmc, total) <= 0 ||
   ArrayResize(buc, total) <= 0 || ArrayResize(blc, total) <= 0 ||
   ArrayResize(macd_delta, total) <= 0)
    return;

double extremum = -1;
for(int i = ArraySize(zz) - 2; i >= 0; i--)
{
    if(zz[i + 1] > 0 && zz[i + 1] != EMPTY_VALUE)
        extremum = zz[i + 1];
    if(i >= total)
        continue;
    target2[i] = extremum - close[i];
    target1[i] = (target2[i] >= 0);
    oc[i] = close[i] - open[i];
    sar[i] -= close[i];
    bands_low[i] = close[i] - bands_low[i];
    bands_up[i] -= close[i];
    bands_medium[i] -= close[i];
    macd_delta[i] = macd_main[i] - macd_signal[i];
}
```

After completing the preparatory work, we will proceed to check the data correlation. Since we'll be performing the same operation for different indicator data, it makes sense to encapsulate this iteration in a separate function. From the body of the *Start* function, we will only make the function call, passing different source data to it. The results of the correlation analysis will be saved to a CSV file for further processing.

```
int handle = FileOpen("correlation.csv", FILE_WRITE | FILE_CSV | FILE_ANSI,
                    "\\t", CP_UTF8);

string message = "Indicator\\tTarget 1\\tTarget 2";
if(handle != INVALID_HANDLE)
    return;
FileWrite(handle, message);
//---
Correlation(target1, target2, oc, "Close - Open", handle);
Correlation(target1, target2, hc, "High - Close %.5f", handle);
Correlation(target1, target2, lc, "Close - Low", handle);
Correlation(target1, target2, cci, "CCI %.5f", handle);
Correlation(target1, target2, rsi, "RSI", handle);
Correlation(target1, target2, atr, "ATR", handle);
Correlation(target1, target2, sar, "SAR", handle);
Correlation(target1, target2, macd_main, "MACD Main", handle);
Correlation(target1, target2, macd_signal, "MACD Signal", handle);
Correlation(target1, target2, macd_delta, "MACD Main-Signal", handle);
```

3. Building the first neural network model in MQL5

```
Correlation(target1, target2, bands_medium, "BB Main", handle);
Correlation(target1, target2, bands_low, "BB Low", handle);
Correlation(target1, target2, bands_up, "BB Up", handle);
Correlation(target1, target2, stoch, "Stochastic Main", handle);
Correlation(target1, target2, ssig, "Stochastic Signal", handle);
Correlation(target1, target2, mfi, "MFI", handle);
//---
FileFlush(handle);
FileClose(handle);
}
```

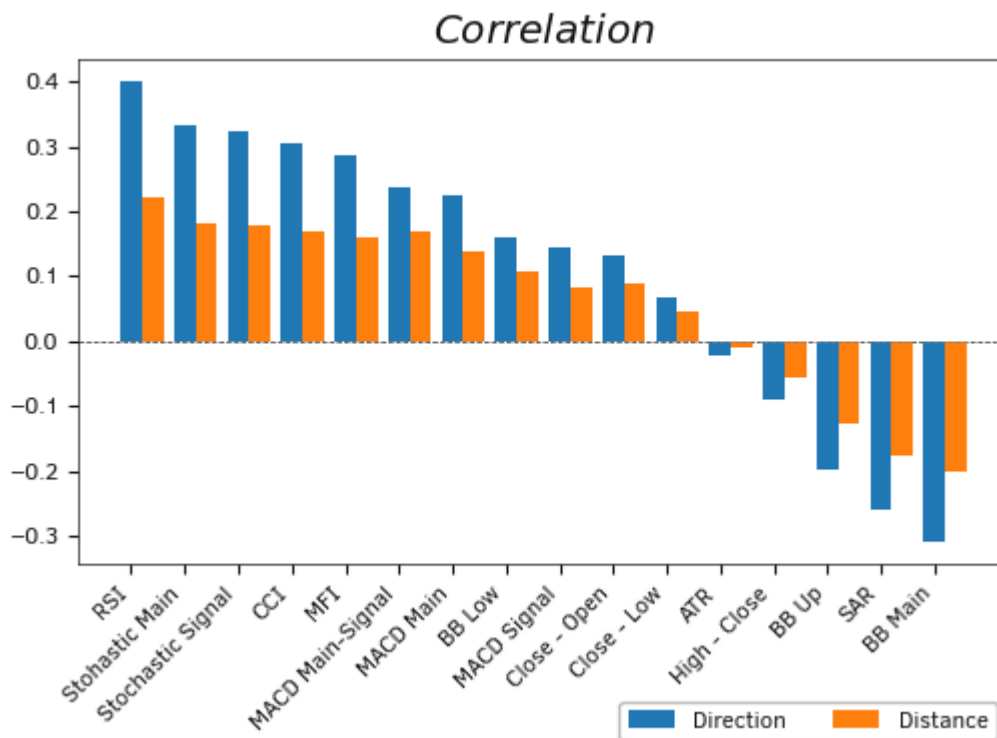
The algorithm of our correlation test method is pretty straightforward. The correlation coefficient is calculated using the **MathCorrelationPearson** function from the MQL5 standard statistical analysis library. We will call this function sequentially for two sets of data:

- Indicator and direction of the upcoming movement; and
- Indicator and strength of the upcoming movement.

The results of the analysis are used to form a text message, which is then written to a local file.

```
void Correlation(double &target1[], double &target2[],
                double &indicator[], string name,
                int handle)
{
//---
double correlation=0;
string message="";
if(MathCorrelationPearson(target1,indicator,correlation))
    message=StringFormat("%s\t%.5f",name,correlation);
if(MathCorrelationPearson(target2,indicator,correlation))
    message=StringFormat("%s\t%.5f",message,correlation);
if(handle!=INVALID_HANDLE)
    FileWrite(handle,message);
}
```

The results of my analysis are presented in the graph below. The data show that there is no correlation between our target data and the ATR indicator values. The deviation from the extremes of the candlestick to its closing price (**High – Close**, **Close – Low**) also shows a low correlation with the expected price movement. Consequently, we can safely exclude these figures from our further analysis.



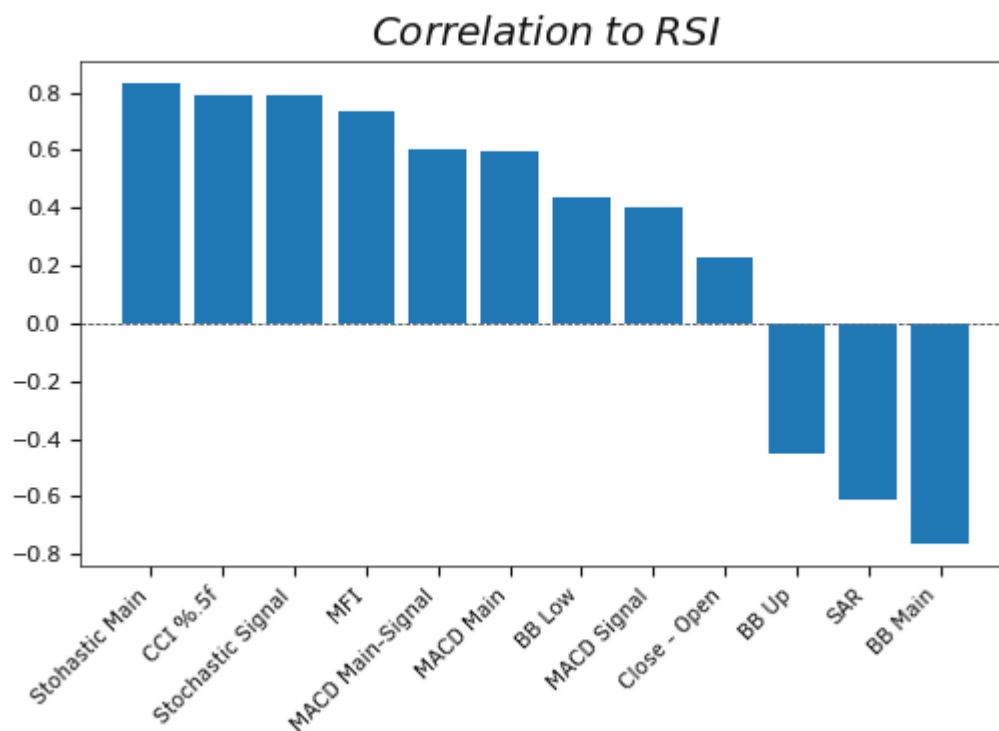
Correlation of indicator values with expected price movements

In general, the conducted analysis shows that determining the direction of the upcoming movement is much easier than predicting its strength. All indicators showed a higher correlation with the direction rather than the magnitude of the upcoming movement. However, the correlation with all values remains rather low. The RSI indicator demonstrated the highest correlation, with a value of 0.40 for the direction and 0.22 for the magnitude of the movement.

The correlation coefficient takes values from -1 (inverse relationship) to 1 (direct relationship), with 0 indicating a complete absence of dependence between random variables.

It's worth noting that among the three arrays of data obtained from the MACD indicator (histogram, signal line, and the difference between them), it's the distance between the MACD lines that demonstrated the highest correlation with the target data. This only confirms the validity of the classical approach to using indicator signals.

The next step is to test the correlation between data from different indicators. To avoid comparing each indicator with all others, we will analyze the correlation of indicators with RSI (the winner of the previous stage). We will perform the task using the previously created script with minor modifications. The new script will be saved in the file *initial_data_rsi.mq5* in our [subdirectory](#).



Correlation of indicator values to RSI

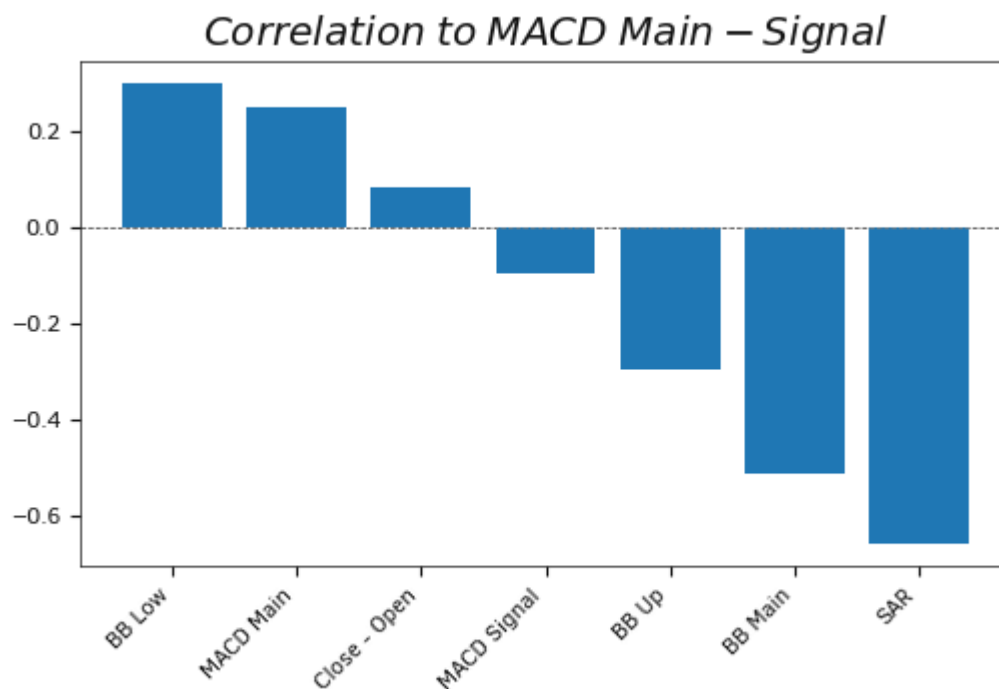
The analysis showed a strong correlation of RSI with a range of indicators. Stochastic, CCI, and MFI have a correlation coefficient with RSI which is greater than 0.70, while the main line of Bollinger Bands showed an inverse correlation of -0.76 with RSI. This indicates that the indicators mentioned above will only duplicate signals. Including them for analysis in our neural network will only complicate its architecture and maintenance. The expected impact of their use will be minimal. Therefore, we are excluding the aforementioned indicators from further analysis.

The indicators that show the minimum correlation with *RSI* are the two deviation variables:

- MACD signal line (0.40);
- Between opening and closing prices (0.23).

The deviation of the MACD signal line from the histogram in the first step showed a strong correlation with the target data of the upcoming price movement. Based on this data, it is MACD that will be taken into our indicator basket. Next, we will check its correlation with the remaining indicators.

The updated script is saved in the file *initial_data_macd.mq5* in the [subdirectory](#).



Correlation of indicator values to MACD Main-Signal

The SAR indicator here shows some interesting data. With moderate levels of inverse correlation to the target data, it shows a relatively high negative correlation with both selected indicators. The correlation coefficient with MACD was -0.66 and for RSI it was -0.62. This gives us reason to exclude the SAR indicator from the basket of analyzed indicators.

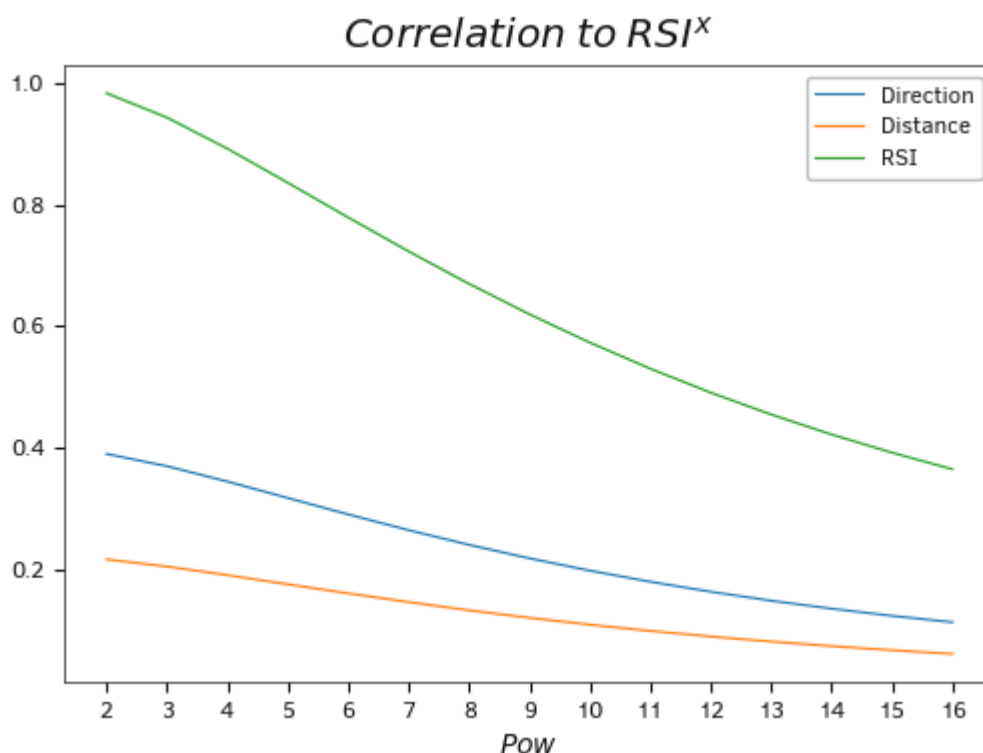
A similar situation is observed for all three Bollinger Bands indicator lines.

So far, we have selected two indicators for our indicator basket for further training of the neural network.

But this is not the end of the road to initial data selection. It should be noted that the neural network analyzes linear dependencies between the target values and the initial data in their pure form. So, it analyzes the data that is input into it. Each indicator is analyzed in isolation from other data available on the neuron input layer.

Hence, the neural network won't be able to capture the relationship between the source data and target values if it's not a linear relationship, such as being power law or logarithmic instead. To find such relationships, we need to prepare data beforehand. And to test the usefulness of such work, we need to test the correlation between such values.

In the script *initial_data_rsi_pow.mq5*, we will analyze the change in correlation with the expected price movement when the RSI indicator values are set to different degrees. The new script can be saved in the appropriate [subdirectory](#).



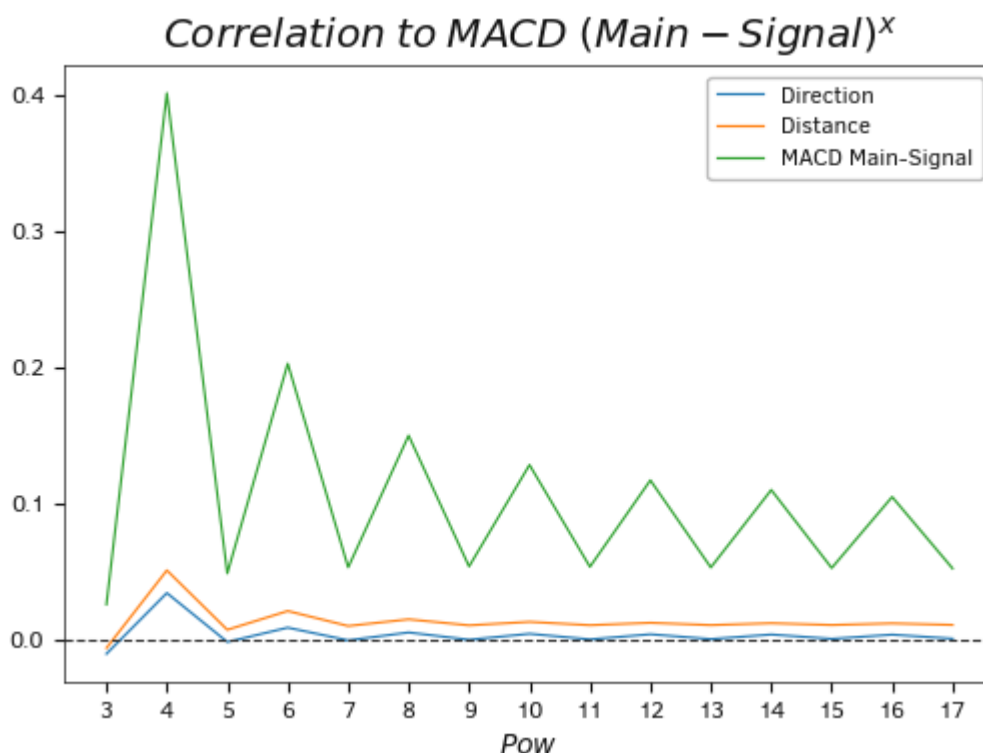
The dynamics of how the correlation of RSI values changes in relation to the expected movement when raising the indicator to a power.

The presented graph clearly shows that as the exponentiation of the indicator values increases, the correlation with the original values decreases much faster than with the expected price movement. This observation gives us a potential opportunity to expand the basket of original data indicators with exponential values of the selected indicators. A little later, we will be able to see how it works in practice.

It's important to note that when using exponential operations on indicators, you need to consider the properties of exponents and the nature of indicator values. Indeed, when raising any number to an even exponent, the result will always be positive. In other words, we lose the sign of the number.

For example, if we have two equal-magnitude values of a certain indicator with opposite signs, and they correlate with the target data. If the indicator is positive, the target function grows and if it is negative, it falls. Squared values for such an indicator will give us the same value. In this case, we will observe a decrease in correlation or a complete absence of it, as our target function remains unchanged while the indicator loses its sign.

This effect is noticeable when analyzing the change in correlation when raising the power of the difference between the lines of the MACD indicator, which we conducted in the script `initial_data_macd_pow.mq5` from our [subdirectory](#).



Dynamics of changing the correlation of MACD values to expected movement when raising the indicator to the power

Similarly, you can test the correlation of various indicators with the target values. The only limit is your ability and common sense. In addition to standard indicators and price quotes, these could also include custom indicators, and quotes from other instruments, including synthetic ones. Don't be afraid to experiment. Sometimes you can find good indicators in the most unexpected places.

Effect of the time shift on the correlation coefficient

After selecting the indicators for analysis, let's remember that we are dealing with time series data. One of the features of time series is their "historical memory". Each subsequent value depends not only on one previous value but also on a certain depth of historical values.

Certainly, one approach could be experimental – building a neural network and conducting a series of experiments to identify the optimal configuration. In this approach, it will take time to create and train several experimental models. Optionally, we could use our sample to test the correlation of the data with the historical shift.

To solve this problem, let's modify the script slightly and replace the *Correlation* function with **ShiftCorrelation**. The new function is a complete descendant of the *Correlation* function and is built using the same algorithm.

In the function parameters, we add a new variable *max_shift* to which we pass the maximum shift for analysis.

Time offsets are organized by copying the data to the new shifted arrays. The initial data will be copied without offset but to a lesser extent. Data reduction corresponds to time offset. At the same time, we will transfer the target values data to new arrays with offsets. However, since the size of the target data in our dataset is constant, when shifting, the number of elements for correlation analysis

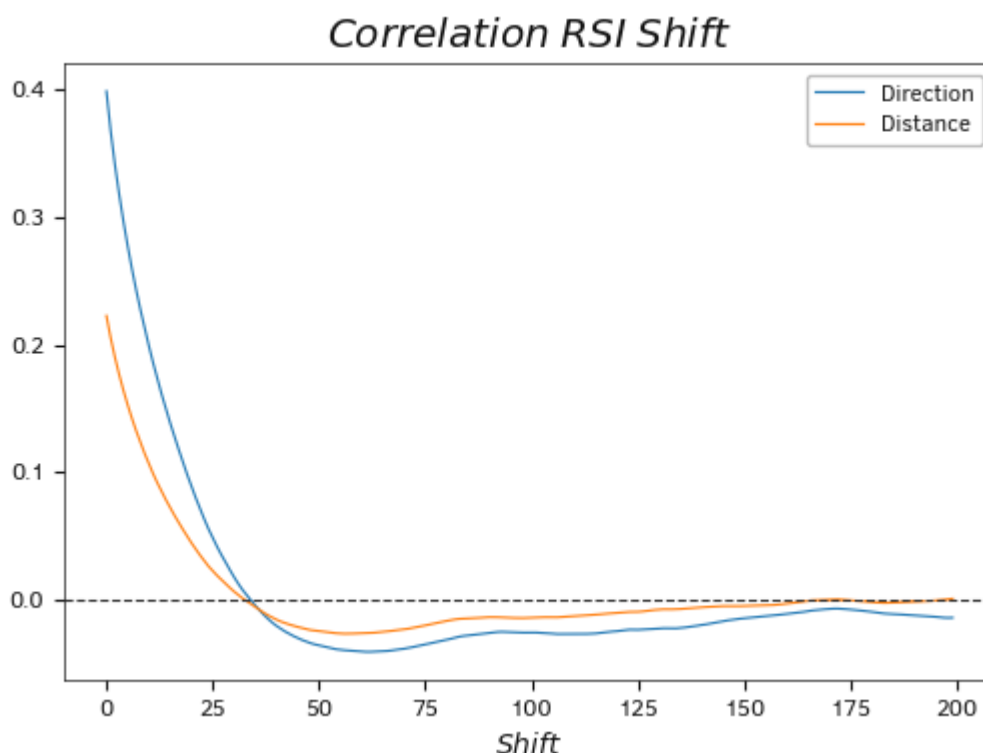
decreases. Therefore, after copying the data, we get data sets that are comparable in size and with a given time shift.

All we need to do is call the correlation coefficient calculation function and write the resulting data to a file.

To analyze the change in correlation with the increase in displacement over time, wrap all operations in a loop. The number of looping iterations corresponds to the *max_shift* parameter.

```
void ShiftCorrelation(double &targ1[], double &targ2[],
                    double &signal[], string name,
                    int max_shift, int handle)
{
    int total = ArraySize(targ1);
    if(max_shift > total)
        max_shift = total - 10;
    if(max_shift < 10)
        return;
    double correlation = 0;
    for(int i = 0; i < max_shift; i++)
    {
        double t1[], t2[], s[];
        if(ArrayCopy(t1, targ1, 0, i, total - i) <= 0 ||
           ArrayCopy(t2, targ2, 0, i, total - i) <= 0 ||
           ArrayCopy(s, signal, 0, 0, total - i) <= 0)
        {
            continue;
        }
        //---

        string message;
        if(MathCorrelationPearson(s, t1, correlation))
            message = StringFormat("%d\t%.5f", i, correlation);
        if(MathCorrelationPearson(s, t2, correlation))
            message = StringFormat("%s\t%.5f", message, correlation);
        if(handle != INVALID_HANDLE)
            FileWrite(handle, message);
    }
    //---
    return;
}
```



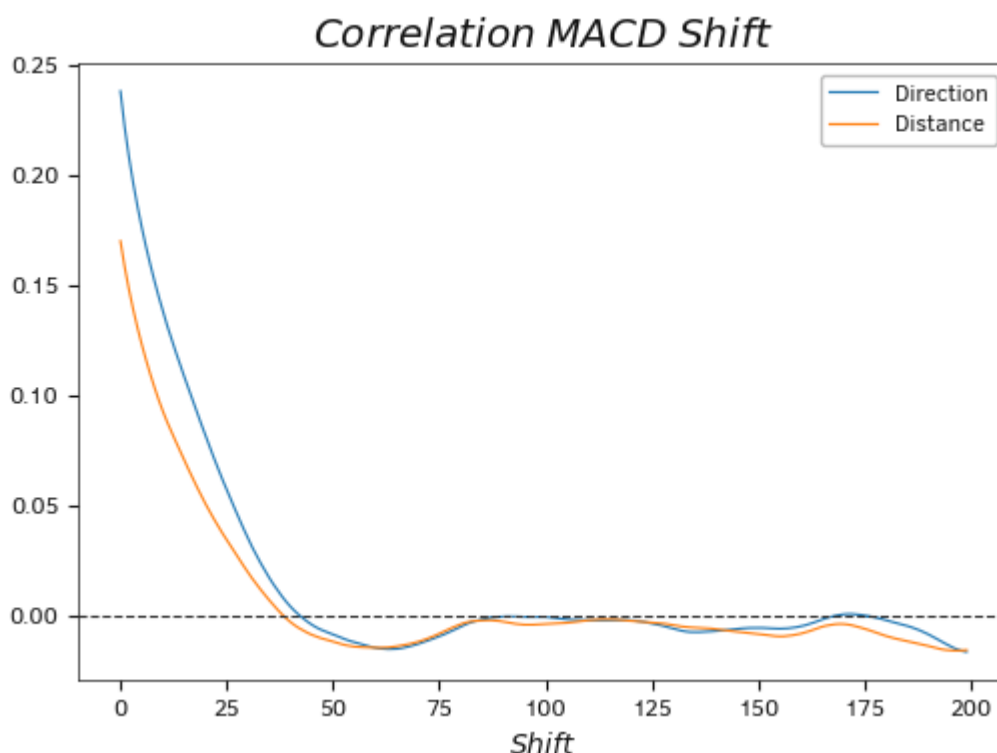
Dynamics of the correlation between RSI values and the expected movement, with time shift

To analyze the effect of shifting RSI indicator values over time on the correlation with target data, we will create a new script in the file `initial_data_rsi_shift.mq5` in the specified *initial_data_rsi_shift.mq5* subdirectory.

The results of the analysis show a rapid decline in the correlation up to the 30th bar. Then, a slight inverse correlation with a peak coefficient of -0.042 is observed around the 60th bar, followed by a gradual approach to 0. In such a situation, the use of the first 30 bars would be most effective. Further expansion of the analysis depth may lead to a decrease in the efficiency of utilizing computational resources. The value of such a solution can be tested in practice.

A similar analysis of MACD indicator data in the script `initial_data_macd_shift.mq5` showed a similar dynamic with a slight shift in the transition zone from direct to inverse correlation at around the 40th bar.

Thus, conducting a correlation analysis of available source data and target values allows us to choose the optimal set of indicators and historical depth during the preparatory phase. This helps in analyzing data and forecasting target values more effectively. This enables us to significantly reduce expenses during the neural network creation and training phases with relatively low effort spent on the preparatory stage.



Dynamics of the correlation between MACD values and the expected movement, with time shift

3.4 Creating the framework for the future MQL5 program

In the previous section, we talked about preparatory work and methods for selecting indicators for analysis by a neural network. After conducting the analysis, we have determined a set of indicators to train the neural network and the depth of historical data to be loaded.

Now let's move on to the practical part of our book. We will look at various neural network algorithms and architectures. You will learn the specifics of constructing and implementing a fully connected perceptron, convolutional neural networks, and recurrent neural networks. After that, we will discuss the features and advantages of attention mechanisms. Finally, we will look at the GPT architecture, which, at the time of writing the book, demonstrates the best results in natural language processing problems.

As we explore the algorithms, step by step, we will create a tool for designing and organizing neural networks using MQL5. Each algorithm under consideration will be implemented in three versions: MQL5, OpenCL, and Python.

We will build and train neural networks using all the learned algorithms. Then, we will practically assess the strengths and weaknesses of their use for time series forecasting. We will train and test the built models on real data. And, of course, during the training process, we will discuss the nuances of this process.

The book will showcase the practical results of using neural networks to solve the problem defined in the previous sections, using real-world data. During testing, we will conduct a comparative analysis of various implementations and evaluate the practical effectiveness of each implementation in solving the given problem.

Let's begin working on refining the architecture of our future tool. It's quite logical to consolidate our entire development into a single entity (class) that can be easily integrated into any program. This way, we'll be able to configure the entire operation of our model within this class.

At the same time, we need to ensure the ability to create models of various architectures within our main model. The architecture of the model itself will be defined in the main program and passed to the class through created interfaces. To make this process convenient and easy to use, it is necessary to standardize it. To address standardization matters, we will use constants and named enumerations.

3.4.1 Defining constants and enumerations

The process of defining constants is one of those basic processes that is often overlooked. Moreover, it enables the organization and systematization of future work on creating a software product. Particular attention should be given to it when creating complex, structured products with a multi-block branched architecture.

Here, we won't discuss specific local variables and constants, as their scope will often be determined by separate blocks or functions. We will discuss creating constants that will serve as a common thread throughout our program and will frequently be used for organizing interactions both between blocks within our product and for data exchange with external programs.

Starting a large project by creating constants and enumerations is a very useful practice. Here, we can also include the creation of global variables. Primarily, this is one of the integral parts of developing project architecture. When contemplating the list of global constants and enumerations, we are re-evaluating our project as a whole, reconsidering its objectives and the means to achieve them. Even in broad strokes, we conceptualize the project structure and define the tasks of each block and the flow of information between them. We also understand what information needs to be obtained from an external program, what information needs to be returned, and at which stage of the process.

The work done at this stage will be our roadmap when creating the project. A detailed examination of data exchange interface organization allows us to assess the necessity of having specific information at each stage. This also provides the opportunity to identify the sources of information and uncover potential data deficits. Eliminating data deficits during the design stage will be much easier than during the implementation phase. At that point, we would have to revisit the design stage to search for the necessary data sources. Next, it will be necessary to consider possible ways of transmitting information from the source to the processing location and attempt to seamlessly integrate them into the established architecture with minimal adjustments. This will lead to an unpredictable number of revisions in the already established processes, and it will be necessary to assess the impact of these revisions on adjacent processes.

We will collect all the files of the library to be built in the *NeuroNetworksBook\realization* subdirectory according to the [file structure](#).

All global constants of our project will be collected in one file, *defines.mqh*.

So what constants are we going to define?

Let's take a look at the architecture of the project. As we've discussed, the result of our work will be a class that encompasses the complete organization of a neural network's operation. In the MQL5 architecture, all objects are inherited from the base class *CObject*. It includes the virtual *Type* method which is defined for class identification and which returns an integer value. Consequently, for a unique identification of our class, we should define a certain constant, preferably distinct from the constants of

3. Building the first neural network model in MQL5

existing classes. This will serve as a prototype for the business card of our class within the program. To create named constants, we will utilize the mechanism of macro substitution.

```
#define defNeuronNet 0x8000
```

Next, our neural network will consist of neurons. Neurons are organized into layers, and a neural network may consist of multiple layers. Since we are constructing a universal constructor, at this stage, we don't know the number of layers in the neural network or the number of neurons in each layer. Therefore, we assume that there will be a dynamic array for storing pointers to neuron layers. Most likely, in addition to simple storage of pointers to neural layer objects, we will need to create additional methods for working with them. Based on these considerations, we will create a separate class for such storage. Consequently, we will also create a business card for it.

```
#define defArrayLayers 0x8001
```

Next in the structure, we will create a separate class for the neural layer. Later, when we approach the implementation of computation algorithms using the OpenCL technology, we will discuss the organization of vector computations and the means of transferring data to the GPU memory. In this context, creating classes for each individual neuron might not be very convenient, but we will need a class for storing information and organizing data exchange buffering. Thus, we must create "business cards" for these objects as well.

It should be noted that the book will explore several architectural solutions for organizing neurons. Each architecture has its own peculiarities in terms of forward and backward propagation algorithms. However, we have already decided that we will not create distinct objects for neurons. So, we need to introduce identification at the level of neural layers. Therefore, we will create separate identifiers for each architecture of the neural layer.

```
#define defBuffer 0x8002
#define defActivation 0x8003
#define defLayerDescription 0x8004
#define defNeuronBase 0x8010
#define defNeuronConv 0x8011
#define defNeuronProof 0x8012
#define defNeuronLSTM 0x8013
#define defNeuronAttention 0x8014
#define defNeuronMHAttention 0x8015
#define defNeuronGPT 0x8016
#define defNeuronDropout 0x8017
#define defNeuronBatchNorm 0x8018
```

We have defined constants and object identifiers and can move further. Let's recall what this book starts with. At the very beginning of the book, we considered a mathematical model of a neuron. Each neuron has an [activation function](#). We've seen several options for activation functions, and all of them are valid choices. Due to the absence of a derivative, we'll exclude the threshold function from the list. However, we'll implement the remaining discussed activation functions using the OpenCL technology. In the case of working with the CPU, we will use vector operations in which activation functions are already implemented. To maintain consistency in approaches and to indicate the used activation function, we use the standard enumeration [ENUM_ACTIVATION_FUNCTION](#).

However, it's worth noting that later, when discussing convolutional neural network algorithms, we will become familiar with the organization of a pooling layer. It utilizes other functions.

3. Building the first neural network model in MQL5

```
//--- pooling layer activation functions
enum ENUM_PROOF
{
    AF_MAX_POOLING,
    AF_AVERAGE_POOLING
};
```

Take a look at the chapter [Training a neural network](#). In it, we discussed various options for loss functions and optimization methods for neural networks. In my understanding, we should provide the user with the ability to choose what they want to use. However, we need to restrict the choices to the capabilities of our library. For a loss function, we can use the standard enumeration [ENUM_LOSS_FUNCTION](#) by analogy with the activation function. For model optimization methods, we will create a new enumeration.

As you can observe, in the enumeration of optimization methods, I added the *None* element to allow the option of disabling training for a specific layer. Such an approach is often utilized when using a pre-trained network on new data. For instance, we might have a trained and functioning neural network that works well on one financial instrument, and we would like to replicate it for other instruments or timeframes. In all likelihood, without retraining, its performance will drop dramatically.

```
enum ENUM_OPTIMIZATION
{
    None=-1,
    SGD,
    MOMENTUM,
    AdaGrad,
    RMSProp,
    AdaDelta,
    Adam
};
```

In this case, we have a choice: to train the neural network from scratch or to retrain the existing network. The second option usually requires less time and resources. However, to avoid disrupting the entire network, the retraining process starts with a low learning rate and focuses on the final layers (decision-making neurons), while leaving the initial analytical layers untrained.

Along with the learning methods, we discussed [techniques for improving the convergence](#) of neural networks. In this regard, normalization and dropout will be organized as separate layers – for them, we have already defined constants when discussing neural layers. We will implement one regularization – *Elastic Net*. The process will be controlled through the variables λ_1 and λ_2 . If both variables are zero, regularization is disabled. In the case where one of the parameters is equal to zero, we will obtain L1 or L2 regularization, depending on the non-zero parameter.

Have you noticed that in this chapter we have refreshed our memories of the major milestones of the material we have studied? In addition, behind each constant or enumeration element, there is a specific functionality that we still need to implement.

But I'd like to add one more point. When introducing the OpenCL technology, we discussed that not all OpenCL-enabled devices work with the *double* type. It would probably be foolish to create copies of the library for different data types.

Here it's important to understand that different data types provide different levels of precision for computations. Therefore, when creating a model, it's important to ensure consistent conditions for all

scenarios of model operation, both with and without using the OpenCL technology. To address this issue, we will introduce data type macros along with corresponding types for vectors and matrices.

```
#define TYPE double
#define MATRIX matrix<TYPE>
#define VECTOR vector<TYPE>
```

We organize a similar macro substitution for an OpenCL program.

```
#resource "opencl_program.cl" as string OCLprogram
//---
#define LOCAL_SIZE 256
const string ExtType=StringFormat("#define TYPE %s\r\n"
                                   "#define TYPE4 %s4\r\n"
                                   "#define LOCAL_SIZE %d\r\n",
                                   typename(TYPE), typename(TYPE), LOCAL_SIZE);

#define cl_program ExtType+OCLprogram
```

Here we can also add to the models various hyperparameters. For example, it could be a learning rate. You can also add parameters for optimization and regularization methods.

```
#define defLossSmoothFactor 1000
#define defLearningRate (TYPE)3.0e-4
#define defBeta1 (TYPE)0.9
#define defBeta2 (TYPE)0.999
#define defLambdaL1 (TYPE)0
#define defLambdaL2 (TYPE)0
```

However, it's important to keep in mind that the hyperparameter values mentioned here are just default values. During the operation of the model, we will use variables that will be initialized with these values when the model is created. However, the user has the right to specify different values without changing the library code. We will discuss the mechanism of such a process when constructing classes and their methods.

3.4.2 Mechanism for describing the structure of the future neural network

We have already decided that we will build a universal constructor for the convenient creation of neural networks of various configurations. Hence, we need some mechanism (interface) to be able to pass the model configuration to be built. Let's think about what information we need to get from the user to unambiguously understand what kind of neural network is supposed to be created.

First of all, we need to understand how many layers of neurons our network will have. There should be at least two such layers: an input layer with initial data and an output layer with results. Additionally, the new neural network may include a varying quantity of hidden layers. Their quantity may vary, and we will not limit them now.

To create each layer of the neural network, we need to know the number of neurons in that layer. Hence, in addition to the number of neural layers, the user must specify the number of neurons in each layer.

Now let's recall that in the previous section, we defined constants for several types of neural layers, which will differ by the type of neurons. To understand what kind of layer the user wants to create, you

3. Building the first neural network model in MQL5

need to get that initial information. So, the user should be able to specify it for each layer that is created.

In addition, we considered different variants of activation functions. Which one should be used when creating neurons?

When creating a universal tool, we must provide the user with the option to choose the activation function. Hence, we add the activation function to the list of parameters that the user should specify.

Then there is another question: will all neurons in the same layer use the same activation function? Or will there be options to use different activation features within a single layer? I propose to focus on the first option, where all neurons of one layer use one activation function.

Let me explain my point. While discussing techniques to improve the convergence of neural networks and, in particular, data [normalization](#), we talked about the importance of data comparability at the input of the neural layer. The use of different activation functions, on the other hand, is highly likely to lead to data imbalance. This is due to the nature of the activation functions themselves. Remember, the sigmoid returns data in the range from 0 to 1. The value range of the hyperbolic tangent lies in the range from -1 to 1. ReLU can return values from 0 to $+\infty$. Evidently, different activation functions will produce significantly different values and only complicate the training and operation of the neural network.

Additionally, from a technical perspective, there are also advantages to using one activation function for the entire neural layer. In this case, we can then limit ourselves to a single integer value to store the activation code of neurons in a layer regardless of the number of neurons. To store individual activation functions, we would have had to create a whole vector of values the size of the number of neurons in the layer.

The next thing we need to know when creating the architecture of a neural network is the weight optimization method. In the chapter [Neural network optimization methods](#), we covered six optimization methods. In the previous chapter, we set up an enumeration to identify them. Now you can take advantage of this enumeration and let the user choose one of them.

Why is it important for us to know the optimization method now, at the stage of creating the neural network, rather than during its training? It's very simple. Different optimization methods require different amounts of objects to store information, so when creating a neural network, it is necessary to create all the required objects. Given that we have memory constraints on our computing machine, we need to use it rationally and not create unnecessary objects.

When creating layers such as normalization and *Dropout*, we will need some specific information. For normalization, we need the normalization sample size (batch), and for *Dropout*, we need to specify the probability of "dropping out" neurons during training.

Looking ahead, for some types of neural layers, we will still need the size of the input and output window, as well as the step size from the beginning of one input window to the beginning of the next window.

To make it easier for the user to create consecutively identical layers, let's add another parameter to specify such a sequence.

As a result, we have accumulated a dozen parameters that the user needs to specify for each layer. Let's add to this the total number of layers to create in a neural network. These are all things we want to get from the user before creating the neural network. We will not overly complicate the data transfer

3. Building the first neural network model in MQL5

process, and to describe one neural layer, we will create a class named ***CLayerDescription*** with elements to store the specified parameters.

```
class CLayerDescription      : public CObject
{
public:
    CLayerDescription(void);
    ~CLayerDescription(void) {};

    //---
    int         type;           // Type of neural layer
    int         count;          // Number of neurons in a layer
    int         window;         // Source data window size
    int         window_out;     // Results window size
    int         step;           // Input data window step
    int         layers;         // Number of neural layers
    int         batch;          // Weight Matrix Update Packet Size
    ENUM_ACTIVATION_FUNCTION activation; // Activation function type
    VECTOR       activation_params[2]; // Array of activation function parameters
    ENUM_OPTIMIZATION optimization; // Weight matrix optimization type
    TYPE         probability;    // masking probability, Dropout only
};
```

Note that the created class is inherited from the *CObject* class, which is the base class for all objects in MQL5. It's a small point that we'll exploit a little later.

We will not complicate the class constructor in any way, but only set some default values. You can use any of your values here. I recommend, however, that you specify the most commonly used parameters. This will make it easier for you to specify them later in the program code.

```
CLayerDescription::CLayerDescription(void)    : type(defNeuronBase),
                                                count(100),
                                                window(100),
                                                step(100),
                                                layers(1),
                                                activation(AF_TANH),
                                                optimization(Adam),
                                                probability(0.1),
                                                batch(100)
{
    activation_params = VECTOR::Ones(2);
    activation_params[1] = 0;
}
```

Now let's get back to why it was important to inherit from *CObject*. Here everything is quite straightforward: we have created an object to describe one neural layer but not the whole neural network. We have not yet specified the total number of layers and their sequence.

I decided not to complicate the process and use the *CArrayObj* class from the standard MQL5 library. This is a dynamic array class for storing pointers to *CObject* objects and their successors. Hence, we can write our neural layer description objects into it. In this way, we address the issue of a container for storing and transmitting information about neural networks. The sequence of neural layers will correspond to the sequence of stored descriptions from the zero-index input layer to the output layer.

In my opinion, this is a rather simple and intuitive way to describe the structure of a neural network. But every reader can make use of their own developments.

3.4.3 Neural network base class and organization of forward and backward pass processes

We have already done preparatory work to create constants and an interface for transferring the architecture of the created neural network. Let's continue. Now I propose to move on to creating a top-level class **CNet**, which will act as the manager of our neural network.

To do this work, we will create a new included library file *neuronnet.mqh* in a [subdirectory](#) of our library. In it, we will collect all the code of our **CNet** neural network class. Next, we will create a separate file for each new class. File names will correspond to the names of the classes – this will allow for structuring the project and quickly accessing the code of a specific class.

We won't be able to write the complete code for the methods of this class right now, as during their implementation we will need to refer to the neural layer classes and their methods. There are currently no such classes. Why have I decided to start by creating the top-level object instead of creating the lower-level objects first? Here, I am addressing the issue of the integrity of the structure and the standardization of methods and data transfer interfaces between individual blocks of our neural network.

Later, when examining the architectural features of the neural layers, you will be able to notice differences in their functionality and, to some extent, in the information flow. When solving the problem from the bottom up, we run the risk of obtaining quite different methods and interfaces, which will then be difficult to integrate into a unified system. On the contrary, I want to create a top-level "skeleton" of our development right from the beginning, and later fill it with functionality. By early planning the architecture and functionality of the interfaces, we will simply integrate new neural layer architectures into the already established information flow.

Let's define the functionality of the **CNet** class. The first thing this class should do is directly assemble the neural network with the architecture provided by the user. This can be done in the class constructor, or you can create a separate method, **Create**. I picked the second option. Using the base class constructor without parameters will allow us to create an "empty" class instance, for example, to load a previously trained neural network. It will also make it easier to inherit the class for possible future development.

Since we have started on the issue of loading a pre-trained network, the following class functionality follows from here: saving (**Save**) and loading (**Load**) our model.

Whether it is newly created (generated) neural layers or loaded from a file, we will need to store them and work with them. When elaborating and defining constants, we allocated a separate constant for the dynamic array storing the neural layers. We will add an instance of this object to the class variables (**m_cLayers**).

Let's take a look at how the work of the neural network is organized. Here we need to implement feed-forward pass (**FeedForward**) and backpropagation pass (**Backpropagation**) algorithms. Let's display the process of updating the weights **UpdateWeights** as a separate method.

Of course, you can update the weights in the backpropagation method, which is what is most commonly encountered in practice. But we're talking about a universal constructor. At the time of writing the code, we don't know if batch normalization (batch size) will be used. Therefore, there is no clear understanding at what point it will be necessary to update the weights.

3. Building the first neural network model in MQL5

A complex problem is always easier to solve step by step. Dividing a process into smaller subprocesses makes it easier to both write code and debug it. Therefore, I decided to separate the process of updating the weights.

Let's recall the [neuron optimization](#) methods. Almost all methods use a learning rate, and some require additional parameters, such as decay coefficients. We also need to allow the user to specify them. In this case, the user specifies once, and we will need them at each iteration. So we need to store them somewhere. Let's add a method for specifying learning parameters (***SetLearningRates***) and variables for storing data (***m_dLearningRate*** and ***m_adBeta***). For the decay coefficients, we will create a vector of two elements, which, in my opinion, will make the code more readable.

In the process of practical use of a neural network, the user may need to obtain the results of processing of the same source data several times. This option should be possible. However, in order not to make a direct pass every time, we will output the possibility of obtaining the results of the last direct pass using a separate ***GetResults*** method.

In addition, in the process of training and operating the neural network, we will need to control the process of accuracy and correctness of the forward pass data. The main indicator of the neural network's correct operation is the value of the loss function. The actual calculation of the loss function will be carried out in the ***Backpropagation*** method. The calculated value of the loss function will be stored in the ***m_dNNLoss*** variable. Let's add the ***GetRecentAverageLoss*** method to display the variable value at the user's request.

Now, speaking of the loss function. A specific loss function should be selected by the user. Therefore, we need a method to be able to get it from the user (***LossFunction***). The actual calculation of the value of the loss function will be carried out by standard means of matrix operations in MQL5. Here we will create a variable to store the type of the loss function (***m_eLossFunction***).

When defining constants, we didn't create a separate enumeration for regularization methods. Then we agreed to implement *Elastic Net* and manage the process through regularization coefficients. I suggest adding the specification of regularization coefficients to the loss function method. After all, look at how the number of class methods grows. Therefore, the question is not only in the implementation of our constructor. On the contrary, when building the constructor, all possible usage scenarios should be anticipated. This will help make it more flexible.

At the same time, the actual use of such a constructor should be as easy and intuitive as possible. In other words, we should provide the user with an interface that allows for the most flexible configuration of a new neural network with the minimum number of iterations required from the user.

Note that the algorithm of the normalization layers and *Dropout* differ depending on the mode of use (training or operation). Of course, this could have been done as a separate parameter in the forward and backward pass methods, but it's important to have a clear correspondence between the operations of the forward and backward passes. Performing a backward pass in training mode after a working forward pass and vice versa can only destabilize the neural network. Therefore, to avoid overloading the aforementioned methods with additional checks, we'll create separate functions to set and query the ***TrainMode*** operating mode.

There's another aspect regarding the operating mode of the neural network, specifically, the choice of tool for conducting computational operations. We have already discussed the topic of using [OpenCL](#) technology for parallel computing. This will allow parallel computation of mathematical operations on the GPU and speed up calculations during the operation of the neural network. The standard MQL5 library *OpenCL.mqh* provides the ***COpenCL*** class for working with OpenCL.

In the process of working with this class, I decided to slightly supplement its functionality, for which I created a new class **CMyOpenCL** that inherits the standard **COpenCL** class. Inheritance allowed me to write the code for just a couple of methods while still utilizing the full power of the parent class.

To use the **CMyOpenCL** class, add a pointer to an instance of the **m_cOpenCL** class. We will also add the **m_bOpenCL** flag, which will inform you if the functionality is enabled in our neural network. We will also add methods for initializing the functionality and managing it (**InitOpenCL**, **UseOpenCL**).

Let's not forget that we plan to use neural networks to work with timeseries. This leaves a certain imprint on their work. Do you remember the time-shift correlation score plot of the [initial data](#)? As the time lag increases, the impact of the indicator on the target result decreases. This once again confirms the importance of taking into account the position of the analyzed indicator on the timeline. Therefore, it will be necessary to implement such a mechanism.

We will talk about the method itself a little later. For now, let's create an instance of the **CPositionEncoder** class to implement positional encoding. We will also create a flag for controlling the activity of the function and declare methods for managing the function.

Let's add another class identification method to our list and get the following **CNet** class structure.

```
class CNet : public CObject
{
protected:
    bool                m_bTrainMode;
    CArrayLayers*       m_cLayers;
    CMyOpenCL*         m_cOpenCL;
    bool                m_bOpenCL;
    TYPE                m_dNNLoss;
    int                 m_iLossSmoothFactor;
    CPositionEncoder*   m_cPositionEncoder;
    bool                m_bPositionEncoder;
    ENUM_LOSS_FUNCTION  m_eLossFunction;
    VECTOR              m_adLambda;
    TYPE                m_dLearningRate;
    VECTOR              m_adBeta;

public:
    CNet(void);
    ~CNet(void);

    //--- Methods for creating an object
    bool                Create(.....);
    //--- Organization of work with OpenCL
    void                UseOpenCL(bool value);
    bool                UseOpenCL(void) const { return(m_bOpenCL); }
    bool                InitOpenCL(void);
}
```

3. Building the first neural network model in MQL5

```
//--- Methods of working with positional coding
void          UsePositionEncoder(bool value);
bool          UsePositionEncoder(void) const { return(m_bPositionEncoder); }
//--- Organization of the basic algorithms of the model
bool          FeedForward(.....);
bool          Backpropagation(.....);
bool          UpdateWeights(.....);
bool          GetResults(.....);
void          SetLearningRates(TYPE learning_rate, TYPE beta1 = defBeta1,
                                TYPE beta2 = defBeta2);

//--- Methods of the loss function
bool          LossFunction(ENUM_LOSS_FUNCTION loss_function,
                           TYPE lambda1 = defLambdaL1, TYPE lambda2 = defLambdaL2);
ENUM_LOSS_FUNCTION LossFunction(void) const { return(m_eLossFunction); }
ENUM_LOSS_FUNCTION LossFunction(TYPE &lambda1, TYPE &lambda2);

TYPE          GetRecentAverageLoss(void) const { return(m_dNNLoss); }
void          LossSmoothFactor(int value) { m_iLossSmoothFactor = value; }
int           LossSmoothFactor(void) const { return(m_iLossSmoothFactor); }
//--- Model operation mode control
bool          TrainMode(void) const { return m_bTrainMode; }
void          TrainMode(bool mode);
//--- Methods for working with files
1 virtual bool Save(.....);
virtual bool Load(.....);
//--- object identification method
virtual int    Type(void) const { return(defNeuronNet); }
//--- Retrieving pointers to internal objects
virtual CBufferType* GetGradient(uint layer) const;
virtual CBufferType* GetWeights(uint layer) const;
virtual CBufferType* GetDeltaWeights(uint layer) const;
};
```

You can note that in the declaration of several methods, I left ellipsis instead of specifying parameters. Now we will analyze the class methods and add the missing data.

Let's start with the class constructor. In it, we initialize the variables with initial values and create instances of the classes used.

3. Building the first neural network model in MQL5

```
CNet::CNet(void)      : m_bTrainMode(false),
                      m_bOpenCL(false),
                      m_bPositionEncoder(false),
                      m_dNNLoss(-1),
                      m_iLossSmoothFactor(defLossSmoothFactor),
                      m_dLearningRate(defLearningRate),
                      m_eLossFunction(LOSS_MSE)
{
    m_adLambda.Init(2);
    m_adBeta.Init(2);
    m_adLambda[0] = defLambdaL1;
    m_adLambda[1] = defLambdaL2;
    m_adBeta[0]   = defBeta1;
    m_adBeta[1]   = defBeta2;
    m_cLayers     = new CArrayLayers();
    m_cOpenCL     = new CMyOpenCL();
    m_cPositionEncoder = new CPositionEncoder();
}
```

In the class destructor, we will clear the memory by deleting the instances of the previously created objects.

```
CNet::~~CNet(void)
{
    if(!m_cLayers)
        delete m_cLayers;
    if(!m_cPositionEncoder)
        delete m_cPositionEncoder;
    if(!m_cOpenCL)
        delete m_cOpenCL;
}
```

Let's consider the **Create** method that creates a neural network. I omitted the parameters of this method earlier, and now I suggest we discuss them.

The interface for passing the structure of a neural network to a class was described in the previous chapter. Of course, we will pass it to this method. But is this data enough or not? From a technical perspective, this data is quite sufficient to specify the architecture of the neural network. We have provided additional methods for specifying learning rates and loss functions.

But if we look at the question from the user's perspective: how convenient is it to use three methods to specify all the necessary parameters when initializing the neural network? In fact, it is a matter of personal habits and preferences of the user. Some prefer to use multiple methods specifying one or two parameters and monitor the process at each step. Others would prefer to 'throw' all the parameters into one method in a single line of code, check the result once, and move on.

When we work directly with the customer, we can discuss their preferences and make the product convenient for them. But when creating a universal product, it's logical to try to satisfy the preferences of all potential users. Moreover, the user can choose different options depending on the task at hand. Therefore, we will use the ability to overload functions and create several methods with the same name to satisfy all possible usage scenarios.

3. Building the first neural network model in MQL5

First, we'll create a method with a minimal number of parameters, which will only receive a dynamic array describing the architecture of the neural network. At the beginning of the method, we will check the validity of the pointer to the object received in the method parameter. Then we check the number of neural layers in the passed description.

We already mentioned earlier that there cannot be less than two layers, as the first input layer is used to input the initial data, and the last layer is for outputting the result of the neural network's operation. If at least one check fails, we exit the method with a **false** result.

```
bool CNet::Create(CArrayObj *descriptions)
{
    //--- Control block
    if(!descriptions)
        return false;
    //--- Check the number of layers to be created
    int total = descriptions.Total();
    if(total < 2)
        return false;
```

After successfully passing the controls, we initialize the class to work with the OpenCL technology. Unlike the previous checks, we will not return **false** in the case of initialization errors. We will simply disable this functionality and continue operating in the standard mode. This approach is implemented to enable the replication of the finished product on various computing machines without altering the program code. This, in general, expands the potential customer base for distributing the end product.

```
    //--- Initialize OpenCL objects
    if(m_bOpenCL)
        m_bOpenCL = InitOpenCL();
    if(!m_cLayers.SetOpencl(m_cOpenCL))
        m_bOpenCL = false;
```

For all objects of our neural network to work in the same OpenCL context, we will pass a pointer to an instance of the *CMyOpenCL* class to the storage array of neural layers. From there, it will subsequently be passed to each neural layer.

Then we will organize a loop with the number of iterations equal to the number of layers of our network. In it, we will sequentially iterate through all the elements of the dynamic array describing neural layers. During this process, we will validate the validity of the description object for each layer, as well as ensure that the specified parameters adhere to the model's integrity. In the method's code, you can observe the validation of specific parameters for various types of neural layers, which we will become acquainted with a little later.

After that, we will call the method to create the corresponding layer. It is worth noting that we will entrust the creation of the neural layer directly to the element creation method, **CreateElement** of the *m_cLayers* dynamic storage array of neural layers.

3. Building the first neural network model in MQL5

```
//--- Organize a loop to create neural layers
for(int i = 0; i < total; i++)
{
    CLayerDescription *temp = descriptions.At(i);
    if(!temp)
        return false;
    if(i == 0)
    {
        if(temp.type != defNeuronBase)
            return false;
        temp.window = 0;
    }
    else
    {
        CLayerDescription *prev = descriptions.At(i - 1);
        if(temp.window <= 0 || temp.window > prev.count ||
            temp.type == defNeuronBase)
        {
            switch(prev.type)
            {
                case defNeuronConv:
                case defNeuronProof:
                    temp.window = prev.count * prev.window_out;
                    break;
                case defNeuronAttention:
                case defNeuronMHAttention:
                    temp.window = prev.count * prev.window;
                    break;
                case defNeuronGPT:
                    temp.window = prev.window;
                    break;
                default:
                    temp.window = prev.count;
                    break;
            }
        }
        switch(temp.type)
        {
            case defNeuronAttention:
            case defNeuronMHAttention:
            case defNeuronGPT:
                break;
            default:
                temp.step = 0;
        }
    }
    if(!m_cLayers.CreateElement(i, temp))
        return false;
}
```


3. Building the first neural network model in MQL5

At the end of the method, we initialize the positional encoding class. Please note that the actual code for each position remains unchanged throughout the training and utilization of the neural network. The elements will change, but the size of the input layer of neurons will stay the same. That means, upon creating the network, we can calculate and store the position code for each element right away, and subsequently use the saved values instead of repeatedly recalculating the code.

```
//--- Initialize positional coding objects
if(m_bPositionEncoder)
{
    if(!m_cPositionEncoder)
    {
        m_cPositionEncoder = new CPositionEncoder();
        if(!m_cPositionEncoder)
            m_bPositionEncoder = false;
        return true;
    }
    CLayerDescription *temp = descriptions.At(0);
    if(!m_cPositionEncoder.InitEncoder(temp.count, temp.window))
        UsePositionEncoder(false);
}
//---
return true;
}
```

When organizing method overloads for **Create**, we won't rewrite the entire code; we'll only carry out the user's tasks and make calls to the necessary methods with the received parameters. Below are the possible variations of the overloaded method.

```
bool CNet::Create(CArrayObj *descriptions,
                 TYPE learning_rate,
                 TYPE beta1,TYPE beta2,
                 ENUM_LOSS_FUNCTION loss_function,
                 TYPE lambda1,TYPE lambda2)
{
    if(!Create(descriptions))
        return false;
    SetLearningRates(learning_rate,beta1,beta2);
    if(!LossFunction(loss_function,lambda1,lambda2))
        return false;
//---
return true;
}
```

3. Building the first neural network model in MQL5

```
bool CNet::Create(CArrayObj *descriptions,
                  ENUM_LOSS_FUNCTION loss_function,
                  TYPE lambda1,TYPE lambda2)
{
    if(!Create(descriptions))
        return false;
    if(!LossFunction(loss_function,lambda1,lambda2))
        return false;
    //---
    return true;
}

bool CNet::Create(CArrayObj *descriptions,
                  TYPE learning_rate,
                  TYPE beta1,TYPE beta2)
{
    if(!Create(descriptions))
        return false;
    SetLearningRates(learning_rate,beta1,beta2);
    //---
    return true;
}
```

When creating overloaded methods, be sure to declare any method overloads that you use in the class declaration.

Let's move on. Let's talk about the **FeedForward** feed forward method. The method parameters are omitted in the declaration above. Let's think about what data we need to perform a direct pass. First of all, we need initial data. They must be transferred to the neural network from the outside. We are adding the dynamic array *CBufferType* to the *parameters*. We will create this class later; it will serve all our data buffers.

During the forward pass, the input data is multiplied by the weights stored in the neural layer objects. This means that the neural network already knows them. The obtained values are passed through an activation function. The functions used for each layer are specified during the neural network's creation stage in the architecture description.

Thus, to implement the direct pass, it is enough for us to receive an array of initial data at the input.

In the method body, we will validate the pointers to the array of input data and the first neural layer of our network. We will not create a separate type of neural layer for the initial data. Instead, we take a basic fully connected neural layer and write the received initial data to the buffer of output (resulting) values of neurons. Thus, we get the unification of neural layers.

```
bool CNet::FeedForward(const CBufferType *inputs)
{
    //--- control block
    if(!inputs)
        return false;
    CNeuronBase *InputLayer = m_cLayers.At(0);
    if(!InputLayer)
        return false;
```

In the next step, if necessary, we will position the initial values.

3.4 Creating the framework for the future MQL5 program

3. Building the first neural network model in MQL5

```
CBufferType *Inputs = InputLayer.GetOutputs();
if(!Inputs)
    return false;
if(Inputs.Total() != inputs.Total())
    return false;
//--- Transfer the source data to the neural layer
Inputs.m_mMatrix = inputs.m_mMatrix;
//--- Apply positional coding
if(m_bPositionEncoder && !m_cPositionEncoder.AddEncoder(Inputs))
    return false;
if(m_bOpenCL)
    Inputs.BufferCreate(m_cOpenCL);
```

At this stage, the preparation of the initial data can be considered complete. Let's proceed directly to the forward pass: we will organize a loop that iterates through all the neural layers in our network sequentially, from the first to the last. For each layer, we will call its corresponding forward pass method. Note that the loop starts at layer index 1. The neural layer with the initial data recorded has an index of 0.

Another point to which you should also pay attention. In the process of enumeration, we use one class *CNeuronBase* for all objects of neural layers. This is our base class for the neural layer. All other classes of neural layers will inherit from it.

In addition, we will create the virtual method *FeedForward* that will be overridden in all other types of neural layers. This implementation allows us to use the neural layer base class and call the forward pass virtual method. The task of distributing and utilizing the specific type of neuron's forward pass method will be handled by the compiler and system on our behalf.

```
//--- Create a loop with a complete search of all neural layers
//--- and call the forward pass method for each of them
CNeuronBase *PrevLayer = InputLayer;
int total = m_cLayers.Total();
for(int i = 1; i < total; i++)
{
    CNeuronBase *Layer = m_cLayers.At(i);
    if(!Layer)
        return false;
    if(!Layer.FeedForward(PrevLayer))
        return false;
    PrevLayer = Layer;
}
```

It should be noted here that when using the OpenCL technology, when the kernel is sent for execution, it is queued. To "push" its execution, we need to initiate the retrieval of the operation results. We have previously discussed the need to minimize the exchange of data between RAM and the OpenCL context. Therefore, we will not retrieve data after each kernel is added to the queue. Instead, we will enqueue the entire chain of operations and only after completing the loop iterating through all the neural layers, we will request the results of the operations from the last neural layer. Since our data is passed sequentially from one layer to another, the entire queue of operations will be pulled along. But do not forget that data loading is only necessary when using the OpenCL technology.

3. Building the first neural network model in MQL5

```
if(m_bOpenCL)
    if(!PrevLayer.GetOutputs().BufferRead())
        return false;
//---
return true;
}
```

During the feed-forward pass, we obtained certain calculated data. On an untrained neural network, the obtained result will be quite random. We aim for our neural network to produce results that are as close as possible to real outcomes. And in order to get closer to them, we need to train [a neural network](#). The supervised learning process is based on an iterative approach with the gradual adjustment of weights to the correct answers. As we said earlier, this process consists of two stages: forward and [backward \(backpropagation\) pass](#). We have already written about the forward pass method. Let's look at the **backpropagation** method.

Above, when describing the class, I also omitted the parameters of this method. Please take another look at the algorithm for the [backward pass](#). Here we need only correct answers from the external system. Therefore, we will add a dynamic array of correct answers to the method parameters. But at the input of the method, we will receive only reference values for the output neural layer. Therefore, we need to calculate the error gradient for each neuron in our network. The only exception is the neurons in the input layer: their values are provided by an external system and are independent of the neural network state. Hence, calculating the error gradient for the input data is unnecessary work that has no practical value and logical meaning.

At the beginning of the method, as always, we will perform data validation for the method operation. In this block, we will validate the received pointer to the dynamic array of target values and compare the result buffer size with the size of the obtained vector of target values. After that, we calculate the value of the loss function. The calculation of the loss function itself is hidden in the standard MQL5 matrix operations. The algorithm for calculating the value of the function was shown when considering possible options for the [loss function](#). We will check the obtained loss function value and calculate the smoothed error over the entire training period.

```
bool CNet::Backpropagation(CBufferType *target)
{
//--- Control block
if(!target)
    return false;
int total = m_cLayers.Total();
CNeuronBase *Output = m_cLayers.At(total - 1);
if(!Output || Output.Total() != target.Total())
    return false;
//--- Calculate the value of the loss function
TYPE loss = Output.GetOutputs().m_mMatrix.Loss(target.m_mMatrix,
                                                m_eLossFunction);

if(loss == FLT_MAX)
    return false;
m_dNNLoss = (m_dNNLoss < 0 ? loss :
            m_dNNLoss + (loss - m_dNNLoss) / m_iLossSmoothFactor);
```

In the next block of our backward pass method, we will bring the error gradient to each neuron of our network. To achieve this, we will first calculate the error gradient at the output layer and then set up a backward loop. While iterating from the output of the neural network to its input, for each neural layer, we will invoke the gradient calculation method. We will discuss the differences in gradient calculation

algorithms for the output and hidden layers of the neural network a bit later while exploring the [fully connected neural layer](#).

Right here, we will calculate how the weights of our neural network should change in order for it to produce correct results for the current set of input data. In the sequential enumeration of neural layers, for each layer we will call the method for calculating deltas.

```
//--- Calculate the error gradient at the output of a neural network
CBufferType* grad = Output.GetGradients();
grad.m_mMatrix = target.m_mMatrix;
if(m_cOpenCL)
{
    if(!grad.BufferWrite())
        return false;
}
if(!Output.CalcOutputGradient(grad, m_eLossFunction))
    return false;
//--- Create a loop with enumeration of all neural layers in reverse order
for(int i = total - 2; i >= 0; i--)
{
    CNeuronBase *temp = m_cLayers.At(i);
    if(!temp)
        return false;
    //--- Call the method for distributing the error gradient through the hidden la
    if(!Output.CalcHiddenGradient(temp))
        return false;
    //--- Call the method for distributing the error gradient to the weight matrix
    if(!Output.CalcDeltaWeights(temp, i == 0))
        return false;
    Output = temp;
}
```

Similarly to the forward pass, in the case of using OpenCL technology, we need to download the results of the operations of the last kernel in the queue.

```
if(m_cOpenCL)
{
    for(int i = 1; i < m_cLayers.Total(); i++)
    {
        Output = m_cLayers.At(i);
        if(!Output.GetDeltaWeights() || !Output.GetDeltaWeights().BufferRead())
            continue;
        break;
    }
}
//---
return true;
}
```

The goal of training a neural network is not to find deviations, but to adjust it for the maximum likelihood of producing accurate results. A neural network is tuned by adjusting the correct weights. Therefore, after calculating the deltas, we must update the weights. For the above reasons, I moved the update of the weights into a separate method **UpdateWeights**.

When declaring a method in the class description, the parameters are not specified. Let's think: we have already calculated the deltas for updating the weights, and the training and regularization coefficients are set when initializing the neural network. At first glance, we have everything we need to update the weights. But look at the deltas. At each iteration, we will summarize them. If a batch of a certain size is used for updating coefficients, there is a high likelihood of obtaining an exaggerated delta. In such a situation, it is logical to use the average delta. To get the average of the sum of the packet deltas, it is enough to divide the available sum by the packet size. Of course, mathematically speaking, batch size can be factored into the learning rate. If we pre-divide the learning rate by the batch size, the final result will remain unchanged.

$$\alpha \frac{\Delta}{Batchsize} = \frac{\alpha}{Batchsize} \Delta$$

But this is manual control, and as always, it's a matter of user preference. We will give the opportunity to use both options: we will add a parameter to the method to specify the batch size and set its default value to one. Thus, the user can specify the batch size in the method parameters or can call the method without specifying parameters. In that case, the batch size will be set to the default value, and the delta will be adjusted only by the learning coefficient.

The algorithm of the method is quite straightforward. First, we will validate the specified batch size as it must be a positive integer value. Next, we will set up a loop to iterate through all the neural layers in our network, calling the corresponding method for each layer. The very process of updating the weights will be carried out at the level of the neural layer.

```
bool CNet::UpdateWeights(uint batch_size = 1)
{
    //--- Control block
    if(batch_size <= 0)
        return false;
    //--- Organize a loop of enumeration of all hidden layers
    int total = m_cLayers.Total();
    for(int i = 1; i < total; i++)
    {
        //--- Check the validity of the pointer to the neural layer object
        CNeuronBase *temp = m_cLayers.At(i);
        if(!temp)
            return false;
        //--- Call the method of updating the matrix of the weights of the inner layer
        if(!temp.UpdateWeights(batch_size, m_dLearningRate, m_adBeta, m_adLambda))
            return false;
    }
    //---
    return true;
}
```

Of course, the user should have the ability to obtain the results of the neural network operation after the forward pass is executed. This will be implemented by the *GetResult* method.

What external data should the method receive? Logically reasoning, the function should not receive but rather return data to an external program. However, we do not know what this data will be and in what numbers. Knowing the possible options for the neuron activation functions, it is logical to assume that the output of each neuron will be a certain number. The number of such values will be equal to the number of neurons in the output layer. Accordingly, it will be known at the stage of generation of the

neural network. The logical way out of this situation would be a dynamic array of the appropriate type. Previously we used the data buffer class *CBufferType* for passing data into our model. Here we will use a similar object. Thus, for data exchange between the main program and the model, we will always use one dynamic array class.

In the method body, we first obtain a pointer to the array of output layer neuron values and validate this pointer. Then we check the validity of the pointer to the dynamic array for storing the results. We received a link to the last array in the method parameters from an external program. If the pointer is invalid, then we initiate the creation of a new instance of the data buffer class. After successfully creating a new buffer, we copy the values from the output layer neurons into it and exit the method.

```
bool CNet::GetResults(CBufferType *&result)
{
    int total = m_cLayers.Total();
    CNeuronBase *temp = m_cLayers.At(total - 1);
    if(!temp)
        return false;
    CBufferType *output = temp.GetOutputs();
    if(!output)
        return false;
    if(!result)
    {
        if(!(result = new CBufferType()))
            return false;
    }
    if(m_cOpenCL)
        if(!output.BufferRead())
            return false;
    result.m_mMatrix = output.m_mMatrix;
    //---
    return true;
}
```

It's important to note that depending on the complexity of the task, neural networks can vary significantly in terms of architectural complexity and the number of synaptic connections. The training time of the network heavily depends on its complexity. Retraining the neural network every time is inefficient and is impossible in most cases. Therefore, the once-trained neural network must be saved and, at the next start, all the coefficients should be loaded from the file. Only after that, if necessary, you can retrain the neural network for the current realities.

The method responsible for saving the trained neural network is called **Save**. This virtual method is created in the *CObject* base class and is overridden in every new class. I intentionally did not immediately rewrite the method parameters from the parent class. The reason is that the parameters there are designed to receive a file handle for writing the object. That is, the file must first be opened in an external program, and after saving the data, the external program closes the file.

In other words, the control over opening and closing the file is removed from the class and placed onto the calling program. This approach is convenient when the object is part of a larger project and allows sequentially writing all project objects into a single shared file. And we will definitely use this when saving the objects that make up our neural network.

However, when we're talking about the top level of our program, it would be desirable to have a single method for saving the entire project. This method should handle the task of opening and closing the file,

3. Building the first neural network model in MQL5

iterating through and saving all the necessary information for reconstructing the entire neural network from the file. At the same time, we cannot exclude the possibility that the neural network will be just a part of something larger.

Taking into consideration the ideas presented above, we will create two methods with the same name: one will receive a file handle in its parameters similar to the parent class method, and the other will be passed a file name for data writing.

Now, let's think about the minimum information we need to fully reconstruct a trained neural network. Of course, we need the architecture of the network, the number of layers and the number of neurons in them. Besides, we need all weights. To do this, we need to save the entire array of neural layers.

However, it's important to understand that a trained neural network will work correctly only within the environment for which it was trained. Therefore, we will save information about the loss function and position encoding.

I propose to write information about the symbol and timeframe in the name of the file. This will allow the Expert Advisor to quickly determine the presence of a pre-trained network on the disk in the future. Moreover, changing just the file name would be sufficient to transfer and test a pre-trained neural network on a different tool or timeframe. In most cases, fine-tuning a neural network will be easier than training it from random weights.

To gauge the extent of training for the neural network saved in the file, let's add the final average loss value and the smoothing coefficient. For convenient continuation of training, we will save the training and regularization parameters. To complete the picture, we will also add a flag indicating whether to use OpenCL.

Let's look at the algorithm of the method with the file handle in the parameters. At the beginning of the method, we will check the validity of the received file handle for data writing, as well as the pointers to the instances of loss functions and the dynamic array of neural layers.

```
bool CNet::Save(const int file_handle)
{
    if(file_handle == INVALID_HANDLE ||
        !m_cLayers)
        return false;
```

Next, we will save the above parameters.

```
//--- Storing constants
if(!FileWriteInteger(file_handle, (int)m_bOpenCL) ||
    !FileWriteDouble(file_handle, m_dNNLoss) ||
    !FileWriteInteger(file_handle, m_iLossSmoothFactor) ||
    !FileWriteInteger(file_handle, (int)m_bPositionEncoder) ||
    !FileWriteDouble(file_handle, (double)m_dLearningRate) ||
    !FileWriteDouble(file_handle, (double)m_adBeta[0]) ||
    !FileWriteDouble(file_handle, (double)m_adBeta[1]) ||
    !FileWriteDouble(file_handle, (double)m_adLambda[0]) ||
    !FileWriteDouble(file_handle, (double)m_adLambda[1]) ||
    !FileWriteInteger(file_handle, (int)m_eLossFunction))
    return false;
```

Let's check the flag for using the positional encoding of the input sequence and, if necessary, call the *CPositionEncoder* class instance saving method. At the end of the method, let's call the method that

saves a dynamic array of neural layers. We will get acquainted with the called methods in more detail while analyzing the classes containing them.

```
//--- Save the positional coding object if necessary
if(m_bPositionEncoder)
{
    if(!m_cPositionEncoder ||
        !m_cPositionEncoder.Save(file_handle))
        return false;
}
//-- Call the method for saving the data of a dynamic array of neural layers
return m_cLayers.Save(file_handle);
}
```

The algorithm method with the file name in the parameters will be a bit simpler. We will not rewrite the data saving algorithm in full. We will simply set up the file for writing information, and then pass the obtained file handle to the method discussed above. After the method execution is complete, we will close the file.

Please note that if an empty file name is provided in the parameters, we will replace it with the default file name and then proceed to execute the method in the standard mode.

Also, after executing the file opening function, we should check the success of the operation by checking the received handle. I deliberately omitted this step as it is the first operation in the *Save* method discussed above, and doing the same operation twice will only slow things down.

```
bool CNet::Save(string file_name = NULL)
{
    if(file_name == NULL || file_name == "")
        file_name = defFileName;
    //---
    int handle = FileOpen(file_name, FILE_WRITE | FILE_BIN);
    //---
    bool result = Save(handle);
    FileClose(handle);
    //---
    return result;
}
```

For the reverse operation of loading neural network data from a file, we will create two similar **Load** methods with a handle and a file name in the parameters. While the algorithm for loading data with a specified file name in the parameters is identical to the corresponding data saving method, the algorithm for the second method becomes slightly more complex due to the initialization operations of objects.

At the beginning of the method, just like during saving, we validate the validity of the received file handle for loading data.

```
bool CNet::Load(const int file_handle)
{
    if(file_handle == INVALID_HANDLE)
        return false;
}
```

3. Building the first neural network model in MQL5

Then we load all the previously saved parameters of the neural network. At the same time, we make sure that the sequence of reading data strictly corresponds to the sequence of their recording.

```
//--- Reading constants
m_bOpenCL = (bool)FileReadInteger(file_handle);
m_dNNLoss = FileReadDouble(file_handle);
m_iLossSmoothFactor = FileReadInteger(file_handle);
m_bPositionEncoder = (bool)FileReadInteger(file_handle);
m_dLearningRate = (TYPE)FileReadDouble(file_handle);
m_adBeta[0] = (TYPE)FileReadDouble(file_handle);
m_adBeta[1] = (TYPE)FileReadDouble(file_handle);
m_adLambda[0] = (TYPE)FileReadDouble(file_handle);
m_adLambda[1] = (TYPE)FileReadDouble(file_handle);
m_eLossFunction = (ENUM_LOSS_FUNCTION) FileReadInteger(file_handle);
```

Please note that when saving the data, we wrote the positional encoding object to the file only when the function was enabled. Consequently, we first check if the function was enabled when saving the data, and if necessary, initiate the process of reading the positional encoding method. We check the existence of the corresponding created object. If it has not been created before, then before loading the data, we initiate the creation of an instance of the object.

```
//--- Load the positional coding object
if(m_bPositionEncoder)
{
    if(!m_cPositionEncoder)
    {
        m_cPositionEncoder = new CPositionEncoder();
        if(!m_cPositionEncoder)
            return false;
    }
    if(!m_cPositionEncoder.Load(file_handle))
        return false;
}
```

To initialize the OpenCL context object, we won't repeat the entire initialization code. Instead, we will use the appropriate method. We just need to call it and control the result of the operations.

```
//--- Initialize the object for working with OpenCL
if(m_bOpenCL)
{
    if(!InitOpenCL())
        m_bOpenCL = false;
}
else
    if(!m_cOpenCL)
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
    }
```

Next, we need to load the neural layers of the model and their parameters directly. To load this information, it would be sufficient to call the method for loading the dynamic array of neural layers. But before accessing the class method, we need to ensure the validity of the pointer to the class instance.

3. Building the first neural network model in MQL5

Otherwise, we risk getting a critical program execution error. Therefore, we validate the pointer validity and create a new instance of the dynamic array object if necessary. Here we pass a valid pointer to the object to work with the OpenCL context into the object. Only after the preparatory work is done, we call the method that loads the dynamic array of neural layers.

```
//--- Initialize and load the data of a dynamic array of neural layers
if(!m_cLayers)
{
    m_cLayers = new CArrayLayers();
    if(!m_cLayers)
        return false;
}
if(m_bOpenCL)
    m_cLayers.SetOpencl(m_cOpenCL);
//---
return m_cLayers.Load(file_handle);
}
```

Perhaps, here we should explain why we're only loading the dynamic array instead of all the neural layers. The reason is that our dynamic array of neural layers serves as a container containing pointers to all the neural layer objects in the model. During saving, all the neural layers were sequentially stored in the array. Now, when loading the data, objects will also be sequentially created while preserving the pointers in the array. We will get acquainted with this mechanism in more detail when considering the methods of this class.

So, we've covered the main methods of our neural network class. In conclusion, taking into account everything mentioned above, its final structure will look as follows.

```
class CNet : public CObject
{
protected:
    bool                m_bTrainMode;
    CArrayLayers*       m_cLayers;
    CMyOpenCL*         m_cOpenCL;
    bool                m_bOpenCL;
    TYPE                m_dNNLoss;
    int                 m_iLossSmoothFactor;
    CPositionEncoder*   m_cPositionEncoder;
    bool                m_bPositionEncoder;
    ENUM_LOSS_FUNCTION  m_eLossFunction;
    VECTOR              m_adLambda;
    TYPE                m_dLearningRate;
    VECTOR              m_adBeta;
```

```

public:
    CNet(void);
    ~CNet(void);

//--- Methods for creating an object
bool    Create(CArrayObj *descriptions);
bool    Create(CArrayObj *descriptions, TYPE learning_rate,
              TYPE beta1, TYPE beta2);

bool    Create(CArrayObj *descriptions,
              ENUM_LOSS_FUNCTION loss_function, TYPE lambda1, TYPE lambda2);
bool    Create(CArrayObj *descriptions, TYPE learning_rate,
              TYPE beta1, TYPE beta2,
              ENUM_LOSS_FUNCTION loss_function, TYPE lambda1, TYPE lambda2);

//--- Implement work with OpenCL
void    UseOpenCL(bool value);
bool    UseOpenCL(void)          const { return(m_bOpenCL);          }
bool    InitOpenCL(void);

//--- Methods for working with positional coding
void    UsePositionEncoder(bool value);
bool    UsePositionEncoder(void) const { return(m_bPositionEncoder);}

//--- Implement the main algorithms of the model
bool    FeedForward(const CBufferType *inputs);
bool    Backpropagation(CBufferType *target);
bool    UpdateWeights(uint batch_size = 1);
bool    GetResults(CBufferType *&result);
void    SetLearningRates(TYPE learning_rate, TYPE beta1 = defBeta1,
                        TYPE beta2 = defBeta2);

//--- Loss Function Methods
bool    LossFunction(ENUM_LOSS_FUNCTION loss_function,
                    TYPE lambda1 = defLambdaL1, TYPE lambda2 = defLambdaL2);
ENUM_LOSS_FUNCTION LossFunction(void)          const { return(m_eLossFunction); }
ENUM_LOSS_FUNCTION LossFunction(TYPE &lambda1, TYPE &lambda2);

TYPE    GetRecentAverageLoss(void) const { return(m_dNNLoss);          }
void    LossSmoothFactor(int value)    { m_iLossSmoothFactor = value;}
int     LossSmoothFactor(void)         const { return(m_iLossSmoothFactor);}

//--- Model operation mode control
bool    TrainMode(void)                const { return m_bTrainMode;      }
void    TrainMode(bool mode);

//--- Methods for working with files
virtual bool    Save(string file_name = NULL);
virtual bool    Save(const int file_handle);
virtual bool    Load(string file_name = NULL, bool common = false);
virtual bool    Load(const int file_handle);

//--- object identification method
virtual int     Type(void)               const { return(defNeuronNet);      }

//--- Retrieve pointers to internal objects
virtual CBufferType* GetGradient(uint layer)    const;
virtual CBufferType* GetWeights(uint layer)     const;
virtual CBufferType* GetDeltaWeights(uint layer) const;
};

```

3.4.4 Dynamic array of neural layer storage

It is worth mentioning a few words about the dynamic array **CArrayLayers** that stores neural layers. As previously announced, it is based on the standard **CArrayObj** object array class.

The functionality of the parent class almost entirely meets our requirements for a dynamic array. When examining the source code of the parent class, you can find all the functionality related to the dynamic array operations and accessing its elements. Additionally, methods for working with files (writing and reading an array) are also implemented. For this, special thanks to the [MetaQuotes](#) team.

When examining the algorithm of the method that reads an array from the **Load** file in detail, pay attention to the **CreateElement** method which creates a new element.

In the previous section, when discussing the method for reading a neural network from a file, prior to reading the data, we instantiated an object of the corresponding class. The mentioned method performs similar functionality, but it is not implemented in the parent class. This is quite understandable and reasonable, as the creators of the class couldn't anticipate the specific objects their array would store, and thus couldn't create a method generating an unknown class. Therefore, they left a virtual method to be overridden in the user-defined class.

And as consumers of their product, we create our own dynamic array class by inheriting the core functionality from the parent class. In this case, we override the method of creating a new array element.

```
class CArrayLayers : public CArrayObj
{
protected:
    CMyOpenCL*      m_cOpenCL;
    int             m_iFileHandle;
public:
    CArrayLayers(void) : m_cOpenCL(NULL),
                        m_iFileHandle(INVALID_HANDLE)
    { }
    ~CArrayLayers(void) { };

    //---
    virtual bool     SetOpencl(CMyOpenCL *opencl);
    virtual bool     Load(const int file_handle) override;
    //--- method creating an element of an array
    virtual bool     CreateElement(const int index) override;
    virtual bool     CreateElement(const int index,
                                   CLayerDescription* description);
    //--- method identifying the object
    virtual int      Type(void) override const { return(defArrayLayers); }
};
```

One more point should be noted. In order for our overridden method to be called from the parent class method, its definition must fully match the definition of the parent class method, including parameters and return value. Of course, there is nothing complex in this, but we are faced with the same question that the team of creators of the parent class had: what object to create?

We know it will be a neural layer object, but we don't know what type. We can save the type of the required neural layer to a file before writing the contents of the object itself. However, how can we read it from the file if the method doesn't receive a file handle for loading data?

At the same time, we pass the file handle when we call the data loading method *Load*. Evidently, we need to override the load method as well. But I wouldn't want to rewrite the whole method. Therefore, I added the variable *m_iFileHandle*, in which I save the file handle for loading data when the *Load* method is called. Then I call a similar method of the parent class.

```
bool CArrayLayers::Load(const int file_handle)
{
    m_iFileHandle = file_handle;
    return CArrayObj::Load(file_handle);
}
```

Now let's look directly at the method of creating a new neural layer in a dynamic array. In the parameters, the method receives the index of the element to be created. At the beginning of the method, we check that the resulting index is not negative, because the index of an element of a dynamic array cannot be less than zero. We will also check the saved file handle for loading – without it, we won't be able to determine the type of the element being created.

Next, we reserve an element in our array, read the type of the element to be created from the file, and create an instance of the type we need. Let's not forget to check the result of creating a new object, pass a pointer to the OpenCL object into the new element, and save the pointer to the new neural layer into our array. In conclusion, let's ensure that the index of the new element does not exceed the maximum number of elements in the array.

```
bool CArrayLayers::CreateElement(const int index)
{
    //--- source data verification block
    if(index < 0 || m_iFileHandle==INVALID_HANDLE)
        return false;
    //--- reserving an array element for a new object
    if(!Reserve(index + 1))
        return false;
    //--- read the type of the desired object from the file and create the corresponding
    CNeuronBase *temp = NULL;
    int type = FileReadInteger(m_iFileHandle);
    switch(type)
    {
        case defNeuronBase:
            temp = new CNeuronBase();
            break;
        case defNeuronConv:
            temp = new CNeuronConv();
            break;
        case defNeuronProof:
            temp = new CNeuronProof();
            break;
        case defNeuronLSTM:
            temp = new CNeuronLSTM();
            break;
        case defNeuronAttention:
            temp = new CNeuronAttention();
            break;
        case defNeuronMHAttention:
            temp = new CNeuronMHAttention();
```

3. Building the first neural network model in MQL5

```
        break;
    case defNeuronGPT:
        temp = new CNeuronGPT();
        break;
    case defNeuronDropout:
        temp = new CNeuronDropout();
        break;
    case defNeuronBatchNorm:
        temp = new CNeuronBatchNorm();
        break;
    default:
        return false;
    }
//--- control over the creation of a new object
if(!temp)
    return false;
//--- add a pointer to the created object to the array
if(m_data[index])
    delete m_data[index];

temp.SetOpenCL(m_cOpenCL);
m_data[index] = temp;
//---
return true;
}
```

Since a new class has been created, I decided to add a couple more methods to it. The first thing I added was a similar method for generating a new item. The difference is that the new method creates a new layer based on the description obtained in the method parameters. The algorithm of the method is almost completely the same as above, except for some details.

```
bool CArrayLayers::CreateElement(const int index, CLayerDescription *desc)
{
//--- source data verification block
if(index < 0 || !desc)
    return false;
//--- reserve an array element for a new object
if(!Reserve(index + 1))
    return false;
//--- create the corresponding neural layer
CNeuronBase *temp = NULL;
switch(desc.type)
{
    case defNeuronBase:
        temp = new CNeuronBase();
        break;
    case defNeuronConv:
        temp = new CNeuronConv();
        break;
    case defNeuronProof:
        temp = new CNeuronProof();
        break;
}
```

3.4 Creating the framework for the future MQL5 program

```

        case defNeuronLSTM:
            temp = new CNeuronLSTM();
            break;
        case defNeuronAttention:
            temp = new CNeuronAttention();
            break;
        case defNeuronMHAttention:
            temp = new CNeuronMHAttention();
            break;
        case defNeuronGPT:
            temp = new CNeuronGPT();
            break;
        case defNeuronDropout:
            temp = new CNeuronDropout();
            break;
        case defNeuronBatchNorm:
            temp = new CNeuronBatchNorm();
            break;
        default:
            return false;
    }
    //--- control over the creation of a new object
    if(!temp)
        return false;
    //--- add a pointer to the created object to the array
    if(!temp.Init(desc))
        return false;
    if(m_data[index])
        delete m_data[index];
    temp.SetOpenCL(m_cOpenCL);
    m_data[index] = temp;
    m_data_total = fmax(m_data_total, index + 1);
    //---
    return true;
}

```

The second added method is responsible for passing a pointer to the OpenCL object to all previously created layers of our neural network, as the decision to use this technology can be made either before or after the neural network is generated. For example, a neural network can be created and tested for performance without using OpenCL technology. Further, the technology can be leveraged to accelerate the learning process.

The algorithm of the method is quite simple. We first check if the pointer was previously set and delete the old object if necessary. Then we save the new pointer and start the loop for enumerating the elements of the dynamic array. In this case, we will pass a new pointer to an OpenCL object to each element of the array.


```

bool CArrayLayers::SetOpenCL(CMyOpenCL *opencl)
{
    //--- source data verification block
    if(m_cOpenCL)
        delete m_cOpenCL;

    m_cOpenCL = opencl;
    //--- passing a pointer to all array elements
    for(int i = 0; i < m_data_total; i++)
    {
        if(!m_data[i])
            return false;
        if(!((CNeuronBase *)m_data[i]).SetOpenCL(m_cOpenCL))
            return false;
    }
    //---
    return(!!m_cOpenCL);
}

```

3.5 Description of a Python script structure

Python is an interpreted programming language with a minimalistic syntax. Such syntax enables the fast creation of small code blocks and the immediate testing of their functionality. Therefore, Python allows you to focus on solving the problem rather than programming. Perhaps, it's precisely due to this feature that Python has gained such popularity.

Despite the fact that interpreted programming languages run slower than compiled ones, Python has currently become the most popular programming language for creating and conducting experiments with neural networks. The issue of execution speed is solved by using various libraries written, among others, in compiled programming languages. Fortunately, Python has the ability to easily expand and incorporate libraries written in practically all available programming languages.

We, too, won't be constructing complex algorithms and will make use of ready-made solutions, including libraries both for building neural networks and for trading. Let's start by familiarizing ourselves with some of them.

The **os** module contains functions for working with the operating system. Using this library enables the creation of cross-platform applications, as the functionality of this module operates independently of the installed operating system. Here are just some of the functions of the **os** library:

- *os.name* returns the name of the operating system. The following options are possible as a result of executing the function: 'posix', 'nt', 'mac', 'os2', 'ce', 'java'.
- *os.environ* is a function for working with environment variables, allowing you to modify, add and delete environment variables.
- *os.path* contains a number of functions for working with file and directory paths.

The **Pandas** module is a library for processing and analyzing data. The library provides specialized data structures and operations for processing numeric tables and time series. It enables data analysis and modeling without using specialized programming languages for statistical processing, such as *R* or *Octave*.

3. Building the first neural network model in MQL5

The package is designed for data cleaning and initial assessment based on general statistical indicators. It can be used to calculate mean, quantiles, etc. At the same time, the package cannot be considered purely statistical in nature. However, the datasets it creates, such as *DataFrame* and *Series* types, are used as inputs in most data analysis and machine learning modules like *SciPy*, *Scikit-Learn*, and others.

The *DataFrame* object is created in the *Pandas* library. It is designed to work with indexed arrays of two-dimensional data.

In addition, the library provides:

- Tools for data exchange between structures in memory and files of different formats;
- Built-in data matching tools and ways to handle missing information;
- Reformatting datasets, including creating pivot tables;
- Advanced indexing and sampling capabilities from large datasets;
- Grouping capabilities that enable performing three-step operations like “split, apply, combine”;
- Merging and combining different datasets.

The library provides the ability to create hierarchical indexing, allowing you to work with high-dimensional data within structures of lower dimensions. Functions for working with time series allow you to form time periods and change intervals. The library is optimized for high performance, with the most important parts of the code written in *Cython* and *C*.

Another library for working with multidimensional arrays, ***NumPy***, is an open-source library. The main capabilities of this module include support for multidimensional arrays (including matrices) and high-level mathematical functions designed for working with multidimensional arrays.

The *NumPy* library implements computational algorithms in the form of functions and operators that are optimized for working with multidimensional arrays. The library offers the ability to perform vector operations on data. All functions are written in *C* and optimized for maximum performance. As a result, any algorithm that can be expressed as a sequence of operations on arrays (matrices) and implemented using *NumPy* runs as fast as the equivalent code executed in *MATLAB*.

At the same time, *NumPy* can be considered as an alternative to using *MATLAB*. Both languages are interpreted and allow performing operations on arrays.

NumPy is often used as a base for working with multidimensional arrays in other libraries. The aforementioned *Pandas* also uses the *NumPy* library for low-level array operations.

The ***Matplotlib*** module is a comprehensive library for creating static, animated, and interactive visualizations. It can be used to visualize large volumes of data.

We will use the ***TensorFlow*** library to create neural network models. This is a comprehensive open-source platform for machine learning. It has a flexible ecosystem of tools, libraries, and community resources that allow researchers to advance the latest achievements in machine learning while enabling developers to easily create and deploy machine learning-based applications.

The library enables the creation and training of machine learning models using intuitive high-level APIs with eager execution, such as *Keras*. This provides immediate integration of the model and facilitates debugging.

Of course, to integrate with the MetaTrader 5 terminal, we will use the *MetaTrader5* library of the same name. It provides a set of functions for data exchange with the terminal, including functions for retrieving market information and executing trading operations.

3. Building the first neural network model in MQL5

For technical analysis of data, you can make use of the TA-lib library, which offers a wide range of functions for technical indicators.

Before you can use the libraries, you must install them in the *Python* environment you are using. To do this, in the command prompt or *Windows PowerShell* with administrator privileges, you need to execute a series of commands:

- *NumPy* installation
`pip install numpy`
- installing *Pandas*
`pip install pandas`
- installing *Matplotlib*
`pip install matplotlib`
- installing *TensorFlow*
`pip install tensorflow`
- installing *Keras*
`pip install keras`
- installing MetaTrader 5 library
`pip install MetaTrader5`

Moving directly to the structure of our script, let's create a template *template.py*. The script will consist of several blocks. First, we need to connect the necessary libraries to our script.

```
# import libraries
import os
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib as mp
import matplotlib.pyplot as plt
import matplotlib.font_manager as fm
import MetaTrader5 as mt5
```

After training the models, we will create visualization plots to depict the training process and compare the performance of different models. To standardize the plots, we will define common parameters for their construction.

3. Building the first neural network model in MQL5

```
# set parameters for results graphs
mp.rcParams.update({'font.family':'serif',
                    'font.serif':'Clear Sans',
                    'axes.labelsize':'medium',
                    'legend.fontsize':'small',
                    'figure.figsize':[6.0,4.0],
                    'xtick.labelsize':'small',
                    'ytick.labelsize':'small',
                    'axes.titlesize': 'x-large',
                    'axes.titlecolor': '#333333',
                    'axes.labelcolor': '#333333',
                    'axes.edgecolor': '#333333'
                    })
```

We will perform the training and testing of all models on a single dataset, which we will specifically preload into a file on the local disk. This approach will allow us to eliminate the influence of disparate data and assess the performance of different neural network models under consistent conditions.

Hence, in the next step, we will load the initial data from the file into the table. Please note the following: since MetaTrader 5 restricts access to files from its programs within the sandboxed environment, you need to provide the full path to the source data file. It will be stored in the *MQL5\Files* directory of your terminal or its subdirectories if they were specified when saving the data file.

Instead of hardcoding the path to the terminal sandbox in our program code, we will retrieve it from MetaTrader 5 using the provided API. To achieve this, we first establish a connection to the installed terminal and verify the outcome of this operation.

```
# Connecting to the MetaTrader 5 terminal
if not mt5.initialize():
    print("initialize() failed, error code =",mt5.last_error())
    quit()
```

After successfully connecting to the terminal, we will request the sandbox path and then disconnect from the terminal. Subsequent operations involving model creation and training will be conducted using Python tools. We do not plan to perform any trading operations in this script.

```
# Requesting a sandbox path
path=os.path.join(mt5.terminal_info().data_path,r'MQL5\Files')
mt5.shutdown()
```

In the following short data loading block, you can observe the usage of functions from all three aforementioned libraries simultaneously. Using the *os.path.join* function, we concatenate the path to the working directory with the name of the training sample file. With the *read_table* function from the *Pandas* library, we read and convert the contents of the *CSV* file into a table. Then we convert the obtained table into a two-dimensional array using the *NumPy* library function.

3. Building the first neural network model in MQL5

```
# Loading a training sample
filename = os.path.join(path, 'study_data.csv')
data = np.asarray( pd.read_table(filename,
                                sep=',',
                                header=None,
                                skipinitialspace=True,
                                encoding='utf-8',
                                float_precision='high',
                                dtype=np.float64,
                                low_memory=False))
```

The actual reading of the CSV file contents and the transformation of rows into a table are performed using the `read_table` function from the *Pandas* library. This function has quite a few parameters for precise configuration of the methods to transform string data into the desired numerical data type. Their full description can be found in the library [documentation](#). We will only describe those we use:

- *filename* gives the name of the file to be read, specifying full or relative path;
- *sep* specifies the data separator used in the file;
- *header* provides row numbers to be used as column names and the beginning of the data, in the absence of headers we specify the value *None*;
- *skipinitialspace* is a boolean parameter that specifies whether to skip spaces after the delimiter;
- *encoding* specifies the type of encoding used;
- *float_precision* determines which converter should be used for floating point values;
- *dtype* specifies the final data type;
- *low_memory* internally processes the file piecemeal, which will result in less memory usage during parsing.

As a result of these operations, all training sample data were loaded into a two-dimensional array object of type *numpy.ndarray* from the *NumPy* library. Among the loaded data, there are elements of the source data and target values. However, for training a neural network, we need to separately feed the source data as input to the network and then compare the obtained output with the target values after the forward pass. It turns out that the input data and targets for the neural network are separated in time and place of use.

Hence, we need to split this data into separate arrays. Let each data row represent an individual data pattern, with the last two elements of the row containing the target points for that pattern. The *shape* function will show the size of our array, which means we can use it to determine the dimensions of the initial data and target values. Only by knowing these dimensions can we copy specific samples into new arrays.

In the block below, we will divide the training sample into 2 tables. In doing so, we are separating only the columns while preserving the entire structure of rows. Thus, we get the initial data in one array and the target values in the other array. The patterns can be mapped to the corresponding target values by row number.

```
# Dividing the training sample into baseline data and targets
inputs=data.shape[1]-2
targets=2
train_data=data[:,0:inputs]
train_target=data[:,inputs:]
```

3. Building the first neural network model in MQL5

Now that we have the training data, we can start building the neural network model. We will create models using the function *Sequential* from the *Keras* library.

```
# Create a neural network model
model = keras.Sequential([...])
```

The *Sequential* model is a linear stack of layers. You can create a *Sequential* model by passing a list of layers to the model's constructor, and you can also add layers using the *add* method.

First of all, our model needs to know what dimensionality of data to expect at the input. In this regard, the first layer of the *Sequential* model must obtain information about the dimensionality of the input data. All subsequent layers perform an automatic dimensionality calculation.

There are several ways to specify the dimensions of the raw data:

- Pass the *input_shape* argument to the first layer.
- Some 2D layers support the specification of the dimensionality of the input data via the *input_dim* argument. Some 3D layers support *input_dim* and *input_length* arguments.
- Use a special type of neural layer for the original *Input* data with the *shape* parameter that specifies the size of the layer.

```
# Create a neural network model
model = keras.Sequential([keras.Input(shape=inputs),
    # Fill the model with a description of the neural layers
    ])
```

We will become familiar with the types of proposed neural layers as we study their architectures. Now let's look at the general principles of building and organizing models.

Once the model is created, you need to prepare it for training and customize the process. This functionality is performed in the *compile* method which has several parameters:

- *optimizer* – an optimizer, can be specified as a string identifier of an existing optimizer or as an instance of the *Optimizer* class;
- *loss* – a loss function, can be specified by a string identifier of an existing loss function, or an eigenfunction;
- *metrics* – a list of metrics that the model should evaluate during training and testing, for example, 'accuracy' could be used for the classification task;
- *loss_weights* – an optional list or dictionary that defines scalar coefficients for weighting the loss contributions of various model outputs;
- *weighted_metrics* – a list of metrics that will be evaluated and weighted during training and testing.

For each parameter, the *Keras* library offers a different list of possible values, but it does not limit the user to the proposed options. For each parameter, there is a possibility to add custom classes and algorithms.

```
model.compile(optimizer='Adam',
              loss='mean_squared_error',
              metrics=['accuracy'])
```

Next, we can start training the created model using the *fit* method, which allows training a model with a fixed number of epochs. This method has parameters to customize the learning process.

- *x* – an array of initial data;

3. Building the first neural network model in MQL5

- *y* – an array of target results;
- *batch_size* – an optional parameter, specifies the number of sets of "source data - target values" pairs before updating the weights matrix;
- *epochs* – a number of epochs of training;
- *verbose* – an optional parameter, specifies the level of detail of training logging: 0 - no messages, 1 - progress bar, 2 - one line per epoch, *auto* - automatic selection;
- *callbacks* – a list of callbacks to apply during training;
- *validation_split* – carries out allocation of a part of the training sample for validation, specified in fractions from 1.0;
- *validation_data* – a separate sample for validating the learning process;
- *shuffle* – a logical value that indicates the need to shuffle the training sample data before the next epoch;
- *class_weight* – an optional dictionary mapping class indices to a weight value used to weight the loss function (only during training);
- *sample_weight* – an optional array of *NumPy* weights for the training sample used to weight the loss function (only during training);
- *initial_epoch* – a training start epoch, can be useful for resuming a previous training cycle;
- *steps_per_epoch* – a total number of packets before declaring one epoch completed and starting the next, by default equal to the training sample size;
- *validation_steps* – the total number of packets from the validation sample before stopping when performing validation at the end of each epoch, defaults to the validation sample size;
- *validation_batch_size* – a number of samples per validation batch;
- *validation_freq* – an integer, specifies the number of training periods before performing a new validation run.

Of course, we will not use the full set of parameters in the first model. I propose to stop at the parameter *callbacks* which sets a callback list. This option provides methods for interacting with the learning process in an interactive way.

Its usage allows configuring the retrieval of real-time information about the training process and managing the process itself. In particular, you can accumulate the average values of indicators for an epoch or save the results of each epoch to a CSV file. You can also monitor training metrics and adjust the learning rate or even stop the training process when the monitored metric stops improving. At the same time, it is possible to add your own callback classes.

I suggest using early stopping in the training procedure if there is no improvement in the error function metric over five epochs.

```
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=5)
history = model.fit(train_data, train_target,
                    epochs=500, batch_size=1000,
                    callbacks=[callback],
                    verbose=2,
                    validation_split=0.2,
                    shuffle=True)
```

After training is complete, save the trained model to a file on the local disk. To do this, let's use the methods of the *Keras* and *os* libraries.

3. Building the first neural network model in MQL5

```
# Saving the learned model
model.save(os.path.join(path, 'model.h5'))
```

For clarity and understanding of the training process, let's plot the dynamics of metric changes during training and validation. Here we will use the methods of the *Matplotlib* library.

```
# Drawing model learning results
plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Validation')
plt.ylabel('$MSE$ $Loss$')
plt.xlabel('$Epochs$')
plt.title('Dynamic of Models train')
plt.legend(loc='upper right')

plt.figure()
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Dynamic of Models train')
plt.legend(loc='lower right')
```

After training, it's necessary to evaluate our model's performance on the test dataset, as before deploying the model in real conditions, we need to understand how it will perform on new data. To do this, we will load a test sample. The data loading procedure is entirely analogous to loading the training dataset, with the only difference being the filename.

```
# Loading a test sample
test_filename = os.path.join(path, 'test_data.csv')
test = np.asarray( pd.read_table(test_filename,
                                sep=',',
                                header=None,
                                skipinitialspace=True,
                                encoding='utf-8',
                                float_precision='high',
                                dtype=np.float64,
                                low_memory=False))
```

After loading the data, we will split the obtained table into source data and target labels, just as we did with the training dataset.

```
# Dividing the test sample into raw data and targets
test_data=test[:,0:inputs]
test_target=test[:,inputs:]
```

We will check the quality of the trained model on the test sample using the *evaluate* method from the *Keras* library. As a result of calling the specified method, we obtain the loss value and metrics on the test dataset. The method has a number of parameters to customize the testing process:

- *x* – an array of initial data of the test sample;
- *y* – an array of test sample targets;
- *batch_size* – a size of the test batch;
- *verbose* – a mode of process logging detailing (0 - no logging, 1 - progress indication);

- *sample_weight* – an optional parameter used to weight the loss function;
- *steps* – a total number of steps to declare the testing process complete;
- *callbacks* – a list of callbacks used in the training process;
- *return_dict* – a boolean variable that defines the format of the method output (*True* = as a "metric-value" dictionary, *False* = as a list).

Most of the above parameters are optional and also have default values. To initiate the testing process, in most cases, it's sufficient to simply provide the data arrays.

```
# Validation of model results on test sample
ltest_loss, test_acc = model.evaluate(test_data, test_target)
```

Finally, let's output the test results to the log and display the previously plotted graphs.

```
# Logging test results
print('Model in test')
print('Test accuracy:', test_acc)
print('Test loss:', test_loss)

# Output of created charts
plt.show()
```

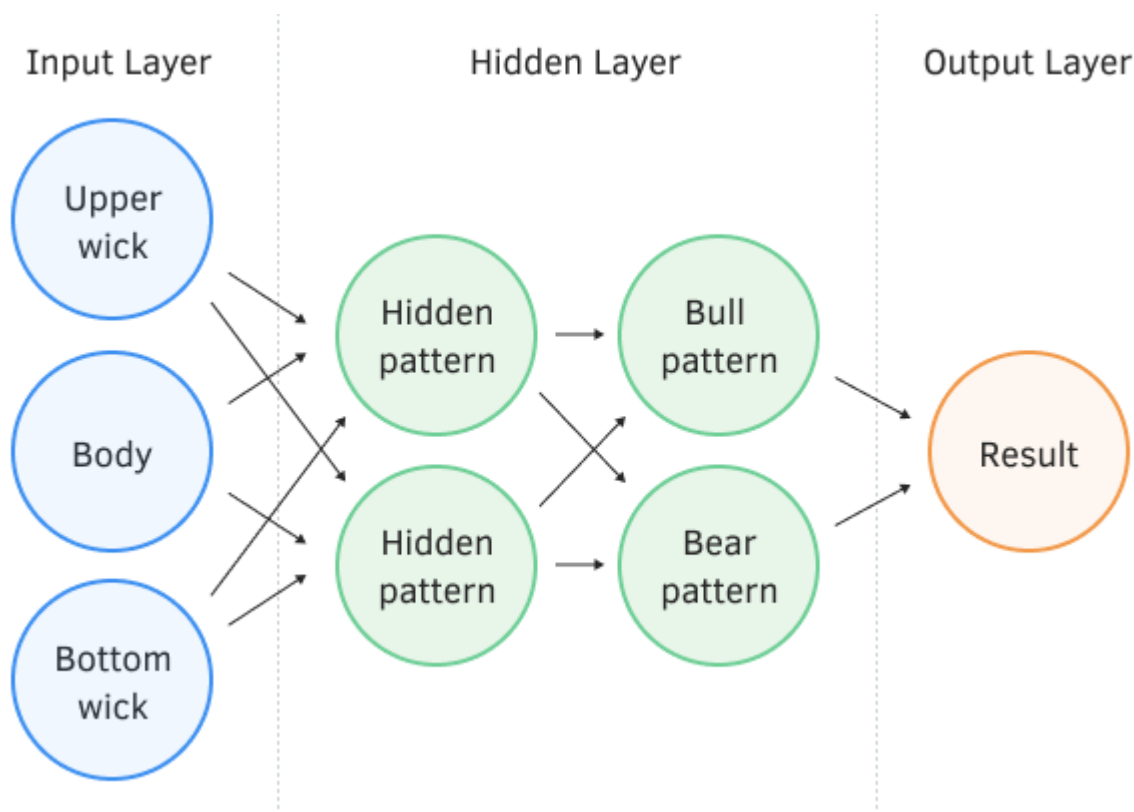
At this point, the basic script template can be considered complete. It's worth noting that attempting to run this script will result in an error. It has nothing to do with errors in template construction. Indeed, we haven't provided a description of our model yet and left the block empty. During the process of exploring various neural network solutions, we will fill in the model architecture description block, allowing us to fully assess the performance of our template.

3.6 Fully connected neural layer

In the previous sections, we established the fundamental logic of organizing and operating a neural network. Now we are moving on to its specific content. We begin the construction of actual working elements – neural layers and their constituent neurons. We will start with constructing a fully connected neural layer.

It was precisely the fully connected neural layers that formed the Perceptron, created by Frank Rosenblatt in 1957. In this architecture, each neuron of the layer has connections with all neurons of the previous layer. Each link has its own weight factor. The figure below shows a perceptron with two hidden fully connected layers. The output neuron can also be represented as a fully connected layer with one neuron. Almost always, at the output of a neural network, we will use at least one fully connected layer.

It can be said that each neuron assesses the overall picture of the input vector and responds to the emergence of a certain pattern. Through the adjustment of various weights, a model is constructed in which each neuron responds to its specific pattern in the input signal. It is this property that makes it possible to use a fully connected neural layer at the output of classifier models.



Perceptron has two hidden fully connected layers.

If we consider a fully connected neural layer from the perspective of vector mathematics, in the framework of which the input values vector represents a certain point in the N -dimensional space (where N is the number of elements in the input vector), then each neuron builds a projection of this point on its own vector. In this case, the activation function decides whether to transmit the signal further, or not.

Here, it's important to pay attention to the displacement of the obtained projection relative to the origin of the coordinates. While the activation function is designed to make decisions within a strict range of input data, this displacement is, for us, a systematic error. To compensate for this bias, an additional constant called **bias** is introduced on each neuron. In practice, this constant is tuned during the training process along with the weights. For this, another element with a constant value of 1 is added to the input signal vector, and the selected weight for this element will play the role of **bias**.

3.6.1 Architecture and principles of implementation of a fully connected layer

When constructing the base class for the neural network and the dynamic array to store pointers to neuron layers, we defined the main methods and interfaces for data exchange between the neural network manager and its components. This is what defines the basic public methods of all our neural layer classes. I suggest summarizing the mentioned sections briefly now. Let's highlight the key class methods we have yet to write and their functionality.

Please note that all neural layer objects must be descendants of the *CObject* base class. This is the fundamental requirement for placing pointers to instances of these objects into the dynamically created array we've designed.

Adhering to the general principles of object organization, in the class constructor, we initialize internal variables and constants. In the destructor, we will perform memory cleanup: deleting all internal instances of various classes and clearing arrays.

3. Building the first neural network model in MQL5

In parameters, the *Init* method receives an instance of the *CLayerDescription* class containing the description of the neural layer to be created. Therefore, this method should be organized to create the entire internal architecture for the proper functioning of our neural layer. We will need to create several arrays to store the data.

It is an array for recording the states at the output of neurons. This array will have a size equal to the number of neurons in our layer.

We will also need an array to store the weights. This will be a matrix, where the size of the first dimension is equal to the number of neurons in our layer, and the size of the second dimension is one more than the size of the input data array. For a fully connected neural layer, the array of input data consists of the output values of the neurons from the previous layer. Consequently, the size of the second dimension will be one element larger than the size of the previous layer. The added element will serve to adjust the *bias*.

For the backward pass, we will need an array to store gradients (deviations of calculated values from reference values at the output of neurons). Its size will correspond to the number of neurons in our layer.

Additionally, depending on the training method, we might need one or two matrices to store accumulated moments. The sizes of these matrices will be equal to the size of the weight matrix.

We will not always update the weights after every iteration of the backward pass. It is possible to update the weights after a full pass of the training sample or based on some batch. We will not store the intermediate states of all neurons and their inputs. On the contrary, after each iteration of the backward pass, we will calculate the necessary change for each weight as if we were updating the weights at every iteration. But instead of changing the weights, we will summarize the resulting deltas into a separate array. If updating is necessary, we will simply take the average delta value over the period and adjust the weights accordingly. For this purpose, we will need another matrix with a size equal to the weight matrix.

For all arrays, we will create a special class called **CBufferType**. It will inherit from the base class **CObject** with the addition of the necessary functionality to organize the operation of the data buffer.

In addition to creating arrays and matrices, we need to fill them with initial values. We will fill all arrays, except the weights, with zeros, and initialize the weight matrix with random values.

In addition to data arrays, our class will also use local variables. We will need to save the activation and optimization parameters of the neurons. We will store the type of optimization method in a variable, and for activation functions, we will create a whole structure of separate classes inheriting from a common base class.

Let me remind you that we are building a universal platform for creating neural networks and their operation in the MetaTrader 5 terminal. We plan to provide users with the ability to utilize multi-threaded computations using the OpenCL technology. All objects in our neural network will operate in the same context. This will reduce the time spent on unnecessary data overload. The actual instance of the class for working with the OpenCL technology will be created in the base neural network class, and a pointer to the created object will be passed to all elements of the neural network. Therefore, all objects that make up the neural network, including our neural layer, should have a method for obtaining the **SetOpenCL** pointer and a variable for storing it.

The forward pass will be organized in the **FeedForward** method. The only parameter of this method will be a pointer to the **CNeuronBase** object of the previous layer of the neural network. We will need the

output states of the neurons from the previous layer, which will form the incoming data stream. To access them, let's create the **GetOutputs** method.

The backward pass, unlike the forward pass, will be divided into several methods:

- **CalcOutputGradient** calculates the error gradient at the output layer of the neural network by reference values.
- **CalcHiddenGradient** skips the error gradient through the hidden layer from output to input. As a result, we will pass the error gradients to the previous layer. To access the array of gradients from the previous layer, we will need a method to access them – **GetGradients**.
- **CalcDeltaWeights** calculates the necessary changes in weights based on the analysis of the last iteration.
- **UpdateWeights** is a method to directly update the weights.

Let's not forget the common for all objects methods of working with files and identification, namely **Save**, **Load**, and **Type**.

In our object detailing, we will focus on the neural layer class and will not create separate objects for each neuron. In fact, there are a number of reasons for this. From what lies on the surface:

- Using the *Softmax* activation function involves working with the entire neural layer.
- Using the *Dropout* and *Layer Normalization* methods requires the processing of the entire neural layer data.
- This approach allows us to efficiently organize multi-threaded computations based on matrix operations.

Let's delve more into matrix operations and see how they allow us to distribute operations across multiple parallel threads. Consider a small example of three elements in the input (vector *Inputs*) and two neurons in the layer. Both neurons have their weight vectors W_1 and W_2 . In this case, each vector of weights contains three elements.

$$\begin{aligned} &Inputs\{i_1, i_2, i_3\} \\ &W_1\{w_{11}, w_{12}, w_{13}\} \\ &W_2\{w_{21}, w_{22}, w_{23}\} \end{aligned}$$

According to the [mathematical model of the neuron](#), we need to element-wise multiply the input data vector with the weight vector, sum up the obtained values, and apply the activation function to them. Essentially, the same process, except for the activation function, is achieved through matrix multiplication.

Matrix multiplication is an operation resulting in a matrix. The elements of the new matrix are obtained by summing the element-wise products of rows from the first matrix with columns from the second matrix.

Thus, to obtain the sum of element-wise products of the input data vector and the weight vector of one of the neurons, it is necessary to multiply the input data row vector by the weight column vector.

$$Inputs * W_1 = [i_1 \quad i_2 \quad i_3] * \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \end{bmatrix} = i_1 w_{11} + i_2 w_{12} + i_3 w_{13} = z_1$$

This rule is applicable to any matrices. The only condition is that the number of columns in the first matrix must be equal to the number of rows in the second matrix. Therefore, we can assemble the weight vectors of all neurons in the layer into a single matrix W , where each column will represent the weight vector of an individual neuron.

$$Inputs * W = \begin{bmatrix} i_1 & i_2 & i_3 \end{bmatrix} * \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} = \begin{bmatrix} z_1 & z_2 \end{bmatrix}$$

You can see that the computation of any element in vector Z is independent of the other elements of the same vector. Accordingly, we can load the matrices of input data and weights into memory and then concurrently compute the values of all elements in the output vector.

We can go even further and load not just a single vector of input data, but a matrix where the rows represent individual states of the system. When working with time series data, each row represents a snapshot of the system's state at a certain moment in time. As a result, we will increase the number of parallel threads of operations and potentially reduce the time to process data.

$$Inputs * W = \begin{bmatrix} i_{1t_1} & i_{2t_1} & i_{3t_1} \\ i_{1t_2} & i_{2t_2} & i_{3t_2} \\ i_{1t_3} & i_{2t_3} & i_{3t_3} \\ i_{1t_4} & i_{2t_4} & i_{3t_4} \end{bmatrix} * \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} = \begin{bmatrix} z_{1t_1} & z_{2t_1} \\ z_{1t_2} & z_{2t_2} \\ z_{1t_3} & z_{2t_3} \\ z_{1t_4} & z_{2t_4} \end{bmatrix}$$

Naturally, we can also use multi-threading to calculate the activation function values for each independent element of matrix Z . An exception might be the use of the [Softmax](#) activation function due to the peculiarities of its computation. However, even in this case, parallelization of computations at different stages of the function calculation is possible.

3.6.2 Creating a neural layer using MQL5 tools

When starting to implement a fully connected neural layer, it should be taken into account that this will be the base class for all subsequent architectural solutions of neural layers. Therefore, we must make it as versatile as possible while also allowing for a potential expansion of functions. At the same time, we should provide the possibility to easily integrate extensions into the existing solution.

Let's start by creating our neural layer base class *CNeuronBase* inherited from the *CObject* class. We define the internal variables of the class:

- *m_cOpenCL* – a pointer to an instance of the class for working with OpenCL technology
- *m_cActivation* – a pointer to an activation function object
- *m_eOptimization* – the type of neuron optimization method during training
- *m_cOutputs* – an array of values at the output of neurons
- *m_cWeights* – an array of weights
- *m_cDeltaWeights* – an array for accumulating outstanding weight updates (cumulative error gradient for each weight since the last update)
- *m_cGradients* – the error gradient at the output of the neural layer as a result of the last iteration of the backward pass

3. Building the first neural network model in MQL5

- *m_cMomentum* – unlike other variables, this will be an array of two elements for recording pointers to arrays of accumulated moments

To facilitate access to variables from derived classes, all variables will be declared in a block called *protected*.

In the class constructor, we initialize the above variables with default parameters. I have specified *Adam* and *Swish* optimization method as an activation function but you can choose your preferred optimization method and activation function. We will leave the pointer to the class working with *OpenCL* empty and create instances for all other classes used.

```
CNeuronBase::CNeuronBase(void)    : m_eOptimization(Adam)
{
    m_cOpenCL = NULL;
    m_cActivation = new CActivationSwish();
    m_cOutputs = new CBufferType();
    m_cWeights = new CBufferType();
    m_cDeltaWeights = new CBufferType();
    m_cGradients = new CBufferType();
    m_cMomentum[0] = new CBufferType();
    m_cMomentum[1] = new CBufferType();
}
```

We immediately create a class destructor so we don't forget about memory cleanup after the class finishes its work.

```
CNeuronBase::~CNeuronBase(void)
{
    if(!m_cActivation)
        delete m_cActivation;
    if(!m_cOutputs)
        delete m_cOutputs;
    if(!m_cWeights)
        delete m_cWeights;
    if(!m_cDeltaWeights)
        delete m_cDeltaWeights;
    if(!m_cGradients)
        delete m_cGradients;
    if(!m_cMomentum[0])
        delete m_cMomentum[0];
    if(!m_cMomentum[1])
        delete m_cMomentum[1];
}
```

Next, we create a neural layer initialization method. In the parameters, the method receives an instance of the *CLayerDescription* class with a description of the layer to be created. To avoid getting lost in the intricacies of the method algorithm, I suggest breaking it down into separate logical blocks

The method starts with a block in which we check input parameters. First, we check the validity of the pointer to the object. Then we check the type of the layer being created and the number of neurons in the layer: each layer should have at least one neuron, because from the logical perspective of constructing a neural network, a layer without neurons blocks the passage of the signal and paralyzes the entire network. Note that when checking the type of the created layer, we use the virtual method *Type* and not the constant *defNeuronBase* which it returns. This is a very important point for future

class inheritance. The fact is that when using a constant, calling such a method for descendant classes would always return *false* when trying to create a layer other than the base one. Using a virtual method allows us to obtain a constant identifier of the final derived class, and the check will yield a true comparison result between the specified type of neural layer and the object being created.

```
bool CNeuronBase::Init(const CLayerDescription *desc)
{
    //--- source data control block
    if(!desc || desc.type != Type() || desc.count <= 0)
        return false;
```

In the next block, we will verify the validity of previously created buffers for recording the data flow coming out of the neural layer and the gradient to them (if necessary, we create new instances of the class). We initialize arrays with zero values.

```
//--- creating a results buffer
if(!m_cOutputs)
    if(!(m_cOutputs = new CBufferType()))
        return false;
if(!m_cOutputs.BufferInit(1, desc.count, 0))
    return false;
//--- creating error gradient buffer
if(!m_cGradients)
    if(!(m_cGradients = new CBufferType()))
        return false;
if(!m_cGradients.BufferInit(1, desc.count, 0))
    return false;
```

After that, we check the number of elements of the input signal. In the case of using the neural layer as an array of incoming signals, we will not have preceding neural layers, and other data buffers will not be required. We can remove them without any problem and clear the memory. Then we check the validity of the pointer to the object in *m_cOpenCL* and, if the result is positive, we create a copy of the data buffer in the *OpenCL* context.

```
//--- removing unused features for the source data layer
if(desc.window <= 0)
{
    if(m_cActivation)
        delete m_cActivation;
    if(m_cWeights)
        delete m_cWeights;
    if(m_cDeltaWeights)
        delete m_cDeltaWeights;
    if(m_cMomenum[0])
        delete m_cMomenum[0];
    if(m_cMomenum[1])
        delete m_cMomenum[1];
    if(m_cOpenCL)
        if(!m_cOutputs.BufferCreate(m_cOpenCL))
            return false;
    m_eOptimization = desc.optimization;
    return true;
}
```

3. Building the first neural network model in MQL5

Further method code is executed only if there are previous neural layers. Let's create and initialize an instance of the activation function method. We have moved this process to a separate method, *SetActivation*, which we are now simply calling. We will examine the algorithm of the *SetActivation* method a bit later.

```
//--- initializing an activation function object
VECTOR ar_temp = desc.activation_params;
if(!SetActivation(desc.activation, ar_temp))
    return false;
```

The next step is to initialize the matrix of weights. We determine the number of elements in the matrix and initialize it with **random values** using the Xavier method. In the case of using LReLU as an activation function, we will use the He method.

```
//--- initializing a weight matrix object
if(!m_cWeights)
    if(!(m_cWeights = new CBufferType()))
        return false;
if(!m_cWeights.BufferInit(desc.count, desc.window + 1, 0))
    return false;
double weights[];
double sigma = (desc.activation == AF_LRELU ?
    2.0 / (double)(MathPow(1 + desc.activation_params[0], 2)
        * desc.window) :
    1.0 / (double)desc.window);
if(!MathRandomNormal(0, MathSqrt(sigma), m_cWeights.Total(), weights))
    return false;
for(uint i = 0; i < m_cWeights.Total(); i++)
    if(!m_cWeights.m_mMatrix.Flat(i, (TYPE)weights[i]))
        return false;
```

We still need to initialize the buffers for deltas and moments. The size of the buffers will be equal to the size of the weight matrix, and we will initialize them with zero values. Remember that not all optimization methods use the moment matrices in the same way. Therefore, we will initialize the matrices of moments depending on the optimization method. We will clear and delete unnecessary arrays to free up memory for productive use.

```
//--- initialization of the gradient accumulation object at the weight matrix level
if(!m_cDeltaWeights)
    if(!(m_cDeltaWeights = new CBufferType()))
        return false;
if(!m_cDeltaWeights.BufferInit(desc.count, desc.window + 1, 0))
    return false;
//--- initializing moment objects
switch(desc.optimization)
{
    case None:
    case SGD:
        for(int i = 0; i < 2; i++)
            if(m_cMomentum[i])
                delete m_cMomentum[i];
        break;
```



```

    case MOMENTUM:
    case AdaGrad:
    case RMSProp:
        if(!m_cMomenum[0])
            if(!(m_cMomenum[0] = new CBufferType()))
                return false;
        if(!m_cMomenum[0].BufferInit(desc.count, desc.window + 1, 0))
            return false;
        if(m_cMomenum[1])
            delete m_cMomenum[1];
        break;

    case AdaDelta:
    case Adam:
        for(int i = 0; i < 2; i++)
        {
            if(!m_cMomenum[i])
                if(!(m_cMomenum[i] = new CBufferType()))
                    return(false);
            if(!m_cMomenum[i].BufferInit(desc.count, desc.window + 1, 0))
                return false;
        }
        break;

    default:
        return false;
        break;
}

//--- saving parameter optimization method
m_eOptimization = desc.optimization;
return true;
}

```

At the end of the method we save the specified weight optimization method.

The *SetOpenCL* method is used to save a pointer to the object of work with the OpenCL context and looks simpler than the initialization method. However, unlike all previously considered methods, we do not terminate the method operation upon receiving an invalid pointer to the object. This is because we do not introduce a flag for the use of OpenCL technology in every neural layer class. Instead, we use a single flag in the base class of the neural network. In turn, to check the use of the technology inside the class, we can verify the validity of the pointer in the *m_cOpenCL* variable.

It should be noted that all objects of the neural network operate within a single OpenCL context. All objects are provided with a pointer to the same object of the *CMyOpenCL* class. With such an approach, deleting an instance of the class in one of the neural network objects will invalidate the pointer in all objects that use it. The flag may not correspond to the current state of the pointer. Additionally, in the case of disabling the use of technology, we leave the possibility of specifying an empty value of the pointer to the object.

Therefore, the code of our method can be conditionally divided into two parts. The first part of the code will be executed when receiving an invalid pointer to the object. In this case, we need to clear all previously created data buffers in the OpenCL context.

3. Building the first neural network model in MQL5

```
bool CNeuronBase::SetOpenCL(CMyOpenCL *opencl)
{
    if(!opencl)
    {
        if(m_cOutputs)
            m_cOutputs.BufferFree();
        if(m_cGradients)
            m_cGradients.BufferFree();
        if(m_cWeights)
            m_cWeights.BufferFree();
        if(m_cDeltaWeights)
            m_cDeltaWeights.BufferFree();
        for(int i = 0; i < 2; i++)
        {
            if(m_cMomenum[i])
                m_cMomenum[i].BufferFree();
        }
        if(m_cActivation)
            m_cActivation.SetOpenCL(m_cOpenCL, Rows(), Cols());
        m_cOpenCL = opencl;
        return true;
    }
}
```

The second part of the method will be executed when receiving a valid pointer to the object working with the OpenCL context. Here, we organize the creation of new data buffers in the specified OpenCL context for all objects of the current class.

```
if(m_cOpenCL)
    delete m_cOpenCL;
m_cOpenCL = opencl;
if(m_cOutputs)
    m_cOutputs.BufferCreate(opencl);
if(m_cGradients)
    m_cGradients.BufferCreate(opencl);
if(m_cWeights)
    m_cWeights.BufferCreate(opencl);
if(m_cDeltaWeights)
    m_cDeltaWeights.BufferCreate(opencl);
for(int i = 0; i < 2; i++)
{
    if(m_cMomenum[i])
        m_cMomenum[i].BufferCreate(opencl);
}

if(m_cActivation)
    m_cActivation.SetOpenCL(m_cOpenCL, Rows(), Cols());
//---
return(!m_cOpenCL);
}
```

Earlier, we talked about isolating the activation function initialization procedure into a separate method. I suggest examining this method to complete the description of the new object initialization process. This is one of the few methods where we don't organize a block for data verification. Verification of the

activation function parameters is not feasible due to the variance in the range of permissible values when using different functions. In most cases, the range of their values is limited only by common sense and the architectural requirements of the model.

As for the choice of the activation function, it exists implicitly, in the form of a list of allowable values. But even if the user inserts a value not from the enumeration, we will create activation function objects within the body of the *switch* statement. This means that we will have implicit control over the type of the activation function, and if the specified value is absent in the selection function, we will create a base class without an activation function.

The need to create a base class is due to maintaining the functionality of the class without using an activation function in standard mode. As you will see a little later, in some cases we will use neural layers without activation functions.

```
bool CNeuronBase::SetActivation(ENUM_ACTIVATION_FUNCTION function, VECTOR &params)
{
    if(m_cActivation)
        delete m_cActivation;

    switch(function)
    {
        case AF_LINEAR:
            if(!(m_cActivation = new CActivationLine()))
                return false;
            break;

        case AF_SIGMOID:
            if(!(m_cActivation = new CActivationSigmoid()))
                return false;
            break;

        case AF_LRELU:
            if(!(m_cActivation = new CActivationLReLU()))
                return false;
            break;

        case AF_TANH:
            if(!(m_cActivation = new CActivationTANH()))
                return false;
            break;

        case AF_SOFTMAX:
            if(!(m_cActivation = new CActivationSoftMAX()))
                return false;
            break;

        case AF_SWISH:
            if(!(m_cActivation = new CActivationSwish()))
                return false;
            break;
    }
}
```

```

default:
    if(!m_cActivation = new CActivation())
        return false;
    break;
}

```

After creating an instance of the required activation function object, we pass the function parameters and a pointer to the OpenCL context object to the new object.

```

if(!m_cActivation.Init(params[0], params[1]))
    return false;
m_cActivation.SetOpenCL(m_cOpenCL, m_cOutputs.Rows(), m_cOutputs.Cols());
return true;
}

```

Feed-forward operations will be implemented in the **FeedForward** method. In the parameters, the method receives a pointer to the object of the previous layer. Since we are planning to build the classes of all neural layers based on one base class, we can use the class of the base neural layer in the method parameters to get a pointer to the previous layer of any type. The use of virtual access methods to the internal objects of the class allows you to build a universal interface without being tied to a specific type of neural layer.

At the beginning of the method, we check the validity of pointers to all objects used in the method. This is our initial data: the pointer to the previous layer received in the parameters, as well as the buffer of the neurons' output states contained in it. Together with them, we will check the pointers to the weight matrix and the buffer for recording the results of the forward pass of the current layer, that is, the buffer of the output states of the neurons of the current layer. Again, it's a good practice to check the pointer to the instance of the class for calculating the values of the activation function.

```

bool CNeuronBase::FeedForward(CNeuronBase * prevLayer)
{
    //--- control block
    if(!prevLayer || !m_cOutputs || !m_cWeights ||
        !prevLayer.GetOutputs() || !m_cActivation)
        return false;
    CBufferType* input_data = prevLayer.GetOutputs();
}

```

Then we check the pointer to the object working with OpenCL. If the pointer is valid, we move on to the block that is using this technology. We will talk about it a little later when considering the organization of the process of parallel computing. In case of an invalid pointer to an object or its absence, we move on to the block of calculations using standard MQL5 tools. Here, we will first check the consistency of matrix sizes and reformat the source data matrix into a vector, adding a unit element for the *bias*. We will perform the operation of matrix multiplication by the weight matrix. The result will be written to the outgoing stream buffer. Before exiting the method, do not forget to compute the values of the activation function at the output of the neural layer.

3. Building the first neural network model in MQL5

```
//---branching of the algorithm depending on the device for performing operations
if(!m_cOpenCL)
{
    if(m_cWeights.Cols() != (input_data.Total() + 1))
        return false;
    //---
    MATRIX m = input_data.m_mMatrix;
    if(!m.Reshape(1, input_data.Total() + 1))
        return false;
    m[0, m.Cols() - 1] = 1;
    m_cOutputs.m_mMatrix = m.MatMul(m_cWeights.m_mMatrix.Transpose());
}

else
{
    //--- Here is the code for accessing the OpenCL program
    return false;
}
//---
return m_cActivation.Activation(m_cOutputs);
}
```

The forward pass is followed by the [backpropagation pass](#). We break down this neural network training procedure into component parts and create four methods:

- **CalcOutputGradient** for calculating the error gradient at the output of the neural network,
- **CalcHiddenGradient** to enable the gradient propagation through the hidden layer,
- **CalcDeltaWeights** for calculating the necessary weight adjustments,
- **UpdateWeights** for updating the weight matrix.

We will move along the data flow path and consider the algorithm of each method.

In the process of supervised learning, after the forward pass, the calculated output values of the neural network are compared with the target values. The deviation on each neuron of the output layer is determined at this moment. We perform this operation in the **CalcOutputGradient** method. The algorithm of this method is quite simple: the method receives an array of target values and the type of the loss function used as parameters. At the beginning of the method, we will validate the pointers to the used objects as well as ensure the compatibility of the array sizes.

```
bool CNeuronBase::CalcOutputGradient(CBufferType * target, ENUM_LOSS_FUNCTION loss)
{
    //--- control block
    if(!target || !m_cOutputs || !m_cGradients ||
        target.Total() < m_cOutputs.Total() ||
        m_cGradients.Total() < m_cOutputs.Total())
        return false;
}
```

Next, similar to the feed-forward method, we will create a branching in the algorithm depending on the device used for calculations. The algorithm using the OpenCL technology will be discussed in the next chapter, and now let's look at the process construction using MQL5.

Let's take a look at the process of computing the error gradient at the output of the neural network. At first glance, we should move in the direction of minimizing the error for each neuron. In other words,

calculate the difference between the reference and calculated values and minimize this difference. In this case, we get a linear dependence of the error and the gradient. This is true when using [mean absolute error](#) as a loss function, with all the resulting advantages and disadvantages.

When we were talking about the [loss function](#), we considered other options and discussed their advantages and disadvantages. But how can we take advantage of them? The answer here is pretty simple. One should consider the loss function and the trainable model as a single complex function. In this case, we should minimize not the deviation for each neuron of the output layer, but directly the value of the loss function. Just as when propagating the error gradient through the neural network, we calculate the derivative of the loss function and multiply it by the deviation of the loss function value from zero. Moreover, for MAE and MSE we can consider only the derivative of the loss function as the error and disregard multiplying it by the value of the loss function since this linear scaling will be compensated by the learning rate, while when using cross-entropy, we are compelled to multiply it by the value of the loss function. The reason is that if the target and calculated values are equal, the loss function will give 0, and its derivative will be equal to -1 . If we don't multiply the derivative by the error, we will continue adjusting the model parameters in the absence of an error.

In this case, it is not at all necessary to fully calculate the value of the loss function. Cross-entropy is commonly used as the loss function in classification tasks. Therefore, as target values, we expect to obtain a vector in which only one element will be set to one, while all others will be zero. For zero values, the derivative will also be zero, and multiplication by 1 doesn't change the result. Therefore, it is enough for us to multiply the derivative by the logarithm of the calculated value. It is the logarithm of 1 that will give 0, indicating that there is no error.

Taking into account the above, to calculate the corresponding error gradient at the model's output, we will use a *switch* statement to create a branching process based on the employed loss function. In case the specified loss function is not present, we will calculate the simple deviation of the calculated results from the target values.

```
//---branching of the algorithm depending on the device for performing operations
if(!m_cOpenCL)
{
    switch(loss)
    {
        case LOSS_MAE:
            m_cGradients.m_mMatrix = target.m_mMatrix - m_cOutputs.m_mMatrix;
            break;
        case LOSS_MSE:
            m_cGradients.m_mMatrix = (target.m_mMatrix - m_cOutputs.m_mMatrix) * 2;
            break;
        case LOSS_CCE:
            m_cGradients.m_mMatrix = target.m_mMatrix /
            (m_cOutputs.m_mMatrix + FLT_MIN) * MathLog(m_cOutputs.m_mMatrix) * (-1);
            break;
```

```

        case LOSS_BCE:
            m_cGradients.m_mMatrix = (target.m_mMatrix-m_cOutputs.m_mMatrix) /
                (MathPow(m_cOutputs.m_mMatrix, 2) - m_cOutputs.m_mMatrix + FLT_MIN);
            break;
        default:
            m_cGradients.m_mMatrix = target.m_mMatrix - m_cOutputs.m_mMatrix;
            break;
    }
}
else
    return false;
//---
return true;
}

```

After obtaining the error at the neural network output, it's necessary to determine the influence of each neuron in our network on this error. To achieve this, we need to propagate the error gradient layer by layer, reaching every neuron. The responsibility for organizing the loop that iterates through the layers of the neural network lies with the network manager, that is, [the neural network base class CNet](#). Now we will examine the organization of the process within a single neural layer.

In the parameters, the **CalcHiddenGradient** method receives a pointer to the previous layer of the neural network. We will need it to write the transmitted error gradient. In the previous method, we determined the error at the neuron output, but the neuron output value depends on the activation function. To determine the influence of each element of the input data on the final result, it's necessary to exclude the influence of the activation function on the error. To achieve this, we will adjust the error gradient using the derivative of the activation function. This operation, like the computation of the activation function itself, is implemented in a separate class.

```

bool CNeuronBase::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    //--- adjusting the incoming gradient to the derivative of the activation function
    if(!m_cActivation.Derivative(m_cGradients))
        return false;
}

```

Next comes the block in which we check pointers of used objects. First, we validate the received pointer to the previous layer. Then, we extract and validate the pointers to the buffers of results and gradients from the previous layer. We also verify the consistency of the number of elements in the specified buffers. Additionally, we check the presence of a sufficient number of elements in the weight matrix. Such a number of preventive checks are necessary for the stable operation of the method and to prevent potential errors when accessing data arrays.

```

//--- checking the buffers of the previous layer
if(!prevLayer)
    return false;
CBufferType *input_data = prevLayer.GetOutputs();
CBufferType *input_gradient = prevLayer.GetGradients();
if(!input_data || !input_gradient ||
    input_data.Total() != input_gradient.Total())
    return false;
//--- checking the correspondence between the size of the source data buffer and the
if(!m_cWeights || m_cWeights.Cols() != (input_data.Total() + 1))
    return false;

```

After successfully passing all the checks, we proceed directly to the computational part. Let me remind you that the derivative of the product of a variable and a constant is a constant. In this case, the derivative with respect to the neuron is the corresponding weight. Consequently, the neuron influence on the result is the product of the error gradient at the output of the function and the corresponding weight. We will calculate the sum of such products for each neuron in the previous layer. We will write the obtained values into the corresponding cell of the gradient buffer of the previous layer.

As in the methods described above, we carry out the separation of the algorithm depending on the computing device used. We will get acquainted with the algorithm for implementing multi-threaded calculations a little later. Let's now consider the implementation of the algorithm using MQL5 tools. As mentioned earlier, we need to calculate the sum of products of error gradients from neurons dependent on a given neuron and their corresponding weights. Performing this operation is easily accomplished using matrix multiplication. In this case, it suffices to multiply the error gradient matrix by the matrix of weights. We will store the result of the operation in a local matrix.

We cannot immediately write the result of the operation to the error gradient matrix of the previous layer. If you look at the forward pass method, you will see how we added the *bias* element. Accordingly, when multiplying matrices, we will get the result, taking into account the error on the bias element. However, the previous layer does not expect this value, and the size of the matrix of gradients is smaller. Therefore, we will first resize the matrix obtained from the multiplication operation to the required dimensions, and then transfer its values to the gradient matrix of the previous layer.

Note that in this method, we do not adjust the gradient obtained at the output of the previous layer by the derivative of the activation function of neurons in the previous layer, as we did with a similar operation at the beginning of this method. Therefore, if the previous layer is the hidden layer of our network, then the first thing that will be done when calling the considered method on the lower layer is to adjust the gradient for the derivative of the activation function. Doubling the operation will lead to errors.

```
//--- branching of the algorithm depending on the device for performing operations
if(!m_cOpenCL)
{
    MATRIX grad = m_cGradients.m_mMatrix.MatMul(m_cWeights.m_mMatrix);
    if(!grad.Reshape(input_data.Rows(), input_data.Cols()))
        return false;
    input_gradient.m_mMatrix = grad;
}
else
    return false;
//---
return true;
}
```

We now have a calculated error gradient on each neuron in our network. There is enough data to update the weights. However, as we know, the weights are not always updated after each iteration of the backpropagation pass. Therefore, we separated the process of updating the weight matrix into two methods. In the first one, we will calculate the error gradient for each weight similarly to how we calculated the error gradient for the neuron in the previous layer. In the second one, we will adjust the weight matrix.

We will calculate the value of the error gradient for the weight matrix in the **CalcDeltaWeights** method. In the parameters of the method, similar to the previous one, there will be a pointer to the preceding

layer of the neural network, but now we will use not the gradient buffer from it, but the array of output values.

Similar to the previously discussed methods, this method starts with a block of checks. It is followed by a block of calculations.

```
bool CNeuronBase::CalcDeltaWeights(CNeuronBase *prevLayer, bool read);
{
    //--- control block
    if(!prevLayer || !m_cDeltaWeights || !m_cGradients)
        return false;
    CBufferType *Inputs = prevLayer.GetOutputs();
    if(!Inputs)
        return false;
```

In the previous method, we have already adjusted the gradient for the derivative of the activation function. Therefore, we will skip this iteration and proceed directly to the calculation of the gradient on the weights. Here, as in other methods, there is a branching of the algorithm based on the computation device. In the MQL5 block, similarly to the previous method, we will employ matrix multiplication, because, in essence, both methods perform a similar operation only for different matrices. But there are a few differences here.

First, in the previous method, we removed the *bias* element. However, in this case, we need to add a unitary element to the vector of the previous layer results in order to determine the error gradient on the corresponding weight.

Second, earlier we multiplied the matrix of gradients by the matrix of weights. Now we multiply the transposed matrix of error gradients by the vector of the previous layer results with the bias element.

In addition, we were overwriting the error gradient of the previous layer, but for the weight gradient, we will sum them up, thereby accumulating the error gradient over the entire period between weight update operations.

```
//--- branching of the algorithm depending on the device for performing operations
if(!m_cOpenCL)
{
    MATRIX m = Inputs.m_mMatrix;
    if(!m.Reshape(1, Inputs.Total() + 1))
        return false;
    m[0, Inputs.Total()] = 1;
    m = m_cGradients.m_mMatrix.Transpose().MatMul(m);
    m_cDeltaWeights.m_mMatrix += m;
}
else
    return false;
//---
return true;
}
```

At the conclusion of the backpropagation process, we need to adjust the weight matrix. To perform this functionality, our class provides the **UpdateWeights** method. However, let's not forget that we have different options available for choosing the optimization method. The question was resolved using a simple and intuitive approach. The public method for updating the weights provides a dispatcher function to select the optimization method based on the user's choice. The actual process of adjusting

the weight matrix is implemented in separate methods, with one method for each optimization method version.

```
bool CNeuronBase::UpdateWeights(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda)
{
    //--- control block
    if(!m_cDeltaWeights || !m_cWeights ||
        m_cWeights.Total() < m_cDeltaWeights.Total() || batch_size <= 0)
        return false;
    //---
    bool result = false;
    switch(m_eOptimization)
    {
        case None:
            result = true;
            break;

        case SGD:
            result = SGDUpdate(batch_size, learningRate, Lambda);
            break;

        case MOMENTUM:
            result = MomentumUpdate(batch_size, learningRate, Beta, Lambda);
            break;

        case AdaGrad:
            result = AdaGradUpdate(batch_size, learningRate, Lambda);
            break;

        case RMSProp:
            result = RMSPropUpdate(batch_size, learningRate, Beta, Lambda);
            break;

        case AdaDelta:
            result = AdaDeltaUpdate(batch_size, Beta, Lambda);
            break;

        case Adam:
            result = AdamUpdate(batch_size, learningRate, Beta, Lambda);
            break;
    }
    //---
    return result;
}
```

Algorithms for each of the [weight optimization methods](#) have already been presented earlier, while we considered their features. We will not duplicate them here, but we will implement them in the **protected** block in our base class of the neural layer.

We have already discussed the implementation of feed-forward and backpropagation operations in a fully connected neural layer. However, we will not re-train the neural network at each launch. Therefore, we need methods for working with files: writing and reading data from the state of the neural layer. We should be resource-efficient, so let's consider which information we need to save. The general rule is to save a minimum amount of information, but it should be sufficient for a quick startup and the functioning of the class without interrupting the process. Let's take a look at internal class variables and critically evaluate the need to save their contents to a file.

- *m_cOpenCL* – a pointer to an instance of the class for working with OpenCL technology, which is responsible for a separate functionality, but does not contain additional information. *Not to be written to file.*
- *m_cActivation* – a pointer to an activation function object. The activation function type is set by the user when constructing a neural network. Using a different activation function can lead to distortion of the results across the entire network. *Save.*
- *m_eOptimization* – a type of neuron optimization method during training, which is specified by the user when constructing a neural network. Influences the learning process. *Save.*
- *m_cOutputs* – an array of neuron output values. The number of elements is set by the neural network architect. The content is overwritten on every forward pass. *It's sufficient to save the number of neurons in the layer and not save the entire array.*
- *m_cWeights* – a weight matrix. The value of the elements is formed in the process of training the neural network. *Save.*
- *m_cDeltaWeights* – a matrix for accumulating failed weight updates (cumulative error gradient for each weight since the last update). Values are accumulated between weights matrix updates and reset to zero after weights adjustments. The size of the array is equal to the weight matrix. *Not to be written to a file.*
- *m_cGradients* – the error gradient at the output of the neural layer as a result of the last iteration of the backward pass. The content is overwritten on every backward pass. The size of the array is equal to the buffer of the output signal. *Not to be written to a file.*
- *m_cMomentum* — unlike other variables, this will be an array of two elements for writing pointers to moment accumulation arrays. The use of buffers depends on the optimization method. The content is accumulated during the training of the neural network. *Save.*

After determining the data to be written to the file, let's proceed to create the file writing method **Save**. This virtual method exists in all descendant classes of the *CObject* class. In the parameters, the method receives the handle of the file to be written.

In the body of the method, we first check the received handle and the validity of the pointer to the result buffer of the neural layer. As we remember, a neural layer can be used with both full functionality and not. When using an object as a layer of input data, we deleted all buffers except for the input data buffer. Therefore, the presence of this buffer is mandatory for the neural layer. If any of the checks fail, we exit the method with a result of false.

Next, we write the type of the neural layer and the size of the result buffer to the file. At the same time, do not forget to check the results of the operations.

```
bool CNeuronBase::Save(const int file_handle)
{
    //--- control block
    if(file_handle == INVALID_HANDLE)
        return false;
    //--- writing result buffer data
    if(!m_cOutputs)
        return false;
    if(FileWriteInteger(file_handle, Type()) <= 0 ||
       FileWriteInteger(file_handle, m_cOutputs.Total()) <= 0)
        return false;
```

After successfully writing the size of the result buffer, we check the validity of the pointers to the activation function objects and the weight matrices. In the absence of at least one object, we consider

3. Building the first neural network model in MQL5

the current neural layer to be the initial data layer. To confirm this, we write 1 as a flag to indicate the preservation of the input data layer in the file. Otherwise, we save 0, which will indicate the preservation of the full-functionality neural layer.

```
//--- checking and writing the source data layer flag
if(!m_cActivation || !m_cWeights)
{
    if(FileWriteInteger(file_handle, 1) <= 0)
        return false;
    return true;
}
if(FileWriteInteger(file_handle, 0) <= 0)
    return false;
```

Then, using the optimization method, we determine the number of moments required for recording to the buffer.

```
int momentums = 0;
switch(m_eOptimization)
{
    case SGD:
        momentums = 0;
        break;
    case MOMENTUM:
    case AdaGrad:
    case RMSProp:
        momentums = 1;
        break;
    case AdaDelta:
    case Adam:
        momentums = 2;
        break;
    default:
        return false;
        break;
}
```

Immediately, we organize a loop to validate the pointers to the momentum buffers.

```
for(int i = 0; i < momentums; i++)
    if(!m_cMomenum[i])
        return false;
```

After the block of checks, there are operations for directly writing data to the file. First, we save the values of variables, and then we call the file writing methods for the objects that need to be saved.

3. Building the first neural network model in MQL5

```
//--- saving a matrix of weighting coefficients, moments, and activation functions
if(FileWriteInteger(file_handle, (int)m_eOptimization) <= 0 ||
   FileWriteInteger(file_handle, momentums) <= 0)
    return false;
if(!m_cWeights.Save(file_handle) || !m_cActivation.Save(file_handle))
    return false;
for(int i = 0; i < momentums; i++)
    if(!m_cMomentum[i].Save(file_handle))
        return false;
//---
return true;
}
```

As seen from the provided code, we simply skip objects that do not need to be saved. However, this approach is not applicable when loading data from a file, as even skipped objects are necessary for the normal functioning of the neural layer. Therefore, the data loading method **Load** must be supplemented with a missing object initialization block. Let's see how it is implemented.

Just like when writing to a file, the method also receives a file handle for data in its parameters. Therefore, at the beginning of the method, we validate the received file handle.

```
bool CNeuronBase::Load(const int file_handle)
{
    //--- control block
    if(file_handle == INVALID_HANDLE)
        return false;
}
```

Reading data from the file should be done in precise accordance with the sequence of data writing. First, we saved the type of neural layer and the number of elements in the buffer in the results buffer. The type of the neural layer will be read by the method of the top-level object (dynamic array of neural layers) to create the required neural layer. In the body of this method, we will read the number of elements in the result buffer and initialize a buffer of the corresponding size.

```
//--- loading result buffer
if(!m_cOutputs)
    if(!(m_cOutputs = new CBufferType()))
        return false;
int outputs = FileReadInteger(file_handle);
if(!m_cOutputs.BufferInit(1, outputs, 0))
    return false;
```

Immediately create a gradient buffer of the same size.

```
//--- creating error gradient buffer
if(!m_cGradients)
    if(!(m_cGradients = new CBufferType()))
        return false;
if(!m_cGradients.BufferInit(1, outputs, 0))
    return false;
```

Next, we check the flag for loading the input data neural layer. In the case of loading it, we delete unused objects and exit the method with a positive result.

3. Building the first neural network model in MQL5

```
//--- checking the source data layer flag
int input_layer = FileReadInteger(file_handle);
if(input_layer == 1)
{
    if(m_cActivation)
        delete m_cActivation;
    if(m_cWeights)
        delete m_cWeights;
    if(m_cDeltaWeights)
        delete m_cDeltaWeights;
    if(m_cMomenum[0])
        delete m_cMomenum[0];
    if(m_cMomenum[1])
        delete m_cMomenum[1];
    if(m_cOpenCL)
        if(!m_cOutputs.BufferCreate(m_cOpenCL))
            return false;
    m_eOptimization = None;
    return true;
}
```

Further code is executed only when loading a fully functional neural layer. At the beginning of this block, we read the optimization method from the file and the number of used momentum buffers.

```
m_eOptimization = (ENUM_OPTIMIZATION)FileReadInteger(file_handle);
int momentums = FileReadInteger(file_handle);
```

After that, we check the pointer to the weights matrix object. If necessary, we will create a new instance of the object and immediately call the data buffer loading method.

```
//--- creating objects before loading data
if(!m_cWeights)
    if(!(m_cWeights = new CBufferType()))
        return false;
//--- loading data from file
if(!m_cWeights.Load(file_handle))
    return false;
```

Then, we read the type of the activation function from the file and initialize an instance of the corresponding class using the *SetActivation* method. The activation function parameters will be loaded by calling the method with the same name for loading data from the activation function object.

```
//--- activation function
if(FileReadInteger(file_handle) != defActivation)
    return false;
ENUM_ACTIVATION_FUNCTION activation =
    (ENUM_ACTIVATION_FUNCTION)FileReadInteger(file_handle);
if(!SetActivation(activation, VECTOR::Zeros(2)))
    return false;
if(!m_cActivation.Load(file_handle))
    return false;
```

Similarly, we will load the data of the momentum buffers.

3. Building the first neural network model in MQL5

```
//---
for(int i = 0; i < momentums; i++)
{
    if(!m_cMomentum[i])
        if(!m_cMomentum[i] = new CBufferType())
            return false;
    if(!m_cMomentum[i].Load(file_handle))
        return false;
}
```

After loading the data, we initialize the *m_cDeltaWeights* buffer. The buffer will be initialized with zero values. In this case, the buffer size is equal to the number of elements in the weights matrix.

First, check the pointer to the object and create a new one if necessary. Then, we will write 0 into all elements of the buffer.

```
//--- initializing remaining buffers
if(!m_cDeltaWeights)
    if(!(m_cDeltaWeights = new CBufferType()))
        return false;
if(!m_cDeltaWeights.BufferInit(m_cWeights.m_mMatrix.Rows(),
                                m_cWeights.m_mMatrix.Cols(), 0))

    return false;
```

At the end of the method, we pass the current pointer *m_cOpenCL* to all internal objects. Here, we are not adding a check for the validity of the pointer. Since all objects of the neural network work within the same OpenCL context, we pass even an invalid pointer to the objects.

```
//--- passing a pointer to the OpenCL context to objects
SetOpenCL(m_cOpenCL);
//---
return true;
}
```

As a result of implementing all the methods described above, the final structure of our class has taken the following form.

```
class CNeuronBase : public CObject
{
protected:
    bool                m_bTrain;
    CMyOpenCL*          m_cOpenCL;
    CActivation*         m_cActivation;
    ENUM_OPTIMIZATION   m_eOptimization;
    CBufferType*         m_cOutputs;
    CBufferType*         m_cWeights;
    CBufferType*         m_cDeltaWeights;
    CBufferType*         m_cGradients;
    CBufferType*         m_cMomentum[2];
```

```

//---
virtual bool    SGDUpdate(int batch_size, TYPE learningRate,
                        VECTOR &Lambda);
virtual bool    MomentumUpdate(int batch_size, TYPE learningRate,
                        VECTOR &Beta, VECTOR &Lambda);
virtual bool    AdaGradUpdate(int batch_size, TYPE learningRate,
                        VECTOR &Lambda);
virtual bool    RMSPropUpdate(int batch_size, TYPE learningRate,
                        VECTOR &Beta, VECTOR &Lambda);
virtual bool    AdaDeltaUpdate(int batch_size,
                        VECTOR &Beta, VECTOR &Lambda);
virtual bool    AdamUpdate(int batch_size, TYPE learningRate,
                        VECTOR &Beta, VECTOR &Lambda);
virtual bool    SetActivation(ENUM_ACTIVATION_FUNCTION function,
                        VECTOR &params);

public:
    CNeuronBase(void);
    ~CNeuronBase(void);

//---
virtual bool    Init(const CLayerDescription *description);
virtual bool    SetOpenCL(CMyOpenCL *opengl);
virtual bool    FeedForward(CNeuronBase *prevLayer);
virtual bool    CalcOutputGradient(CBufferType *target,
                        ENUM_LOSS_FUNCTION loss);
virtual bool    CalcHiddenGradient(CNeuronBase *prevLayer);
virtual bool    CalcDeltaWeights(CNeuronBase *prevLayer);
virtual bool    UpdateWeights(int batch_size, TYPE learningRate,
                        VECTOR &Beta, VECTOR &Lambda);
virtual void    TrainMode(bool flag)          { m_bTrain = flag; }
virtual bool    TrainMode(void) const { return m_bTrain; }

//---
CBufferType    *GetOutputs(void) const { return(m_cOutputs); }
CBufferType    *GetGradients(void) const { return(m_cGradients); }
CBufferType    *GetWeights(void) const { return(m_cWeights); }
CBufferType    *GetDeltaWeights(void) const { return(m_cDeltaWeights); }

virtual bool    SetOutputs(CBufferType* buffer, bool delete_previous = true);
//--- methods for working with files
virtual bool    Save(const int file_handle);
virtual bool    Load(const int file_handle);
//--- method of identifying the object
virtual int     Type(void) const { return(defNeuronBase); }
virtual ulong   Rows(void) const { return(m_cOutputs.Rows()); }
virtual ulong   Cols(void) const { return(m_cOutputs.Cols()); }
virtual ulong   Total(void) const { return(m_cOutputs.Total()); }
};

```

3.6.3 Activation function class

We still have some open questions regarding the implementation of the neural layer base class. One of them is the neuron activation function class.

The activation function class will contain the operations for calculating the activation function and its derivative. There are various types of activation functions, and the book does not provide the full list of such functions, while it only covers the more commonly used ones. New, well-performing activation functions can emerge. So, if you need to add a new activation function to this library, the easiest way would be to do so by creating a new class that inherits from a certain base class. In this way, by overriding a couple of methods responsible for the direct calculation of the function and its derivative, the changes will be propagated to all neural network objects, including those created earlier.

Following this logic, I decided to create not a single activation function class that would cover all the functions discussed earlier, but a class structure in which each class would contain an algorithm for only one activation function. In this structure, there would be one base class at the top, which would define the interfaces for interaction with other objects and serve as an object for accessing methods from other objects without being tied to a specific activation function.

By creating a single branching point in the algorithm during the initialization of a specific activation function class, we move away from checking the used function at each iteration of the forward and backward passes.

The parent class for all activation functions **CActivation** is inherited from the **CObject** class, which is the base class for all objects in *MQL5*.

The *CActivation* class only contains methods for organizing the interface and does not describe any of the activation functions. In turn, to organize the activation function classes, I defined the following methods:

- *CActivation* – a class constructor;
- *~CActivation* – a class destructor;
- *Init* – passing parameters to calculate the activation function;
- *GetFunction* – getting the used activation function and its parameters;
- *Activation* – performs calculation of the activation function value based on the reference value;
- *Derivative* – derivative from the activation function;
- *SetOpenCL* – writing a pointer to an OpenCL object;
- *Save* and *Load* – virtual methods for working with files;
- *Type* – a virtual method for class identification.

In general, the class looks much simpler than those discussed previously. In the constructor of the class, we will set the default activation function parameters.

```
CActivation::CActivation(void) : m_iRows(0),
                                m_iCols(0),
                                m_cOpenCL(NULL)
{
    m_adParams = VECTOR::Ones(2);
    m_adParams[1] = 0;
}
```

Note that to calculate the derivative of certain activation functions, we only need the value of the activation function itself. In other cases, we will need values before the activation function. Therefore, let's introduce two pointers to the corresponding data buffers:

- *m_cInputs*
- *m_cOutputs*

3. Building the first neural network model in MQL5

In the body of this class, we will create only one instance of a buffer, and in another variable, we will save a pointer to the buffer that calls the neural layer. Due to this, in the destructor of the class, we will only delete one object.

```
CActivation::~CActivation(void)
{
    if(!m_cInputs)
        delete m_cInputs;
}
```

In the class initialization method, we store the resulting activation function parameters and create a data buffer object. It's important to note that at this stage, we are merely creating an instance of the class; we are initializing the buffer itself because we don't yet know the required buffer data sizes.

```
bool CActivation::Init(VECTOR &params)
{
    m_adParams = params;
    //---
    m_cInputs = new CBufferType();
    if(!m_cInputs)
        return false;
    //---
    return true;
}
```

The activation parameter reading method is straightforward. We only return the value of the variables.

```
ENUM_ACTIVATION_FUNCTION CActivation::GetFunction(VECTOR &params)
{
    params = m_adParams;
    return GetFunction();
}
```

The *Activation* method that calculates the values of the activation function, in the parameters receives a pointer to the neural layer result buffer. This buffer contains neuron performance data prior to the activation function. We need to activate the obtained values and overwrite them into the specified buffer. However, as we know, the obtained values might be needed when calculating derivatives of certain functions. Therefore, we "play" with the pointers to the buffer objects, saving the obtained pointer in the variable *m_cInputs*. In the variable received in parameters and in the *m_cOutputs* variable, we save the buffer from the *m_cInputs* variable. The current class corresponds to the absence of an activation function, so we don't perform any operations on the obtained data.

However, there is one nuance. Since we don't perform any operations on the obtained data, we need to return them to the calling program. At this point, we have already replaced the buffer that we will return. Therefore, we check the used activation function, and if no further actions are required on the obtained data, we will return the pointer to the buffer back and delete the unnecessary object.

It might seem like there are many unnecessary actions in the method that didn't alter the data in any way. However, these are our small investments in the functionality of the inheriting classes.

3. Building the first neural network model in MQL5

```
bool CActivation::Activation(CBufferType *amp;output)
{
    if(!output || output.Total() <= 0)
        return false;
    m_cOutputs = m_cInputs;
    m_cInputs = output;
    output = m_cOutputs;
    if(GetFunction() == AF_NONE && output != m_cInputs)
    {
        delete output;
        output = m_cInputs;
    }
    //---
    return true;
}
```

At the same time, the method for calculating the derivative of the activation function in this class will remain nominal. In all cases it will return a positive value.

The *SetOpenCL* method for activating multi-threaded computation functionality receives in the parameters a pointer to an object for working with the *OpenCL* context object and the size of the result buffer for the calling neural layer. We will need these buffer sizes for initialization and creation of the buffer in the context.

In the body of the method, we store the resulting dimensions and pointer, then initialize the data buffer of the specified size with null values and create a buffer in the *OpenCL* context.

```
bool CActivation::SetOpenCL(CMyOpenCL *openc1, const_ulong rows, const_ulong cols)
{
    m_iRows = rows;
    m_iCols = cols;
    if(m_cOpenCL != openc1)
    {
        if(m_cOpenCL)
            delete m_cOpenCL;
        m_cOpenCL = openc1;
    }
    //---
    if(!m_cInputs)
    {
        if(!m_cInputs.BufferInit(m_iRows, m_iCols, 0))
            return false;
        m_cInputs.BufferCreate(m_cOpenCL);
    } //---
    return(!m_cOpenCL);
}
```

As you can see, the methods of the class are quite simple. All we have to do is look at file-handling techniques. Their algorithm is also simple. In the body of the *Save* method, as usual, we check the file handle for writing the data we receive in the parameters and store the activation function type and parameter value.

3. Building the first neural network model in MQL5

```
bool CActivation::Save(const int file_handle)
{
    if(file_handle == INVALID_HANDLE)
        return false;
    if(FileWriteInteger(file_handle, Type()) <= 0 ||
        FileWriteInteger(file_handle, (int)GetFunction()) <= 0 ||
        FileWriteInteger(file_handle, (int)m_iRows) <= 0 ||
        FileWriteInteger(file_handle, (int)m_iCols) <= 0 ||
        FileWriteDouble(file_handle, (double)m_adParams[0]) <= 0 ||
        FileWriteDouble(file_handle, (double)m_adParams[1]) <= 0)
        return false;
    //---
    return true;
}
```

The data loading method *Load* also receives a file handle in parameters. In the method body, we check the validity of the received handle and read the values of the constants. After that, we initialize one data buffer. At the same time, we do not forget to control the operation process.

```
bool CActivation::Load(const int file_handle)
{
    if(file_handle == INVALID_HANDLE)
        return false;
    m_iRows = (uint)FileReadInteger(file_handle);
    m_iCols = (uint)FileReadInteger(file_handle);
    m_adParams.Init(2);
    m_adParams[0] = (TYPE)FileReadDouble(file_handle);
    m_adParams[1] = (TYPE)FileReadDouble(file_handle);
    //---
    if(!m_cInputs)
    {
        m_cInputs = new CBufferType();
        if(!m_cInputs)
            return false;
    }
    if(!m_cInputs.BufferInit(m_iRows, m_iCols, 0))
        return false;
    //---
    return true;
}
```

We have reviewed all the methods of the base class of the *CActivation* activation function. So, we have the following class structure.

3. Building the first neural network model in MQL5

```
class CActivation : protected CObject
{
protected:
    ulong          m_iRows;
    ulong          m_iCols;
    VECTOR         m_adParams;
    CMyOpenCL*     m_cOpenCL;
    //---
    CBufferType*   m_cInputs;
    CBufferType*   m_cOutputs;

public:
    CActivation(void);
    ~CActivation(void) {if(!m_cInputs) delete m_cInputs; }

    //---
    virtual bool    Init(VECTOR &params);
    virtual ENUM_ACTIVATION_FUNCTION GetFunction(VECTOR &params);
    virtual ENUM_ACTIVATION_FUNCTION GetFunction(void) { return AF_NONE; }
    virtual bool    Activation(CBufferType*& output);
    virtual bool    Derivative(CBufferType*& gradient) { return true; }
    //---
    virtual bool    SetOpenCL(CMyOpenCL *opencl, const ulong rows,
                                const ulong cols);

    //--- methods for working with files
    virtual bool    Save(const int file_handle);
    virtual bool    Load(const int file_handle);
    //--- object identification method
    virtual int     Type(void) const { return defActivation; }
};
```

However, as we discussed earlier, this class only lays the groundwork for future classes of various activation functions. To add the actual activation function algorithm, you need to create a new class by overriding several methods. For example, let's create a class of a linear activation function. The structure of this class is given below.

```
class CActivationLine : public CActivation
{
public:
    CActivationLine(void) {};
    ~CActivationLine(void) {};

    //---
    virtual ENUM_ACTIVATION_FUNCTION GetFunction(void) override
                                { return AF_LINEAR; }
    virtual bool    Activation(CBufferType*& output) override;
    virtual bool    Derivative(CBufferType*& gradient) override;
};
```

The new *CActivationLine* class is publicly inherited from the created above base class of the *CActivation* activation function. The constructor and destructor of the class are empty. All we have to do is redefine three methods:

- *GetFunction* – gets the used activation function and its parameters;

- *Activation* – performs calculation of the activation function value based on the reference value;
- *Derivative* – a derivative of the activation function.

In the *GetFunction* method, we only change the return type of the activation function to the corresponding class.

The *Activation* method in the parameters receives a pointer to the initial data buffer similar to the method of the parent class. In the body of the method, we don't check the received pointer; we simply call the method of the parent class, where we check the received pointer and "play" with the pointers to data buffers. After this, the algorithm is split into two threads: one using the *OpenCL* technology and the other without it. We will learn about multi-threaded operations a little later. In the block of operations without using multi-threading, we simply invoke the activation function for the matrix of obtained values, specifying the activation function type as *AF_LINEAR* and the function parameters.

```
bool CActivationLine::Activation(CBufferType*& output)
{
    if(!CActivation::Activation(output))
        return false;
    ///---
    if(!m_cOpenCL)
    {
        if(!m_cInputs.m_mMatrix.Activation(output.m_mMatrix, AF_LINEAR,
                                              m_adParams[0], m_adParams[1]))
            return false;
    }
    else // OpenCL block
    {
        return false;
    }
    ///---
    return true;
}
```

The method that calculates the derivative is even more straightforward. In the parameters, the method receives a pointer to the error gradient object. The obtained values must be corrected for the derivative of the activation function. As you know, the derivative of a linear function is its coefficient at the variable. So, the only thing we have to do is multiply the resulting gradient vector by the parameter of the activation function with the index of 0.

3. Building the first neural network model in MQL5

```
bool CActivationLine::Derivative(CBufferType*& gradient)
{
    if(!m_cInputs || !m_cOutputs ||
        !gradient || gradient.Total() < m_cOutputs.Total())
        return false;
    //---
    if(!m_cOpenCL)
    {
        gradient.m_mMatrix = gradient.m_mMatrix * m_adParams[0];
    }
    else // OpenCL block
    {
        return false;
    }
    //---
    return true;
}
```

As you can see, the mechanism for describing the new activation function is quite simple. Let's create a class for using ReLU as the activation function in a similar manner.

```
class CActivationLReLU : public CActivation
{
public:
    CActivationLReLU(void) { m_adParams[0] = (TYPE)0.3; };
    ~CActivationLReLU(void) {};

    //---
    virtual ENUM_ACTIVATION_FUNCTION GetFunction(void) override { return AF_LRELU; }
    virtual bool Activation(CBufferType*& output) override;
    virtual bool Derivative(CBufferType*& gradient) override;
};
```

In the activation function of the new class, we will also use a matrix activation function call specifying the corresponding function type, **AF_LRELU**.

```
bool CActivationLReLU::Activation(CBufferType*& output)
{
    if(!CActivation::Activation(output))
        return false;
    //---
    if(!m_cOpenCL)
    {
        if(!m_cInputs.m_mMatrix.Activation(output.m_mMatrix, AF_LRELU, m_adParams[0]))
            return false;
    }
    else // OpenCL block
    {
        return false;
    }
    //---
    return true;
}
```

We'll use a similar approach in the derivative method of the activation function.

```
bool CActivationLReLU::Derivative(CBufferType*& gradient)
{
    if(!m_cOutputs || !gradient ||
        m_cOutputs.Total() <= 0 || gradient.Total() < m_cOutputs.Total())
        return false;
    //---
    if(!m_cOpenCL)
    {
        MATRIX temp;
        if(!m_cInputs.m_mMatrix.Derivative(temp, AF_LRELU,m_adParams[0]))
            return false;
        gradient.m_mMatrix *= temp;
    }

    else // OpenCL block
    {
        return false;
    }
    //---
    return true;
}
```

The reader may have a reasonable question as to why we should create new classes if we use the activation matrix functions embedded in the MQL5 language. This is done more to ensure a unified approach with and without OpenCL multi-threaded technologies. These methods will incorporate code for organizing multi-threaded computations in the OpenCL context. The use of the described classes enables a unified call to activation function algorithms using both MQL5 tools and multi-threaded computations in the OpenCL context.

3.7 Organizing parallel computing using OpenCL

In the previous chapters, we have already become acquainted with the organization of the operation of a fully connected neural layer using MQL5. I would like to remind you that in our implementation, we used matrix operations to multiply the input data vector by the weight matrix. From one neural layer to another, the signal flows sequentially, and we cannot initiate operations on the subsequent neural layer until the operations on the previous one are fully completed. In contrast to this, the results of operations within one neuron in a layer do not depend on the operations being carried out with other neurons within the same neural layer. Consequently, we can reduce the time cost of processing a single neural layer if we can organize parallel computation. The more neurons we process simultaneously, the less time we spend on processing one signal and training the neural network as a whole.

As we have already discussed earlier, [OpenCL](#) technology will help us in organizing parallel computations. Of course, this will require extra work to customize the process. Let's consider which processes we will transfer to OpenCL to make it as efficient as possible. Let me remind you that due to the overhead time for data transfer between devices, we can achieve real performance improvement only with a large number of concurrent operation threads.

The first thing that can be carried over is the computation of forward pass operations. We can move the execution of operations on each individual neuron into the realm of parallel computing. First, we

calculate the weighted sum of the input signal for each neuron and then calculate the activation function for each neuron.

We can also move the operations of the backward pass into the realm of parallel computations. Let's break down the steps of the backward pass.

Deviation of calculated values from the reference values at the output layer of the neural network can be easily divided into separate threads for each neuron.

Furthermore, we can also adjust the obtained deviation for each neuron based on the derivative of the activation function. As a result of such an operation, we obtain the error gradient before the neuron activation function.

Following the backpropagation process, in the next step we need to distribute the resulting error gradient to the neurons of the previous layer. In a fully connected neural layer, all neurons from the previous layer are connected to all neurons in the subsequent layer. In each element of the error gradient vector, there is a component from every neuron in the previous layer. There are two seemingly equivalent approaches here:

- We can create threads for each element of the error gradient vector, and within each thread, iterate through all neurons of the previous layer and add the value of its gradient error component.
- Conversely, we can divide the threads for each neuron in the previous layer and assemble the gradient error components from the previous layer.

Despite their apparent equivalence, the first approach has several drawbacks. Since we will be summing up the gradient error components from different neurons of the subsequent layer, it's necessary to initialize the value of the current vector to zero before starting the operations. This means additional costs in time and resources. In addition, there are also technical nuances. Working with global memory is slower than working with a thread's private memory. Therefore, it's preferable to assemble values in fast memory and transfer them to global memory once. The most challenging aspect of this approach is that there's a significant likelihood of multiple threads attempting to write values to a single neuron in the previous layer simultaneously. And that is highly undesirable for us.

Based on the combination of the above factors, the second option becomes more attractive for implementation.

Splitting the following two processes into threads (calculating deltas for weight adjustment and directly updating the weight matrix) doesn't raise any questions, as each weight is involved in only one connection between two neurons and doesn't affect the others.

3.7.1 Creating an OpenCL program

We will start porting calculations by creating an OpenCL program. The choice of this approach is very obvious. We have already organized the process using MQL5 tools. Therefore, the whole process of operations is clear and transparent. Computation operations will be performed in the OpenCL program. In the main program, we have to organize the process of transferring data and calling the OpenCL program. The latter process is easier to organize when we already know what data and to which *kernel* we need to transfer.

The program code will be written to a separate file *opencl_program.cl*. The file name and extensions can be anything, as we will later load it in the main program code as a resource. I use the *.cl extension as a common extension to denote OpenCL programs. In general, the use of standard extensions makes it easier to read projects with complex file structures.

For a feed-forward pass, similar to the [MQL5](#) implementation, we will create *kernel PerceptronFeedForward*. To carry out operations, we need the result vector of the previous layer (*inputs*) and the weight matrix (*weights*) as the initial data. The result of the operations will be written to the result vector (*outputs*). In addition to the data sets, we need to know the number of neurons in the previous layer to ensure control over potential output exceeding the array limits.

As we discussed earlier, the number of threads will correspond to the number of neurons in the layer. Therefore, at the beginning of the kernel, we need to call the *get_global_id* function, which will return the thread identifier to us. Let's use this value as the ordinal number of the neuron being processed.

The weight matrix in our buffer is represented as a vector in which *N* weights of the first neuron go sequentially, followed by *N* weights of the second neuron, and so on. The value of *N* is one element greater than the number of neurons in the previous layer since we are using the *bias* element. We already have the neuron's ordinal number and the number of neurons in the previous layer, so we can determine the offset in the weight vector to the first weight of our neuron.

Next, we will create a local variable to accumulate the sum of products and initialize it with the bias offset coefficient. After that, we organize a loop and calculate the sum of products of the original signal by the weights for a specific neuron in our stream. OpenCL supports vector operations which allow the microprocessor to perform one operation simultaneously on multiple values in a single cycle. In the proposed implementation, I used vectors of type *double4*. This is a vector of four elements of type *double*. To convert data from an array of discrete values to a vector, I created a function called *ToVect4*, which we will discuss a bit later. To obtain the sum of products, I used the built-in *dot* function. It belongs to vector operations and allows obtaining the discrete value of the product of two vectors. This allowed us to use a step of 4 in the loop and thereby reduce the number of iterations by 4 times.

It should be noted that *double4* is not the only vector data type supported by OpenCL. The ability to use them depends on the technical specifications of the hardware being used. The *double4* type, in my personal opinion, is the most versatile for use on a wide range of available equipment. When creating your own libraries, you can use a different type of data that is most optimal for your equipment.

After completing the iterations of the loop, we will save the accumulated sum of the vector product to the corresponding element of the result buffer.

The full kernel code is presented below.

```
__kernel void PerceptronFeedForward(__global TYPE *inputs,
                                    __global TYPE *weights,
                                    __global TYPE *outputs,
                                    int inputs_total)
{
    const int n = get_global_id(0);
    const int weights_total = get_global_size(0) * (inputs_total + 1);
    int shift = n * (inputs_total + 1);
    TYPE s = weights[shift + inputs_total];
    for(int i = 0; i < inputs_total; i += 4)
        s += dot(ToVect4(inputs, i, 1, inputs_total, 0),
                 ToVect4(weights, i, 1, weights_total, shift));
    outputs[n] = s;
}
```

Let's examine the algorithm of the *ToVect4* function. In parameters, the function receives a pointer to a vector of discrete values, the position of the first element to copy, the step between two elements to copy, the size of the data array, and the offset in the array to the first copied element. The parameters of the position of the first element and the offset serve a similar purpose but differ in the context of operations. The offset determines the displacement in the data vector from the element with index 0. For the feed-forward function, this offset is to the first weight of the processed neuron. The position of the first element to copy specifies the element without considering the step between elements. In the example given, this is the number of neurons in the preceding layer. When considering an example with a step of one element, one parameter can be easily expressed in terms of the other. The difference in using parameters will be more noticeable when discussing the backpropagation pass, where the weight copying step will be equal to the size of the weight vector for one neuron.

At the beginning of the function, we initialize the result vector with zero values and ensure that the step is at least one element. Then, we check how many elements we can take from the original data array starting from the initial position, considering the offset and step. This operation is necessary to prevent accessing data beyond the array boundaries. Next, we fill the result vectors with the available values, leaving the missing elements as zero. Thus, the size of the original array becomes a multiple of four. Meanwhile, the final value for calling functions remains unchanged, and the use of vector operations overall helps reduce time costs for operations.

```
TYPE4 ToVect4(__global TYPE *array, int start, int step, int size, int shift)
{
    TYPE4 result = (TYPE4)(0, 0, 0, 0);
    step = max(1, step);
    int st = start * step + shift;
    if(st < size)
    {
        int k = (size - shift + step - 1) / step;

        switch(k)
        {
            case 0:
                break;
            case 1:
                result = (TYPE4)(array[st], 0, 0, 0);
                break;
            case 2:
                result = (TYPE4)(array[st], array[st + step], 0, 0);
                break;
            case 3:
                result = (TYPE4)(array[st], array[st + step], array[st + 2 * step], 0);
                break;
            default:
                result = (TYPE4)(array[st], array[st + step], array[st + 2 * step], array[st + 3 * step]);
                break;
        }
    }
    return result;
}
```

To fully understand the forward pass, let's explore the activation functions. In general, they repeat the relevant implementations in MQL5, except for the kernel design. For example, below is the code for the implementation of a sigmoid. The kernel parameters include pointers to the input data buffers, output

3. Building the first neural network model in MQL5

buffers, and activation function parameters. In the kernel body, we first determine the thread identifier, which indicates the ordinal number of the processed element in the data buffer, and then organize the process of calculating the activation function. As you can see, the code for calculating the value of the function is very similar to the relevant MQL5 code presented in the [activation function](#) description section.

```
__kernel void SigmoidActivation(__global TYPE* inputs,
                               __global TYPE* outputs,
                               const TYPE a, const TYPE b)
{
    size_t i = get_global_id(0);
    outputs[i] = a / (1 + exp(-inputs[i])) - b;
}
```

The same can be said about the implementation of the [Swish](#) activation function.

```
__kernel void SwishActivation(__global TYPE* inputs,
                             __global TYPE* outputs,
                             const TYPE b)
{
    size_t i = get_global_id(0);
    TYPE value = inputs[i];
    outputs[i] = value / (1 + exp(-b * value));
}
```

However, there are some difficulties in implementing the *Softmax* function. This is due to the difficulty of transferring data between threads to calculate the sum of all the values of the exponent vector in the neural layer. To resolve the issue, I decided to divide the process into several stages. In addition, we'll take advantage of the Work-group's ability to use shared arrays in local memory.

In the kernel parameters, we pass pointers to the buffers of input data and results, as well as the size of the input data buffer. In the kernel body, we first get all the necessary identifiers. These are the global IDs in two dimensions, and the thread ID in the local group. We will talk about the use of two dimensions in the global task space later.

```
__kernel void SoftMaxActivation(__global TYPE* inputs,
                               __global TYPE* outputs,
                               const ulong total)
{
    uint i = (uint)get_global_id(0);
    uint l = (uint)get_local_id(0);
    uint h = (uint)get_global_id(1);
    uint ls = min((uint)get_local_size(0), (uint)LOCAL_SIZE);
```

Then we will create a local array in which we allocate one element for each thread to sum up exponential values.

Note that OpenCL does not allow the creation of dynamic arrays. Therefore, the size of the local array must be determined before the program is compiled. So, we need to look for some kind of compromise. Excessive size leads to inefficient memory utilization. On the other hand, if the array size is too small, this limits the number of active parallel threads. Of course, solving such a task is much easier when you know the parameters of the device you are using and the architecture of the model. Therefore, we need a mechanism that allows us to easily and quickly change this parameter before compiling the program. The best solution for this is to use macro substitution, just like we did for the data type. In

the code, we specify the *LOCAL_SIZE* constant, the value of which we assign in our constant file *defines.mqh*.

Next, we organize a loop in which each thread sums its part of the exponential values. The resulting value is written to the corresponding element of the local array.

```
__local TYPE temp[LOCAL_SIZE];
uint count = 0;
for(count = 1; (count < total && 1 < ls); count += ls)
{
    uint shift = h * total + count;
    temp[l] = (count > 1 ? temp[l] : 0) + exp(inputs[shift]);
}
barrier(CLK_LOCAL_MEM_FENCE);
```

After the loop operations are complete, we set a *barrier* that allows all the threads of the local group to be synchronized. This operation suspends the execution of each thread, waiting for all threads within the local group to complete their loop iterations.

After obtaining several individual sums computed by each separate thread, it is necessary to combine them into a single sum. To do this, we will organize another cycle. In its body, we will sum the values of the local array in pairs. The trick is that we will divide the entire local array into two equal parts. The first half of the active threads will add the value from the second half to its value in the local array. In the next iteration of the loop, we will halve the number of active threads and sum the values obtained in the previous iteration of the loop. The loop repeats until the sum of all elements is collected in the first element of the array.

Here we insert a *barrier* in the body of the loop because before the start of each subsequent iteration, all threads must finish the previous iteration.

```
count = ls;
do
{
    count = (count + 1) / 2;
    temp[l] += (1 < count && (1 + count) < ls ? temp[l + count] : 0);
    barrier(CLK_LOCAL_MEM_FENCE);
}
while(count > 1);
```

After obtaining the sum of the exponents of all the values in the buffer, we can calculate the value of each element after the activation function. This is what we will do in the next cycle.

```
//---
TYPE sum=temp[0];
for(count = 1; count < total; count += ls)
{
    uint shift = h * total + count;
    outputs[shift] = exp(inputs[shift]) / (sum + 1e-37f);
}
}
```

A forward pass is followed by a backward pass. It begins with the definition of error gradients at the output of the neural layer. This function is performed by the *CalcOutputGradient* kernel. In the kernel parameters, it receives pointers to three vectors: the first two vectors of target values and forward

pass results are the input data for the function, while the third one is used to store the calculation results. The parameters also specify the loss function to be used. The kernel code completely repeats the algorithm of the previously considered relevant method written using MQL5 tools. It also shows the branching of the algorithm depending on the loss function used.

```
__kernel void CalcOutputGradient(__global TYPE *target,
                                __global TYPE *outputs,
                                __global TYPE *gradients,
                                int loss_function)
{
    const int n = get_global_id(0);
    switch(loss_function)
    {
        case 0:
            gradients[n] = target[n] - outputs[n];
            break;
        case 1:
            gradients[n] = 2 * (target[n] - outputs[n]);
            break;
        case 2:
            gradients[n] = -target[n] /
                (outputs[n] + 1e-37f) * log(outputs[n] + 1e-37f);
            break;
        case 3:
            gradients[n] = (target[n] - outputs[n]) /
                (outputs[n] * (outputs[n] - 1) + 1e-37f);
            break;
        default:
            gradients[n] = target[n] - outputs[n];
            break;
    }
}
```

The next step in our backpropagation process is to adjust the error gradient based on the derivative of the activation function. Recall that we have moved the activation function and all related processes to a separate class. Furthermore, each activation function has its own class. We will create a separate kernel for each activation function. Similarly, we will create a separate kernel for determining the derivative of each activation function.

When creating kernels for calculating derivative functions, we take into account the specific features of each of them. For example, the derivative of a linear activation function will always be its parameter a , and I see no reason to put this in a separate kernel. To adjust the error gradient to the derivative of this function, we can use the forward pass kernel by specifying 0 instead of the b parameter.

A similar situation arises when using [LReLU](#). A linear relationship is also used here, but the linearity factor varies from the value to the activation function. Therefore, we need to create a new kernel that, in its parameters, will receive pointers to three data buffers and a leakage coefficient, whereas the forward pass kernel only received pointers to two buffers and the coefficient.

In the kernel body, we define the global thread identifier, which will identify the element to be processed in the data buffers. Then we check the value of the corresponding element before the function. If the number is greater than 0, we use the coefficient of 1. Otherwise, we will use the leakage factor. The error gradient obtained from the subsequent neural layer will be multiplied by the

selected coefficient. The value of the operation will be written to the corresponding element of the result buffer.

```
__kernel void LReLUDerivative(__global TYPE* outputs,
                             __global TYPE* output_gr,
                             __global TYPE* input_gr,
                             const TYPE a)
{
    size_t i = get_global_id(0);
    input_gr[i] = (outputs[i] > 0 ? (TYPE)1 : a) * output_gr[i];
}
```

The value of derivatives of S-shaped functions, such as the sigmoid and the hyperbolic tangent, can be easily calculated through the result of the activation function.

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x)$$

It is this approach that we will embed into the kernel algorithm for calculating the derivatives of these functions. The general approaches to the organization of kernel work remain the same.

```
__kernel void SigmoidDerivative(__global TYPE* outputs,
                               __global TYPE* output_gr,
                               __global TYPE* input_gr,
                               const TYPE a, const TYPE b
                               )
{
    size_t i = get_global_id(0);
    if(a == 0)
        input_gr[i] = 0;
    else
    {
        TYPE z = clamp(outputs[i] + b, (TYPE)0, a);
        input_gr[i] = z * (1 - z / a) * output_gr[i];
    }
}

__kernel void TanhDerivative(__global TYPE* outputs, __global TYPE* output_gr,
                             __global TYPE* input_gr)
{
    size_t i = get_global_id(0);
    input_gr[i] = (1 - pow(outputs[i], 2)) * output_gr[i];
}
```

To calculate the derivative of the Swish function, we will need the values both before and after activation. The mathematical formula of the derivative is presented below.

$$\frac{\partial f(x)}{\partial x} = \beta f(x) + \sigma(\beta x)(1 - \beta f(x))$$

As you can see, the derivative of the function is expressed in terms of the activation function value, sigmoid values, and the activation function's parameter β . Substituting the sigmoid formula, we get the following expression.

$$\frac{\partial f(x)}{\partial x} = \beta f(x) + \frac{1 - \beta f(x)}{1 + e^{-\beta x}}$$

To implement the formula mentioned, you need to pass pointers to four data buffers to the kernel: the corresponding values before and after activation, the error gradient from the subsequent layer, and the results buffer.

In the kernel body, calculate the derivative value using the provided formula and then multiply this value by the error gradient obtained from the subsequent layer. The result of the operation will be written to the corresponding element of the results buffer.

```
__kernel void SwishDerivative(__global TYPE* outputs,
                             __global TYPE* output_gr,
                             __global TYPE* input_gr,
                             const TYPE b,
                             __global TYPE* inputs)
{
    size_t i = get_global_id(0);
    TYPE by = b * outputs[i];
    input_gr[i] = (by + (1 - by) / (1 + exp(-b * inputs[i]))) * output_gr[i];
}
```

The kernel of calculating the derivative of the function *Softmax* is the most difficult. As in the case of the S-shaped functions, the values of the activation function itself are sufficient to calculate the derivative of the *Softmax* function. To compute the value of one element in the vector during the feed-forward pass, we used the values of all elements in the vector before the activation function (calculating the total sum of exponents). Therefore, the value of each element after activation depends on all elements of the vector before activation. This means that each element must receive its share of the error from each element at the output of the neural layer before activation. In general, the derivative of the *Softmax* function is calculated using the formula below.

$$\frac{df(x_i)}{dx_i} = \begin{cases} i = j & f(x_i) * (1 - f(x_i)) \\ i \neq j & -f(x_j)f(x_i) \end{cases}$$

In the parameters of the *SoftMaxDerivative* kernel, we will pass pointers to three data buffers: the values of the activation function, the error gradient from the next layer, and the result buffer.

In the kernel body, we define the global thread ID, which this time only points to an element in the results buffer. The global thread identifier in the second dimension is used when working with a matrix in which the *Softmax* function was applied row-by-row. In this case, this identifier will help us determine the offset to the analyzed data.

Next, we prepare two private variables: one to store the value of the corresponding element after activation, and the other to accumulate the total gradient error value.

After that, we organize a loop for collecting error gradients from all elements at the output of the neural layer. Each specific gradient is calculated using the above formula.

After completing the loop iterations, we will save the accumulated sum of gradients in the corresponding element of the results buffer.

```
__kernel void SoftMaxDerivative(__global TYPE* outputs,
                              __global TYPE* output_gr,
                              __global TYPE* input_gr)
{
    size_t i = get_global_id(0);
    size_t outputs_total = get_global_size(0);
    size_t shift = get_global_id(1) * outputs_total;
    TYPE output = outputs[shift + i];
    TYPE result = 0;
    for(int j = 0; j < outputs_total; j++)
        result += outputs[shift + j] * output_gr[shift + j] *
                ((TYPE)(i == j ? 1 : 0) - output);
    input_gr[shift + i] = result;
}
```

After adjusting the error gradient for the activation function derivative, we need to distribute the obtained values to the neurons of the previous neural layer. As it was mentioned above, here we will divide the threads by the number of neurons in the lower neural layer. For each neuron, we will collect gradients from all neurons dependent on it.

The distribution of the error gradient through the neural layer will be carried out in the *CalcHiddenGradient* kernel. Pointers to 3 arrays are input into the kernel:

- *weights*: a matrix of weights;
- *gradients*: an array of gradients adjusted for the derivative of the activation function;
- *gradient_inputs*: an array for recording the gradients of the preceding layer.

In addition, the parameters indicate the number of neurons in the top layer (the size of the *gradients* array). The kernel construction algorithm is very similar to the forward pass method, as we also use the *dot* and *ToVect4* functions. The difference lies in the arrays being used: during the forward pass, we took the input signal and multiplied it by the weights, whereas now we multiply the error gradient by the weights. There is one more point in using the *ToVect4* function for the matrix of weights. When we considered this function for the feed-forward pass, we talked about a similar function of the parameters of the first element for copying *start* and for shifting *shift*. Then we used step of 1 element. Now, by iterating over the array of gradients, we will select the appropriate weights. However, in the feed-forward pass, neurons and weights followed in order, while in the backward pass, we take the weights across the weight matrix. In the vector expression of the weight matrix, we will use the step between the two elements to copy 1 element more than the number of neurons in the previous layer (the *bias* element). At the same time, the shift will be equal to the ordinal number of the processed neuron of the lower layer.

We do not specify the number of neurons in the lower neural layer in the parameters but use this value as a step to read values from the weight matrix. The *get_global_size* function allows us to get the specified value, which returns the total number of running kernel threads. Since we launched one thread for each neuron of the previous layer, the number of threads in this case will correspond to the number of neurons in the layer. Here, we calculate the number of elements in the weight matrix by multiplying the number of neurons in the layer by the number of neurons in the previous layer plus the *bias* element.

In other respects, we also use vector operations that allow us to utilize a loop in steps of 4 elements.

3. Building the first neural network model in MQL5

```
__kernel void CalcHiddenGradient(__global TYPE *gradient_inputs,
                                __global TYPE *weights,
                                __global TYPE *gradients,
                                int outputs_total)
{
    const int n = get_global_id(0);
    const int inputs_total = get_global_size(0);
    int weights_total = (inputs_total + 1) * outputs_total;
    ///---
    TYPE grad = 0;
    for(int o = 0; o < outputs_total; o += 4)
        grad += dot(ToVect4(gradients, o, 1, outputs_total, 0),
                    ToVect4(weights, o, (inputs_total + 1), weights_total, n));
    gradient_inputs[n] = grad;
}
```

If you look at the [error backpropagation](#) algorithm, we have already reached the finish line. After propagating the error gradient, all we have to do is update the weight matrix. However, we remember from the MQL5 implementation that the weight matrix will not be updated after each iteration. Again, the process of updating the weight matrix will be divided into 2 stages:

1. Accumulating error gradients over a certain interval.
2. Averaging of the accumulated gradient and adjustment of the weight matrix.

We will collect error gradients for each weight in the *CalcDeltaWeights* kernel. In the kernel parameters, pointers to the array of data from the output of the previous layer, the gradient array, and an array for accumulating deltas needed for weight adjustments are passed.

All weights are calculated independently, so we can run the error calculation for each weighting factor in a separate thread. To make the structure of the threads more visual and understandable, we will create a task space in two dimensions. The first dimension will be equal to the number of neurons in the current layer, and the second will be equal to the number of neurons in the previous layer.

In the kernel body, we will determine the dimension of the weight matrix over the problem space and the position of the analyzed element in this matrix. After that, we will determine the offset of the element in the result buffer.

Let's not forget that we accumulate the error gradient until the moment of direct updating of the weights. Therefore, we will add to the previously accumulated sum the product of the corresponding error gradient by the result element of the previous layer.

Note that we are using an additional *bias* element. For this element, a constant value of the incoming element equal to one is used. We didn't take it into account when creating the task space, but we must also accumulate an error gradient for it. From a mathematical point of view, the derivative of multiplication by 1 is equal to 1. This means that the error gradient for a given element is equal to the error gradient before the activation function of the corresponding neuron. To avoid duplicating the iteration for writing the *bias* weight error gradient, we will perform this iteration only for the thread with index 0 in the second dimension.

```
__kernel void CalcDeltaWeights(__global TYPE *inputs,
                               __global TYPE *delta_weights,
                               __global TYPE *gradients)
{
    const int n = get_global_id(0);
```

3. Building the first neural network model in MQL5

```
const int outputs_total = get_global_size(0);
const int i = get_global_id(1);
const int inputs_total = get_global_size(1);
//---
TYPE grad = gradients[n];
int shift = n * (inputs_total + 1);
delta_weights[shift + i] = inputs[i] * grad + delta_weights[shift + i];
if(i == 0)
    delta_weights[shift + inputs_total] += grad;
}
```

Now, we need to organize the process of updating the matrix of weights. We have studied and already implemented several methods for updating weights in the main program. When implementing this process in [MQL5](#), we created a dispatch method that redirected the logical chain of operations to the method corresponding to the selected method for updating the weights. Then, within these methods, we defined the device to perform the operations. To maintain the integrity of this approach, we have to create several kernels by analogy with the main program, implementing all the methods used to optimize the weight matrix.

All kernels were created using a single approach and therefore have many features in common. However, there are differences due to the specific features of each method. Let's start exploring kernels with the [stochastic gradient](#) descent method.

$$w_{il}^j = w_{il}^j - \alpha \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_i^j,$$

In the kernel parameters, we pass pointers to the matrix of accumulated gradients and the matrix of weights. In addition to pointers to matrices, the kernel parameters include the total number of elements in the weight matrix, the batch size for averaging, the learning rate, and regularization parameters.

As before, we will use vector operations, so the number of threads will be four times smaller than the size of the weight matrix. In the kernel body, we first define the offset in the array for the working elements of our stream and load them into vector variables. When reading the accumulated deltas, we immediately divide the obtained values by the batch size, which will give us the average value of the gradient. After that, we adjust the weights for the regularization coefficients and the average value of the accumulated deltas, taking into account the learning rate.

In conclusion, we return the obtained values to the weight matrix and reset the array of accumulated deltas.

```
__kernel void SGDUpdate(__global TYPE *delta_weights,
                        __global TYPE *weights,
                        int total,
                        int batch_size,
                        TYPE learningRate,
                        TYPE Lambda1,
                        TYPE Lambda2
)
{
    int start = 4 * get_global_id(0);
    TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0);
```

3. Building the first neural network model in MQL5

```

TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
TYPE lr = learningRate / ((TYPE4)batch_size);
weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
weights4 += (TYPE4)(lr) * delta4;
D4ToArray(weights, weights4, start, 1, total, 0);
D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
}

```

Next, we studied the [accumulated momentum](#) method. In the same sequence, we will create kernels for the implementation of methods. The *MomentumUpdate* kernel algorithm is very similar to the stochastic gradient descent kernel discussed above. The main differences are the introduction of an additional array for storing the accumulated pulse, updating the weights, and the pulse smoothing parameter.

$$\Delta_t = \alpha \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_i^j + \beta \Delta_{t-1}$$

$$w_{il}^j = w_{il}^j - \Delta_t$$

In the kernel body, as in the previous method, we read the corresponding array values into vector variables. At the same time, we average the accumulated gradient. Then we adjust the weights for the regularization parameters. After this, we first update the momentum of the weight change, taking into account the average gradient and the previously accumulated momentum. Only after this step, we adjust the weights based on the updated momentum. Before exiting the kernel, transfer the values of the vector variables to the corresponding elements of the weights and moments matrices. The cumulative array of deltas will be zeroed.

```

__kernel void MomentumUpdate(__global TYPE* delta_weights,
                             __global TYPE* weights,
                             __global TYPE* momentum,
                             int total, int batch_size,
                             TYPE learningRate,
                             TYPE beta,
                             TYPE Lambda1, TYPE Lambda2)
{
    int start = 4 * get_global_id(0);
    //---
    TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0) /
                  ((TYPE4)batch_size);
    TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
    TYPE4 momentum4 = ToVect4(momentum, start, 1, total, 0);
    weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
    momentum4 = (TYPE4)(learningRate) * delta4 + (TYPE4)(beta) * momentum4;
    weights4 += momentum4;
    D4ToArray(weights, weights4, start, 1, total, 0);
    D4ToArray(momentum, momentum4, start, 1, total, 0);
    D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
}

```

The [AdaGrad](#) optimization method, like the accumulated momentum method, uses a single array to accumulate moments. But unlike the previous method, we will sum up the squares of the gradients and there is no smoothing coefficient.

$$G_{ilt}^j = G_{ilt-1}^j + (grad_{ilt}^j)^2$$

$$w_{il}^j = w_{il}^j - \frac{\alpha}{\sqrt{G_{ilt}^j} + \varepsilon} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_{il}^j$$

The approach to the use of the accumulated moment has also changed. In the previous method, we used accumulated momentum to reduce the randomness of updates and maintain smooth movements in the direction of the anti-gradient. Now, in the adaptive gradient method, the accumulated square of gradients is used to decrease the learning rate with each iteration. This is reflected in the kernel code below.

```
__kernel void AdaGradUpdate(__global TYPE* delta_weights,
                           __global TYPE* weights,
                           __global TYPE* momentum,
                           int total, int batch_size,
                           TYPE learningRate,
                           TYPE Lambda1, TYPE Lambda2)
{
    int start = 4 * get_global_id(0);
    //---
    TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0) /
                                                           ((TYPE4)batch_size);
    TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
    TYPE4 momentum4 = ToVect4(momentum, start, 1, total, 0);
    //---
    weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
    momentum4 = momentum4 + pow(delta4, 2);
    weights4 += learningRate / sqrt(momentum4 + 1.0e-37f);
    D4ToArray(weights, weights4, start, 1, total, 0);
    D4ToArray(momentum, momentum4, start, 1, total, 0);
    D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
}
```

The main problem of the adaptive gradient method is the constant accumulation of the gradient square. With long-term training, this can lead to a decrease in the learning rate to zero and a practical stop in the training of the neural network. This problem is solved in [RMSProp](#) by introducing a smoothing factor for accumulating gradient squares. This allows us to limit the growth of the accumulated sum of squares of gradients and thereby limit the decrease in the learning rate.

$$REMS(G_{il}^j)_t = \gamma (grad_{il(t-k)}^j)^2 + (1 - \gamma) REMS(G_{il}^j)_{t-1}$$

$$w_{il}^j = w_{il}^j - \frac{\alpha}{\sqrt{REMS(G_{il}^j) + \varepsilon}} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_{il}^j$$

Otherwise, the kernel algorithm repeats the previously considered methods for updating weights.

```
__kernel void RMSPropUpdate(__global TYPE* delta_weights,
                           __global TYPE* weights,
                           __global TYPE* momentum,
```

3. Building the first neural network model in MQL5

```

        int total, int batch_size,
        TYPE learningRate,
        TYPE beta,
        TYPE Lambda1, TYPE Lambda2)
{
    int start = 4 * get_global_id(0);
    //---
    TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0) /
                                                            ((TYPE4)batch_size);
    TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
    TYPE4 momentum4 = ToVect4(momentum, start, 1, total, 0);
    //---
    weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
    momentum4 = beta * momentum4 + (1 - beta) * pow(delta4, 2);
    weights4 += delta4 * learningRate / (sqrt(momentum4) + 1.0e-37f);
    D4ToArray(weights, weights4, start, 1, total, 0);
    D4ToArray(momentum, momentum4, start, 1, total, 0);
    D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
}

```

In the *AdaDelta* optimization update method, the authors tried to exclude the learning rate. But the price for this was the introduction of an additional momentum buffer and a second smoothing factor.

$$\begin{aligned}
 REMS(\delta w_{il}^j)_t &= \gamma_w (\delta w_{il(t-k)}^j)^2 + (1 - \gamma_w) REMS(\delta w_{il}^j)_{t-1} \\
 REMS(G_{il}^j)_t &= \gamma_G (grad_{il(t-k)}^j)^2 + (1 - \gamma_G) REMS(G_{il}^j)_{t-1} \\
 w_{il}^j &= w_{il}^j - \frac{REMS(\delta w_{il}^j)}{REMS(G_{il}^j) + \varepsilon} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_{il}^j,
 \end{aligned}$$

The method uses two exponential averages. The first one averages the square values of the corresponding weight, while the second one, like in the two previous methods, calculates the square of gradients on this weight. Instead of the learning rate, the ratio of the square roots of the two specified averages is used. As a result, we obtain a method in which, with an increase in the absolute value of the weight, the learning rate also increases. At the same time, an increase in the absolute value of the error gradient leads to a decrease in the learning rate.

Let's consider the implementation of this method in the *AdaDeltaUpdate* kernel. In the parameters, pointers to four data arrays are passed to the kernel:

- *delta_weights*: an array of accumulated error gradients;
- *weights*: a matrix of weights;
- *momentumW*: a matrix of exponential mean squares of the weights;
- *momentumG*: a matrix of exponential squares of error gradients.

In addition to the pointers to arrays, the kernel parameters include the size of the arrays, batch size, two exponential smoothing coefficients, and regularization parameters.

In the kernel body, we define the shift to the elements to be processed in this thread and read the necessary elements from the arrays into vector variables for further processing. The next step is to adjust the weights based on regularization parameters and update the weight moments and gradients.

After that, we will update the weights themselves. Finally, write the new values to the data arrays and reset the array of accumulated gradients.

```
__kernel void AdaDeltaUpdate(__global TYPE* delta_weights,
                             __global TYPE* weights,
                             __global TYPE* momentumW,
                             __global TYPE* momentumG,
                             int total, int batch_size,
                             TYPE beta1, TYPE beta2,
                             TYPE Lambda1, TYPE Lambda2)
{
    int start = 4 * get_global_id(0);
    //---
    TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0) /
                                                           ((TYPE4)batch_size);
    TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
    TYPE4 momentumW4 = ToVect4(momentumW, start, 1, total, 0);
    TYPE4 momentumG4 = ToVect4(momentumG, start, 1, total, 0);
    //---
    weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
    momentumW4 = beta1 * momentumW4 + (1 - beta1) * pow(weights4, 2);
    momentumG4 = beta2 * momentumG4 + (1 - beta2) * pow(delta4, 2);
    weights4 += delta4 * sqrt(momentumW4) / (sqrt(momentumG4) + 1.0e-37f);
    D4ToArray(weights, weights4, start, 1, total, 0);
    D4ToArray(momentumW, momentumW4, start, 1, total, 0);
    D4ToArray(momentumG, momentumG4, start, 1, total, 0);
    D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
}
```

The last method we studied was the [Adam](#) adaptive moment estimation method. Below are the mathematical formulas of this method.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) grad_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) grad_t^2$$

$$\hat{m} = \frac{m}{1 - \beta_1}$$

$$\hat{v} = \frac{v}{1 - \beta_2}$$

$$w_t = w_{t-1} - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \varepsilon}$$

Compared to the methods discussed earlier, the formulas may appear more complex, but there's nothing daunting about them, and this method is implementable. Like AdaDelta, the method uses two buffers to accumulate moments. We accumulate the momentum of the gradients in the first buffer and the momentum of the gradient squares in the second one. Both buffers use exponential smoothing, but each uses a different smoothing factor. In addition, the learning rate is returned to the method.

Let's consider the implementation of the method in the *AdamUpdate* kernel. In the kernel parameters, we will pass pointers to the data arrays:

- *delta_weights*: accumulated gradient deltas;
- *weights*: a matrix of weights;
- *momentumM*: a matrix of accumulated gradients;
- *momentumV*: a matrix of accumulated gradient squares.

We will also pass the size of the arrays, batch size, learning rate, smoothing coefficients, and regularization parameters in the kernel parameters.

At the beginning of the kernel, as in the implementation of the previous optimization methods, we define the shift to the processed elements in the arrays. To avoid unnecessary confusion, we will synchronously use the elements of the arrays, that is, the size of the arrays and the shift to the corresponding elements will be identical.

Let's copy the processed array elements into vector variables. As always, we will immediately divide the accumulated deltas by the batch size and store the arithmetic mean of the error gradient.

Next, we calculate the updated values of the accumulated pulses and adjust them, as proposed by the authors of the method.

After the preparatory work, we will adjust our weights. As before, we will first adjust for regularization parameters, and then move towards the anti-gradient direction according to the optimization method rules. In other words, we will subtract the product of the learning rate and the ratio of the first moment of the gradient to the square root of its second moment from the current weight.

At the end of the kernel, we will save the obtained values to the corresponding arrays. Do not forget to reset the array of accumulated deltas.

```
__kernel void AdamUpdate(__global TYPE* delta_weights,
                        __global TYPE* weights,
                        __global TYPE* momentumM,
                        __global TYPE* momentumV,
                        int total, int batch_size,
                        TYPE learningRate,
                        TYPE beta1, TYPE beta2,
                        TYPE Lambda1, TYPE Lambda2)
{
    int start = 4 * get_global_id(0);
    //---
    TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0) /
                                                            ((TYPE4)batch_size);
    TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
    TYPE4 momentumM4 = ToVect4(momentumM, start, 1, total, 0);
    TYPE4 momentumV4 = ToVect4(momentumV, start, 1, total, 0);
    //---
    momentumM4 = beta1 * momentumM4 + (1 - beta1) * delta4;
    momentumV4 = beta2 * momentumV4 + (1 - beta2) * pow(delta4, 2);
    TYPE4 m = momentumM4 / (1 - beta1);
    TYPE4 v = momentumV4 / (1 - beta2);
    weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
    weights4 += learningRate * m / (sqrt(v) + 1.0e-37f);
}
```



```

D4ToArray(weights, weights4, start, 1, total, 0);
D4ToArray(momentumM, momentumM4, start, 1, total, 0);
D4ToArray(momentumV, momentumV4, start, 1, total, 0);
D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
}

```

At this stage, we have completed the work on writing the OpenCL program. Of course, we will return to this work when implementing other architectural solutions for neural layers. However, in terms of a fully connected neural layer, this work can be considered complete. We save the code we've written and move on to implementing the processes of data exchange between the main program and the OpenCL kernels, as well as the functions for invoking the kernels. We have to do this work on the side of the main program.

3.7.2 Implementing functionality on the main program side

The implementation of the functionality on the main program side will require some knowledge of process organization and effort. Let's start with the preparatory work. First, in our [file of definitions](#), we need to add the loading of the OpenCL program written above as a resource and assign its contents to a string variable. Here, we will also add predefined macro substitutions for data types and the size of the local array to the program.

```

#resource "opencl_program.cl" as string OCLprogram
//---
#define TYPE float
#define LOCAL_SIZE 256
const string ExtType = StringFormat("#define TYPE %s\r\n"
                                     "#define TYPE4 %s4\r\n"
                                     "#define LOCAL_SIZE %d\r\n",
                                     typename(TYPE), typename(TYPE), LOCAL_SIZE);

#define cl_program ExtType+OCLprogram

```

When declaring kernels in the main program, the [CLKernelCreate](#) function returns a handle. To work with OpenCL technology, we will use the *CMyOpenCL* class, which is derived from the standard *COpenCL* class. The aforementioned classes implement arrays for storing handles. A specific kernel is accessed by an index in the array. To simplify working with these indices and make the program code more readable, let's add constants for the indices of all the kernels created above. To explicitly identify the kernel index in the program code, we will start all named kernel constants with *def_k*.

```

//+-----+
//| OpenCL Kernels |
//+-----+
#define def_k_PerceptronFeedForward 0
#define def_k_LineActivation 1
#define def_k_SigmoidActivation 2
#define def_k_SigmoidDerivative 3
#define def_k_TANHActivation 4
#define def_k_TANHDerivative 5
#define def_k_LReLuActivation 6
#define def_k_LReLuDerivative 7
#define def_k_SoftMAXActivation 8
#define def_k_SoftMAXDerivative 9
#define def_k_SwishActivation 10

```

3. Building the first neural network model in MQL5

```
#define def_k_SwishDerivative      11
#define def_k_CalcOutputGradient  12
#define def_k_CalcHiddenGradient  13
#define def_k_CalcDeltaWeights   14
#define def_k_SGDUpdate           15
#define def_k_MomentumUpdate      16
#define def_k_AdaGradUpdate       17
#define def_k_RMSPropUpdate       18
#define def_k_AdaDeltaUpdate      19
#define def_k_AdamUpdate          20
```

To specify parameters when calling kernels, we can also use indices. However, now they are not specified explicitly. Instead, the serial number in the list of OpenCL kernel parameters is used. All kernels use their own set of parameters, so we will define named constants for all created kernels. To avoid confusion between identical parameters of different kernels, we will include a pointer to the respective kernel in the constant name. For example, the parameter constants for the forward pass kernel of the basic fully connected layer will start with *def_pff*.

```
//--- perceptron feed forward pass
#define def_pff_inputs            0
#define def_pff_weights          1
#define def_pff_outputs          2
#define def_pff_inputs_total     3
```

We will declare constants for all written kernels in a similar way.

```
//--- calculating the error gradient of the result layer
#define def_outgr_target          0
#define def_outgr_outputs        1
#define def_outgr_gradients      2
#define def_outgr_loss_function  3

//--- calculating the error gradient of the hidden layer
#define def_hidgr_gradient_inputs 0
#define def_hidgr_weights        1
#define def_hidgr_gradients      2
#define def_hidgr_outputs_total  3

//--- calculating the error gradient at the level of the weight matrix
#define def_delt_inputs           0
#define def_delt_delta_weights   1
#define def_delt_gradients       2

//--- parameter optimization by stochastic gradient descent
#define def_sgd_delta_weights    0
#define def_sgd_weights         1
#define def_sgd_total            2
#define def_sgd_batch_size      3
#define def_sgd_learningRate     4
#define def_sgd_Lambda1         5
#define def_sgd_Lambda2         6

//--- parameter optimization using the moment method
#define def_moment_delta_weights 0
```

3. Building the first neural network model in MQL5

```
#define def_moment_weights          1
#define def_moment_momentum         2
#define def_moment_total            3
#define def_moment_batch_size       4
#define def_moment_learningRate     5
#define def_moment_beta              6
#define def_moment_Lambda1          7
#define def_moment_Lambda2          8

//--- parameter optimization using the AdaGrad method
#define def_adagrad_delta_weights   0
#define def_adagrad_weights         1
#define def_adagrad_momentum        2
#define def_adagrad_total           3
#define def_adagrad_batch_size      4
#define def_adagrad_learningRate    5
#define def_adagrad_Lambda1         6
#define def_adagrad_Lambda2         7

//--- parameter optimization using the RMSProp method
#define def_rms_delta_weights       0
#define def_rms_weights             1
#define def_rms_momentum            2
#define def_rms_total               3
#define def_rms_batch_size          4
#define def_rms_learningRate        5
#define def_rms_beta                6
#define def_rms_Lambda1             7
#define def_rms_Lambda2             8

//--- parameter optimization using the AdaDelta method
#define def_adadelt_delta_weights   0
#define def_adadelt_weights         1
#define def_adadelt_momentumW       2
#define def_adadelt_momentumG       3
#define def_adadelt_total           4
#define def_adadelt_batch_size      5
#define def_adadelt_beta1           6
#define def_adadelt_beta2           7
#define def_adadelt_Lambda1         8
#define def_adadelt_Lambda2         9

//--- parameter optimization using the Adam method
#define def_adam_delta_weights      0
#define def_adam_weights            1
#define def_adam_momentumM          2
#define def_adam_momentumV          3
#define def_adam_total              4
#define def_adam_batch_size         5
#define def_adam_learningRate       6
#define def_adam_beta1              7
#define def_adam_beta2              8
#define def_adam_Lambda1            9
```

3. Building the first neural network model in MQL5

```
#define def_adam_Lambda2          10

//--- activation functions
#define def_activ_inputs          0
#define def_activ_outputs        1
#define def_activ_param_a        2
#define def_activ_param_b        3

//--- adjusting the gradient to the derivative of the activation function
#define def_deactgr_outputs      0
#define def_deactgr_gradients    1
#define def_deactgr_deact_gradient 2
#define def_deactgr_act_param_a  3
#define def_deactgr_act_param_b  4
```

I intentionally provided a complete set of constants above to offer you a reference guide. It will assist in reading and understanding the code for our next steps in implementing OpenCL technology into the project.

After describing the constants, we will move on to creating classes that will be responsible for servicing OpenCL tools. We have already mentioned them multiple times. It's time to learn more about their features.

First, this is the *CMyOpenCL* class. It inherits from the *COpenCL* class from the MQL5 standard libraries. The standard library is well-written and has sufficient functionality to organize work. However, I found one aspect inconvenient personally: when working with buffers for data exchange between the main program and the OpenCL context, a similar approach is used as with other process objects. When creating a buffer, we have to specify its index in the general array of buffers. This is a perfectly workable option when we know all the buffers and their quantity in advance. However, our case is a little more complicated.

```
class CMyOpenCL : public COpenCL
{
public:
    CMyOpenCL(void) {}
    ~CMyOpenCL(void) {}

    //--- initialization and shutdown
    virtual bool Initialize(const string program, const bool show_log = true);
    //---
    template<typename T>
    int AddBufferFromArray(T &data[], const uint data_array_offset,
                           const uint data_array_count, const uint flags);
    int AddBufferFromArray(MATRIX &data,
                           const uint data_array_offset, const uint flags);
    int AddBuffer(const uint size_in_bytes, const uint flags);
    bool CheckBuffer(const int index);
    //---
    bool BufferFromMatrix(const int buffer_index, MATRIX &data,
                          const uint data_array_offset, const uint flags);
    bool BufferRead(const int buffer_index, MATRIX &data,
                   const uint cl_buffer_offset);
    bool BufferWrite(const int buffer_index, MATRIX &data,
                    const uint cl_buffer_offset);
```

```
};
```

Earlier, we discussed that the number of used buffers for accumulating moments can vary depending on the chosen method for updating weights. In addition, we cannot know in advance how many neural layers the user will use to solve their tasks. Hence, I needed a dynamic array to store handles of data buffers. This problem was solved by adding a small *AddBufferFromArray* method. The parameters of this method are similar to those of the *BufferFromArray* method of the parent class except for the buffer index. The body of the method body a loop to search for empty cells in the buffer handle storage array. The first empty cell is used to create the buffer. When there are no free elements in the array, the method expands the array. The buffer is directly created by calling the above parent class method.

As a result of the operations, the method returns the index of the created buffer. If errors occur during operations, the method will return the *INVALID_HANDLE* constant.

I'd like to point out another aspect, which is that the method is created using the function template pattern. This allows you to use one method to create buffers of different types of data.

```
template<typename T>
int CMyOpenCL::AddBufferFromArray(T &data[], const uint data_array_offset,
                                   const uint data_array_count, const uint flags
                                   )
{
    int result=INVALID_HANDLE;
    for(int i=0; i<m_buffers_total; i++)
    {
        if(m_buffers[i]!=INVALID_HANDLE)
            continue;
        result=i;
        break;
    }
    //---
    if(result<0)
    {
        if(ArrayResize(m_buffers,m_buffers_total+1)>0)
        {
            m_buffers_total=ArraySize(m_buffers);
            result=m_buffers_total-1;
            m_buffers[result]=INVALID_HANDLE;
        }
        else
            return result;
    }
    //---
    if(!BufferFromArray(result,data,data_array_offset,data_array_count,flags))
        return INVALID_HANDLE;
    //---
    return result;
}
```

The method created above allows the creation of buffers from arrays of any data types but it is not applicable when working with matrices. Therefore, the method was overloaded. The method algorithm remains unchanged.

```

int CMyOpenCL::AddBufferFromArray(MATRIX &data,
                                  const uint data_array_offset,
                                  const uint flags
                                  )
{
    //--- Search for a free element in a dynamic array of pointers
    int result = -1;
    for(int i = 0; i < m_buffers_total; i++)
    {
        if(m_buffers[i] != INVALID_HANDLE)
            continue;
        result = i;
        break;
    }
    //--- If a free item is not found, add a new item to the array
    if(result < 0)
    {
        if(ArrayResize(m_buffers, m_buffers_total + 1) > 0)
        {
            m_buffers_total = ArraySize(m_buffers);
            result = m_buffers_total - 1;

            m_buffers[result] = INVALID_HANDLE;
        }
        else
            return result;
    }
    //--- Create a buffer in the OpenCL context
    if(!BufferFromMatrix(result, data, data_array_offset, flags))
        return -1;
    return result;
}

```

Anticipating a bit, I want to mention that we won't always be creating buffers based on ready-made arrays. Sometimes, we just need to create a buffer in the OpenCL context without duplicating it in the main memory. Or, for example, a specific buffer is only used to obtain results, and there is no need to load its data into the context before performing operations. As we've mentioned before, the data copying process is an expensive operation, and we would like to minimize such operations. Therefore, it would be easier for us to simply create a data buffer in the context of a certain size without copying the data. For such cases, we will create the *AddBuffer* method. As you can notice, the algorithm of the method is almost identical to the methods of the previous class. The only difference is that this method receives the buffer size in bytes as a parameter instead of an array. At the end of the method, we call the *BufferCreate* method, which will create a buffer of the specified size in the OpenCL context.

```

int CMyOpenCL::AddBuffer(const uint size_in_bytes, const uint flags)
{
    //--- Search for a free element in a dynamic array of pointers
    int result = -1;
    for(int i = 0; i < m_buffers_total; i++)
    {
        if(m_buffers[i] != INVALID_HANDLE)
            continue;
    }
}

```

```

        result = i;
        break;
    }
    ///--- If a free item is not found, add a new item to the array
    if(result < 0)
    {
        if(ArrayResize(m_buffers, m_buffers_total + 1) > 0)
        {
            m_buffers_total = ArraySize(m_buffers);
            result = m_buffers_total - 1;
            m_buffers[result] = INVALID_HANDLE;
        }

        else
            return result;
    }
    ///--- Create a buffer in the OpenCL context
    if(!BufferCreate(result, size_in_bytes, flags))
        return -1;
    return result;
}

```

We also created methods for reading (*BufferRead*) and writing (*BufferWrite*) data of the OpenCL context buffer to the main memory matrix. The method algorithm is completely identical. Let's consider the data reading method as an example. In the method parameters, it receives the buffer identifier in the dynamic array of our class, a matrix for writing data, and an offset in the context buffer.

Please do not confuse the buffer identifier in the dynamic class array and the buffer handle in the OpenCL context. The class operation is structured in such a way that we only pass the ordinal number of an element in the dynamic array of our class to the external program, which contains the handle of that buffer. As a result, when creating a buffer in the context using the class, the external program does not have direct access to the created buffer in the context. All work with the buffer should be done using class methods.

In the method body, we first check the received buffer ID for the size of our dynamic array. We then check the validity of the specified buffer handle. In addition, we will check the validity of the OpenCL context and program handles. Only after successfully passing all the controls, we call the function for reading data from the buffer. Don't forget to check the results of the operations at every step. At the end of the method, we will return the logical result of the operations.

```

bool CMyOpenCL::BufferRead(const int buffer_index, MATRIX &data,
                           const uint cl_buffer_offset)
{
    ///--- checking parameters
    if(buffer_index < 0 || buffer_index >= m_buffers_total || data.Rows() <= 0)
        return(false);
    if(m_buffers[buffer_index] == INVALID_HANDLE)
        return(false);
    if(m_context == INVALID_HANDLE || m_program == INVALID_HANDLE)
        return(false);
    ///--- reading buffer data from the OpenCL context
    if(!CLBufferRead(m_buffers[buffer_index], cl_buffer_offset, data))
        return(false);
}

```

```
//---
    return(true);
}
```

The second class that we will create and use to transfer data between the main program and the OpenCL context is the *CBufferType* data buffer class. The class was created as a descendant of the *CObject* base class. Since the parent class is the base class, we need to recreate all the necessary functionality.

In addition to creating new methods in the new class, two new variables have appeared:

- *m_cOpenCL* – a pointer to an object of the *CMyOpenCL* class
- *m_myIndex* – the index of the current buffer in the dynamic array for storing buffer handles in the *CMyOpenCL* class.

The *m_mMatrix* matrix for storing data has also been introduced. Here we have slightly deviated from the generally accepted rules for creating classes. It is usually customary to restrict access to internal variables, and all interactions with them are built through class methods. Each such method restricts the degree of freedom to internal variables and requires additional time for executing the method's additional operations. Of course, this approach allows for complete control over changes in variable states. However, in building neural models, we aim to minimize the time spent on each iteration, as milliseconds per iteration can result in significant time overhead due to repeated calls. That is why we announced the *m_mMatrix* data matrix in public space. Of course, the fact that the class will be used to store and transmit data within our global project and that all buffers will be private or protected objects of other classes, minimizes our risks.

```
class CBufferType: public CObject
{
protected:
    CMyOpenCL*      m_cOpenCL;      // OpenCL context object
    int             m_myIndex;      // data buffer index in context
public:
    CBufferType(void);
    ~CBufferType(void);

    //--- data matrix
    MATRIX          m_mMatrix;
    //--- method of initializing the buffer with initial values
    virtual bool    BufferInit(const ulong rows, const ulong columns,
                                const TYPE value = 0);

    //--- create a new buffer in the OpenCL context
    virtual bool    BufferCreate(CMyOpenCL *opencl);
    //--- delete the buffer in the context of OpenCL
    virtual bool    BufferFree(void);
    //--- read buffer data from the OpenCL context
    virtual bool    BufferRead(void);
    //--- write buffer data to the OpenCL context
    virtual bool    BufferWrite(void);
    //--- get the buffer index
    virtual int     GetIndex(void);
    //--- change the buffer index
    virtual bool    SetIndex(int index)
    {
        if(!m_cOpenCL.BufferFree(m_myIndex))

```



```

        return false;
        m_myIndex = index;
        return true;
    }

//--- copy buffer data to an array
virtual int      GetData(TYPE &values[], bool load = true);
virtual int      GetData(MATRIX &values, bool load = true);
virtual int      GetData(CBufferType* values, bool load = true);
//--- calculate the average value of the data buffer
virtual TYPE     MathMean(void);
//--- vector operations
virtual bool     SumArray(CBufferType* src);
virtual int      Scaling(TYPE value);
virtual bool     Split(CBufferType* target1, CBufferType* target2,
                      const int position);
virtual bool     Concatenate(CBufferType* target1, CBufferType* target2,
                             const int positions1, const int positions2);
//--- methods for working with files
virtual bool     Save(const int file_handle);
virtual bool     Load(const int file_handle);
//--- class identifier
virtual int      Type(void) const { return defBuffer; }
//--- methods for working with the data matrix
ulong           Rows(void) const { return m_mMatrix.Rows(); }
ulong           Cols(void) const { return m_mMatrix.Cols(); }
uint            Total(void) const { return (uint)(m_mMatrix.Rows() *
                                                  m_mMatrix.Cols()); }

TYPE            At(uint index) const { return m_mMatrix.Flat(index); }
TYPE            operator[](ulong index) const { return m_mMatrix.Flat(index); }
VECTOR          Row(ulong row) { return m_mMatrix.Row(row); }
VECTOR          Col(ulong col) { return m_mMatrix.Col(col); }
bool            Row(VECTOR& vec, ulong row) { return m_mMatrix.Row(vec, row); }
bool            Col(VECTOR& vec, ulong col) { return m_mMatrix.Col(vec, col); }
bool            Activation(MATRIX& mat_out, ENUM_ACTIVATION_FUNCTION func)
                { return m_mMatrix.Activation(mat_out, func); }
bool            Derivative(MATRIX& mat_out, ENUM_ACTIVATION_FUNCTION func)
                { return m_mMatrix.Derivative(mat_out, func); }
bool            Reshape(ulong rows, ulong cols)
                { return m_mMatrix.Reshape(rows, cols); }

//---
bool            Update(uint index, TYPE value)
{
    if(index >= Total())
        return false;
    m_mMatrix.Flat(index, value);
    return true;
}

```

3. Building the first neural network model in MQL5

```
bool Update(uint row, uint col, TYPE value)
{
    if(row >= Rows() || col >= Cols())
        return false;
    m_mMatrix[row, col] = value;
    return true;
}

};
```

The structure of the class methods is quite diverse. Some of them are similar to matrix functions and perform the same functionality – designed to work with a data matrix. Others carry out the functionality of interacting with the OpenCL context. Let's take a closer look at some of them.

In the class constructor, we will only set the initial values of the new variables. They are filled with empty values.

```
CBufferType::CBufferType(void) : m_myIndex(-1)
{
    m_cOpenCL = NULL;
}
```

In the class destructor, we will perform memory cleaning operations. Here we'll clear the buffer in the context of OpenCL.

```
CBufferType::~CBufferType(void)
{
    if(m_cOpenCL && m_myIndex >= 0 && m_cOpenCL.BufferFree(m_myIndex))
    {
        m_myIndex = -1;
        m_cOpenCL = NULL;
    }
}
```

We have already used the *BufferInit* buffer initialization method in the neural layer class constructor. The main functionality of this method is to create a matrix of a specified size and populate it with initial values. The buffer size and initial values are specified in the method parameters. As part of this project, we will fill arrays with zero values during the initialization of the neural network and reset the buffers of accumulated deltas after updating the weight matrix.

```
bool CBufferType::BufferInit(ulong rows, ulong columns, TYPE value)
{
    if(rows <= 0 || columns <= 0)
        return false;
    m_mMatrix = MATRIX::Full(rows, columns, value);
    if(m_cOpenCL)
    {
        CMyOpenCL *openc1=m_cOpenCL;
        BufferFree();
        return BufferCreate(openc1);
    }
    //---
    return true;
}
```

The next method is to create a buffer in the OpenCL context. In parameters, the method receives a pointer to an instance of the *CMyOpenCL* class in the context of which the buffer should be created.

The method starts with a control block. First, we check the validity of the obtained pointer - in case of receiving an invalid pointer, we delete the buffer previously created in the OpenCL context and exit the method.

```
bool CBufferType::BufferCreate(CMyOpenCL *openc1)
{
    //--- initial data validation block
    if(!openc1)
    {
        BufferFree();
        return false;
    }
}
```

Then we check that it matches the previously saved pointer. If the pointers are identical and the buffer index is already saved, we won't create a new buffer in the *OpenCL* context but will simply copy the data from the matrix to the data exchange buffer again. To do this, we call the *BufferWrite* method. This method has its own set of checks, which we will become familiar with a bit later, and it returns a logical result of the operation. We exit the method with the result of the method of writing data to the OpenCL context.

```
//--- if the received pointer matches the one previously saved,
//--- simply copy the buffer contents into the context memory
if(openc1 == m_cOpenCL && m_myIndex >= 0)
    return BufferWrite();
```

The subsequent code of the method will be executed only if we have not exited the method during the preceding operations. Here, we check the validity of the previously saved pointer to an instance of the *CMyOpenCL* class and the presence of an index in the dynamic array storing handles of data buffers. If this condition is met, we must clear the memory and delete the existing buffer using the *BufferFree* method before continuing operations. Only after successfully deleting the old buffer do we have the right to open a new one. Otherwise, uncontrolled use of memory resources will lead to memory shortages and corresponding consequences.

```
//--- checking for a previously saved pointer to the OpenCL context
//--- if available, remove the buffer from the unused context
if(m_cOpenCL && m_myIndex >= 0)
{
    if(m_cOpenCL.BufferFree(m_myIndex))
    {
        m_myIndex = -1;
        m_cOpenCL = NULL;
    }
    else
        return false;
}
```

At the end of the method, we initiate the creation of a new data buffer in the specified context. To do this, we call the *AddBufferFromArray* method discussed above. The index obtained in response to the call will be stored in the *m_myIndex* variable. If the buffer opening operation is successful, we will save the *CMyOpenCL* instance pointer received as input to the method before exiting.

3. Building the first neural network model in MQL5

```
//--- create a new buffer in the specified OpenCL context
    if((m_myIndex = opencl.AddBufferFromArray(m_mMatrix, 0, CL_MEM_READ_WRITE)) < 0)
        return false;
    m_cOpenCL = opencl;
//---
    return true;
}
```

In this method, we used two new methods: one for clearing the buffer and the other for writing data. The *BufferFree* method is responsible for clearing the buffer. The method algorithm is quite simple. First, we check for the presence of a stored pointer to an instance of the *CMyOpenCL* class and an index in the dynamic buffer array. If they are available, call the *CMyOpenCL* class buffer cleaning method and specify the buffer index to delete. If the buffer is successfully removed from the context, clear the pointer to the *CMyOpenCL* class instance and the buffer index variable.

It should be noted that calling this method clears memory and deletes the buffer only in the context of *OpenCL*. At the same time, the data matrix itself and its contents remain in RAM. We will be able to exploit this property to use *OpenCL* context memory more efficiently a little later.

```
bool CBufferType::BufferFree(void)
{
//--- checking for a previously saved pointer to the OpenCL context
//--- if available, remove the buffer from the unused context
    if(m_cOpenCL && m_myIndex >= 0)
        if(m_cOpenCL.BufferFree(m_myIndex))
        {
            m_myIndex = -1;
            m_cOpenCL = NULL;
            return true;
        }
    if(m_myIndex >= 0)
        m_myIndex = -1;
//---
    return false;
}
```

Next, I suggest considering methods for transferring information between the main program and the OpenCL context. This work is done in two similar methods: *BufferRead* and *BufferWrite*. Despite the differences in the operation directions, the algorithm of the methods is identical. At the beginning of the methods, a control block is organized that checks the validity of the pointer to an instance of the *CMyOpenCL* class and the presence of an index in the dynamic buffer array. And only after the control block has been successfully passed, the OpenCL context class method of the same name is called, specifying the buffer index, matrix, and offset in the OpenCL buffer.

```
bool CBufferType::BufferRead(void)
{
    if(!m_cOpenCL || m_myIndex < 0)
        return false;
//---
    return m_cOpenCL.BufferRead(m_myIndex, m_mMatrix, 0);
}

bool CBufferType::BufferWrite(void)
```

3. Building the first neural network model in MQL5

```
{
    if(!m_cOpenCL || m_myIndex < 0)
        return false;
//---
    return m_cOpenCL.BufferWrite(m_myIndex, m_mMatrix, 0);
}
```

We have separately created methods for obtaining and directly specifying the buffer index in the dynamic array of *GetIndex* and *SetIndex* buffer handles. Their code is straightforward, so I don't even move them outside the class declaration block.

We've added three *GetData* methods of the same name to the class. They all perform the same function which is copying matrix data into a given structure. The difference is in the data receiver. This can be a dynamic array, matrix, or another instance of the *CBufferType* class.

In the first case, the method parameters contain a reference to the array and a flag that indicates the need to read data from the OpenCL context before copying the data. The introduction of the flag is a necessary measure. As you may have noticed when considering a method for reading data from the context, if there is no pointer to the *CMyOpenCL* object or index in the dynamic buffer array, the method will return *false*. This will block receiving data from an array without a buffer created in the OpenCL context. The introduction of a flag allows you to control this process.

At the beginning of the method, we check the flag and read data from the context, if necessary. Only then do we change the size of the receiver array and create a data copying cycle. Finally, the method returns the number of copied items.

```
int CBufferType::GetData(TYPE &values[], bool load = true)
{
    if(load && !BufferRead())
        return -1;
    if(ArraySize(values) != Total() &&
        ArrayResize(values, Total()) <= 0)
        return false;
//---
    for(uint i = 0; i < Total(); i++)
        values[i] = m_mMatrix.Flat(i);
    return (int)Total();
}
```

The other two methods are built on the basis of a similar algorithm but they take into account the specifics of the receiver object.

```
int CBufferType::GetData(MATRIX &values, bool load = true)
{
    if(load && !BufferRead())
        return -1;
//---
    values = m_mMatrix;
    return (int)Total();
}

int CBufferType::GetData(CBufferType *values, bool load = true)
{
    if(!values)
```

```

        return -1;
    if(load && !BufferRead())
        return -1;
    values.m_mMatrix.Copy(m_mMatrix);
    return (int)values.Total();
}

```

Now that we have prepared constants and classes for working with the OpenCL context, we can continue to work on organizing the process directly in our neural network classes.

When creating methods for our [neural network base class](#), we did not add two methods, *UseOpenCL* and *InitOpenCL*. As can be seen from the names of the methods, they are designed to initialize and control the process of working with OpenCL. The first one is used to switch the operating mode and enables and disables the use of OpenCL. The second one initializes the operation of an instance of the *CMyOpenCL* class.

Let's take a step back and fill these gaps. In the parameters of the *UseOpenCL* method, we will specify the new state as a logical value. Using a logical value to convey a binary state to enable/disable a function seems intuitive to me. It is quite logical to use *true* to enable the functionality and *false* to turn it off.

In the method body, we will organize the algorithm to branch out depending on the state being set. When we receive a command to disable the functionality, we will check the current pointer to an instance of the *CMyOpenCL* class that is stored in the *m_Copencl* variable. If the pointer is invalid, the functionality has not been initialized before, and we have nothing to disable. In this case, we will just update the state of the technology usage flag and exit the method.

If the functionality was previously activated and a signal to deactivate it has now been received, we will initiate the process of cleaning up the object and deleting it. After that, we will distribute a new (empty) pointer to neural network objects, save the flag, and exit the method.

```

void CNet::UseOpenCL(bool value)
{
    if(!value)
    {
        if(!m_cOpenCL)
        {
            m_bOpenCL = value;
            return;
        }
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        if(!m_cLayers)
            m_cLayers.SetOpencl(m_cOpenCL);
        m_bOpenCL = value;
        return;
    }
}

```

Further operations will be performed only when the OpenCL functionality is enabled. When we receive a signal to enable the use of OpenCL, we start the process of creating and initializing a new instance of the *CMyOpenCL* class, which is placed in a separate *InitOpenCL* method.

Before exiting the method, save the new flag for using OpenCL and distribute the pointer to the new object across all objects of the neural network. To do this, we will pass a new pointer into the dynamic

array object storing the layers of the neural network, and from there, the pointer will be passed down the hierarchical chain to each object in the neural network.

```
//---
if(!m_cOpenCL)
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
}
m_bOpenCL = InitOpenCL();
if(!m_cLayers)
    m_cLayers.SetOpencl(m_cOpenCL);
return;
}
```

The actual process of creating a new instance of the *CMyOpenCL* class and initializing it is placed in a separate *InitOpenCL* method.

At the beginning of the method, we check for the existence of a previously saved pointer to an object of the *CMyOpenCL* class. At this point, the question arises about what we want to do next if there is a previously instantiated object. We can continue using a previously initialized instance of the class or create a new one. Using an existing facility seems less labor-intensive at this stage. However, in this case, we may need an additional method to restart the functionality in the event of an error of some kind. This is an additional effort that is likely to require developing an additional control system for the entire project code.

We chose the forced restart option. Therefore, if we have a valid pointer to a previously created instance of the *CMyOpenCL* class, we start the process of deleting its contents from memory, and then the object itself. Only after clearing the memory, we start the process of creating and initializing a new object. The process of creating an OpenCL context and program is implemented in the *COpenCL::Initialize* method. As parameters to this method, we will pass a text variable containing our program. Remember, we wrote our program code from a file resource into it?

```
bool CNet::InitOpenCL(void)
{
    //--- Delete previously created OpenCL objects
    if(!m_cOpenCL)
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
    }
    //--- Create a new object to work with OpenCL
    m_cOpenCL = new CMyOpenCL();
    if(!m_cOpenCL)
        return false;
    //--- Initialize the object for working with OpenCL
    if(!m_cOpenCL.Initialize(cl_program, true))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        return false;
    }
}
```

Next, let's specify the number of kernels and buffers used. Above, we have declared constants for 20 kernels, each using no more than 4 data buffers. I intentionally don't specify a large number of buffers at this stage, as thanks to our new method, the array will automatically expand when a new data buffer is created. However, the number of kernels in the program is static and does not depend on the neural network architecture.

```
if(!m_cOpenCL.SetKernelsCount(20))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}
if(!m_cOpenCL.SetBuffersCount(4))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}
```

After that, we will initialize all program kernels and save the handles for calling them into an array within the *CMyOpenCL* class object.

We are not creating all the data buffers one by one at this stage for one simple reason: their quantity depends on the architecture of the neural network and may exceed the available OpenCL context memory capacity. If it is insufficient, dynamic memory allocation can be used. This implies loading buffers as needed and subsequently freeing memory when a specific data buffer is not planned to be used. However, this approach leads to an increase in the overhead of copying data between the main memory and the OpenCL context. Therefore, its use is justified only if there is a lack of GPU memory.

The kernel creation algorithm is identical. Here are just a few examples.

```
if(!m_cOpenCL.KernelCreate(def_k_PerceptronFeedForward, "PerceptronFeedForward"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}

if(!m_cOpenCL.KernelCreate(def_k_CalcOutputGradient, "CalcOutputGradient"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}

if(!m_cOpenCL.KernelCreate(def_k_CalcHiddenGradient, "CalcHiddenGradient"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}

if(!m_cOpenCL.KernelCreate(def_k_CalcDeltaWeights, "CalcDeltaWeights"))
{

```


3. Building the first neural network model in MQL5

```
m_cOpenCL.Shutdown();
delete m_cOpenCL;
return false;
}
```

So we have come to the stage of organizing work with the OpenCL context directly in the neural layer class. When creating many class methods, we branched the method algorithm depending on the device for performing operations. Then we created the process organization code using MQL5 and left gaps in the process organization on the OpenCL side. Let's go back and fill in these gaps.

We will start with the direct pass method. We have previously discussed the organization of operations using MQL5. Now let's look at the implementation of working with the OpenCL context.

```
bool CNeuronBase::FeedForward(CNeuronBase * prevLayer)
{
    ///--- control block
    if(!prevLayer || !m_cOutputs || !m_cWeights ||
        !prevLayer.GetOutputs() || !m_cActivation)
        return false;
    CBufferType *input_data = prevLayer.GetOutputs();
    ///--- algorithm branching depending on the operating device
    if(!m_cOpenCL)
    {
        if(m_cWeights.Cols() != (input_data.Total() + 1))
            return false;
        ///---
        MATRIX m = input_data.m_mMatrix;
        if(!m.Reshape(1, input_data.Total() + 1))
            return false;
        m[0, m.Cols() - 1] = 1;
        m_cOutputs.m_mMatrix = m.MatMul(m_cWeights.m_mMatrix.Transpose());
    }
}
```

First, we'll check that the initial data array, the weight matrix, and the result buffer have a buffer index. The logic here is simple. If we receive a pointer to a data array with an existing buffer in the method's parameters, we assume that the data is already loaded into the OpenCL context. Above, when creating a data buffer in the *CBufferType* class, we immediately created a buffer in the OpenCL context. Therefore, the absence of a buffer index may indicate an error. Because of this, in such a case, we end the method with a *false* result. If you use dynamic memory allocation, then at this point you will need to create copies of all data buffers used in this kernel and copy the contents of the source data buffers into the OpenCL context.

```
else // OpenCL block
{
    ///--- checking data buffers
    if(input_data.GetIndex() < 0)
        return false;
    if(m_cWeights.GetIndex() < 0)
        return false;
    if(m_cOutputs.GetIndex() < 0)
        return false;
}
```

Then we will specify the parameters for the feed-forward kernel. Here we will specify their indices for buffers and specific values for discrete parameters.

```
//--- passing arguments to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_PerceptronFeedForward, def_pff_inputs,
                                input_data.GetIndex()))

    return false;

if(!m_cOpenCL.SetArgumentBuffer(def_k_PerceptronFeedForward, def_pff_weights,
                                m_cWeights.GetIndex()))

    return false;

if(!m_cOpenCL.SetArgumentBuffer(def_k_PerceptronFeedForward, def_pff_outputs,
                                m_cOutputs.GetIndex()))

    return false;

if(!m_cOpenCL.SetArgument(def_k_PerceptronFeedForward, def_pff_inputs_total,
                           input_data.Total()))

    return false;
```

In the `NDRange` array, we will specify the number of parallel threads required by the number of neurons in the current layer and launch the kernel for execution. Note that the *Execute* method does not literally start kernel execution, but only queues it for execution. The kernel is launched directly when you try to read the results of its operation. However, we will not download the results of each kernel's operations. Instead, we'll queue up a forward pass through the entire section and download only the result of the model's work from the last layer. This will take up the entire queue of operations. Thus, we will reduce the amount of data transferred and the time it takes to download it.

In the case of dynamic memory allocation, after queuing the kernel, it will be necessary to load all changes from the OpenCL context into the data matrices and delete unused buffers from the context. Note that you need to download the contents of all buffers whose data changes during the kernel operation.

```
//--- putting the kernel in the execution queue
uint off_set[] = {0};
uint NDRange[] = {m_cOutputs.Total()};
if(!m_cOpenCL.Execute(def_k_PerceptronFeedForward, 1, off_set, NDRange))
    return false;
}
//---
return m_cActivation.Activation(m_cOutputs);
}
```

After performing the above-described operations, we call the activation method of the required activation function class and exit the method.

It is also necessary to supplement the code for backpropagation methods. In the gradient computation kernel at the output of the neural network, three buffers are used: for target values, for the results of the last feed-forward pass, and for writing the obtained gradients. We'll check them at the beginning of the OpenCL block.

```
bool CNeuronBase::CalcOutputGradient(CBufferType* target, ENUM_LOSS_FUNCTION loss)
{
    //--- control block
    if(!target || !m_cOutputs || !m_cGradients ||
```

3. Building the first neural network model in MQL5

```
target.Total() < m_cOutputs.Total() ||
m_cGradients.Total() < m_cOutputs.Total())
return false;

//--- algorithm branching depending on the operating device
if(!m_cOpenCL)
{
    switch(loss)
    {
        case LOSS_MAE:
            m_cGradients.m_mMatrix = target.m_mMatrix - m_cOutputs.m_mMatrix;
            break;
        case LOSS_MSE:
            m_cGradients.m_mMatrix = (target.m_mMatrix - m_cOutputs.m_mMatrix) * 2;
            break;
        case LOSS_CCE:
            m_cGradients.m_mMatrix=target.m_mMatrix/(m_cOutputs.m_mMatrix+FLT_MIN)*
                log(m_cOutputs.m_mMatrix) * (-1);

            break;
        case LOSS_BCE:
            m_cGradients.m_mMatrix = (target.m_mMatrix-m_cOutputs.m_mMatrix)/
                (MathPow(m_cOutputs.m_mMatrix,2) -
                m_cOutputs.m_mMatrix+FLT_MIN);

            break;
        default:
            m_cGradients.m_mMatrix = target.m_mMatrix - m_cOutputs.m_mMatrix;
            break;
    }
}

else // OpenCL block
{
    //--- checking data buffers
    if(target.GetIndex() < 0)
        return false;
    if(m_cOutputs.GetIndex() < 0)
        return false;
    if(m_cGradients.GetIndex() < 0)
        return false;
}
```

Next, we will specify their indices in our kernel parameters. We will also specify the loss function used in the kernel parameters.

```
//--- pass arguments to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcOutputGradient, def_outgr_target,
                                target.GetIndex()))

    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcOutputGradient, def_outgr_outputs,
                                m_cOutputs.GetIndex()))

    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcOutputGradient,def_outgr_gradients,
                                m_cGradients.GetIndex()))

    return false;
```

```

        if(!m_cOpenCL.SetArgument(def_k_CalcOutputGradient, def_outgr_loss_function,
                                   (int)loss))

            return false;

```

The number of independent operation threads launched equals the number of neurons at the output of our model.

Start the kernel execution and complete the method.

```

        //--- put the kernel in the execution queue
        uint NDRange[] = { m_cOutputs.Total() };
        uint off_set[] = {0};
        if(!m_cOpenCL.Execute(def_k_CalcOutputGradient, 1, off_set, NDRange))
            return false;
    }
    //---
    return true;
}

```

The process of distributing the gradient through the hidden layer to the neurons of the previous layer is divided into two sub-processes. In the first buffer, we will adjust the error gradient based on the derivative of the activation function, and in the second one, we will distribute the error gradient values to the neurons of the previous layer according to their influence on the final result. We have created a separate kernel for each sub-process. We placed the correction of the error gradient for the derivative of the activation function into a separate class of the activation function. Therefore, in the *CalcHiddenGradient* method, we will only have to launch the error gradient distribution kernel in the OpenCL program.

```

bool CNeuronBase::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    //--- adjust the incoming gradient by the derivative of the activation function.
    if(!m_cActivation.Derivative(m_cGradients))
        return false;
    //--- check the buffers of the previous layer
    if(!prevLayer)
        return false;
    CBufferType *input_data = prevLayer.GetOutputs();
    CBufferType *input_gradient = prevLayer.GetGradients();
    if(!input_data || !input_gradient ||
        input_data.Total() != input_gradient.Total())
        return false;
    //--- check the match between the size of the input data buffer and the weight matrix
    if(!m_cWeights || m_cWeights.Cols() != (input_data.Total() + 1))
        return false;
    //--- algorithm branching depending on the operating device
    if(!m_cOpenCL)
    {
        MATRIX grad = m_cGradients.m_mMatrix.MatMul(m_cWeights.m_mMatrix);
        grad.Reshape(input_data.Rows(), input_data.Cols());
        input_gradient.m_mMatrix = grad;
    }
}

```

Again, at the beginning of the OpenCL block, we check for the availability of previously created buffers in the OpenCL context for the current kernel to work.

```
else // OpenCL block
{
    //--- check data buffers
    if(m_cWeights.GetIndex() < 0)
        return false;
    if(input_gradient.GetIndex() < 0)
        return false;
    if(m_cGradients.GetIndex() < 0)
        return false;
```

After successfully passing the control block, we will pass the buffer handles and the number of neurons in the layer to the kernel.

```
//--- pass arguments to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcHiddenGradient,
                                def_hidgr_gradient_inputs, input_gradient.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcHiddenGradient, def_hidgr_weights,
                                m_cWeights.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcHiddenGradient, def_hidgr_gradients,
                                m_cGradients.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_CalcHiddenGradient, def_hidgr_outputs_total,
                            m_cGradients.Total()))
    return false;
```

The number of threads in this case will be equal to the number of neurons in the previous layer. We will write their value to the first element of the *NDRange* array. Let's start kernel operations.

```
//--- put the kernel in the execution queue
uint NDRange[] = {input_data.Total()};
uint off_set[] = {0};
if(!m_cOpenCL.Execute(def_k_CalcHiddenGradient, 1, off_set, NDRange))
    return false;
}
//---
return true;
}
```

After propagating the error gradient across all neurons in our network based on their influence on the final result, the next step is to organize the process of updating the weight matrix. We have divided this process into two sub-processes. The weight matrix will not always be updated after every iteration. Therefore, at each iteration, we calculate the error gradient for each weight and add it to a separate buffer. Upon receiving a command from the main program, we adjust the weight matrix by the size of the batch, which gives us the average value from the accumulated error gradient.

Error gradients are accumulated in the *CalcDeltaWeights* method. To perform the kernel operations of this method, we need three buffers:

- the buffer of the results of the last direct pass of the previous layer,

3. Building the first neural network model in MQL5

- the current layer's gradient buffer,
- the buffer for accumulating weight gradients.

```
bool CNeuronBase::CalcDeltaWeights(CNeuronBase *prevLayer, bool read);
{
//--- control block
    if(!prevLayer || !m_cDeltaWeights || !m_cGradients)
        return false;
    CBufferType *Inputs = prevLayer.GetOutputs();
    if(!Inputs)
        return false;
//--- algorithm branching depending on the operating device
    if(!m_cOpenCL)
    {
        MATRIX m = Inputs.m_mMatrix;
        m.Resize(1, Inputs.Total() + 1);
        m[0, Inputs.Total()] = 1;
        m = m_cGradients.m_mMatrix.Transpose().MatMul(m);
        m_cDeltaWeights.m_mMatrix += m;
    }
}
```

First, as usual, we check the availability of used buffers in the OpenCL context.

```
else // OpenCL block
{
//--- check data buffers
    if(m_cGradients.GetIndex() < 0)
        return false;
    if(m_cDeltaWeights.GetIndex() < 0)
        return false;
    if(Inputs.GetIndex() < 0)
        return false;
}
```

We pass the pointers to them to the kernel parameters.

```
//--- pass arguments to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcDeltaWeights,
                                def_delt_delta_weights, m_cDeltaWeights.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcDeltaWeights, def_delt_inputs,
                                Inputs.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcDeltaWeights, def_delt_gradients,
                                m_cGradients.GetIndex()))
    return false;
```

In this case, we will use a two-dimensional task space to launch the kernel. In one dimension, we specify the number of neurons in the current layer, and in the other dimension, the number of neurons in the previous layer.

After the preparatory work is completed, we will start the kernel execution.

Then we will check the data reading flag and, if necessary, load the result of operations from the context.

And of course, do not forget to monitor the process of performing operations at every step.

```

    //--- put the kernel in the execution queue
    uint NDRange[] = {m_cGradients.Total(), Inputs.Total()};
    uint off_set[] = {0, 0};
    if(!m_cOpenCL.Execute(def_k_CalcDeltaWeights, 2, off_set, NDRange))
        return false;
    if(read && !m_cDeltaWeights.BufferRead())
        return false;
}
//---
return true;
}

```

We are successfully moving forward in the process of creating our project. To complete the work on the fully connected neuron, we need to describe the sub-process of updating the weight matrix. In our project, we decided to implement several algorithms for updating the weights. We have created our own kernel for each algorithm for updating the weight matrix. Let's add calls to these kernels to the corresponding methods of our class.

We will start with the stochastic gradient descent method. The implementation of this method requires only two buffers: accumulated deltas and the weight matrix. We check the availability of these buffers in the OpenCL context.

```

bool CNeuronBase::SGDUpdate(int batch_size, TYPE learningRate, VECTOR &Lambda)
{
    //--- algorithm branching depending on the operating device
    if(!m_cOpenCL)
    {
        TYPE lr = learningRate / ((TYPE)batch_size);
        m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
        m_cWeights.m_mMatrix += m_cDeltaWeights.m_mMatrix * lr;
        m_cDeltaWeights.m_mMatrix.Fill(0);
    }
    else // OpenCL block
    {
        //--- check data buffers
        if(m_cWeights.GetIndex() < 0)
            return false;
        if(m_cDeltaWeights.GetIndex() < 0)
            return false;
    }
}

```

Then we will pass pointers to them to the kernel parameters. In addition, we need to transfer training parameters to the kernel:

- *batch_size*
- *learningRate*
- *Lambda* vector (regularization parameters)

```

    //--- pass arguments to the kernel
    if(!m_cOpenCL.SetArgumentBuffer(def_k_SGDUpdate, def_sgd_delta_weights,
                                    m_cDeltaWeights.GetIndex()))
        return false;
}

```

3. Building the first neural network model in MQL5

```
if(!m_cOpenCL.SetArgumentBuffer(def_k_SGDUpdate, def_sgd_weights,
                                m_cWeights.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_SGDUpdate, def_sgd_total,
                           (int)m_cWeights.Total()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_SGDUpdate, def_sgd_batch_size, batch_size))
    return false;
if(!m_cOpenCL.SetArgument(def_k_SGDUpdate, def_sgd_learningRate,
                           learningRate))
    return false;
if(!m_cOpenCL.SetArgument(def_k_SGDUpdate, def_sgd_Lambda1, Lambda[0]))
    return false;
if(!m_cOpenCL.SetArgument(def_k_SGDUpdate, def_sgd_Lambda2, Lambda[1]))
    return false;
```

Let's determine the number of threads to be launched. There will be four times fewer elements in these buffers than in the weight matrix. This effect is achieved through the use of vector operations.

Please note the following while working with the algorithm for determining the number of threads. We can't just divide the number of neurons by four because we can't be sure that the number of neurons will always be a multiple of four. But we must be sure that the number of threads covers all neurons in our layer. So we need a function similar to rounding up to an integer. Instead, we will use the property of integer division to discard the fractional part, in other words, rounding down. To get the result we want, before dividing by the vector size, we'll increase the number of neurons by a value one greater than the vector size. After such a small mathematical trick, the result of integer division will be the required number of threads. When using this trick, you should be particularly careful with the data type used because the desired effect can only be achieved when all variables in the operation are integers.

```
//--- put the kernel in the execution queue
int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
int off_set[] = {0};
if(!m_cOpenCL.Execute(def_k_SGDUpdate, 1, off_set, NDRange))
    return false;
}
return true;
}
```

After the preparatory work, we will request the kernel to be completed.

In the description of the weight matrix update process using the accumulated momentum method, we have an additional buffer for storing moments and a momentum averaging coefficient. For the rest, the principles of constructing the algorithm laid down in the previous method are preserved.

```
bool CNeuronBase::MomentumUpdate(int batch_size, TYPE learningRate,
                                  VECTOR &Beta, VECTOR &Lambda)
{
    if(Beta[0] == 0)
        return SGDUpdate(batch_size, learningRate, Lambda);
    //--- control block
    if(!m_cMomenum[0])
        return false;
    if(m_cMomenum[0].Total() < m_cWeights.Total())
```


3. Building the first neural network model in MQL5

```
        return false;
//--- algorithm branching depending on the operating device
if(!m_cOpenCL)
{
    TYPE lr = learningRate / ((TYPE)batch_size);
    m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
    m_cMomenum[0].m_mMatrix = m_cDeltaWeights.m_mMatrix * lr +
                                m_cMomenum[0].m_mMatrix * Beta[0] ;
    m_cWeights.m_mMatrix += m_cMomenum[0].m_mMatrix;
    m_cDeltaWeights.m_mMatrix.Fill(0);
}

else // OpenCL block
{
    //--- check data buffers
    if(m_cWeights.GetIndex() < 0)
        return false;
    if(m_cDeltaWeights.GetIndex() < 0)
        return false;
    if(m_cMomenum[0].GetIndex() < 0)
        return false;

    //--- pass arguments to the kernel
    if(!m_cOpenCL.SetArgumentBuffer(def_k_MomentumUpdate,
                                    def_moment_delta_weights, m_cDeltaWeights.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_MomentumUpdate, def_moment_weights,
                                    m_cWeights.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_MomentumUpdate,
                                    def_moment_momentum, m_cMomenum[0].GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_total,
                              (int)m_cWeights.Total()))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_batch_size,
                              batch_size))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_learningRate,
                              learningRate))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_Lambda1,
                              Lambda[0]))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_Lambda2,
                              Lambda[1]))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_beta, Beta[0]))
        return false;
}
```

We will set the number of threads to 4 times less than the number of elements in the weight matrix and start performing operations.

3. Building the first neural network model in MQL5

```
//--- put the kernel in the execution queue
int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
int off_set[] = {0};
if (! m_copengl. Execute (def_k_momentumUpdate, 1, off_set, ndRange))
    return false;
}
return true;
}
```

Please note the constants used in kernels and their parameters. Despite the similarity of operations, a small detail or a typo with a constant can often lead to a critical error and program termination.

Let's move on to the next implementation. The *AdaGrad* optimization method is implemented in the *AdaGradUpdate* method and in the respective kernel, which we will identify by the *def_k_AdaGradUpdate* constant. To avoid possible errors when specifying parameters, all parameter constants for this kernel start with *def_adagrad_*. As you can see, all constant names are intuitive and logically connected. This reduces the risk of a possible error. This method is very convenient when there are a large number of constants.

The *AdaGrad* method, like the cumulative pulse method, uses a moment accumulation buffer. However, unlike the previous method, there is no averaging factor here. At this point, we don't care about differences in the use of parameters and buffers. We are only interested in their availability: the use of buffers and parameters is already described in the OpenCL program kernel, and here we organize the process of transferring data from the main program to the OpenCL context.

The algorithm for organizing the process of working with the OpenCL context in the *AdaGradUpdate* method is similar to that used in the methods described earlier.

- First, check for buffers in the OpenCL context.
- Then we will send pointers to buffers and optimization parameters to the kernel.
- Start kernel execution.

```
bool CNeuronBase::AdaGradUpdate(int batch_size, TYPE learningRate, VECTOR &Lambda)
{
//--- control block
if(!m_cMomenum[0])
    return false;
if(m_cMomenum[0].Total() < m_cWeights.Total())
    return false;
//--- algorithm branching depending on the operating device
if(!m_cOpenGL)
{
    m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
    MATRIX delta = m_cDeltaWeights . m_mMatrix /((TYPE) batch_size);
    MATRIX G = m_cMomenum[0].m_mMatrix = m_cMomenum[0].m_mMatrix + delta.Power(2);
    G = MathPow(MathSqrt(G) + 1e-32, -1);
    G = G * learningRate;
    m_cWeights.m_mMatrix += G * delta;
    m_cDeltaWeights.m_mMatrix.Fill(0);
}
else // OpenCL block
{

```

3. Building the first neural network model in MQL5

```
//--- check data buffers
if(m_cWeights.GetIndex() < 0)
    return false;
if(m_cDeltaWeights.GetIndex() < 0)
    return false;
if(m_cMomenum[0].GetIndex() < 0)
    return false;

//--- pass arguments to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaGradUpdate,
                                def_adagrad_delta_weights, m_cDeltaWeights.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaGradUpdate, def_adagrad_weights,
                                m_cWeights.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaGradUpdate, def_adagrad_momentum,
                                m_cMomenum[0].GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_AdaGradUpdate, def_adagrad_total,
                           (int)m_cWeights.Total()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_AdaGradUpdate, def_adagrad_batch_size,
                           batch_size))
    return false;
if (! m_copencil. SetArgument (Def_K_AdaGradUpdate, Def_Adagrad_LearningRate,
                               learningRate))
    return false;
if(!m_cOpenCL.SetArgument(def_k_AdaGradUpdate, def_adagrad_Lambda1,
                           Lambda[0]))
    return false;
if(!m_cOpenCL.SetArgument(def_k_AdaGradUpdate, def_adagrad_Lambda2,
                           Lambda[1]))
    return false;

//--- put the kernel in the execution queue
int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
int off_set[] = {0};
if(!m_cOpenCL.Execute(def_k_AdaGradUpdate, 1, off_set, NDRange))
    return false;
}
return true;
}
```

The RMSProp optimization method is functionally similar to AdaGrad, but it includes a coefficient for averaging the accumulated momentum.

We're following the established framework: check the availability of OpenCL context buffers, then send pointers to buffers and optimization parameters to the kernel while also ensuring the use of the proper method and constant naming:

- **RMS PropUpdate** method
- **def_k_ RMSPropUpdate** kernel constant
- **def_rms_** parameter constants

After specifying the parameters, launch the kernel.

```
bool CNeuronBase::RMSPropUpdate(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda)
{
    //--- control block
    if(!m_cMomenum[0])
        return false;
    if(m_cMomenum[0].Total() < m_cWeights.Total())
        return false;
    //--- algorithm branching depending on the operating device
    if(!m_cOpenCL)
    {
        TYPE lr = learningRate;
        m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
        MATRIX delta = m_cDeltaWeights.m_mMatrix / ((TYPE) batch_size);
        MATRIX G = m_cMomenum[0].m_mMatrix = m_cMomenum[0].m_mMatrix * Beta[0] +
                                                    delta.Power(2) * (1 - Beta[0]);

        G = MathPow(MathSqrt(G) + 1e-32, -1);
        G = G * learningRate;
        m_cWeights.m_mMatrix += G * delta;
        m_cDeltaWeights.m_mMatrix.Fill(0);
    }

    else // OpenCL block
    {
        //--- check data buffers
        if(m_cWeights.GetIndex() < 0)
            return false;
        if(m_cDeltaWeights.GetIndex() < 0)
            return false;
        if(m_cMomenum[0].GetIndex() < 0)
            return false;

        //--- pass arguments to the kernel
        if(!m_cOpenCL.SetArgumentBuffer(def_k_RMSPropUpdate, def_rms_delta_weights,
                                         m_cDeltaWeights.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgumentBuffer(def_k_RMSPropUpdate, def_rms_weights,
                                         m_cWeights.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgumentBuffer(def_k_RMSPropUpdate, def_rms_momentum,
                                         m_cMomenum[0].GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_total,
                                   (int)m_cWeights.Total()))
            return false;
        if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_batch_size,
                                   batch_size))
            return false;
    }
}
```

3. Building the first neural network model in MQL5

```
        if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_learningRate,
                                   learningRate))
            return false;
        if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_Lambda1, Lambda[0]))
            return false;
        if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_Lambda2, Lambda[1]))
            return false;
        if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_beta, Beta[0]))
            return false;

        //--- put the kernel in the execution queue
        int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
        int off_set[] = {0};
        if(!m_cOpenCL.Execute(def_k_RMSPropUpdate, 1, off_set, NDRange))
            return false;
    }
    //---
    return true;
}
```

The developers of the *AdaDelta* method opted to not use a learning rate but compensated for it by introducing an additional buffer for moments with an additional averaging coefficient. Accordingly, we will use one more buffer in this kernel.

When setting kernel parameters, again, mind the naming:

- ***AdaDeltaUpdate*** method
- *def_k_AdaDeltaUpdate* kernel constant
- *def_adadelt* parameter constants

Furthermore, for seamless portability of the constructed neural network, we need to ensure the consistency of buffer usage in terms of performing operations using MQL5 and in the OpenCL context. When used within the same platform, changing the sequence in which the momentum arrays are used will not have an effect. Whatever we call them, their content will be appropriate to the context of use. However, when transferring a pre-trained neural network to another platform, we will likely get unexpected results. At the same time, we should remember the purpose and functionality of arrays. The moments are only used during the weight matrix update process in the training of the neural network and do not participate in the feed-forward pass. So, the impact of mixed-up buffers will only become apparent when attempting to retrain the neural network. This should not be neglected. If we use a once built neural network for a long time, we will need to periodically refine it. This is necessary to keep weights relevant in our changing world.

Taking into account the above, we will pass pointers to the loaded buffers and training parameters to the kernel.

Let's calculate the number of required threads and launch the kernel.

```
bool CNeuronBase::AdaDeltaUpdate(int batch_size, VECTOR &Beta, VECTOR &Lambda)
{
    //--- control block
    for(int i = 0; i < 2; i++)
    {
        if(!m_cMomenum[i])
            return false;
    }
}
```

3. Building the first neural network model in MQL5

```
        if(m_cMomenum[i].Total() < m_cWeights.Total())
            return false;
    }
//--- algorithm branching depending on the operating device
if(!m_cOpenCL)
{
    MATRIX delta = m_cDeltaWeights . m_mMatrix /((TYPE) batch_size);
    MATRIX W = m_cMomenum[0].m_mMatrix = m_cMomenum[0].m_mMatrix * Beta[0] +
            m_cWeights.m_mMatrix.Power(2) * (1 - Beta[0]);
    m_cMomenum[1].m_mMatrix = m_cMomenum[1].m_mMatrix * Beta[1] +
            delta.Power(2) * (1 - Beta[1]);
    m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
    W = MathSqrt(W) / (MathSqrt(m_cMomenum[1].m_mMatrix) + 1e-32);
    m_cWeights.m_mMatrix += W * delta;
    m_cDeltaWeights.m_mMatrix.Fill(0);
}

else // OpenCL block
{
    //--- create data buffers
    if(m_cWeights.GetIndex() < 0)
        return false;
    if(m_cDeltaWeights.GetIndex() < 0)
        return false;
    if(m_cMomenum[0].GetIndex() < 0)
        return false;
    if(m_cMomenum[1].GetIndex() < 0)
        return false;

    //--- pass arguments to the kernel
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaDeltaUpdate,
            def_adadelt_delta_weights, m_cDeltaWeights.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaDeltaUpdate, def_adadelt_weights,
            m_cWeights.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaDeltaUpdate, def_adadelt_momentumW,
            m_cMomenum[0].GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaDeltaUpdate, def_adadelt_momentumG,
            m_cMomenum[1].GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_total,
            (int)m_cWeights.Total()))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_batch_size,
            batch_size))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_Lambda1,
            Lambda[0]))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_Lambda2,
```

```

                                                                    Lambda[1]))
    return false;
    if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_beta1, Beta[0]))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_beta2, Beta[1]))
        return false;

    //--- put the kernel in the execution queue
    int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
    int off_set[] = {0};
    if(!m_cOpenCL.Execute(def_k_AdaDeltaUpdate, 1, off_set, NDRange))
        return false;
}
//---
return true;
}

```

Our description of the operations performed in the fully connected neural layer is nearing completion. One method remains to be described, and it's the weight update method – specifically, the *Adam* optimization algorithm. This method, though the last on the list, is not of lesser importance. Like *AdaDelta*, the *Adam* method also employs two momentum buffers, but in addition, it returns the learning rate.

Let's recap the main stages of our algorithm and highlight key checkpoints:

- Verify the presence of the necessary data in the OpenCL context memory.
- Pass pointers to data buffers and training parameters to the kernel. Ensure naming consistency: Method ***AdamUpdate*** → kernel constant ***def_k_AdamUpdate*** → parameter constants ***def_adam_...***
- Monitor the consistent use of buffers between MQL5 and the OpenCL context.
- Execute the kernel.

```

bool CNeuronBase::AdamUpdate(int batch_size, TYPE learningRate,
                             VECTOR &Beta, VECTOR &Lambda)
{
    //--- control block
    for(int i = 0; i < 2; i++)
    {
        if(!m_cMomenum[i])
            return false;
        if(m_cMomenum[i].Total() != m_cWeights.Total())
            return false;
    }
    //--- algorithm branching depending on the operating device
    if(!m_cOpenCL)
    {
        MATRIX delta = m_cDeltaWeights . m_mMatrix / ((TYPE) batch_size);
        m_cMomenum[0].m_mMatrix = m_cMomenum[0].m_mMatrix * Beta[0] +
                                delta * (1 - Beta[0]);
        m_cMomenum[1].m_mMatrix = m_cMomenum[1].m_mMatrix * Beta[1] +
                                MathPow(delta,2) * (1 - Beta[1]);
        MATRIX M = m_cMomenum[0].m_mMatrix / (1 - Beta[0]);
    }
}

```

3. Building the first neural network model in MQL5

```
MATRIX V = m_cMomenum[1].m_mMatrix / (1 - Beta[1]);
m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
m_cWeights.m_mMatrix += M * learningRate / MathSqrt(V);
m_cDeltaWeights.m_mMatrix.Fill(0);
}

else // OpenCL block
{
    //--- check data buffers
    if(m_cWeights.GetIndex() < 0)
        return false;
    if(m_cDeltaWeights.GetIndex() < 0)
        return false;
    if(m_cMomenum[0].GetIndex() < 0)
        return false;
    if(m_cMomenum[1].GetIndex() < 0)
        return false;

    //--- pass arguments to the kernel
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AdamUpdate, def_adam_delta_weights,
                                     m_cDeltaWeights.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AdamUpdate, def_adam_weights,
                                     m_cWeights.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AdamUpdate, def_adam_momentumM,
                                     m_cMomenum[0].GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AdamUpdate, def_adam_momentumV,
                                     m_cMomenum[1].GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_total,
                              (int)m_cWeights.Total()))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_batch_size,
                              batch_size))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_Lambda1, Lambda[0]))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_Lambda2, Lambda[1]))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_beta1, Beta[0]))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_beta2, Beta[1]))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_learningRate,
                              learningRate))
        return false;

    //--- put the kernel in the execution queue
    int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
    int off_set[] = {0};
    if(!m_cOpenCL.Execute(def_k_AdamUpdate, 1, off_set, NDRange))
```



```

        return false;
    }
    //---
    return true;
}

```

We have completed a description of the processes of a fully connected neural layer. Now, we've reached the stage where we can look at the work done and assess the initial results. In fact, we already have enough created base classes to build a small perceptron model with several fully connected layers. One of them will serve as the receiver of input data (input layer), the last neural layer will produce the results (output layer), and hidden layers will be in between.

3.8 Implementing the perceptron model in Python

To implement the fully connected perceptron model in Python, we will use the [template](#) we created earlier. As you may recall, in this template, we left the description of the neural layers in our model unfilled.

```

# Create a neural network model
model = keras.Sequential([keras.Input(shape=inputs),
                           # Fill the model with a description of the neural layers
                           ])

```

To create fully connected layers in a neural network, we will use the *layers.Dense* class from the *Keras* library. The following operation is performed within this layer:

$$Output = activation(dot(input, kernel) + bias)$$

where:

- **activation** = activation function, set in parameters
- **input** = an array of source data
- **kernel** = a weight matrix
- **dot** = a vector multiplication operation
- **bias** = a displacement element

Dense provides parameters to control the neural layer creation process:

- *units* – the dimension of the output space (the number of neurons in the layer);
- *activation* – the activation function used;
- *use_bias* – is an optional parameter that indicates whether to use a vector of displacement elements;
- *kernel_initializer* – the method of initializing the matrix of weights;
- *bias_initializer* – a method for initializing a vector of displacement elements;
- *kernel_regularizer* – a weight matrix regularization method;
- *bias_regularizer* – a method for regularizing the displacement vector;
- *activity_regularizer* – a method of regularizing the activation function;
- *kernel_constraint* – a weight matrix restriction function;
- *bias_constraint* – a displacement vector restriction function.

3. Building the first neural network model in MQL5

Please note that you cannot change the settings after the first access to the layer.

In addition to the above parameters, *Dense* can take an *input_shape* parameter that indicates the size of the input array. This parameter is valid only for the first layer of the neural network. When this parameter is used, an input layer is created to be inserted in front of the current layer. The operation can be considered as an equivalent to explicitly defining the input layer.

We'll start implementing our first neural network model by copying our script template into a new *perceptron.py* file. In the created file, we will create the first model with one hidden layer of 40 neurons and 2 neurons in the results layer. In the hidden layer, we'll use *Swish* as an activation function. The neurons in the output layer will be activated by the hyperbolic tangent.

```
# Create a neural network model
model1 = keras.Sequential([keras.Input(shape=inputs),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(targetts, activation=tf.nn.tanh)
                           ])
```

In theory, this is enough to start training the model. However, we study the operation of various models and want to understand the influence of changing the neural network architecture on the model ability to learn and generalize the initial data. So I've added two more models. One model has two additional hidden layers. The result is a model with three hidden layers. All three hidden layers are completely identical: they have 40 elements each and are activated by the *Swish* function. The first and last layers remain unchanged.

```
# Create a model with three hidden layers
model2 = keras.Sequential([keras.Input(shape=inputs),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(targetts, activation=tf.nn.tanh)
                           ])
```

The following steps should be repeated for each model. First, let's prepare the model for training using the *compile* method.

```
model2.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
```

After that, we will start the model training process and save the trained model.

```
history2 = model2.fit(train_data, train_target,
                     epochs=500, batch_size=1000,
                     callbacks=[callback],
                     verbose=2,
                     validation_split=0.2,
                     shuffle=True)
model2.save(os.path.join(path, 'perceptron2.h5'))
```

We will build the third model on the basis of the second model with the addition of regularization. For each neural layer, we will specify in the *kernel_regularizer* parameter the *keras.regularizers.l1_l2* class object with the *L1* and *L2*-regularization parameters. As you can see from the class name, we'll be using *ElasticNet*.

3. Building the first neural network model in MQL5

```
# Add regularization to the model with three hidden layers
model3 = keras.Sequential([keras.Input(shape=inputs),
    keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(targets, activation=tf.nn.tanh)
])
```

Next, we'll compile and train the model. All three models use identical training parameters. This will make it possible to directly assess the impact of the model architecture on the learning outcome. At the same time, we will eliminate the influence of other factors as much as possible.

```
model3.compile(optimizer='Adam',
    loss='mean_squared_error',
    metrics=['accuracy'])
history3 = model3.fit(train_data, train_target,
    epochs=500, batch_size=1000,
    callbacks=[callback],
    verbose=2,
    validation_split=0.2,
    shuffle=True)
model3.save(os.path.join(path, 'perceptron3.h5'))
```

Since we are training not one but three models in this script, we also need to correct the visualization unit. Let's display the training results of all three models on one graph. This will demonstrate differences in the training and validation process. We will make changes to the blocks for constructing both graphs.

```
# Plot the training results of the three models
plt.figure()
plt.plot(history1.history['loss'],
    label='Train 1 hidden layer')
plt.plot(history1.history['val_loss'],
    label='Validation 1 hidden layer')
plt.plot(history2.history['loss'],
    label='Train 3 hidden layers')
plt.plot(history2.history['val_loss'],
    label='Validation 3 hidden layers')
plt.plot(history3.history['loss'],
    label='Train 3 hidden layers vs regularization')
plt.plot(history3.history['val_loss'],
    label='Validation 3 hidden layer vs regularization')
plt.ylabel('$MSE$ $Loss$')
plt.xlabel('$Epochs$')
plt.title('Dynamic of Models train')
plt.legend(loc='lower left')

plt.figure()
plt.plot(history1.history['accuracy'],
    label='Train 1 hidden layer')
```

3. Building the first neural network model in MQL5

```
plt.plot(history1.history['val_accuracy'],
         label='Validation 1 hidden layer')
plt.plot(history2.history['accuracy'],
         label='Train 3 hidden layers')
plt.plot(history2.history['val_accuracy'],
         label='Validation 3 hidden layers')
plt.plot(history3.history['accuracy'],
         label='Train 3 hidden layers\nvs regularization')
plt.plot(history3.history['val_accuracy'],
         label='Validation 3 hidden layer\nvs regularization')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Dynamic of Models train')
plt.legend(loc='upper left')
```

After training, our template tests the model performance on a test sample. Here, we also have to test three models under similar conditions. I'll skip the test sample loading block, as it moved from the template unchanged. Here is just a code for directly testing models.

```
# Check the results of models on a test sample
test_loss1, test_acc1 = model1.evaluate(test_data,
                                       test_target,
                                       verbose=2)
test_loss2, test_acc2 = model2.evaluate(test_data,
                                       test_target,
                                       verbose=2)
test_loss3, test_acc3 = model3.evaluate(test_data,
                                       test_target,
                                       verbose=2)
```

The test results in the template were published in a journal. Now we have the results of testing three models. It will be more efficient to compare the results on the graph. We will use the *Matplotlib* library to build graphs.

In this case, we will not display the dynamics of the process, as before, but compare the values. Therefore, it will be more convenient to use a column chart to display values. The library offers the *bar* method for constructing diagrams. This method takes two arrays in its parameters: in the first, we will specify the labels of the compared parameters, and in the second, their values. To complete the picture, let's add the title of the graph and the vertical axis using the *title* and *ylabel* methods, respectively.

```
plt.figure()
plt.bar(
    ['1 hidden layer', '3 hidden layers', '3 hidden layers\nvs regularization'],
    [test_loss1, test_loss2, test_loss3])
plt.ylabel('$MSE$ $Loss$')
plt.title('Result of test')

plt.figure()
plt.bar(
    ['1 hidden layer', '3 hidden layers', '3 hidden layers\nvs regularization'],
    [test_acc1, test_acc2, test_acc3])
plt.ylabel('$Accuracy$')
```

```
plt.title('Result of test')
```

We will see how the script works a little later. In the next chapter, we'll prepare data for training and testing models.

3.9 Creating training and testing samples

We have come quite a long way in building our library for building neural networks. We have completed the work on constructing the basic dispatcher class for our neural network and have created everything necessary for building a fully connected neural layer. We still have a lot of work to do. However, we can already build our first neural network and test its performance using real data. Since we will have several implementations of various architectural solutions, to compare the results of model performance, we will take a small data subset. Let's create two data samples: a larger one for training the neural network and a smaller one for testing the trained neural network.

Allocating a separate sample for training is a common practice. During the training process of a neural network, weights are adjusted in such a way that the neural network accurately describes the training dataset to the best extent possible. By using a sufficiently large number of weights, the neural network is able to learn the training sample down to the smallest detail. However, in doing so, the neural network loses the ability to generalize the data. In such a state, the neural network is called "*overfitted*". It is not possible to detect this in a training sample. However, if you compare the performance of the neural network on the training dataset and on data that is not part of the training dataset, the difference in results will clearly indicate this. A slight deterioration of the results on the test sample is allowed, but it should not be drastic. Of course, the data in the samples should be comparable. Most often, to achieve this, the overall available data is randomly divided into two sets in a ratio of 70-80% for the training dataset and 20-30% for the testing dataset. In most cases, it will be necessary to divide the general population into three subsamples:

- training 60%
- validation 20%
- test 20%

The validation dataset is used to select the best training parameters and neural network architecture. However, we will not be using a validation dataset at this point, as we would like to compare different implementations under otherwise equal conditions.

To generate samples, let's create the script *create_initial_data.mq5*. In the script we will specify the following parameters:

- The period for loading data is specified as a start date and an end date; within this period, we will retrieve historical data and indicator data from the server;
- Timeframe used to load the analyzed data;
- Number of analyzed historical bars per pattern;
- Name of files for recording training and test samples;
- Data normalization flag.

Earlier, we discussed extensively the importance of normalizing the data that is fed into the neural network as input. Now, we can practically verify how data normalization affects the results of training the neural network. It is to assess the impact of this factor that I introduced the data normalization parameter. Here, it's important to note that the data fed into the neural network as input should be comparable both in the testing and training datasets, as well as during the practical application

operation of the neural network. Therefore, in practice, it will be necessary to store the normalization parameters and use them when normalizing data coming in during the practical application of the neural network.

Recall that in the section on [selecting the input data](#) to feed into the neural network, we selected two indicators: RSI and MACD. We will use them during the process of training neural networks within the practical experiments outlined in this book.

Let's look at the script algorithm. Initially, following the analogy with the scripts discussed while selecting the source data, we will connect the selected indicators to the chart and obtain handles for accessing the indicator data.

```
//+-----+
//| External parameters for script operation |
//+-----+
// Beginning of the period of the general population
input datetime Start = D'2015.01.01 00:00:00';
// End of the period of the general population
input datetime End = D'2020.12.31 23:59:00';
// Timeframe for data loading
input ENUM_TIMEFRAMES TimeFrame = PERIOD_M5;
// Number of historical bars in one pattern
input int BarsToLine = 40;
// File name for recording the training sample
input string StudyFileName = "study_data.csv";
// File name for recording the test sample
input string TestFileName = "test_data.csv";
// Data normalization flag
input bool NormalizeData = true;
//+-----+
//| Beginning of the script program |
//+-----+
void OnStart(void)
{
//--- Connect indicators to the chart
int h_ZZ = iCustom(_Symbol, TimeFrame, "Examples\\ZigZag.ex5", 48, 1, 47);
int h_RSI = iRSI(_Symbol, TimeFrame, 12, PRICE_TYPICAL);
int h_MACD = iMACD(_Symbol, TimeFrame, 12, 48, 12, PRICE_TYPICAL);
double close[];
if(CopyClose(_Symbol, TimeFrame, Start, End, close) <= 0)
return;
```

After that, we check the validity of the obtained handles and load the historical data of indicators into dynamic arrays. It should be noted that for the ZigZag indicator, we will load a bit more data. The reason for this is the specifics of this indicator. The buffer of this indicator points only to the found extrema. In other cases, the indicator returns zero values. Therefore, for the last patterns analyzed, the target values may be outside the analyzed period.

```
//--- Load indicator data into dynamic arrays
double zz[], macd_main[], macd_signal[], rsi[];
datetime end_zz = End + PeriodSeconds(TimeFrame) * 500;
if(h_ZZ == INVALID_HANDLE ||
CopyBuffer(h_ZZ, 0, Start, end_zz, zz) <= 0)
```

3. Building the first neural network model in MQL5

```
{
    PrintFormat("Error loading indicator %s data", "ZigZag");
    return;
}
if(h_RSI == INVALID_HANDLE ||
    CopyBuffer(h_RSI, 0, Start, End, rsi) <= 0)
{
    PrintFormat("Error loading indicator %s data", "RSI");
    return;
}
if(h_MACD == INVALID_HANDLE ||
    CopyBuffer(h_MACD, MAIN_LINE, Start, End, macd_main) <= 0 ||
    CopyBuffer(h_MACD, SIGNAL_LINE, Start, End, macd_signal) <= 0)
{
    PrintFormat("Error loading indicator %s data", "MACD");
    return;
}
```

In addition to the selected indicators, let's load the candlestick closing prices. We will use these handles to determine the direction of price movement towards the nearest extremum and the strength of the upcoming movement.

After loading the data, we organize the process of determining the target values at each step of the historical data. To do this, we will create a reverse loop and loop through all the values of the ZigZag indicator and if the value differs from zero, we will save it to the extremum variable. Simultaneously, we will iterate through closing price values, and by measuring the deviation of the last recorded extremum from the closing price, we will determine the direction and strength of the upcoming movement. Let's save the obtained values into dynamic arrays target1 and target2.

```
int total = ArraySize(close);
double target1[], target2[], macd_delta[], test[];
if(ArrayResize(target1, total) <= 0 ||
    ArrayResize(target2, total) <= 0 ||
    ArrayResize(test, total) <= 0 ||
    ArrayResize(macd_delta, total) <= 0)
    return;
//--- Calculate targets: direction and distance
//--- to the nearest extremum
double extremum = -1;
for(int i = ArraySize(zz) - 2; i >= 0; i--)
{
    if(zz[i + 1] > 0 && zz[i + 1] != EMPTY_VALUE)
        extremum = zz[i + 1];
    if(i >= total)
        continue;
    target2[i] = extremum - close[i];
    target1[i] = (target2[i] >= 0 ? 1 : -1);
    macd_delta[i] = macd_main[i] - macd_signal[i];
}
```

Here, it's important to note that on a time chart, the extremum should always come after the analyzed closing price. Therefore, the closing price is taken from the previous bar compared to the last checked value of the ZigZag indicator.

In the same loop, we will determine the distance between the main and signal lines of the MACD indicator and store them in a separate dynamic array called `macd_delta`.

After calculating the targets and the distance between the MACD indicator lines, we normalize the data. Of course, we will perform normalization only when this requirement is specified by the user in the script parameters. The purpose of normalization is to transform the original data so that its values are in the range of -1 to 1 centered on point 0. It's important to pay attention to a series of introductory aspects that stem from the characteristics of the indicators themselves.

The RSI indicator is constructed in such a way that its values are normalized within a range from 0 to 100. Hence, we do not need to determine the maximum and minimum data value of this indicator to normalize it. Therefore, the algorithm for normalizing the readings of this indicator is limited by the constant 50 which is the middle of the range of possible indicator values. The formula for normalizing the values is as follows.

$$RSI_i^{Norm} = \frac{RSI_i - 50}{50}$$

The values of the MACD indicator do not have an upper and lower boundary of the range, but they are centered around point 0. This is because, based on the construction principles of the indicator, it reflects whether the price is above or below the moving average. The same can be said about the calculated distance between the base and signal lines of the indicator. The signal line can be either above or below the base line. However, at the moment the lines cross, the distance between them is 0. Therefore, for normalizing the data, we will take the value of the indicator and divide it by the absolute value of the maximum deviation over the analyzed period.

$$MACD_i^{Norm} = \frac{MACD_i}{Max(Abs(MACD_{min}), MACD_{max})}$$

Here, I want to once again emphasize the importance of data comparability for the training, testing dataset, and data used during practical application. If we normalize the training and test sample data now, we will have to keep the normalization parameters of all three indicators of the MACD indicator for practical application.

After defining the normalization parameters, we will organize a cycle for enumeration and appropriate correction of historical values of indicators.

Only the initial data is normalized, not the target values.

```
//--- Data normalization
if(NormalizeData)
{
    double main_norm = MathMax(MathAbs(macd_main[ArrayMinimum(macd_main)]),
                                macd_main[ArrayMaximum(macd_main)]);
    double sign_norm = MathMax(MathAbs(macd_signal[ArrayMinimum(macd_signal)]),
                                macd_signal[ArrayMaximum(macd_signal)]);
    double delt_norm = MathMax(MathAbs(macd_delta[ArrayMinimum(macd_delta)]),
                                macd_delta[ArrayMaximum(macd_delta)]);
    for(int i = 0; i < total; i++)
    {
        rsi[i] = (rsi[i] - 50.0) / 50.0;
        macd_main[i] /= main_norm;
        macd_signal[i] /= sign_norm;
        macd_delta[i] /= delt_norm;
    }
}
```



```
}  
}
```

Certainly, sometimes it can be useful to normalize the target values to fit them into the range of a specific activation function. However, in such cases, similar to normalizing input data, it's crucial to preserve the normalization parameters for decoding the neural network's outputs in industrial applications. These considerations lie at the interface between the neural network and the main program, and the solution largely depends on the specific task.

After preparing the dataset, the next step is to split it into training and testing sets. A common practice is to randomly select records from the entire dataset for the test set, with the remaining data used for training. It is highly discouraged to take consecutive patterns for the test set, whether they are the first or last in the dataset. This is primarily because a small subset of data is more susceptible to the influence of local trends. Such a sample may not be representative for extrapolating the evaluation to the entire dataset. On the other hand, randomly selecting records from the entire dataset provides a higher probability of extracting patterns that differ significantly for the test set. This kind of sample will be more independent of local trends and more representative to enable the evaluation of the neural network performance on the global dataset. However, it should be noted that there are cases where consecutive patterns are chosen for the test set, but these are specific instances related to the architecture of certain models.

To split the dataset into training and testing sets, we will create an array of flags called `test`. This array will have the same size as our global dataset. The values of its elements will indicate the usage direction of the pattern:

- 0 means the training sample
- 1 means the testing sample

For binary classification, you can also use an array of logical values. However, when we need to add a validation dataset, we can easily use the value 2 for it, whereas using an array of logical values doesn't provide us with such flexibility.

Our flag array will be first initialized with zero values. In other words, we establish that by default the pattern belongs to the training dataset. We then determine the number of patterns for the test set. Then we create a loop based on the number of elements for the testing dataset, generating random values within this loop. The random value generator should return an integer number between 0 and the size of the general population. In my solution, I used the MQL5 built-in *MathRand* function to generate pseudo-random numbers. This function returns an integer value in the range of 0 to 32767. However, since the size of the dataset is expected to be over 33,000 elements, I multiplied two random numbers. Such a version is capable of generating more than 1 billion random values. To scale the obtained random number to the size of our population, we first divide the generated random number by the square of 32767, thereby normalizing the random number within the range of 0 to 1. Then multiply by the number of elements in our general population. The resulting number will tell us the ordinal number of the pattern for the test sample.

All we have to do is write 1 to the corresponding element of the flag array. However, there is still a chance of landing twice (or even more times) on the same element of the flag array. If we do not control for this, we are very likely to get a test sample smaller than expected. Therefore, before writing 1 to the selected element of the flag array, we first check its current state. If it already contains 1, we decrease the loop iteration count by 1 and generate the next random number. Thus, if we hit the same element, the loop iteration counter will not be incremented, ensuring that we obtain a testing dataset of the expected size as output.

```
//--- Generate randomly the data indices for the test sample
```

```

ArrayInitialize(test, 0);
int for_test = (int)((total - BarsToLine) * 0.2);
for(int i = 0; i < for_test; i++)
{
    int t = (int)((double)(MathRand() * MathRand()) / MathPow(32767.0, 2) *
                (total - 1 - BarsToLine)) + BarsToLine;
    if(test[t] == 1)
    {
        i--;
        continue;
    }
    test[t] = 1;
}

```

This is the end of the preparatory work. The only thing left to do is to save the prepared data into the appropriate files. To write the data, we open two files for writing according to the names specified in the script parameters. An obvious thing to do would be to create binary files to record numeric data. They take up less disk space and are faster to work with. But since we are going to load data from applications written in other programming languages, in particular from Python scripts, the most universal approach is to use CSV files.

We open two CSV files for writing and immediately check the resulting handles for accessing the files. Erroneous handles will signal a file opening error. The corresponding message will be displayed in the terminal log.

```

//--- Open the training sample file for writing
int Study = FileOpen(StudyFileName, FILE_WRITE |
                    FILE_CSV |
                    FILE_ANSI, ",", CP_UTF8);

if(Study == INVALID_HANDLE)
{
    PrintFormat("Error opening file %s: %d", StudyFileName, GetLastError());
    return;
}

//--- Open the test sample file for writing
int Test = FileOpen(TestFileName, FILE_WRITE |
                   FILE_CSV |
                   FILE_ANSI, ",", CP_UTF8);

if(Test == INVALID_HANDLE)
{
    PrintFormat("Error opening file %s: %d", TestFileName, GetLastError());
    return;
}

```

After successfully opening the files, we set up a loop to iterate through all elements of the population. Note that the loop does not start from the zeroth element, but from the element corresponding to the number of bars in the pattern. After all, for a complete pattern record, we must specify the data of several previous candles. We will divide the training and test samples at the stage of writing to the file. By checking the value of the corresponding element in the flag array, we will replace the file handle for pattern recording with the file handle of the correct dataset. The actual pattern writing to the file is encapsulated in a separate function, which we will review a little later. To track the process, we will output the percentage of completion in the comments on the chart.

Upon completion of the loop, we will clear the comments on the chart, close the files, and log information about the file names and paths to the journal.

```
//--- Write samples to files
for(int i = BarsToLine - 1; i < total; i++)
{
    Comment(StringFormat("%.2f%%", i * 100.0 / (double)(total - BarsToLine)));
    if(!WriteData(target1, target2, rsi, macd_main, macd_signal, macd_delta, i,
        BarsToLine, (test[i] == 1 ? Test : Study)))
    {
        PrintFormat("Error to write data: %d", GetLastError());
        break;
    }
}

//--- Close the files
Comment("");
FileFlush(Study);
FileClose(Study);
FileFlush(Test);
FileClose(Test);
PrintFormat("Study data saved to file %s\\MQL5\\Files\\%s",
    TerminalInfoString(TERMINAL_DATA_PATH), StudyFileName);
PrintFormat("Test data saved to file %s\\MQL5\\Files\\%s",
    TerminalInfoString(TERMINAL_DATA_PATH), TestFileName);
}
```

To write information about the pattern to a file, we will create a function called *WriteData*. In the function parameters, we will pass pointers to arrays of source and target data, the sequential number of the last bar in the pattern in the data arrays, the number of bars to analyze for one pattern, and the file handle for writing data. The choice of the last bar in the pattern instead of the first is made in an attempt to approximate the pattern construction to the real conditions of neural network operation. When working with real-time stock price time series, we are always on the latest known bar at the current moment. We analyze information from several recent bars, which already constitute history, and try to understand the most probable upcoming price movement. Similarly, the bar specified in the parameters here represents the "current moment" for us. We take the specified number of bars before it, and all of this constitutes the pattern we analyze. Based on this pattern, our neural network should determine the probable price movement and its strength.

```
//+-----+
//| Function for writing a pattern to a file |
//+-----+
bool WriteData(double &target1[], // Buffer 1 of target values
    double &target2[], // Buffer 2 target values
    double &data1[], // Buffer 1 of historical data
    double &data2[], // Buffer 2 of historical data
    double &data3[], // Buffer 2 of historical data
    double &data4[], // Buffer 2 of historical data
    int cur_bar, // Current bar of the end of the pattern
    int bars, // Number of historical bars
    // in one pattern
    int handle) // Handle of the file to be written
{
    }
```

Let's first collect the information on the pattern into a string variable of type *string*. In doing so, we don't forget to insert a delimiter between the values of the elements. The delimiter must match the delimiter specified when opening the CSV file. Collecting data into a string variable is a forced compromise. The point is that the *FileWrite* function for writing to a text file has a limit of 63 parameters to write, and each call to write is terminated with an end-of-line character. Now, we have two problems before us:

1. By specifying all pattern data within one call of the *WriteData* function, using 4 indicators per 1 bar, we will be able to describe no more than 15 candlesticks.
2. We have to collect information on all the bars at once.

We cannot use a loop to iterate through the array values. We need to manually specify all the elements to be written in the parameters of the data-writing function. The use of a string variable helps address these issues. In a simple loop, we can collect all values into one text string. In this process, we are not limited in the number of included parameters. Of course, during the collection of indicators into the string, we will need to insert a delimiter between them, thus simulating a CSV file string. Moreover, we will write the already assembled string to the file only once. Consequently, the function will insert an end-of-line character at the end of the write once. Thus, the entire pattern in our file will be recorded in a single line.

```
//--- check the file handle
if(handle == INVALID_HANDLE)
{
    Print("Invalid Handle");
    return false;
}

//--- determine the index of the first record of the historical data of the pattern
int start = cur_bar - bars + 1;
if(start < 0)
{
    Print("Too small current bar");
    return false;
}
```

```
//--- Check the correctness of the index of the data and the data written to the file
int size1 = ArraySize(data1);
int size2 = ArraySize(data2);
int size3 = ArraySize(data3);
int size4 = ArraySize(data4);
int sizet1 = ArraySize(target1);
int sizet2 = ArraySize(target2);
string pattern = (string)(start < size1 ? data1[start] : 0.0) + "," +
    (string)(start < size2 ? data2[start] : 0.0) + "," +
    (string)(start < size3 ? data3[start] : 0.0) + "," +
    (string)(start < size4 ? data4[start] : 0.0);
for(int i = start + 1; i <= cur_bar; i++)
{
    pattern = pattern + "," + (string)(i < size1 ? data1[i] : 0.0) + "," +
        (string)(i < size2 ? data2[i] : 0.0) + "," +
        (string)(i < size3 ? data3[i] : 0.0) + "," +
        (string)(i < size4 ? data4[i] : 0.0);
}
return (FileWrite(handle, pattern,
    (double)(cur_bar < sizet1 ? target1[cur_bar] : 0),
    (double)(cur_bar < sizet2 ? target2[cur_bar] : 0)) > 0);
}
```

As a result, we obtain a structured CSV file in which a delimiter is placed between every two adjacent elements and each row represents a separate pattern for analyzing the data.

It should also be noted that to prevent an array out-of-bounds error, we should check the index values against the array sizes before accessing the data arrays. In case of an incorrect index, we write 0 instead of the indicator value. During the operation of the neural algorithm, all values of the input indicator vector are multiplied by the weights, and the resulting products are summed into a common sum. Multiplying any weight by 0 always returns 0. Therefore, zero-valued indicators have no direct effect on the outcome of the neuron performance. Of course, we can talk about indirect influence here. Indeed, there could be a situation where the contribution of a particular indicator to the overall sum is insufficient to activate the neuron. However, this is a lesser evil, and we accept these risks.

Perhaps, it is worth mentioning that for future tests of our models, we will immediately create two sets of training data:

- We will write the training samples with **non-normalized** initial data to the files *study_data_not_norm.csv* and *test_data_not_norm.csv*.
- We will write the training datasets with **non-normalized** source data into files named *study_data.csv* and *test_data.csv*.

To create the aforementioned training datasets, we will use the previously described script from the file *create_initial_data.mq5*. We will run it twice to collect the same historical data but change the filenames for data recording and the "Data normalization flag."

3.10 Gradient distribution verification

Now is the moment when we will assemble the first neural network using MQL5. However, I won't raise your hopes too much as our first neural network will not analyze or predict anything. Instead, it will perform a control function and verify the correctness of the work done earlier. The reason is that

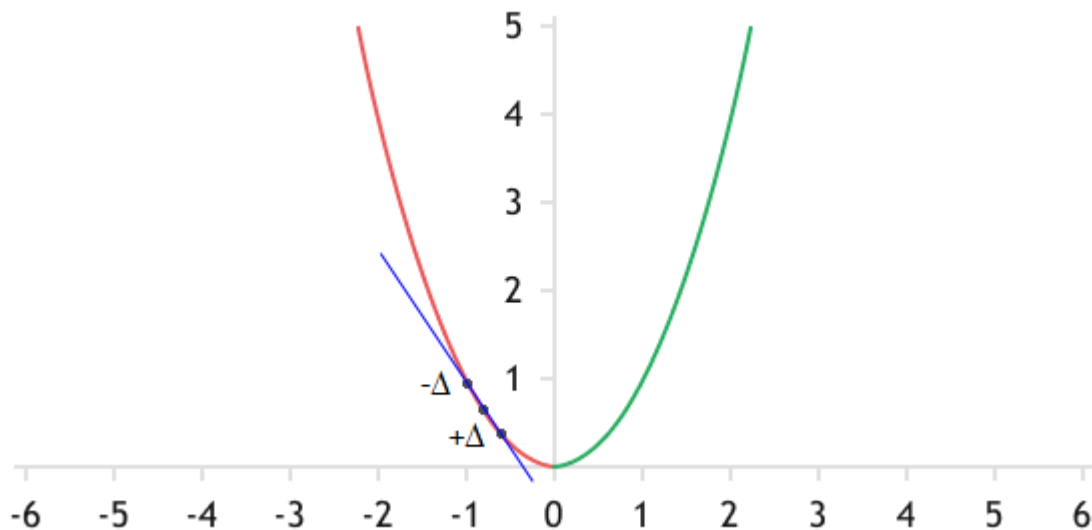
3. Building the first neural network model in MQL5

before we proceed directly to training the neural network, we need to check the correctness of the error gradient distribution throughout the neural network. I believe it is clear that the correctness of implementing this process significantly affects the overall result of the neural network training. After all, it is the error gradient on each weight that determines the magnitude and direction of its change.

To verify the correctness of gradient distribution, we can use the fact that there are two options to determine the derivative of a function:

- *Analytical*: determining the gradient of a function based on its first-order derivative. This method is implemented in our backward pass method.
- *Empirical*: under other equal conditions, the value of one indicator is changed and its effect on the final result of the function is evaluated.

In its geometric interpretation, the gradient is the slope of the tangent to the graph of the function at the current point. It indicates how the value of the function changes with a change in the parameter value.



In geometry terms, the gradient is the slope of the tangent to the graph of the function at the current point

To draw a line, we need two points. Therefore, in two simple iterations, we can find these points on the graph of the function. First, we need to add a small number to the current value of the parameter and calculate the value of the function without changing the other parameters. This will be the first point. We repeat the iteration, but this time we subtract the same number from the current value and get the second point. The line passing through these two points will approximate the desired tangent with a certain degree of error. The smaller the number used to change the parameter, the smaller this error will be. This is the basis of the *empirical* method for determining the gradient.

If this method is so simple, why not use it consistently? Everything is quite simple here. The method simplicity hides a large number of operations:

1. Perform a forward pass and save its result.
2. Slightly increase one parameter and repeat the forward pass with the result saved.
3. Slightly decrease one parameter and repeat the forward pass with the result saved.
4. Based on the found points, construct a line and determine its slope.

All these steps are performed to determine the gradient at only one step for one parameter. Imagine how much time and computational resources we would need if we used this method in training a neural network with even just a hundred parameters. Do not forget that modern neural networks contain significantly more parameters. For instance, a giant model like GPT-3 contains 175 billion parameters. Of course, we will not build such giants on a home computer. However, the use of the analytical method greatly reduces the number of necessary iterations and the time for their execution.

At the same time, we can build a small neural network and compare the results of the two methods on it. Their similarity will indicate the correctness of the implemented analytical method algorithm. Significant discrepancies in the results of the two methods will indicate the need to reevaluate the backward pass algorithm implemented in the analytical method.

To implement this idea in practice, let's create the script *check_gradient_percp.mq5*. This script will receive three external parameters:

- the size of the initial data vector,
- flag for using OpenCL technology,
- function to activate the hidden layer.

Please note that we haven't specified the source of the original data. The reason is that for this work, it doesn't matter at all what data will be input into the model. We only check the correctness of the backward pass methods. Therefore, we can use a vector of random values as initial data.

```
//+-----+
//| External parameters for the script |
//+-----+
// Source data vector size
input int      BarsToLine      = 40;
// Use OpenCL
input bool     UseOpenCL       = true;
// Hidden Layer Activation Function
input ENUM_ACTIVATION_FUNCTION HiddenActivation = AF_SWISH;
```

In addition, in the global scope of the script, we will connect our library and declare a neural network object.

```
//+-----+
//| Connecting the Neural Network Library |
//+-----+
#include "..\..\Include\NeuroNetworksBook\realization\neuronnet.mqh"
CNet Net;
```

At the beginning of the script body, let's define the architecture of a small neural network. However, since we will need to perform similar tasks multiple times to validate the correctness of the process in different architectural solutions, we will encapsulate the model creation in a separate procedure called *CreateNet*. In the parameters, this procedure receives a pointer to the object of the neural network model being created.

Let me remind you that earlier we created the *CNet::Create* method to create a neural network model. In parameters, this method takes a dynamic array of descriptions of the neural network architecture. Therefore, we need to organize a similar description of the new model. Let's collect the description of each neural layer into a separate instance of the *CLayerDescription* class. We will combine them into a dynamic array *CArrayObj*. When adding neuron descriptions to a dynamic array, make sure that their sequence strictly corresponds to the arrangement of neural layers in the neural network. In my

practice, I simply create layer descriptions sequentially in the order of their arrangement in the neural network and add them to the array as I create them.

```
bool CreateNet(CNet &net)
{
    CArrayObj *layers = new CArrayObj();
    if(!layers)
    {
        PrintFormat("Error creating CArrayObj: %d", GetLastError());
        return false;
    }
}
```

To check the correctness of the implemented error propagation algorithm, we will create a three-layer neural network. All layers will be built on the basis of the *CNeuronBase* class we created. The size of the first neural layer of the initial data was specified by the user in the external parameter *BarsToLine*. We will create it without an activation function and weight update method. In theory, this is the basic approach to creating a source data layer.

```
//--- source data layer
CLayerDescription *descr = new CLayerDescription();
if(!descr)
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    delete layers;
    return false;
}
descr.type = defNeuronBase;
descr.count = BarsToLine;
descr.window = 0;
descr.activation = AF_NONE;
descr.optimization = None;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete layers;
    delete descr;
    return false;
}
```

We will set the number of neurons in the second (hidden) neural layer to be 10 times greater than the input data layer. However, the actual size of the neural layer does not directly affect the process of analyzing the algorithm performance. This layer will already receive the activation function that the user specifies in the external parameter of the *HiddenActivation* script. For example, I used Swish. I would recommend experimenting with all the activation functions you're using. At this stage, we want to verify the correctness of all the methods we've written so far. Exactly, the more diverse your testing is, the more potential issues you can address at this stage. This will help you avoid distractions during the actual training of the neural network and focus on improving its performance.

At this stage, we will not perform weight updates. Therefore, the specified method of updating the weights will not affect our testing results in any way.

```
//--- hidden layer
descr = new CLayerDescription();
```



```

if(!descr)
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    delete layers;
    return false;
}
descr.type = defNeuronBase;
descr.count = 10 * BarsToLine;
descr.activation = HiddenActivation;
descr.optimization = Adam;
descr.activation_params[0] = (TYPE)1;
descr.activation_params[1] = (TYPE)0;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete layers;
    delete descr;
    return false;
}

```

The third neural layer will contain only one output neuron and a linear activation function.

```

//--- result layer
descr = new CLayerDescription();
if(!descr)
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    delete layers;
    return false;
}
descr.type = defNeuronBase;
descr.count = 1;
descr.activation = AF_LINEAR;
descr.optimization = Adam;
descr.activation_params[0] = 1;
descr.activation_params[1] = 0;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete layers;
    delete descr;
    return false;
}

```

Having collected a complete description of the neural network in a single dynamic array, we generate a neural network. To do this, we call the *CNet::Create* method of our base neural network class, in which the neural network is generated according to the passed description. At each step, we check the correctness of performing operations on the returned results. Receiving the boolean value *true* corresponds to the correct execution of the method's operations. If any of the errors occur, the method will return *false*.

We will specify the flag for using OpenCL. For full testing, we have to check the correctness of the backpropagation method in both modes of operation of the neural network.

3. Building the first neural network model in MQL5

```
//--- initialize the neural network
if(!net.Create(layers, (TYPE)3.0e-4, (TYPE)0.9, (TYPE)0.999, LOSS_MAE, 0, 0))
{
    PrintFormat("Error of init Net: %d", GetLastError());
    delete layers;
    return false;
}
delete layers;
net.UseOpenCL(UseOpenCL);
PrintFormat("Use OpenCL %s", (string)net.UseOpenCL());
//---
return true;
}
```

We conclude or work with the model creation procedure and move on to the main procedure of our *OnStart* script. In it, to create a neural network model, we just need to call the above procedure.

```
void OnStart()
{
    //--- create a model
    if(!CreateNet(Net))
        return;
}
```

At this stage, the neural network object is ready for testing. However, we still need initial data for testing. As mentioned above, we will simply populate them with random values. We will create the *CBufferType* data buffer to store a sequence of initial data. Target results are not of interest at this point. When generating the neural network, we filled the weight matrix with random values and do not expect to hit the target values. We also do not plan to train the neural network at this stage. Therefore, we will not waste resources on downloading unnecessary information.

```
//--- create a buffer to read the source data
CBufferType *pattern = new CBufferType();
if(!pattern)
{
    PrintFormat("Error creating Pattern data array: %d", GetLastError());
    return;
}
```

In the loop, fill the entire buffer with random values.

```
//--- generate random source data
if(!pattern.BufferInit(1, BarsToLine))
    return;
for(int i = 0; i < BarsToLine; i++)
    pattern.m_mMatrix[0, i] = (TYPE)MathRand() / (TYPE)32767;
```

Now there is enough information to conduct a forward pass of the neural network. We will implement it by calling the *FeedForward* method of our neural network. The results of the direct pass will be stored in a separate data buffer of reference values. Probably the name *reference* for a randomly obtained result sounds strange. But within the scope of our testing, this will be the reference against which we will consider deviations when changing the input data or weights.

```
//--- perform a forward and reverse pass to obtain analytical gradients
const TYPE delta = (TYPE)1.0e-5;
```

```

TYPE dd = 0;
CBufferType *init_pattern = new CBufferType();
init_pattern.m_mMatrix.Copy(pattern.m_mMatrix);
if(!Net.FeedForward(pattern))
{
    PrintFormat("Error in FeedForward: %d", GetLastError());
    return;
}
CBufferType *etalon_result = new CBufferType();
if(!Net.GetResults(etalon_result))
{
    PrintFormat("Error in GetResult: %d", GetLastError());
    return;
}

```

In the next step, we add a small constant to the result of the feed-forward pass and run the backpropagation pass of our neural network to calculate the error gradients analytically. In the above example, I used the constant $1 \cdot 10^{-5}$ as a deviation.

```

//--- create a result buffer
CBufferType *target = new CBufferType();
if(!target)
{
    PrintFormat("Error creating Pattern Target array: %d", GetLastError());
    return;
}
//--- save obtained data to separate buffers
target.m_mMatrix.Copy(etalon_result.m_mMatrix);
target.m_mMatrix[0, 0] = etalon_result.m_mMatrix[0, 0] + delta;
if(!Net.Backpropagation(target))
{
    PrintFormat("Error in Backpropagation: %d", GetLastError());
    delete target;
    delete etalon_result;
    delete pattern;
    delete init_pattern;
    return;
}
CBufferType *input_gradient = Net.GetGradient(0);
CBufferType *weights = Net.GetWeights(1);
CBufferType *weights_gradient = Net.GetDeltaWeights(1);
if(UseOpenCL)
{
    input_gradient.BufferRead();
    weights.BufferRead();
    weights_gradient.BufferRead();
}

```

Please note that we need to keep the reference result unchanged. That's why we needed to create another data buffer object, into which we copied values from the benchmark values buffer. In this buffer, we correct the data for the backpropagation pass.

According to the results of the backpropagation pass, we will save the error gradients obtained analytically at the level of the initial data and weights. We also save the weights themselves, which we will need when analyzing the distribution of error gradients at the level of the weight matrix.

Perhaps it's worth mentioning that since we adjust the weights during the training process, obtaining accurate gradients at the weight matrix level is most informative for us. Accurate gradients at the level of input data mostly serve as indirect evidence of the correctness of gradient distribution throughout the entire neural network. This is due to the fact that before determining the error gradient at the level of the initial data, we have to consistently draw it analytically through all layers of our neural network.

We obtained the error gradients by an analytical method. Next, we need to determine the gradients empirically and compare them with the results of the analytical method.

Let's look at the initial data level first. To achieve this, we will copy our pattern of input data into a new dynamic array, which will allow us to modify the required indicators without the fear of losing the original pattern.

We will organize a loop to enumerate all the indicators of our pattern. Inside the loop, we will first add our constant $1 \cdot 10^{-5}$ to each indicator of the original pattern in turn and implement a feed-forward pass neural network. After the feed-forward pass, we take the result obtained and compare it with the reference one that was saved earlier. The difference between the results of the feed-forward pass will be stored in a separate variable. Then we subtract a constant from the initial value of the same indicator and repeat the feed-forward pass. The result of the feed-forward pass is also comparable to the reference result. Let's find the arithmetic mean of two passes.

```
//--- in the loop, alternately change the elements of the source data and compare
//--- empirical result with the value of the analytical method
for(int k = 0; k < BarsToLine; k++)
{
    pattern.m_mMatrix.Copy(init_pattern.m_mMatrix);
    pattern.m_mMatrix[0, k] = init_pattern.m_mMatrix[0, k] + delta;
    if(!Net.FeedForward(pattern))
    {
        PrintFormat("Error in FeedForward: %d", GetLastError());
        return;
    }
    if(!Net.GetResults(target))
    {
        PrintFormat("Error in GetResult: %d", GetLastError());
        return;
    }
    TYPE d = target.At(0) - etalon_result.At(0);

    pattern.m_mMatrix[0, k] = init_pattern.m_mMatrix[0, k] - delta;
    if(!Net.FeedForward(pattern))
    {
        PrintFormat("Error in FeedForward: %d", GetLastError());
        return;
    }
    if(!Net.GetResults(target))
    {
        PrintFormat("Error in GetResult: %d", GetLastError());
        return;
    }
}
```

3. Building the first neural network model in MQL5

```
    }  
    d -= target.At(0) - etalon_result.At(0);  
    d /= 2;  
    dd += input_gradient.At(k) - d;  
}  
delete pattern;
```

At this point, you should be careful with the signs of the operation and deviations. In the first case, we added a constant and obtained some deviation in the results. The deviation is considered as the current value minus the reference.

$$\Delta = R_{Current} - R_{Reference}$$

In the second case, we subtracted the constant from the original value. Similarly, using the same formula for calculating the deviation, we will obtain a value with the opposite sign. Therefore, to combine the obtained results in magnitude and preserve the correct direction of the gradient, we need to subtract the second deviation from the first one.

The result is divided by 2 to get the mean deviation. The result obtained is comparable to the result of the analytical method.

We repeat the operations described above for all parameters of the initial pattern.

There is another aspect that we should take into account. The gradient indicates how the function value changes when the parameter changes by 1. Our constant is much smaller. Therefore, the empirically calculated gradient we obtained is significantly underestimated. To compensate for this, let's divide the empirically obtained value by our constant and display the total result in the MetaTrader 5 log.

```
//--- display the total value of deviations at the level of the initial data in the jo  
PrintFormat("Delta at input gradient between methods %.5e", dd / delta);
```

Similarly, we determine the empirical gradient at the level of the weight matrix. Note that to access the matrix of weights, we obtain not a copy of the matrix but a pointer to the object. This is a very important point. Thanks to this, we can modify the values of weights directly in our script without creating additional methods to update buffer values in the neural network and neural layer classes. However, this approach is more of an exception than a general practice. The reason is that with this approach, we cannot track changes to the weight matrix from the neural layer object.

The cumulative result of comparing empirical and analytical gradients at the weight matrix level is also printed to the MetaTrader 5 log.

```
//--- reset the value of the sum and repeat the loop for the gradients of the weights  
dd = 0;  
CBufferType *initial_weights = new CBufferType();  
if(!initial_weights)  
{  
    PrintFormat("Error creating reference weights buffer: %d", GetLastError());  
    return;  
}  
if(!initial_weights.m_mMatrix.Copy(weights.m_mMatrix))  
{  
    PrintFormat("Error copying weights to initial weights buffer: %d",  
                GetLastError());  
}
```

```

        return;
    }

    for(uint k = 0; k < weights.Total(); k++)
    {
        if(k > 0)
            weights.Update(k - 1, initial_weights.At(k - 1));
        weights.Update(k, initial_weights.At(k) + delta);
        if(UseOpenCL)
            if(!weights.BufferWrite())
                return;
        if(!Net.FeedForward(init_pattern))
        {
            PrintFormat("Error in FeedForward: %d", GetLastError());
            return;
        }
        if(!Net.GetResults(target))
        {
            PrintFormat("Error in GetResult: %d", GetLastError());
            return;
        }
        TYPE d = target.At(0) - etalon_result.At(0);

        weights.Update(k, initial_weights.At(k) - delta);
        if(UseOpenCL)
            if(!weights.BufferWrite())
                return;
        if(!Net.FeedForward(init_pattern))
        {
            PrintFormat("Error in FeedForward: %d", GetLastError());
            return;
        }
        if(!Net.GetResults(target))
        {
            PrintFormat("Error in GetResult: %d", GetLastError());
            return;
        }
        d -= target.At(0) - etalon_result.At(0);
        d /= 2;
        dd += weights_gradient.At(k) - d;
    }

    --- display the total value of deviations at the level of weights in the journal
    PrintFormat("Delta at weights gradient between methods %.5e", dd / delta);

```

After completing all the iterations, we clear the memory by deleting the used objects and exit the program.

```
//--- clear the memory before exiting the script
delete init_pattern;
delete etalon_result;
delete initial_weights;
delete target;
}
```

The results of my testing are shown in the screenshot below. Based on the testing results, I obtained a deviation in the 11th to 12th decimal place. For comparison, deviations in the 8–9th decimal places are considered acceptable in different sources. And it's not worth noting that when using OpenCL, the deviation turned out to be an order of magnitude smaller. This is not an advantage of using technology, but rather the influence of a random factor. At each run, a random matrix of weights and initial data was re-generated. As a result, the comparison was carried out on different parts of the neural network function with different curvature.

OpenCL: CPU device '11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz' selected
Use OpenCL false
Delta at input gradient between methods -2.09701e-11
Delta at weights gradient between methods 5.76474e-11
OpenCL: CPU device '11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz' selected
Use OpenCL true
Delta at input gradient between methods -5.97421e-12
Delta at weights gradient between methods -2.70610e-12

Results of comparing analytical and empirical error gradients

In general, we can say that the testing confirmed the correctness of our implementation of the error backpropagation algorithm both by means of MQL5 and using the OpenCL multi-threaded computing technology. Our next step is to assemble a more complex perceptron, and we will try to train it on a training set.

3.11 Comparative testing of implementations

In the previous section, we verified the correctness of the backpropagation algorithm operation. Now we can safely move on to training our perceptron. We will perform this work in the script *perceptron_test.mq5*. The first block of the script will remind you of the script from the previous section. This is a consequence of using our library to create neural networks. We will create a neural network using it. Hence, the algorithm for initializing and using the neural network will be identical in all cases.

To enable various testing scenarios, we will add the following external parameters to the script:

- The name of the file containing the training sample.
- The name of the file to record the dynamics of the error change. Using these values, we will be able to plot the error change graph during the training process, which will help us visualize the neural network learning process.
- The number of historical bars used in the description of one pattern.
- The number of input layer neurons per bar.
- Switch flag for using OpenCL technology in the process of training a neural network.
- Batch size for one iteration of weight matrix update.

3. Building the first neural network model in MQL5

- Learning rate.
- The number of hidden layers.
- The number of neurons in a single hidden layer.
- The number of iterations of updating the weight matrix.

Just like in the previous section, after declaring the external parameters in the global scope of the script, we will include our library for creating a neural network and declare an object of the base class *CNet*.

```
//+-----+
//| External parameters for script operation |
//+-----+
// Name of the file with the training sample
input string StudyFileName = "study_data.csv";
// File name for recording the error dynamics
input string OutputFileName = "loss_study.csv";
// Number of historical bars in one pattern
input int BarsToLine = 40;
// Number of input layer neurons per 1 bar
input int NeuronsToBar = 4;
// Use OpenCL
input bool UseOpenCL = false;
// Packet size for updating the weights matrix
input int BatchSize = 10000;
// Learning rate
input double LearningRate = 3e-5;
// Number of hidden layers
input int HiddenLayers = 1;
// Number of neurons in one hidden layer
input int HiddenLayer = 40;
// Number of iterations of updating the weights matrix
input int Epochs = 1000;
//+-----+
//| Connect the neural network library |
//+-----+
#include <NeuroNetworksBook\realization\neuronnet.mqh>
CNet *net;
```

Before moving on to writing the script code, let's consider what functionality we need to incorporate into it.

First, we need to create the model. For this, we will define the model architecture and call the model initialization method. Similar operations were performed in the script to check the correctness of the error gradient distribution.

Next, to train our model, we need to load the previously created training dataset, which will contain a set of input data and target values.

Only after successfully completing these steps, we can start the model training process. This is a cyclic process that includes a feed-forward pass, a backpropagation pass, and weight matrix updates. There are several approaches to the duration of model training. The most common one involves limiting the number of training epochs and tracking changes in the model error. We will use the first approach. The

analysis of the error dynamics during the training process will allow us to develop criteria for applying the second method. Therefore, during training, we need to record the model error change and save the collected sequence after training. At the end of the training, we will save the obtained model.

Thus, we have defined the necessary functionality for our script. To create clear and readable code, we will divide it into blocks corresponding to the tasks mentioned above. In the body of the main function `OnStart`, we will sequentially call the corresponding functions with control over the execution of operations.

First, we will create a vector to record the dynamics of the model error during training. Its size will be equal to the number of training epochs.

```
//+-----+
//| Beginning of the script program |
//+-----+
void OnStart()
{
    //--- prepare a vector to store the network error history
    VECTOR loss_history = VECTOR::Zeros(Epochs);
```

Next, we initialize our model for training. Here we instantiate a neural network class and pass the object pointer to the model initialization function. Be sure to check the result of the operation.

```
//--- 1. Initialize model
CNet net;
if(!NetworkInitialize(net))
    return;
```

The next step is to load the training sample. For this purpose, we will need two dynamic arrays: one for loading the patterns of source data, and the other for the target values. Both arrays will be synchronized.

The data loading is performed in the *LoadTrainingData* function, in the parameters of which we will pass the file for data loading and pointers to the created dynamic array objects.

```
//--- 2. Load the training sample data
CArrayObj data;
CArrayObj targets;
if(!LoadTrainingData(StudyFileName, data, targets))
    return;
```

As mentioned earlier, after creating the model and loading the training dataset, we can start the training process. This functionality will be assigned to the *NetworkFit* method, in the parameters to which we will pass pointers to our model, a training sample with target values, and a vector recording the dynamics of the model error variation during training.

```
//--- 3. Train model
if(!NetworkFit(net, data, targets, loss_history))
    return;
```

After completing the model training process, we save the history of the model error change during training. We will also keep the trained model. We do not need to create a separate function to save the trained model. We can use the previously created method of our base neural network class to save the model.

3. Building the first neural network model in MQL5

```
//--- 4. Save the error history of the model
    SaveLossHistory(OutputFileName, loss_history);
//--- 5. Save the obtained model
    net.Save("Study.net");
    Print("Done");
}
```

To confirm the successful completion of all operations, we will print an informational message to the log and terminate the script.

As you can see, the code of the main function of the script turned out to be quite short, but clearly structured. This distinguishes it from the gradient distribution correctness check script in the previous section. The choice of programming style remains with the programmer and does not affect the functionality of our library. We go back to our script and now we will write the functions that we called above from the main function of the script.

First on the list is the *NetworkInitialize* model initialization function. In the parameters of this function, we pass a pointer to the object of the model being created. In the body of the function, we have to initialize the model before training. To initialize the model, we need to provide a description of the model to be created. I remind you that we create the model description in a dynamic array, each element of which contains a pointer to an instance of the object *CLayerDescription* with the description of the architecture of a specific neural layer. The very operation of creating such a model description has been moved to a separate function, *CreateLayersDesc*, which is a natural extension of the structured code concept.

```
//+-----+
//| Initializing the model |
//+-----+
bool NetworkInitialize(CNet &net)
{
    CArrayObj layers;
//--- create a description of the network layers
    if(!CreateLayersDesc(layers))
        return false;
}
```

After creating the model architecture description, we call the initialization method of our neural network *CNet::Create*. Into it, we pass the description of the model architecture, the learning rate, optimization parameters, loss function, and regularization parameters. Don't forget to check the result of the model creation operations.

```
//--- initialize the network
if(!net.Create(&layers, (TYPE)LearningRate, (TYPE)0.9, (TYPE)0.999, LOSS_MSE, 0, 0))
{
    PrintFormat("Error of init Net: %d", GetLastError());
    return false;
}
net.UseOpenCL(UseOpenCL);
net.LossSmoothFactor(BatchSize);
return true;
}
```

After the successful initialization of the model, we set the flag for using OpenCL and set the batch size for model error averaging. In the provided example, regularization is set at the level of the weight matrix update batch.

To complete the description of the model initialization process, let's take a look at the *CreateLayersDesc* function, which is responsible for creating the architecture description of the model. In the parameters, the method receives a pointer to a dynamic array, into which we will write the architecture of the created model.

We first create a description of the initial data layer. The number of neurons in the input layer of the raw data depends on two external parameters: the number of historical bars in one pattern (*BarsToLine*) and the number of input layer neurons per bar (*NeuronsToBar*). The quantity is determined by their product. The input layer will be without an activation function and will not be trained. That's clear and should not raise any questions. In this layer, you're essentially storing the initial parameters from an external system in the results array of the layer. Within the layer, no operations are performed on the data.

```
bool CreateLayersDesc(CArrayObj &layers)
{
    ///--- create source data layer
    CLayerDescription *descr;    if(!(descr = new CLayerDescription()))
    {
        PrintFormat("Error creating CLayerDescription: %d", GetLastError());
        return false;
    }
    descr.type          = defNeuronBase;
    descr.count         = NeuronsToBar * BarsToLine;
    descr.window        = 0;
    descr.activation    = AF_NONE;
    descr.optimization  = None;
    if(!layers.Add(descr))
    {
        PrintFormat("Error adding layer: %d", GetLastError());
        return false;
    }
}
```

When using fully-connected neural layers, each neuron in the hidden layer can be considered as a specific pattern that the neural network learns during the training process. In this logic, the number of neurons in the hidden layer represents the number of patterns that the neural network is capable of memorizing. Certainly, you can establish a logical relationship between the number of elements in the previous layer and the number of possible combinations, which will represent patterns. But let's not forget that our previous neural layer results are non-binary quantities, and the range of variation is quite large. Therefore, the total number of possible combinatorial variants of patterns will turn out to be very large. The average probability of their occurrence will vary greatly. Indeed, most of the time, the number of neurons in each hidden layer will be determined by the neural network architect within a certain range, and the exact number is often fine-tuned based on the best performance on a validation dataset. For this reason, we gave the user the ability to specify the number of neurons in the hidden layer in the external parameter *HiddenLayer*. But let's say right away that we will create all neural layers of the same architecture and size.

The number of hidden layers depends on the complexity of the problem being solved and is also determined by the neural network architect. In this test, I will use a neural network with one hidden layer. However, I suggest that you independently conduct a few experiments with different numbers of layers and assess the impact of changing this parameter on the results. To perform such experiments, we have derived a separate external parameter – the number of *HiddenLayers*.

In practice, we create one hidden layer description and then add it to the dynamic array of architecture descriptions as many times as we need to create hidden neural layers.

```
//--- hidden layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
descr.type          = defNeuronBase;
descr.count         = HiddenLayer;
descr.activation     = AF_SWISH;
descr.optimization   = Adam;
descr.activation_params[0] = 1;
for(int i = 0; i < HiddenLayers; i++)
    if(!layers.Add(descr))
    {
        PrintFormat("Error adding layer: %d", GetLastError());
        return false;
    }
```

Within this section, I do not aim to fully train the neural network with the best possible results. We will only compare the performance of our library in different modes and their impact on learning outcomes. Let's also see in practice the impact of some of the approaches we discussed in the theoretical part of the book. Therefore, we will not delve deeply into the careful selection of architectural parameters for the neural network to achieve maximum results at this moment.

I have specified *Swish* as the activation function for the hidden layer. This is one of those functions whose range of values is limited at the bottom and not limited at the top. In this case, the function is differentiable over the whole range of permitted values. However, we will be able to evaluate other activation features during the testing process.

Choosing the activation function for the output layer is a compromise. The challenge here is that we have two goals: direction and strength of movement. This is not a standard approach to solving the problem, as our neural network output consists of two neurons with completely different values. One might consider the direction of movement as a binary classification (buy or sell), while determining the strength of movement is a regression task. It would probably be logical to train the neural network only to determine the strength of the movement, and the direction would correspond to the sign of the result. However, we are learning and experimenting. Let's observe the behavior of the neural network in such a non-standard situation. We will try to activate the neurons with a linear function, which is standard for solving regression tasks.

I have specified Adam as the training method for both neural layers.

The algorithm for describing neural layers is completely identical to the one discussed in the previous section. First, we describe each layer in an object of the *CLayerDescription* class. The sequence of describing layers corresponds to their sequence in the neural network, from the input layer of raw data to the output layer of results. As the layers are getting their descriptions, add them to the collection of the previously created dynamic array.

```
//--- results layer
if(!(descr = new CLayerDescription()))
{
```

```

        PrintFormat("Error creating CLayerDescription: %d", GetLastError());
        return false;
    }
    descr.type          = defNeuronBase;
    descr.count         = 2;
    descr.activation     = AF_LINEAR;
    descr.optimization   = Adam;
    descr.activation_params[0] = 1;
    if(!layers.Add(descr))
    {
        PrintFormat("Error adding layer: %d", GetLastError());
        return false;
    }
    return true;
}

```

The next step in our script was loading the training dataset in the *LoadTrainingData* function. We will load it from the file specified in the function parameters. In the body of the function, we immediately open the specified file for reading and check the result of the operation based on the value of the obtained handle.

```

//+-----+
//| Uploading training data |
//+-----+
bool LoadTrainingData(string path, CArrayObj &data, CArrayObj &targets)
{
    CBufferType *pattern;
    CBufferType *target;
    //-- open the file with the training sample
    int handle = FileOpen(path, FILE_READ | FILE_CSV | FILE_ANSI | FILE_SHARE_READ,
                          ",", CP_UTF8);

    if(handle == INVALID_HANDLE)
    {
        PrintFormat("Error opening study data file: %d", GetLastError());
        return false;
    }
}

```

We will carry out the operation of loading the training sample in two steps. First, we will first patterns and target values into the two *CBufferType* buffers, piece by piece. We will collect the source data elements of one pattern in the *pattern* buffer and the relevant target results in the *target* buffer.

```

//-- display the progress of training data loading in the chart comment
uint next_comment_time = 0;
enum
{
    OutputTimeout = 250 // no more than once every 250 milliseconds
};
//-- organize the cycle of loading the training sample
while(!FileIsEnding(handle) && !IsStopped())
{
    if(!(pattern = new CBufferType()))
    {
        PrintFormat("Error creating Pattern data array: %d", GetLastError());
    }
}

```

```

        return false;
    }
    if(!pattern.BufferInit(1, NeuronsToBar * BarsToLine))
        return false;
    if(!(target = new CBufferType()))
    {
        PrintFormat("Error creating Pattern Target array: %d", GetLastError());
        return false;
    }
    if(!target.BufferInit(1, 2))
        return false;
    for(int i = 0; i < NeuronsToBar * BarsToLine; i++)
        pattern.m_mMatrix[0, i] = (TYPE)FileReadNumber(handle);
    for(int i = 0; i < 2; i++)
        target.m_mMatrix[0, i] = (TYPE)FileReadNumber(handle);

```

After loading information about one pattern from the file, we will store pointers to objects with the data in two dynamic arrays, *CArrayObj*. We also got pointers to them in the function parameters. One array is used for source data patterns (*data*) and the second array is used for target values (*targets*). We repeat the operations in a loop until we reach the end of the file. To allow the user to monitor the process, we will display information about the number of loaded patterns on the chart in the comments field.

Note that since we are passing pointers to data objects into dynamic arrays, we need to create new instances of *CBufferType* objects after writing the pointers to the array. Otherwise, we will fill the entire dynamic array with a pointer to the same instance of an object, and the buffer will contain generic information about all patterns, the manipulation of which will require a different algorithm. Consequently, the entire neural network will not work correctly.

```

    if(!data.Add(pattern))
    {
        PrintFormat("Error adding study data to array: %d", GetLastError());
        return false;
    }
    if(!targets.Add(target))
    {
        PrintFormat("Error adding study data to array: %d", GetLastError());
        return false;
    }
    //--- output download progress in chart comment
    //--- (not more than once every 250 milliseconds)
    if(next_comment_time < GetTickCount())
    {
        Comment(StringFormat("Patterns loaded: %d", data.Total()));
        next_comment_time = GetTickCount() + OutputTimeout;
    }
}
FileClose(handle);
return true;
}

```

After completing the loop for reading the data, we will obtain two arrays of objects with the same number of elements. In these, elements with the same index will constitute the source-target pair of the pattern data. Here we close the training sample file.

Now that we have the neural network already created and the training sample loaded, we can start training in the *NetworkFit* function. In its parameters, this method receives pointers to objects of the neural network and the training dataset. Additionally, it receives a pointer to a vector recording the dynamics of the model's error changes during the training process. To train the neural network, we will create two nested loops. We will initiate the first loop with the number of iterations equal to the external parameter *Epochs* which is the number of weight matrix updates. In the nested loop, we will create a number of iterations equal to *BatchSize*, i.e. the batch size to update the weights.

```
bool NetworkFit(CNet &net, const CArrayObj &data,
               const CArrayObj &target, VECTOR &loss_history)
{
    //--- training
    int patterns = data.Total();
    //--- loop through the eras
    for(int epoch = 0; epoch < Epochs; epoch++)
    {
        ulong ticks = GetTickCount64();
        //--- teach by batches
        for(int i = 0; i < BatchSize; i++)
        {
            //--- check to see if the training has stopped
            if(IsStopped())
            {
                Print("Network training stopped by user");
                return true;
            }
        }
    }
}
```

In the body of the nested loop, we will randomly select one pattern from the training dataset. For each selected pattern, we will first make a forward pass on the corresponding input data. Then open the target values and do a backward pass.

```
//--- select a random pattern
int k = (int)((double)(MathRand() * MathRand()) / MathPow(32767.0, 2) *
                                                    patterns);

if(!net.FeedForward(data.At(k)))
{
    PrintFormat("Error in FeedForward: %d", GetLastError());
    return false;
}
if(!net.Backpropagation(target.At(k)))
{
    PrintFormat("Error in Backpropagation: %d", GetLastError());
    return false;
}
}
```

By repeating iterations of feed-forward and backpropagation passes, we accumulate the error gradient on each element of the weight matrix. After completing the specified number of iterations of feed-forward and backpropagation passes up to the batch size for weight matrix updates, we exit the inner

3. Building the first neural network model in MQL5

loop. Then we update the weights in the direction of the average gradient of the error, clear the buffer of accumulated error gradients, and save the current value of the loss function in a vector to monitor the training process. After that, we enter a new loop of training iterations.

```
//--- reconfigure the network weights
net.UpdateWeights(BatchSize);
printf("Use OpenCL %s, epoch %d, time %.5f sec", (string)UseOpenCL,
        epoch, (GetTickCount64() - ticks) / 1000.0);
//--- report on a bygone era
TYPE loss = net.GetRecentAverageLoss();
Comment(StringFormat("Epoch %d, error %.5f", epoch, loss));
//--- remember the epoch error to save to file
loss_history[epoch] = loss;
}
return true;
}
```

In the proposed example, the training process is constrained by an external parameter of the number of iterations of updating the weight matrix. In practice, a common approach is often used where the training process stops upon achieving specified performance metrics. This could be the value of the loss function, the accuracy rate of hitting expected results, and so on. A hybrid approach can also be employed, where both metrics are monitored while also setting a maximum number of training iterations.

After the training process is completed, we save the dynamics of the loss function to a file. This functionality is performed by the *SaveLossHistory* function, in the parameters of which we will pass the name of the file to record the data and the vector of dynamics of changes in the model error during training.

In the body of the function, we open or create a new CSV file to record the data and in a loop store all the model error values received during training.

```
void SaveLossHistory(string path, const VECTOR &loss_history)
{
    int handle = FileOpen(OutputFileName, FILE_WRITE | FILE_CSV | FILE_ANSI,
                          ",", CP_UTF8);

    if(handle == INVALID_HANDLE)
    {
        PrintFormat("Error creating loss file: %d", GetLastError());
        return;
    }
    for(ulong i = 0; i < loss_history.Size(); i++)
        FileWrite(handle, loss_history[i]);
    FileClose(handle);
    printf("The dynamics of the error change is saved to a file %s\\MQL5\\Files\\%s",
          TerminalInfoString(TERMINAL_DATA_PATH), OutputFileName);
}
```

After writing the data to the file, we close the file and output an informational message to the log indicating the full path of the saved file.

The presented example of the script implementation shows a full loading of the training sample into memory. Of course, working with RAM is always faster than accessing permanent memory. However,

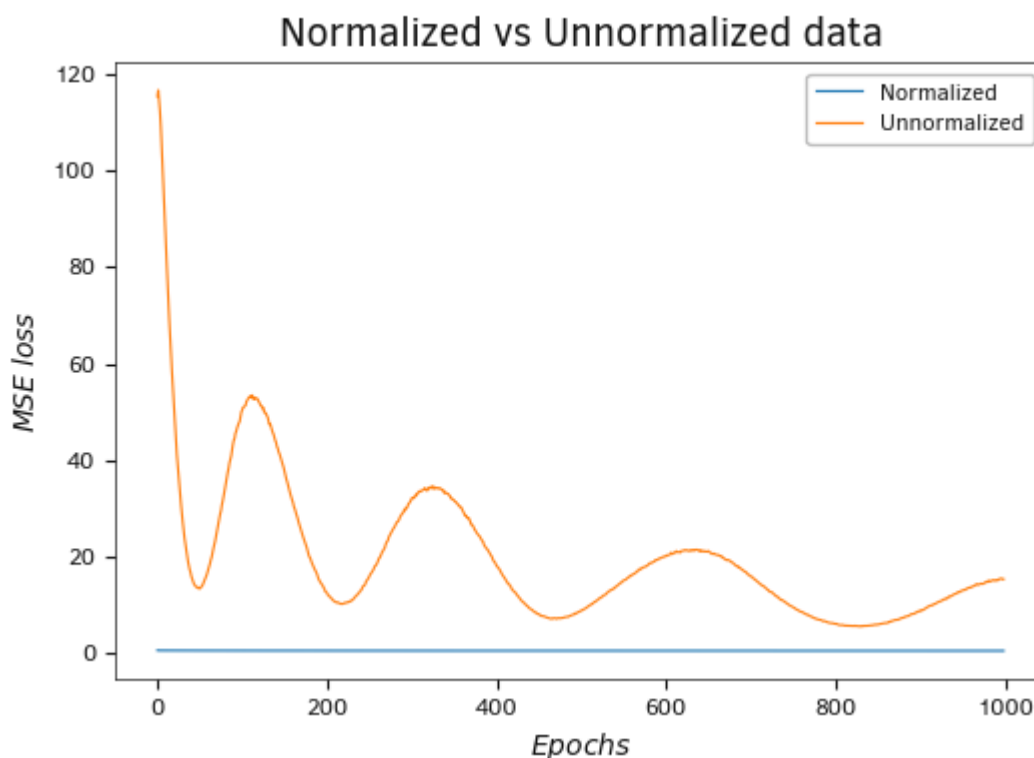
the sizes of the training dataset do not always allow it to be fully loaded into RAM. In such cases, the training sample is loaded and processed in batches.

Normalizing data at the neural network output

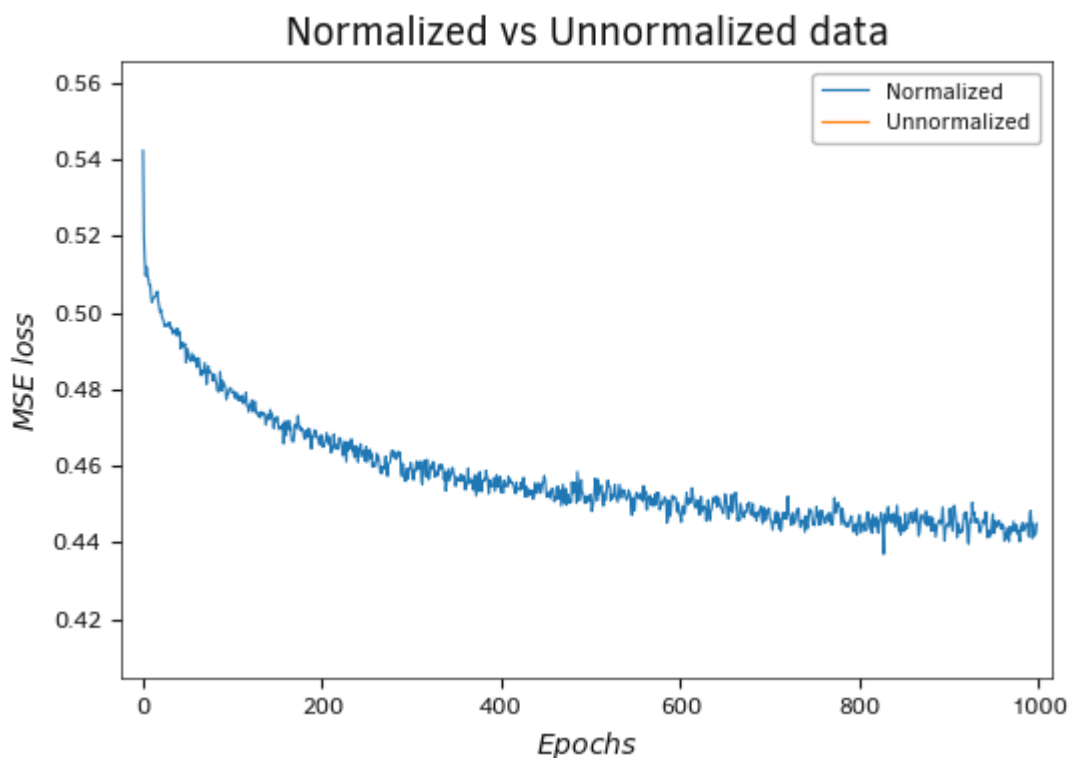
After creating such a script, we can conduct several instructive experiments. For example, we have previously discussed the importance of normalizing the initial data before feeding it to the input of a neural network. But how important is that? Why is it not possible to adjust the appropriate weights during the neural network training process to account for the data scale? Yes, we were talking about the impact of large values. But now we can do a practical experiment and see the effect of input data normalization on the model training result.

Let's take historical data for the EURUSD instrument, with a five-minute timeframe covering the period from 01.01.2015 to 12.31.2020, and create two training datasets: one with normalized data and the other with unnormalized data. Let's run the above neural network training script on both samples. We made the script for creating the training sample in Section 3.9.

The graph depicting the dynamics of the loss function says it all. The error on normalized data is much lower, even if we start with random weights. If the initial value of the loss function on unnormalized data is around 120, then on normalized data, it's only 0.6. Of course, during the training process, the value of the loss function on non-normalized data drops rapidly and after 200 iterations of weighting factor updates it drops to 6, and after 1000 iterations it reaches 4.5. But despite such a rapid rate of decline in the loss function index, it still significantly outperforms that for normalized data. On the final iterations, after 1000 weight matrix update iterations, the loss function approaches approximately 0.44.



Graph of the dynamics of the MSE loss function during the training of a neural network, on both normalized and unnormalized data



Graph of the dynamics of the MSE loss function during the training of a neural network, on both normalized and unnormalized data (scale)

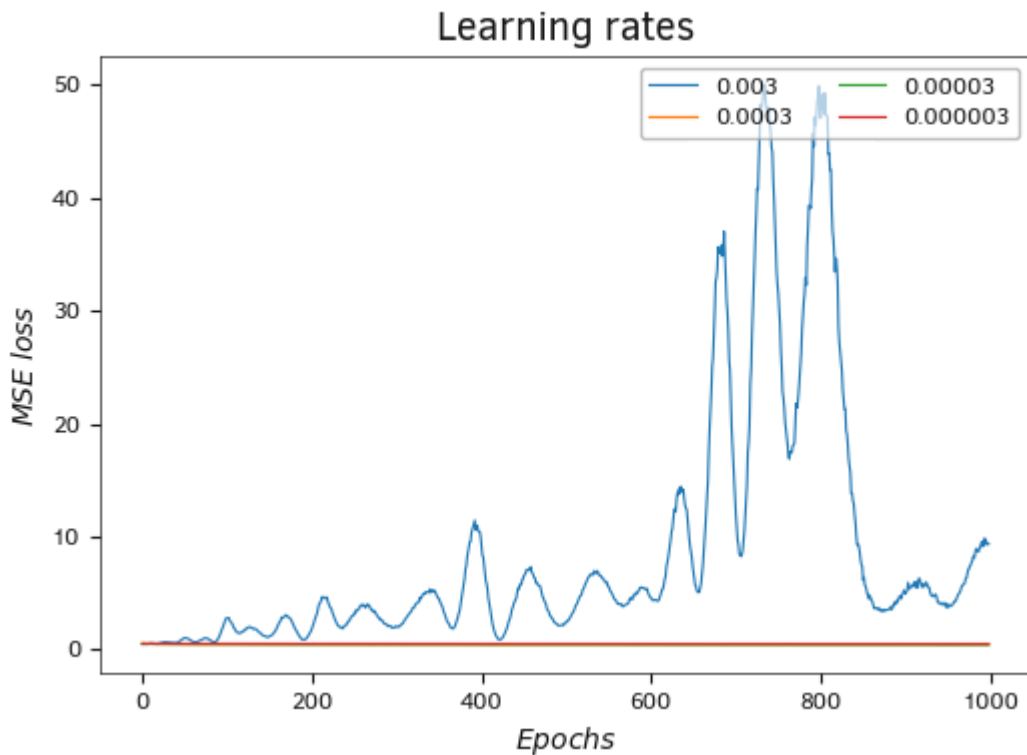
I conducted a similar experiment both with and without using OpenCL technology. The results of the neural network were comparable. But in terms of performance on such a small neural network, the CPU won. Obviously, the data transfer overhead was much higher than the performance gains from utilizing multithreading technology. These results were expected. As we discussed earlier, using such technology is justified for large neural networks when the costs of data transmission between devices are offset by the performance gains achieved by splitting computational iterations into parallel threads.

I suggest repeating a similar experiment with your data – then you won't have any questions about the necessity of normalizing the input data. I believe that after conducting the experiment it is obvious that further testing should be performed on normalized data.

Choosing the learning rate

The next question that always arises for creators of neural networks is the choice of the learning rate. When tackling this issue, it's essential to strike a balance between performance and the quality of learning. Choosing an intentionally high learning rate allows for faster error reduction at the beginning of training. But then the rate of learning rapidly declines and at best stops far from the intended goal. In the worst case, the error starts to increase. Choosing an excessively small learning rate reduces the training speed. The process takes more time, and there's an increased risk of getting stuck in a local minimum without reaching the desired goal.

For experimental testing of the impact of learning rate on the neural network training process, let's train the previously created neural network using four different learning rates: 0.003, 0.0003, 0.00003 and 0.000003. The results of the test are shown in the graph below.

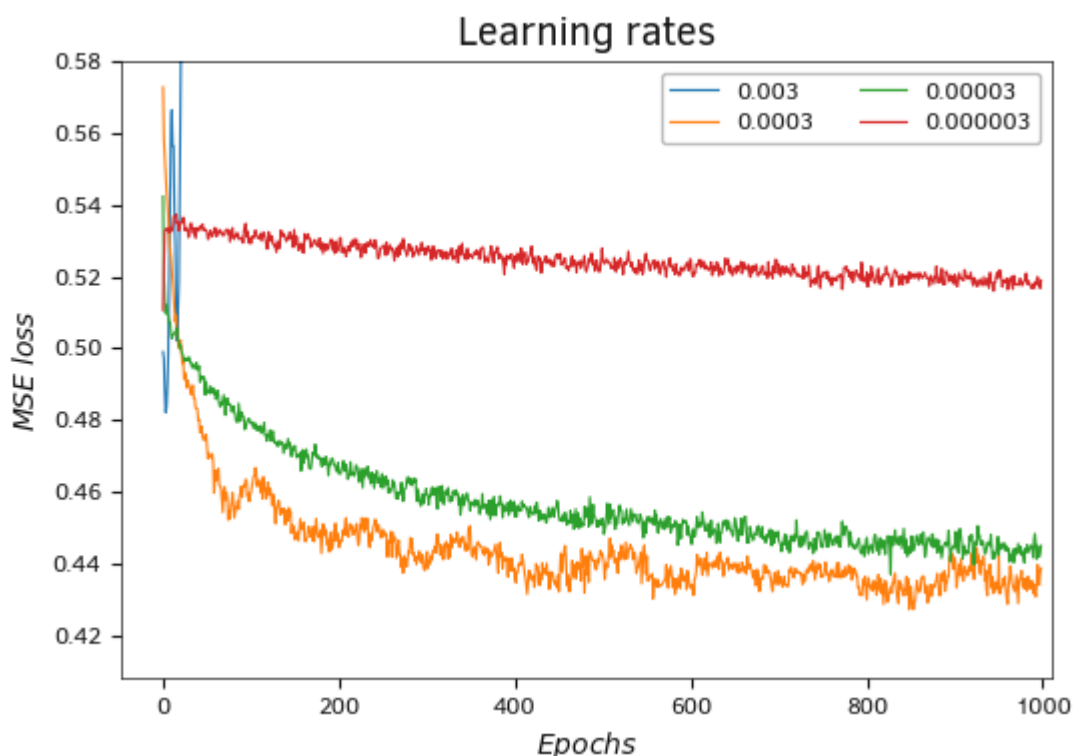


Comparison of the loss function dynamics when using different learning rates

During the training of the neural network using a learning rate of 0.003, fluctuations in the loss function are observed. During the learning process, the amplitude of the oscillations increases. In general, there is a tendency for the model error to increase. Such behavior is characteristic of an excessively high learning rate.

Reducing the learning rate makes the training schedule smoother. However, at the same time, the rate of decrease in the loss function value diminishes with each weight matrix update. The most gradual decrease in the loss function value is demonstrated by the training process with a learning rate of 0.000003. However, achieving the smoothness of the graph came at the cost of increasing the number of weight matrix update iterations required to reach the optimal result. Throughout the entire training process with 1000 weight matrix update iterations, a learning rate of 0.000003 exhibited the worst result among all.

Training the neural network with coefficients of 0.0003 and 0.00003 showed similar results. The loss function graph with a learning rate of 0.00003 turned out to be more jagged. But at the same time, the best result in terms of error value was shown by training with a rate of 0.0003.



Comparison of the loss function dynamics when using different learning rates (scale)

Selecting the number of neurons in the hidden layer

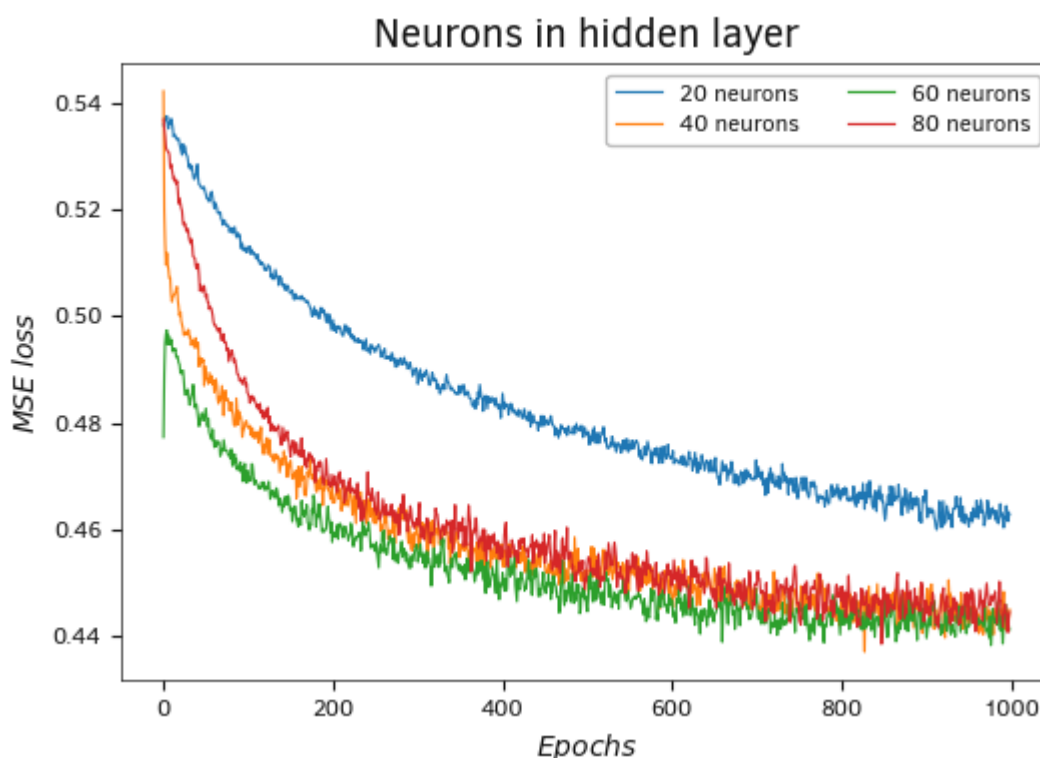
The next aspect I'd like to demonstrate in practice is the impact of the number of neurons in the hidden layer on the training process and its outcome. When we talk about fully connected neural layers, where each neuron in the subsequent layer has connections to all neurons in the previous layer, and each connection is individual and independent, it's logical to assume that each neuron will be activated by its own combination of states from the neurons in the previous layer. Thus, each neuron responds to a different state pattern of the previous layer. Consequently, having a greater number of neurons in the hidden layer has the potential to memorize more such patterns and make them more detailed. At the same time, we are not programming pattern variations; we allow the neural network to learn them autonomously from the presented training dataset. It would seem that in this logic, increasing the number of neurons in the hidden layer can only increase the quality of training of the neural network.

But in practice, not all patterns have an equal probability of occurrence. The goal of training a neural network is not to memorize each individual state down to the finest details. Their goal is to use the training dataset to generalize the presented data, identify and highlight dependencies and regularities. The obtained data should allow the construction of a function that describes the relationship between the target values and the input data with the required accuracy. Therefore, an excessive increase in neurons in the hidden layer reduces the neural network's ability to generalize and leads to overfitting.

The other aspect of increasing the number of neurons in the hidden layer is the increase in the consumption of time and computational resources. The point is that adding one neuron in the hidden layer adds as many elements to the weight matrix as the previous layer contains plus one element for *bias*. Therefore, when choosing the number of neurons in the hidden layer, it's important to consider the balance between the achieved learning quality and the training costs for such a neural network. At the same time, you need to think about the risk of overfitting.

Certainly, there are established methods to combat overfitting in neural networks. These primarily include increasing the training dataset size and [regularization](#) techniques. We have discussed theoretical aspects of regularization earlier, and we will talk about practical applications a little later.

Now I suggest looking at the graphs of the error function values during the training of a neural network with a single hidden layer, where the number of neurons changes while keeping other conditions constant. When testing, I compared the training of 4 neural networks with 20, 40, 60 and 80 neurons in the hidden layer. Of course, such a number of neurons is too small to get any decent training results on a sample of 350 thousand patterns. Moreover, there is no risk of overtraining here. But they are enough to look at the impact of this factor on learning.



Comparison of the loss function dynamics when using different numbers of neurons in the hidden layer

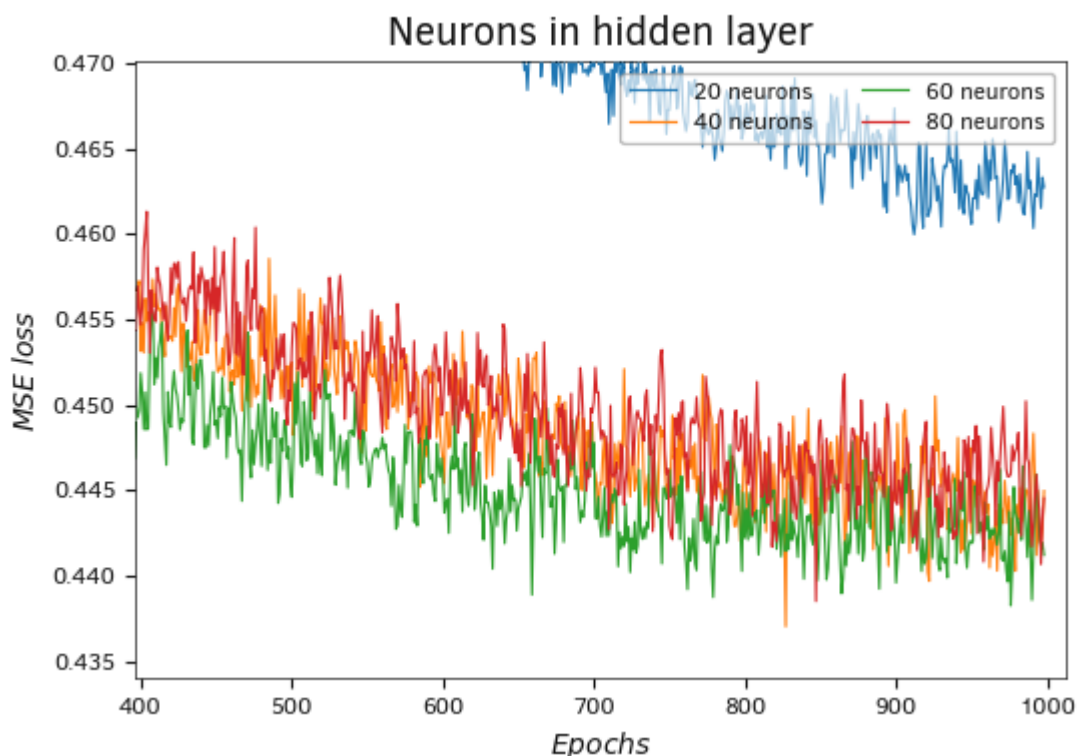
As can be seen in the graph, the model with 20 neurons in the hidden layer showed the worst result. They are clearly insufficient for such a task.

Regarding the other three models, it can be said that the variation in the graphs during the first 100 weight update iterations can be attributed to the randomness factor due to initializing the models with random weights. After about 250-300 iterations of updates to the weight matrix, the graphs are intertwined into a single bundle and from this point go on together.

Increasing the scale of the graph allows us to identify the main trend: as the number of neurons in the layer increases, the number of iterations required to reach local minima and overall train the neural network also increases. At the same time, local minima of neural networks with a large number of neurons fall lower, and their graphs have a lower frequency of oscillations.

Overall, throughout the entire training process, the model with 60 neurons demonstrates the best performance. Slightly behind, almost in parallel, is the graph of the model with 40 neurons in the hidden layer. For a model with 80 neurons in the hidden layer, 1000 iterations of updating the weight matrix were insufficient. This model shows a slower decrease in the value of the loss function. At the same

time, the dynamics of the loss function values graph demonstrate the potential for further reducing the loss function value with continued training of the model. There are valid reasons to expect a decrease in the performance achieved by the model with 60 neurons in the hidden layer.



Comparison of loss function dynamics using different numbers of neurons in the hidden layer (scale)

However, it's important to note that a decrease in error on the training dataset could also be associated with model overfitting. Therefore, before the practical deployment of a trained model, it's always important to test it on "unseen" data.

Training, validation, testing.

During training, we adjust the weight matrix parameters to achieve the minimum error on the training dataset. But how will the model behave on new data beyond the training sample? It's also important to consider that we are dealing with non-static data that is constantly changing, influenced by a large number of factors. Some of these factors are known to us, while we might not even be aware of others. And even about the factors known to us, we cannot say with certainty how they will change in the future. Moreover, we don't know how this will impact the variation of the data we are studying. It is most likely that the performance of the neural network will deteriorate on the new data. But what will that deterioration be? Are we willing to accept such risks?

The first step towards addressing this issue is the validation of the model's training parameters. For model validation, a dataset that is not part of the training set is used. Most often, the entire set of initial data is divided into three blocks:

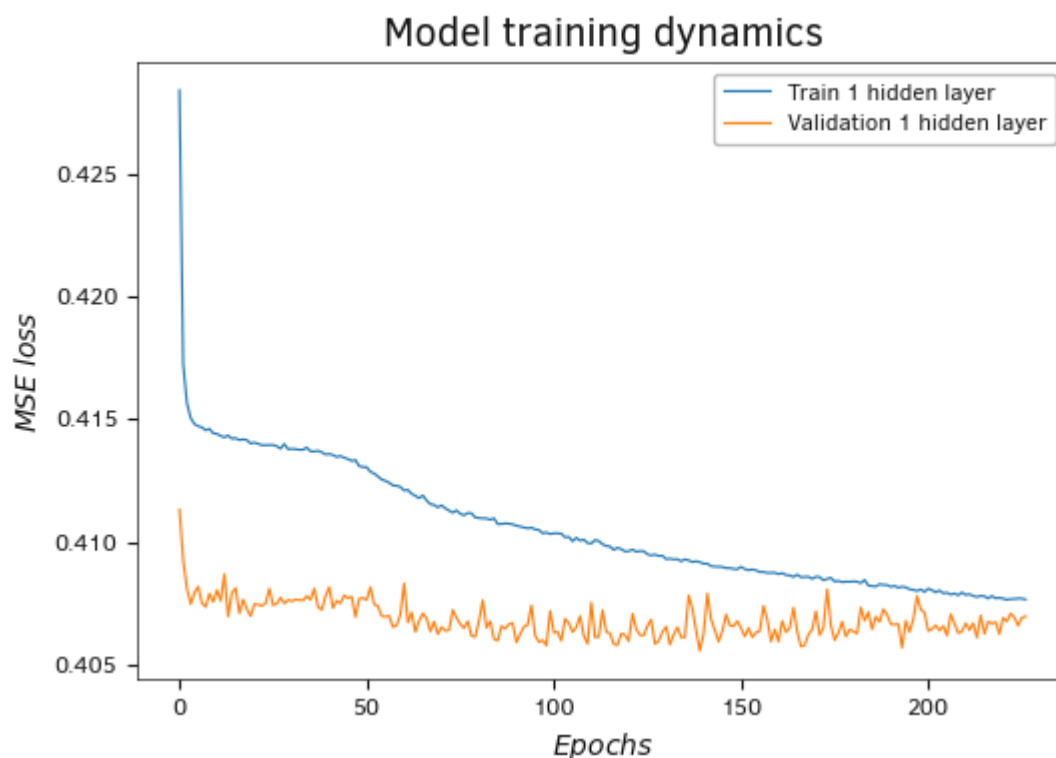
- training sample (~60%)
- validation sample (~20%)
- test sample (~20%)

The percentage distribution for each dataset is given as an example and can vary significantly depending on the specific task at hand.

3. Building the first neural network model in MQL5

The essence of the validation process lies in testing the parameters of the trained model on data that is not part of the training dataset. During validation, hyperparameters of the trained model are tuned to achieve the best possible performance.

When writing a script for a [fully connected perceptron](#) in Python, we allocated 20% of the training dataset for validation. The training of the first model demonstrated results similar to those obtained when training the model created in MQL5. That's a positive signal for us. Obtaining similar results when training models created in three different programming languages can indicate the correctness of the algorithm we have implemented.



Change in performance of a model with a single hidden layer on validation at pace with its training

Evaluating the graphs of the test results, one can notice a tendency for the error to decrease during the learning process. This is a positive signal that indicates the model's ability to learn and establish relationships between input data and target labels. At the same time, there's an increase in error on the validation data, which could indicate both the overfitting of the model and the non-representativeness of the validation dataset.

The issue might be that we specified a portion of the data for validation within the training dataset. In this case, the TensorFlow library takes the latest data in the training sample set. However, this approach doesn't always yield accurate results, as the outcomes of individual periods can be significantly influenced by local trends.

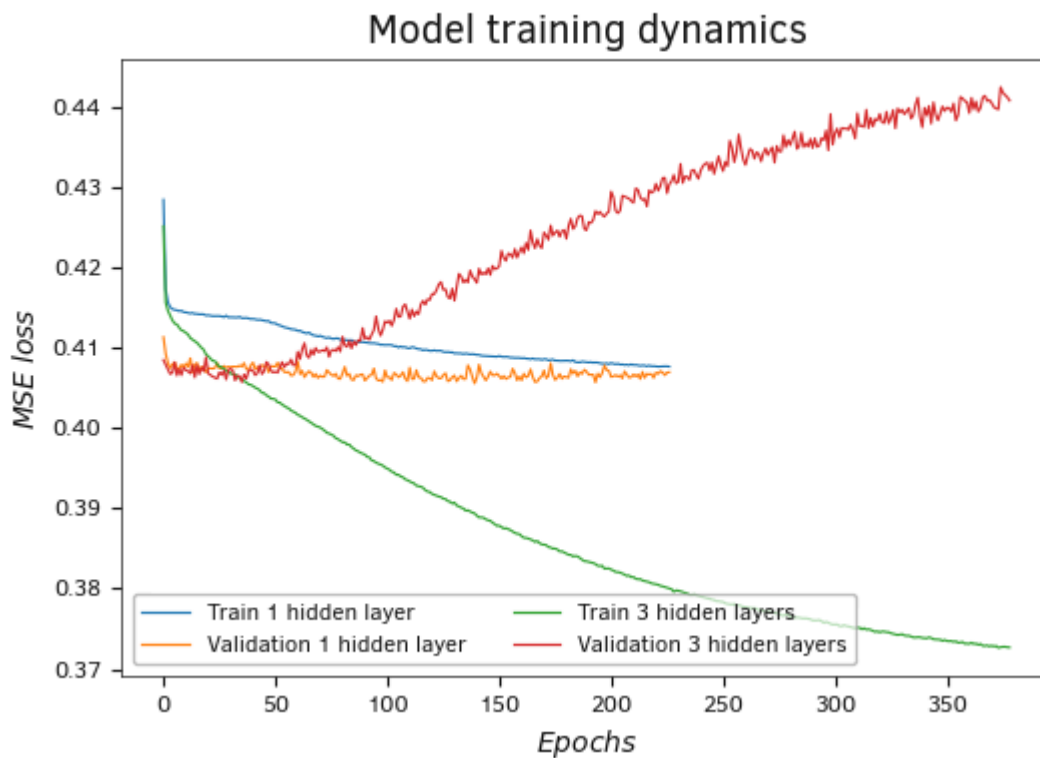
In the graph below, I can see the impact of both factors. The overall trend of increasing error on the validation data indicates the model's tendency to overfit, while the initial validation error being lower than the training error might be due to the influence of local trends.



Change in performance of a model with a single hidden layer on validation at pace with its training

The graph of the *accuracy* metric shows similar trends. The metric itself reflects the model accuracy as a proportion of correct answers in the total number of results. Here we observe an increase in the indicator during training with an almost unchanged indicator on validation. This may indicate that the model learns patterns that do not occur in the validation sample.

In theory, adding hidden layers should enhance the model's ability to learn and recognize more complex patterns and structures. We created the second model in Python with three hidden layers. Indeed, in this model in training, the error decreased significantly. But at the same time, it has further increased in the validation process. This is a clear sign of the model overfitting. When the model, due to its capabilities, does not generalize dependencies and simply "memorizes" pairs "initial data - target values", the result appears randomly on new data not included in the training sample.



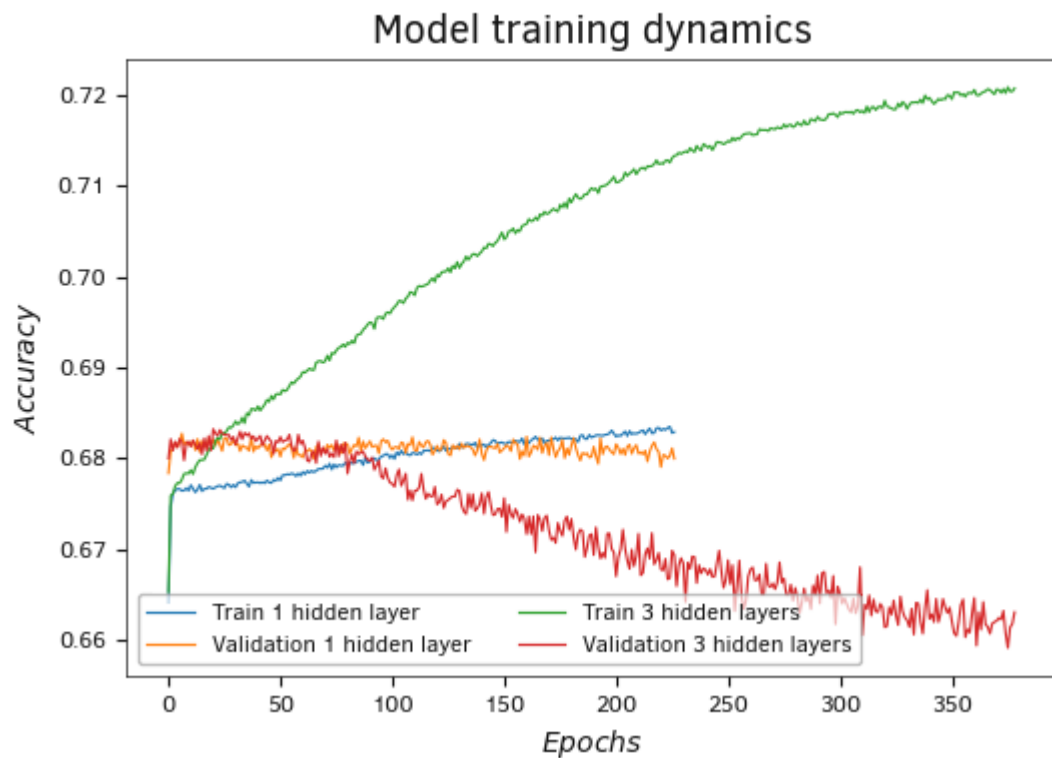
Change in performance of a model with three hidden layers on validation at pace with its training

The dynamics of the *accuracy* metric have trends similar to those of the loss function. The only difference is that the loss function decreases during the training process, while the *accuracy* increases.

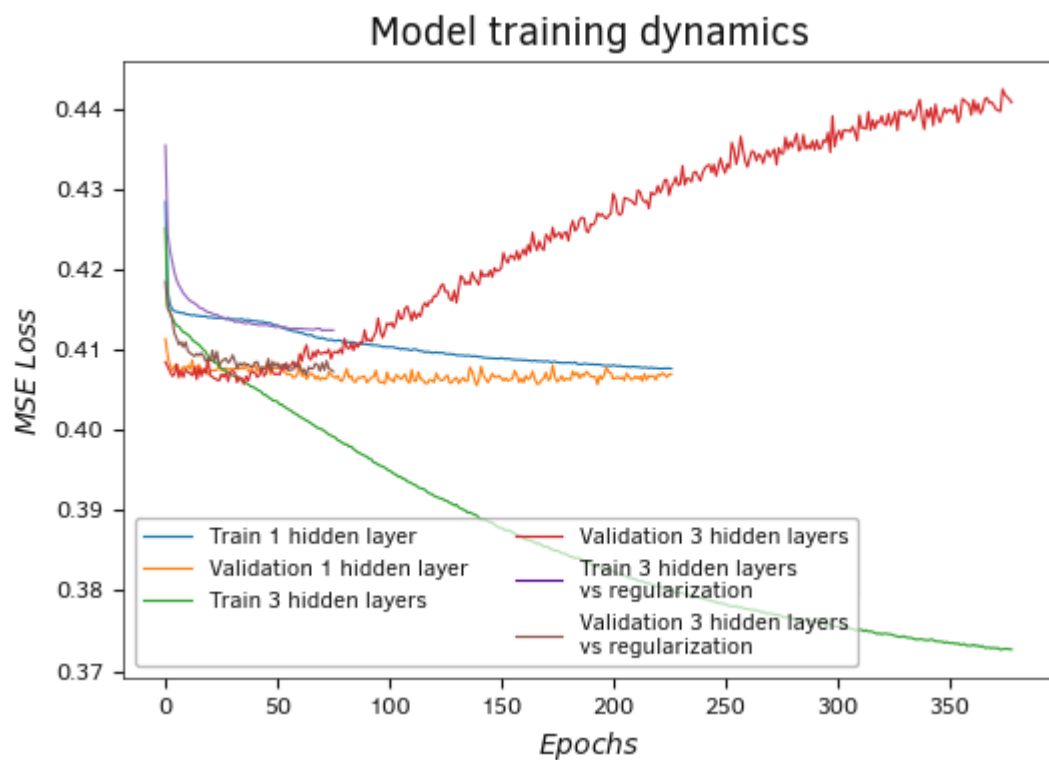
One way to combat overfitting is to regularize the model. We added *ElasticNet* regularization to the third model, and it did the job. When the model was trained with regularization, the error decreased at a slower rate. At the same time, the increase in error on the validation set has slowed down.

Once again, on the accuracy metric graph, we observe the same trends as on the loss function graph.

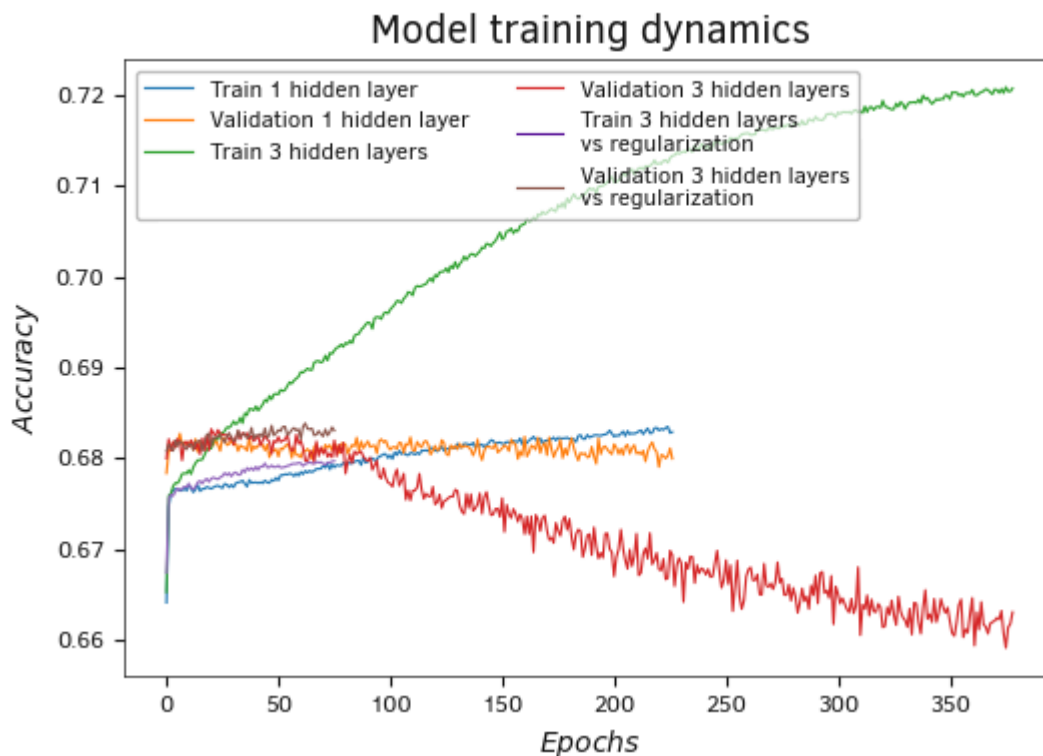
Note that the training and regularization parameters were not meticulously tuned. Therefore, learning outcomes cannot be considered definitive.



Change in performance of a model with three hidden layers on validation at pace with its training



The change in the performance of the model with three hidden layers and regularization on validation is occurring at a similar pace to its training.

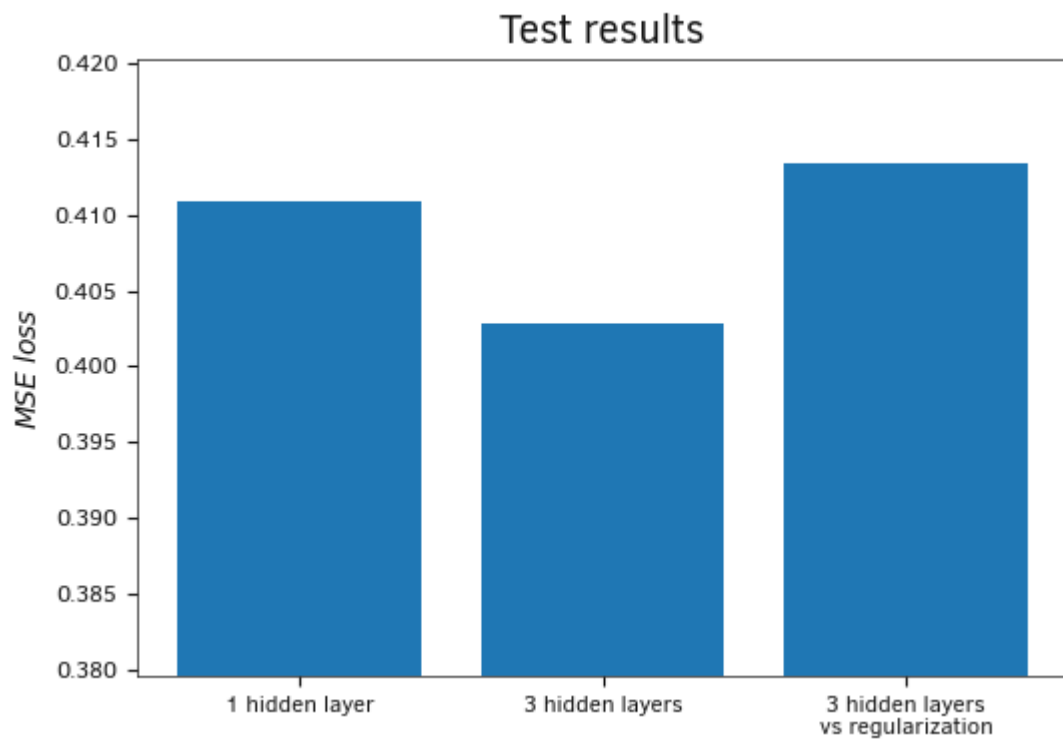


The change in the performance of the model with three hidden layers and regularization on validation is occurring at a similar pace to its training.

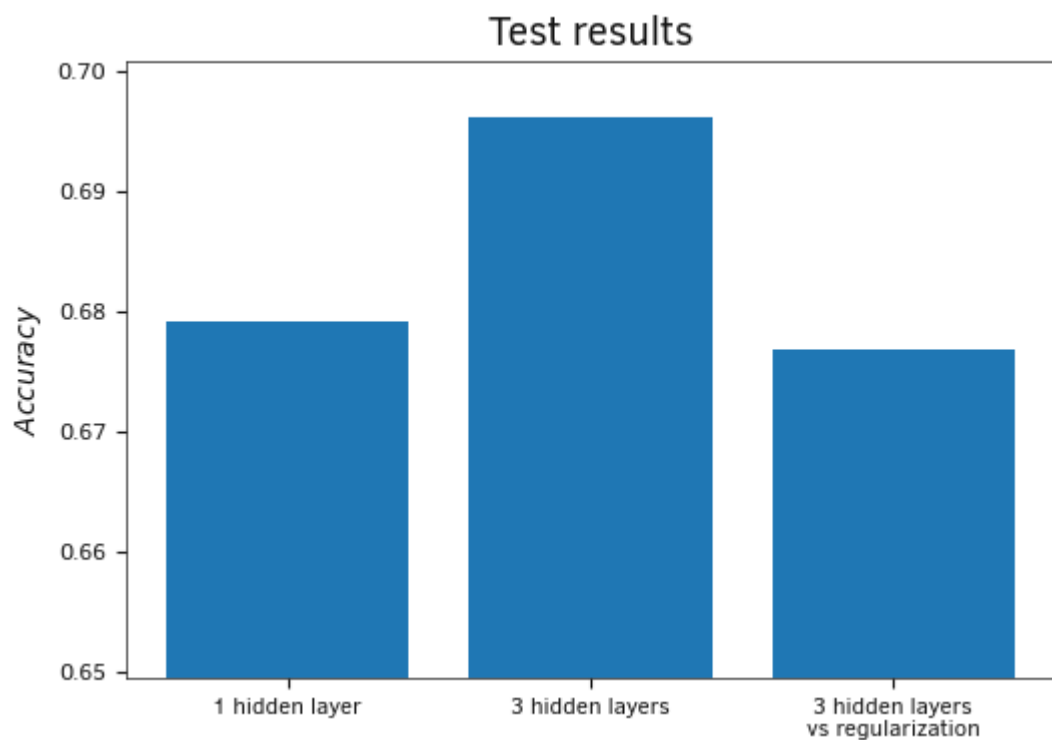
After training the models, we will evaluate them using the test dataset. In contrast to validation, the model with three hidden layers and no regularization demonstrated the lowest error on the test dataset. The model with regularization showed the maximum error. Such differences in results between the test and validation datasets can possibly be explained by the way the datasets were created. While the validation sample included only the most recent data from the training sample, the test sample collected random data sets from the entire population. Thus, the test sample can be considered more representative, as it is deprived of the influence of local tendencies.

Measuring the *accuracy* metric on the test sample showed similar results. The best result was obtained on the model with three hidden layers.

3. Building the first neural network model in MQL5



Comparison of model results on a test sample



Comparison of model results on a test sample

We can summarize the results of our practical work.

1. Normalizing the raw data before feeding it to the input of the neural network greatly increases the chances of convergence of the neural network and reduces the training time.

2. The learning rate should be carefully selected experimentally. Too high learning rates lead to unbalancing of the neural network and an increase in error. Too low learning rates lead to more time and computational resources spent on training the neural network. This increases the risk of stopping the learning process at a local minimum without achieving the desired result.
3. Increasing the number of neurons in the hidden layer gives improved results of training. But at the same time, training costs are also rising. When choosing the size of the hidden layer, it is necessary to find a balance between the training error and the resource cost of conducting the training of the neural network. It should be kept in mind that an excessive increase in the number of neurons in the hidden layer increases the risk of neural network overfitting.
4. Increasing the number of hidden layers also increases the model's ability to learn and recognize more complex shapes and structures. In this case, the model's propensity to overfitting increases significantly.
5. The use of a set of recent training sample values for validation is not always able to show true trends, as such a validation sample is strongly influenced by local trends and cannot be representative.

However, we are building a model for financial markets. It is important to us to make a profit both in the long term and in the present moment. Of course, there may be some localized losses, but they should not be large and frequent. Therefore, it is important to obtain acceptable results both on a single localized dataset and on a more representative sample. Probably, getting better results on a local segment has a higher impact: after making a local profit, we can retrain the model to adapt to new trends and make a profit on a new local segment. At the same time, if training costs exceed possible local losses, the profitability over a long period using a representative sample becomes more significant.

4. Basic types of neural layers

In the previous sections, we got acquainted with the architecture of a fully connected perceptron and constructed our first neural network model. We tested it in various modes, received our first results, and gained our first experience. However, the fully connected neural layers used in the perceptron, despite their merits, also possess certain drawbacks. For instance, a fully connected layer analyzes only the current data without any connection to previously processed data. Thus, each packet of information is analyzed in an informational vacuum. To expand the volume of analyzed data, it is necessary to continually increase the size of the model. Consequently, the expenses for training and operation grow exponentially. A fully connected layer analyzes the entire aggregate as a whole and fails to reveal dependencies between individual elements.

In this chapter, we will explore various architectural solutions for constructing neural layers aimed at overcoming the drawbacks of fully connected layers that we studied earlier. Fully connected neural networks analyze data without considering their context and interconnections, which can lead to insufficient efficiency and an increase in model volume. We will consider the following architectural approaches:

- **Convolutional Neural Networks (CNN)**: we will delve into their architecture and implementation principles, as well as examine ways to build them using MQL5 and OpenCL. Next, we will explore practical testing of convolutional models aimed at evaluating their performance and efficiency.
- **Recurrent Neural Networks (RNN)**: the architecture and implementation principles; ways to build LSTM blocks using MQL5 and organize parallel calculations using OpenCL. From this section, you will also learn how to implement RNNs in Python and test them.

Thus, in this chapter, we will study convolutional and recurrent neural networks, their operation and application in practical problems. We will also examine various methods of their construction and optimization.

4.1 Convolutional neural networks

We continue our exploration of neural network architectures, focusing now on the principles of operation and construction of convolutional neural networks (CNNs). These neural networks are widely used in tasks involving object recognition in photos and videos. It is considered that convolutional neural networks are resistant to changes in scale, shifts in perspective, and other spatial distortions of images. Their architecture enables them to equally effectively detect objects in any part of a scene.

In addition to the architectural differences we will discuss in the next chapter, there is a fundamental distinction in the logical processing of incoming data streams between convolutional and fully connected neural networks. As we have seen earlier, each neuron in a fully connected network is linked to all neurons in the previous layer, responding to distinct patterns in input data. Let's try to translate this to image pattern recognition.

Imagine training a neural network to recognize digits printed on a piece of paper. Each digit is printed on a perfectly clean sheet of paper, and your neural network has learned to recognize them perfectly. But when you input a signal with a slight noise, its output becomes unpredictable, as the noise alters the image, making it deviate from the ideal patterns in the training dataset.

The opposite situation is also possible. When you train a neural network on noisy images, where the object of interest is on some background, a fully connected neural network with a sufficient number of neurons and an adequate training dataset can solve such a task. At the same time, it perceives the picture as a whole. Changing or removing the background can disturb the equilibrium of a fully connected neural network, causing its output to become unpredictable. Once again, it's all about the integrity of the perception of the world. When using supervised learning, if we provided a neural network with a noisy image during training and gave it the correct answer, the neural network associated the image with the answer and memorized it. But it didn't pick out the right image from the picture; instead, it memorized the whole picture. Therefore, the absence of the background is perceived by the neural network as the absence of some component of the image. In such a case, it will be difficult to give the correct result.

Similarly, issues arise with rotations, zooms, or any minor alteration in input data, treating each change as new and unseen by a fully connected network. This demands additional resources for processing and memorization, posing performance challenges as the size of processed images increases.

In addition to the recognition problems mentioned above, there is also a performance problem. As the size of the processed images increases, the size of the incoming data stream also grows. As a consequence, the weight matrices also grow. Therefore, more memory is required to store the weight matrix and more time is required to train the neural network. That said, in many cases, a significant portion of the image does not carry useful information. Consequently, resources are being spent inefficiently.

To address these, convolutional neural networks were designed. Instead of examining the whole image, CNNs divide it into small components, scrutinizing each as if under a microscope. Each convolutional layer contains filters checking individual parts for correspondence to desired patterns, mitigating noise and background influence. The use of small weight matrices reduces memory requirements, and the size remains unchanged with larger processed images.

The use of small weight matrices for each filter allows for a significant reduction in the memory requirement for storing them. Moreover, in this approach, the size of the weight matrix does not depend on the size of the original image, but on the size of the filter image. Therefore, as the size of the processed images increases, the size of the weight matrix remains unchanged.

In addition to the aforementioned benefits, CNNs reduce the processed data with each layer since for each small constituent part of the image, only one value is returned, indicating the degree of similarity between the image and the desired pattern.

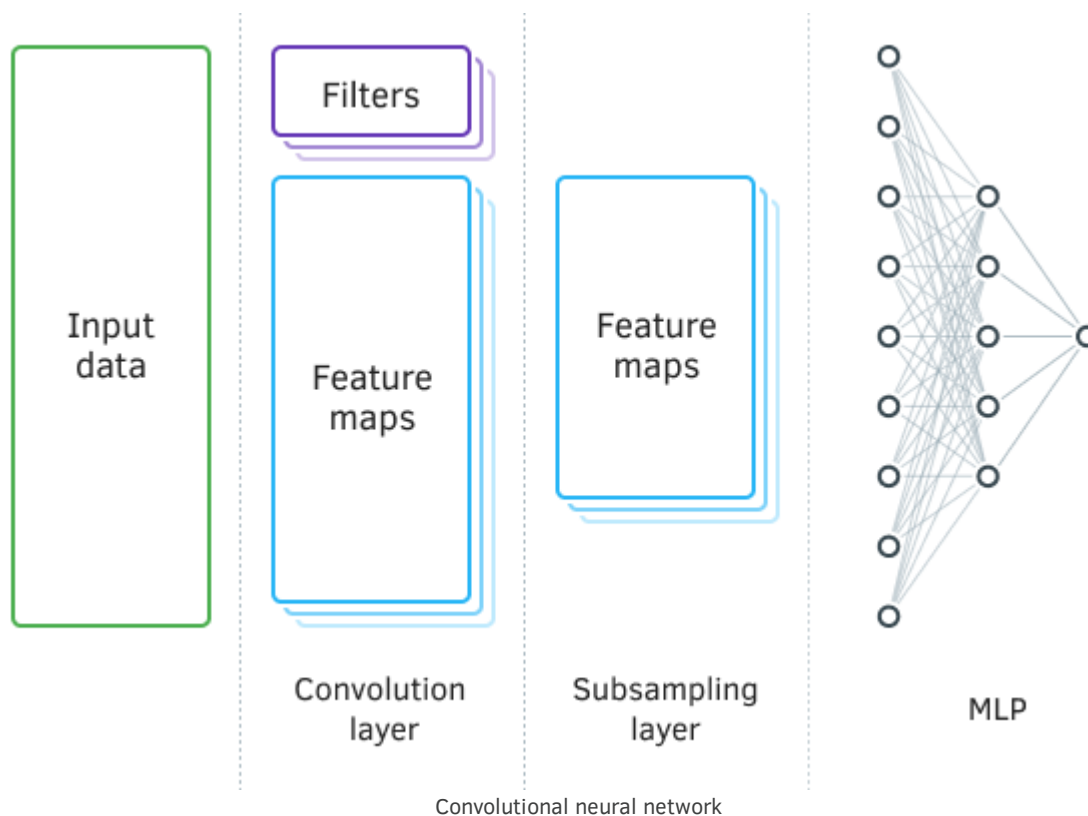
In terms of trading, I have tried to translate the advantages of CNNs into a new plane. Instead of searching for small constituents of the desired pattern in an image, we will be looking for small components of patterns from price candles and indicator values in the input data stream. By doing so, we will try to eliminate the influence of noise fluctuations on the overall result.

Furthermore, one of the major advantages of convolutional neural networks lies in the fact that the network learns the filters during the training process.

Let's delve into the architectures of this solution and get to know it more closely.

4.1.1 Description of architecture and implementation principles

Convolutional networks, in comparison with a fully connected perceptron, have two new types of layers: convolutional (filter) and pooling (subsampling). Alternating, the specified layers are designed to highlight the main components and filter out noise in the original data while simultaneously reducing the dimensionality (volume) of the data, which is then fed into a fully connected perceptron for decision-making. Depending on the tasks to be solved, it is possible to consistently use several groups of alternating convolutional and subsample layers.



4. Basic types of neural layers

The Convolution layer is responsible for recognizing objects in the source data set. In this layer, sequential operations of mathematical convolution of the original data with a small template (filter) are carried out, acting as the convolution kernel.

Convolution is an operation in functional analysis that, when applied to two functions f and g , returns a third function corresponding to the cross-correlation function of $f(x)$ and $g(-x)$. The operation of convolution can be interpreted as the "similarity" of one function to the mirrored and shifted copy of another.

In other words, the convolutional layer searches for a template element in the entire original sample. At the same time, on each iteration, the template is shifted across the array of original data with a specified step, which can be from 1 to the size of the template. If the magnitude of the shift step is smaller than the size of the template, then such convolution is called overlapping.

As a result of the convolution operation, we obtain a feature array that shows the similarity of the original data to the desired template at each iteration. Activation functions are used for data normalization. The size of the obtained array will be smaller than the array of original data, and the number of such arrays is equal to the number of templates (filters).

$$Size_{Conv} = \frac{Size_{Input} - Size_{Filter}}{Step} + 1$$

It is also important for us to note that the templates themselves are not specified during the design of the neural network but are selected during the training process.

Next subsample layer is used to reduce the size of the feature array and filter noise. The application of this iteration is based on the assumption that the similarity of the original data to the template is primary, and the exact coordinates of the feature in the array of original data are not so important. This allows addressing the scaling issue, as it permits some variability in the distance between the sought-after objects.

At this stage, data is condensed by maintaining the maximum or average value within a specified window. This way, only one value per data window is saved. Operations are carried out iteratively, shifting the window by a specified step with each new iteration. Data compaction is performed separately for each array of features.

Pooling layers with a window and a step of two are often used, which makes it possible to halve the size of the feature array. However, in practice, it is also possible to use a larger window. Furthermore, consolidation iterations can be carried out both with overlapping (when the step size is smaller than the window size) and without.

At the output of the pooling layer, we obtain arrays of features of smaller dimensions.

Depending on the complexity of the tasks being solved, after the pooling layer, it is possible to use one or several groups of convolutional and pooling layers. The principles of their construction and functionality comply with the principles described above.

In the general case, after one or several groups of "convolution + compaction", arrays of features obtained from all filters are gathered into a single vector and fed into a multilayer perceptron for the neural network's decision-making.

Convolutional neural networks are trained by the well-known method of error backpropagation. This method belongs to the unsupervised learning methods and imply propagating the error gradient from

the output layer of neurons through hidden layers to the input layer of neurons with adjustment of weights towards the anti-gradient.

Convolutional neural networks are trained by the well-known method of error backpropagation.

In the pooling layer, the error gradient is calculated for each element in the feature array, analogous to the gradients of neurons in a fully connected perceptron. The algorithm for transferring the gradient to the previous layer depends on the compaction operation used. If only the maximum value is taken, then the entire gradient is passed to the neuron with the maximum value. For the other elements within the consolidation window, a zero gradient is set, as during the forward pass they did not influence the final result. If the operation of averaging is used within the window, then the gradient is evenly distributed to all elements within the window.

Weights are not used in the compaction operation, therefore, nothing is adjusted during the training process.

Operations are somewhat more complicated when training the neurons of the convolutional layer. The error gradient is calculated for each element of the feature array and descends to the corresponding neurons of the previous layer. The process of training the convolutional layer is based on convolution and reverse convolution operations.

To propagate the error gradient from the pooling layer to the convolutional layer, first, the edges of the error gradient array, obtained from the pooling layer, are padded with zero elements, and then a convolution of the resulting array is performed with the convolution kernel rotated by 180° . The output is an array of error gradients equal to the input data array, in which the gradient indices will correspond to the index of the corresponding neuron of the previous layer.

To obtain the weight deltas, convolution is performed between the matrix of input values and the matrix of error gradients of this layer, rotated by 180° . The output is an array of deltas with a size equal to the convolution core. The resulting deltas should be adjusted for the derivative of the activation function of the convolutional layer and the learning coefficient. After that, the weights of the convolution core change by the value of the corrected deltas.

It probably sounds rather hard to understand. Let's try to clarify these points while considering the code in detail.

4.1.2 Construction using MQL5

As we have already seen in the description of the convolutional network architecture. To construct it, we need to create two new types of neural layers: convolutional and pooling. The first is responsible for data filtering and extracting the desired data, while the second is for pinpointing the points of maximum correspondence to the filter and reducing the data array's dimensionality. The convolutional layer has a weight matrix, but it is much smaller than the weight matrix of a fully connected layer due to the fact that it is searching for a small pattern. As for the pooling layer, it has no weighting coefficients at all. This reduction in the dimension of the weight matrix makes it possible to reduce the number of mathematical calculations and thereby increase the speed of information processing. At the same time, the number of operations decreases both during the forward and backward passes. Therefore, the time required to train the neural network is significantly reduced. The ability of the algorithm to filter out noise allows you to improve the quality of the neural network.

Pooling layer

We begin the implementation of the algorithm by constructing a pooling layer. To do this, we will create a class *CNeuronProof*. We have previously voiced the idea that for the continuity of neural layers, they will all inherit from one base class. Adhering to this concept, we will inherit the new neural layer from the previously created *CNeuronBase* class. The inheritance will be public. Therefore, all methods not overridden within the *CNeuronProof* class will be accessible from the parent class.

To cover additional requirements due to the peculiarities of the convolutional network algorithm, we will add variables to the new class to store additional information:

- *m_iWindow* – window size at the input of the neural layer
- *m_iStep* – step size of the input window
- *m_iNeurons* – output size of one filter
- *m_iWindowOut* – number of filters
- *m_eActivation* – activation function

Note that, unlike the base class *CNeuronBase*, we did not use a separate activation function class *CActivation* but introduced a new variable *m_eActivation*. The reason is that the pooling layer does not use the activation function in the previously considered form. Its functionality is slightly different here. Usually, the result of the pooling layer is the maximum or the arithmetic mean value of the analyzed window. Therefore, we implement new functionality within the methods of this class and will create a new enumeration with two elements:

- *AF_AVERAGE_POOLING* – the arithmetic mean of the input data window
- *AF_MAX_POOLING* – the maximum value of the input data window

At the same time, we deliberately will not make changes to the code of the base class regarding new activation functions, as they will not be used in other neural layer architectures.

```
//--- activation functions of the pooling layer
enum ENUM_PROOF
{
    AF_MAX_POOLING,
    AF_AVERAGE_POOLING
};
```

Another feature of the pooling layer is the absence of a weight matrix. Therefore, the layer will not participate in the process of training and updating the weights. In this case, we can even delete some objects to free up memory. At the same time, the pooling layer cannot be completely excluded from the backward pass, as it will be involved in the propagation of the error gradient. To avoid cluttering the dispatcher class methods with excessive checks and at the same time to exclude the invocation of unnecessary parent class methods, we will replace a number of methods with "stubs" that will return the value required for the normal operation of the integrated neural network algorithm.

- *CalcOutputGradient* always returns *false* because it is not intended to use the layer as an output layer for the neural network.
- *CalcDeltaWeights* and *UpdateWeights* always return *true*. The absence of a weight matrix makes these methods redundant, but for the correct operation of the entire model, it is necessary to return a positive result from the methods.
- *GetWeights* and *GetDeltaWeights* always return *NULL*. Methods have been overridden to prevent errors due to accessing a non-existent object.

4. Basic types of neural layers

Let's add another method to return the number of elements in the output of one filter and we will get the following class structure.

```
class CNeuronProof : public CNeuronBase
{
protected:
    uint          m_iWindow;           //Window size at the input of the neural
    uint          m_iStep;             //Input window step size
    uint          m_iNeurons;          //Output size of one filter
    uint          m_iWindowOut;        //Number of filters
    ENUM_PROOF    m_eActivation;       //Activation function
public:
    CNeuronProof(void);
    ~CNeuronProof(void) {};

    //---
    virtual bool  Init(const CLayerDescription *desc) override;
    virtual bool  FeedForward(CNeuronBase *prevLayer) override;
    virtual bool  CalcOutputGradient(CBufferType *target) override;
                                                { return false; }
    virtual bool  CalcHiddenGradient(CNeuronBase *prevLayer) override;
    virtual bool  CalcDeltaWeights(CNeuronBase *prevLayer) { return true; }
    virtual bool  UpdateWeights(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda) override
                                                { return true; }

    //---
    virtual CBufferType *GetWeights(void) const { return(NULL); }
    virtual CBufferType *GetDeltaWeights(void) const { return(NULL); }
    virtual uint      GetNeurons(void) const { return m_iNeurons; }
    //--- Methods for working with files
    virtual bool      Save(const int file_handle) override;
    virtual bool      Load(const int file_handle) override;
    //--- Object identification method
    virtual int        Type(void) override const { return(defNeuronProof); }
};
```

In the class constructor, we only initialize the added variables using initial values.

```
CNeuronProof::CNeuronProof(void) : m_eActivation(AF_MAX_POOLING),
                                    m_iWindow(2),
                                    m_iStep(1),
                                    m_iWindowOut(1),
                                    m_iNeurons(0)
{
}
```

We did not add any new objects, and the destructor of the base class is responsible for deleting those created in the base class. Therefore, the destructor of our class will remain empty.

Let's look further at the methods of the new class of pooling layer *CNeuronProof*. Let's examine the *Init* method that initializes the neural layer. In the parameters, the method, similar to the method of the parent class, receives a layer description object. At the beginning of the method, we check the validity of the received object as well as the match between the required layer and the current neural network class.

4. Basic types of neural layers

```
bool CNeuronProof::Init(const CLayerDescription *description)
{
    //--- control block
    if(!description || description.type != Type() ||
        description.count <= 0)
        return false;
```

After successfully passing the initial check, we will save and verify the parameters of the created layer:

- input window size
- input window step
- the number of filters
- the number of elements at the output of one filter

All specified parameters must be non-zero positive values.

```
//--- Save constants
m_iWindow = description.window;
m_iStep = description.step;
m_iWindowOut = description.window_out;
m_iNeurons = description.count;
if(m_iWindow <= 0 || m_iStep <= 0 || m_iWindowOut <= 0 || m_iNeurons <= 0)
    return false;
```

Let's also check the specified activation function. For the pooling layer, we can only use two variants of the activation function, `AF_AVERAGE_POOLING` and `AF_MAX_POOLING`. In other cases, we will exit the method with the result *false*.

```
//--- Checking the activation function
switch((ENUM_PROOF)description.activation)
{
    case AF_AVERAGE_POOLING:
    case AF_MAX_POOLING:
        m_eActivation = (ENUM_PROOF)description.activation;
        break;
    default:
        return false;
        break;
}
```

After successfully passing all the control blocks, we proceed directly to the initialization of the neural layer. First, we initialize the results vector *m_cOutputs* with zero values. We will create this buffer in the form of a rectangular matrix, with its rows representing individual filters.

```
//--- Initializing the results buffer
if(!m_cOutputs)
    if(!(m_cOutputs = new CBufferType()))
        return false;
if(!m_cOutputs.BufferInit(m_iWindowOut, m_iNeurons, 0))
    return false;
```

The use of matrices allows us to distribute data across filters within the scope of a single object. This gives us the opportunity to use a transparent data structure and exchange data between CPU and

OpenCL context. This will allow us to gain a little time when transferring data and organize parallel processing of data by all filters at once.

A similar approach is used for the *m_cGradients* error gradient buffer.

```
//--- Initialize the error gradient buffer
if(!m_cGradients)
    if(!(m_cGradients = new CBufferType()))
        return false;
if(!m_cGradients.BufferInit(m_iWindowOut, m_iNeurons, 0))
    return false;
```

After completing the initialization of the result and gradient buffers, we will remove unused objects and exit the method with a positive result.

```
//---
m_eOptimization = None;
//--- Deleting unused objects
if(!m_cActivation)
    delete m_cActivation;
if(!m_cWeights)
    delete m_cWeights;
if(!m_cDeltaWeights)
    delete m_cDeltaWeights;
for(int i = 0; i < 2; i++)
    if(!m_cMomenum[i])
        delete m_cMomenum[i];
//---
return true;
}
```

Now that we have completed the initialization of the neural layer, let's move on to implementing the feed-forward pass in the *FeedForward* method. Similar to the previous method, the forward pass method is constructed following the concept of inheritance and overriding virtual methods of the base class while adding new functionality. In its parameters, the method receives a pointer to an object of the previous neural layer. As always, at the beginning of the method, we will set up a validation block to check the input data. Here, we are checking the validity of pointers to the previous neural layer and the result buffers of both the previous and current neural layers.

```
bool CNeuronProof::FeedForward(CNeuronBase *prevLayer)
{
//--- Control block
if(!prevLayer || !m_cOutputs ||
    !prevLayer.GetOutputs())
    return false;
CBufferType *input_data = prevLayer.GetOutputs();
```

After successfully passing the control block, we will save a pointer to the result buffer of the previous layer and create a branching algorithm in the method based on the computational device in use: CPU or OpenCL context. We will return to the multi-threaded calculation algorithm a little later. Now, let's consider the implementation in MQL5.

Once again, we emphasize that the subsample layer does not have a weight matrix. And just like all other neural layers, it uses the same activation function for all neurons and filters. So, the difference

between the filter outputs can only occur when different input data is used. In other words, the number of filters in the pooling layer must match the number of filters in the preceding convolutional layer. So, we will first copy the original data matrix and reformat it if necessary.

```
//--- Branching of the algorithm depending on the execution device
if(!m_cOpenCL)
{
    MATRIX inputs = input_data.m_mMatrix;
    if(inputs.Rows() != m_iWindowOut)
    {
        ulong cols = (input_data.Total() + m_iWindowOut - 1) / m_iWindowOut;
        if(!inputs.Reshape(m_iWindowOut, cols))
            return false;
    }
}
```

It should be noted that despite the assumption of using a pooling layer after convolutional layers, our method allows for its use after the base class of a fully connected neural layer. That is why we copy the initial data matrix. This allows us to seamlessly reformat it into the desired format without the fear of disrupting the structure of the preceding layer.

It must be noted that MQL5 does not support three-dimensional matrices. Therefore, from this point on, we will need to work separately for each filter. First, we will create a local matrix with the number of rows and columns equal to the dimensions of the results of one filter and the input window, respectively. We organize two nested loops: an outer loop with a number of iterations equal to the number of filters, and an inner loop with a number of iterations equal to the number of elements in one filter of the current layer.

```
//--- Create a local matrix to collect data from one filter
MATRIX array = MATRIX::Zeros(m_iNeurons, m_iWindow);
m_cOutputs.m_mMatrix.Fill(0);
//--- Filter iteration cycle
for(uint f = 0; f < m_iWindowOut; f++)
{
    //--- Loop through the elements of the results buffer
    for(uint o = 0; o < m_iNeurons; o++)
    {
        uint shift = o * m_iStep;
        for(uint i = 0; i < m_iWindow; i++)
            array[o, i] = ((shift + i) >= inputs.Cols() ? 0 :
                           inputs[f, shift + i]);
    }
}
```

In the inner loop, we implement another nested loop. In its body, we will distribute the input data of one filter into the previously created matrix according to the size of the data window and its step. The use of a loop is driven by the need for a unified approach in cases where the size and stride are not equal.

After distributing the initial data, we will use matrix operations according to the given activation function. The resulting vector is stored in the results matrix. The row of the results matrix corresponds to the number of the analyzed filter.

```
//--- Saving the current result in accordance with the activation function
switch(m_eActivation)
{
    case AF_MAX_POOLING:
```

```

        if(!m_cOutputs.Row(array.Max(1), f))
            return false;;
        break;
    case AF_AVERAGE_POOLING:
        if(!m_cOutputs.Row(array.Mean(1), f))
            return false;
        break;
    default:
        return false;
    }
}
}

```

I use the term 'filter' to maintain a clear chain in your understanding: the filter from the convolutional layer transitions to the filter in the pooling layer. Iterations of the pooling layer can hardly be called a filter. At the same time, I want it to be clear in your understanding that the convolutional and pooling layers, while organized into two objects of neural layers, form a single integrated structure. Therefore the same terminology is used.

After successfully completing all iterations of the loop system, we exit the method with the result *true*.

```

    else
    {
        ///--- The multi-threaded calculation block will be added in the next chapter
        return false;
    }
    ///--- Successful completion of the method
    return true;
}

```

The feed-forward pass is followed by the backpropagation pass. The absence of a weight matrix in the pooling layer allows the backpropagation pass to be organized in a single method, unlike the base class of the neural network *CNeuronBase*, in which the backpropagation pass is divided into several functional methods.

Essentially, for the pooling layer, the backpropagation pass is the *CalcHiddenGradient* method that propagates the error gradient to the hidden layer. We have replaced the remaining methods with placeholders, as mentioned earlier.

The *CalcHiddenGradient* method itself is built within the framework of our concept of using a single format of virtual methods for all classes of neural networks with common inheritance from a single base class of the neural layer. Therefore, similar to the method of the base class of the neural layer *CNeuronBase::CalcHiddenGradient*, the method receives a pointer to the object of the previous neural layer in its parameters. At the beginning of the method, a control block for checking incoming data is organized. Here, we are checking the correctness of the pointer received as a parameter, which points to the object of the previous neural layer, and the presence of active result buffers and error gradients in the previous layer. We also check the correctness of the result buffers and error gradients of the current layer.

```

bool CNeuronProof::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    ///--- Control block
    if(!prevLayer || !m_cOutputs ||
        !m_cGradients || !prevLayer.GetOutputs() ||

```

```

        !prevLayer.GetGradients())
    return false;
CBufferType *input_data = prevLayer.GetOutputs();
CBufferType *input_gradient = prevLayer.GetGradients();
if(!input_gradient.BufferInit(input_data.Rows(), input_data.Cols(), 0))
    return false;

```

After successfully passing the control block, similar to the forward pass method, we will copy and reformat the matrix of input data. We will also create a zero local matrix of similar size, to accumulate error gradients.

Note that in the base neural layer class, we did not pre-zero the gradient buffer. The difference lies in the approach to passing the error gradients to the previous layer. The base class algorithm includes recalculation and saving of the gradient value for each element. With this approach, pre-clearing the buffer doesn't make sense because any value will be overwritten with a new one. In the pooling layer algorithm, recording the error gradient into each buffer element of the previous layer is only envisaged when using *Average Pooling* (arithmetic mean value). In the case of *Max Pooling* (maximum value), the error gradient is transferred only to the element with the maximum value, because only it affects the subsequent result of the neural network. The remaining elements receive a zero error gradient. Therefore, we immediately clear the entire buffer and only insert the gradient value for elements that affect the result.

Next, we divide the algorithm depending on the computational device. We will not now discuss the implementation of multi-threaded calculations in OpenCL but will focus on implementation using MQL5.

Here, just like in the forward pass, we organize a system of nested loops to iterate through filters and their elements. Inside the loops, the error gradient is distributed to the elements of the previous layer depending on the activation function.

```

//--- Branching of the algorithm depending on the execution device
if(!m_cOpenCL)
{
    MATRIX inputs = input_data.m_mMatrix;
    ulong cols = (input_data.Total() + m_iWindowOut - 1) / m_iWindowOut;
    if(inputs.Rows() != m_iWindowOut)
    {
        if(!inputs.Reshape(m_iWindowOut, cols))
            return false;
    }
}
//--- Create a local matrix to collect data from one filter
MATRIX inputs_grad = MATRIX::Zeros(m_iWindowOut, cols);

//--- Filter iteration cycle
for(uint f = 0; f < m_iWindowOut; f++)
{
    //--- Loop through the elements of the results buffer
    for(uint o = 0; o < m_iNeurons; o++)
    {
        uint shift = o * m_iStep;
        TYPE out = m_cOutputs.m_mMatrix[f, o];
        TYPE gradient = m_cGradients.m_mMatrix[f, o];
        //--- Propagate the gradient in accordance with the activation function
        switch(m_eActivation)

```



```

{
    case AF_MAX_POOLING:
        for(uint i = 0; i < m_iWindow; i++)
        {
            if((shift + i) >= cols)
                break;
            if(inputs[f, shift + i] == out)
            {
                inputs_grad[f, shift + i] += gradient;
                break;
            }
        }
        break;
    case AF_AVERAGE_POOLING:
        gradient /= (TYPE)m_iWindow;
        for(uint i = 0; i < m_iWindow; i++)
        {
            if((shift + i) >= cols)
                break;
            inputs_grad[f, shift + i] += gradient;
        }
        break;
    default:
        return false;
}
}

}

//--- copy the gradient matrix to the buffer of the previous neural layer
if(!inputs_grad.Reshape(input_gradient.Rows(), input_gradient.Cols()))
    return false;
input_gradient.m_mMatrix = inputs_grad;
}

```

When using the arithmetic average (*AF_AVERAGE_POOLING*), the error gradient is equally distributed to all elements in the input data window corresponding to the result element.

When using the maximum value (*AF_MAX_POOLING*), the entire error gradient is passed on to the element with the maximum value. Moreover, when there are multiple elements with the same maximum value, the error gradient is passed to the element with the minimum index in the result buffer of the previous layer. This choice was made deliberately to enhance the overall efficiency of the neural network. The reason for this is that when passing the same gradient to elements with the same value, we risk getting into a situation where two or more neurons will work synchronously, producing identical results. Duplicating the signal with different neurons doesn't increase the significance of the signal; it only reduces the efficiency of the neural network's operation. After all, when working synchronously, the efficiency of such neurons becomes equal to the work of one neuron. Therefore, by passing the error gradient to only one neuron, we hope that the next time, another element will receive a different gradient value and disrupt the synchronization of neurons' operation.

After filling the local gradient matrix, we transfer the obtained result to the gradient buffer of the previous layer and exit the method with the result of the operations.

4. Basic types of neural layers

```
    else
    {
//--- The multi-threaded calculation block will be added in the next chapter
        return false;
    }
//--- Successful completion of the method
    return true;
}
```

The methods discussed above describe the main functionality of the pooling layer. For the completeness of the class functionality, it's necessary to add methods for working with files to save information about the trained neural network to a file. The main characteristic of the pooling layer is the absence of a weight matrix. Hence, there are no trainable elements and no need to store any data buffers. To fully restore the functionality of the layer, it's sufficient to save the values of its variables that define the operational parameters of the class.

- *m_iWindow* – window size at the input of the neural layer
- *m_iStep* – step size of the input window
- *m_iNeurons* – output size of one filter
- *m_iWindowOut* – number of filters
- *m_eActivation* – activation function

```
bool CNeuronProof::Save(const int file_handle)
{
//--- Control block
    if(file_handle == INVALID_HANDLE)
        return false;
//--- Save constants
    if(FileWriteInteger(file_handle, Type()) <= 0)
        return false;
    if(FileWriteInteger(file_handle, (int)m_iWindow) <= 0)
        return false;
    if(FileWriteInteger(file_handle, (int)m_iStep) <= 0)
        return false;
    if(FileWriteInteger(file_handle, (int)m_iWindowOut) <= 0)
        return false;
    if(FileWriteInteger(file_handle, (int)m_iNeurons) <= 0)
        return false;
    if(FileWriteInteger(file_handle, (int)m_eActivation) <= 0)
        return false;
//--- Successful completion of the method
    return true;
}
```

The method for restoring the layer from a file is slightly more complex than the method for saving it. In this case, I think the term 'recovery' is more appropriate than 'loading'. This is due to the fact that we will not read any information about training and development of the method from the file. From the file, we first read the layer parameters, which contain roughly the same amount of information as we pass in the initialization method in the layer description object. Then we initialize the result and error gradient buffers.

```
bool CNeuronProof::Load(const int file_handle)
```

```

{
//--- Control block
    if(file_handle == INVALID_HANDLE)
        return false;
//--- Load constants
    m_iWindow = (uint)FileReadInteger(file_handle);
    m_iStep = (uint)FileReadInteger(file_handle);
    m_iWindowOut = (uint)FileReadInteger(file_handle);
    m_iNeurons = (uint)FileReadInteger(file_handle);
    m_eActivation = (ENUM_PROOF)FileReadInteger(file_handle);
//--- Initialize the results buffer
    if(!m_cOutputs)
    {
        m_cOutputs = new CBufferType();
        if(!m_cOutputs)
            return false;
    }
    if(!m_cOutputs.BufferInit(m_iWindowOut, m_iNeurons, 0))
        return false;
//--- Initialize the error gradient buffer
    if(!m_cGradients)
    {
        m_cGradients = new CBufferType();
        if(!m_cGradients)
            return false;
    }
    if(!m_cGradients.BufferInit(m_iWindowOut, m_iNeurons, 0))
        return false;
//---
    return true;
}

```

At this stage, we can say that we have completed the first part of the work on constructing convolutional neural network objects. Now we will move on to the second stage, building a convolutional layer class.

Convolutional layer

The construction of the convolutional layer is carried out in the *CNeuronConv* class, which we will inherit from the *CNeuronProof* pooling layer class created above. Inheriting from the pooling layer class does not violate our concept of having all classes in our neural network inherit from a common base class. The pooling layer class is a direct descendant of the base class, and all its descendants will also be descendants of the base class.

At the same time, by inheriting from the pooling layer class, we immediately gain access to all the added and overridden functionality, including variables for working with data windows. Moreover, inheriting objects and variables reinforces the connection between classes and underscores the unity of approaches in data processing.

Thus, thanks to inheritance, in the convolutional layer class *CNeuronConv*, we will use objects and variables declared in the parent classes. We don't need to declare any new objects and variables. As a consequence, the constructor and destructor of our class remain empty methods. At the same time,

the convolutional layer class uses the weight matrix. In this case, we will need to override some previously set stubs.

- *UpdateWeights* completely satisfies the algorithm of the method of the base class *CNeuronBase*, so let's call its execution.
- *GetWeights* and *GetDeltaWeights* return pointers to the corresponding data buffers.

As a result, the class structure will take the following form.

```
class CNeuronConv      : public CNeuronProof
{
public:
    CNeuronConv(void) {};
    ~CNeuronConv(void) {};

    //---
    virtual bool      Init(const CLayerDescription *desc) override;
    virtual bool      FeedForward(CNeuronBase *prevLayer);
    virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer);
    virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer);
    virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                    VECTOR &Beta, VECTOR &Lambda)
    {
        return CNeuronBase::UpdateWeights(batch_size, learningRate,
                                            Beta, Lambda);
    }
    //---
    virtual CBufferType* GetWeights(void)      const { return(m_cWeights); }
    virtual CBufferType* GetDeltaWeights(void) const { return(m_cDeltaWeights); }
    bool      SetTransposedOutput(const bool value);
    //--- methods for working with files
    virtual bool      Save(const int file_handle);
    virtual bool      Load(const int file_handle);
    //--- object identification method
    virtual int      Type(void)      const { return(defNeuronConv); }
};
```

Let's examine the implementation of the *Init* method that initializes the convolutional layer. It partially combines the initialization methods of both parent classes. Unfortunately, we cannot use any of them: in the initialization method of the base class, buffers of incorrect sizes will be created and will still need to be overridden, and in the initialization method of the pooling layer, objects that will need to be recreated later are deleted. Therefore, we will write the entire algorithm into the method.

Like similar methods in the parent classes, the initialization method receives a pointer to an object describing the created neural layer in its parameters. As before, the method starts with a control block in which we validate the received pointer, the specified type of the layer being created, and the layer parameters.

```
bool CNeuronConv::Init(const CLayerDescription *desc)
{
    //--- control block
    if(!desc || desc.type != Type() || desc.count <= 0 || desc.window <= 0)
        return false;
```

After executing the control block, we save the layer parameters into special variables and initialize the necessary buffers.

```
//--- save constants
m_iWindow = desc.window;
m_iStep = desc.step;
m_iWindowOut = desc.window_out;
m_iNeurons = desc.count;
//--- save parameter optimization method
m_eOptimization = desc.optimization;
```

First, we initialize the results buffer *m_cOutputs*. Similar to the pooling layer, we set the number of rows and columns of the buffer matrix equal to the number of filters and the number of elements in one filter, respectively. The buffer is initialized with zero values.

Next, we initialize the *m_cGradients* error gradient buffer with zero values. We set its size equal to the size of the *m_cOutputs* results buffer.

```
//--- initialize the results buffer
if(!m_cOutputs)
    if(!(m_cOutputs = new CBufferType()))
        return false;
//--- initialize the error gradient buffer
if(!m_cGradients)
    if(!(m_cGradients = new CBufferType()))
        return false;
if(!m_cOutputs.BufferInit(m_iWindowOut, m_iNeurons, 0))
    return false;
if(!m_cGradients.BufferInit(m_iWindowOut, m_iNeurons, 0))
    return false;
```

Next, we will need to initialize an instance of the activation function object. As you may recall, during the development of the base neural layer class, we decided to separate all the work related to initializing the activation function instance into a separate method called *SetActivation*. Here we just call this method of the parent class, and check the result of the operations.

```
//--- initialize the activation function class
VECTOR params=desc.activation_params;
if(!SetActivation(desc.activation, params))
    return false;
```

Then we initialize the weight matrix with random values. The number of rows in the weight matrix is equal to the number of filters being used, and the number of columns in the matrix is one greater than the size of the analyzed window. The added element is used for *bias*. The matrix is initialized with random values.

```
//--- initialize the weight matrix buffer
if(!m_cWeights)
    if(!(m_cWeights = new CBufferType()))
        return false;
if(!m_cWeights.BufferInit(desc.window_out, desc.window + 1))
    return false;
double weights[];
double sigma = desc.activation == AF_LRELU ?
```

```

        2.0 / (double)(MathPow(1 + desc.activation_params[0], 2) *
                                desc.window) :

        1.0 / (double)desc.window;
    if(!MathRandomNormal(0, MathSqrt(sigma), m_cWeights.Total(), weights))
        return false;
    for(uint i = 0; i < m_cWeights.Total(); i++)
        if(!m_cWeights.m_mMatrix.Flat(i, (TYPE)weights[i]))
            return false;

```

At the end of the method, we initialize the buffers involved in the learning process. These are: a buffer of weight deltas (also known as a buffer of accumulated gradients) and moment buffers. Recall that the number of moment buffers used depends on the user-specified method for optimizing model parameters. The sizes of the specified buffers will correspond to the size of the weights matrix.

```

//--- initialize the gradient buffer at the weight matrix level
    if(!m_cDeltaWeights)
        if(!(m_cDeltaWeights = new CBufferType()))
            return false;
    if(!m_cDeltaWeights.BufferInit(desc.window_out, desc.window + 1, 0))
        return false;
//--- initialize moment buffers
    switch(desc.optimization)
    {
    case None:
    case SGD:
        for(int i = 0; i < 2; i++)
            if(m_cMomenum[i])
                delete m_cMomenum[i];
        break;
    case MOMENTUM:
    case AdaGrad:
    case RMSProp:
        if(!m_cMomenum[0])
            if(!(m_cMomenum[0] = new CBufferType()))
                return false;
        if(!m_cMomenum[0].BufferInit(desc.window_out, desc.window + 1, 0))
            return false;
        if(m_cMomenum[1])
            delete m_cMomenum[1];
        break;
    case AdaDelta:
    case Adam:
        for(int i = 0; i < 2; i++)
        {
            if(!m_cMomenum[i])
                if(!(m_cMomenum[i] = new CBufferType()))
                    return false;
            if(!m_cMomenum[i].BufferInit(desc.window_out, desc.window + 1, 0))
                return false;
        }
        break;
    default:

```

```

        return false;
        break;
    }
    return true;
}

```

After initializing the class, we will move on to the forward pass method, which we will create in the overridden virtual method *FeedForward*. This way, we continue to exploit the concepts of inheritance and virtualization of class methods. In its parameters, the feed-forward pass method receives a pointer to the object of the previous layer, just like all the similar methods in the parent classes.

At the beginning of the method, as usual, we will insert a control block for checking the source data. In this method, we validate the received pointer to the object of the preceding neural layer and check for the presence of an 'active' result buffer in it. We also check whether the result buffer and weight matrix of the current layer have been created. To simplify the data access procedure for the result buffer of the preceding layer, we will store a pointer to this object in a local variable.

```

bool CNeuronConv::FeedForward(CNeuronBase *prevLayer)
{
    ///--- control block
    if(!prevLayer || !m_cOutputs || !m_cWeights || !prevLayer.GetOutputs())
        return false;
    CBufferType *input_data = prevLayer.GetOutputs();
    ulong total = input_data.Total();
}

```

Next, we divide the algorithm into two threads depending on the execution device. We will discuss the algorithm for constructing multi-threaded calculations using OpenCL technology in the next chapter. Now let's look at the algorithm for arranging operations using MQL5.

The forward pass convolutional layer algorithm will somewhat resemble the similar pooling layer method. This is quite understandable: both layers work with a data window, which moves through the initial data array with a given step. Differences exist in the methods for processing the set of values that fall into the window.

Another difference lies in the approach to the perception of the array of initial data. The pooling layer in the convolutional neural network algorithm is placed after the convolutional layer, which can contain multiple filters. Consequently, the result buffer will contain the results of processing the data by multiple filters. The pooling layer is supposed to separate the results of one filter from another. In the convolutional layer, I chose to simplify this aspect, so I treat the entire input array as a single vector of input data. This approach allows us to simplify the method algorithm without losing the quality of the neural network in general.

Let's return to the algorithm. Before using matrix operations, we need to transform the vector of input data into a matrix with a number of rows equal to the number of elements in one filter. The number of columns should correspond to the size of the analyzed window of input data. Here, there are two possible scenarios: whether the size of the analyzed window is equal to its step or not.

In the first case, we can simply reformat the vector into a matrix. In the second case, we need to create a loop system for copying data.

```

///--- branching of the algorithm depending on the execution device
if(!m_cOpenCL)
{
    MATRIX m;
}

```

4. Basic types of neural layers

```
if(m_iWindow == m_iStep && total == (m_iNeurons * m_iWindow))
{
    m = input_data.m_mMatrix;
    if(!m.Reshape(m_iNeurons, m_iWindow))
        return false;
}
else
{
    if(!m.Init(m_iNeurons, m_iWindow))
        return false;
    for(ulong r = 0; r < m_iNeurons; r++)
    {
        ulong shift = r * m_iStep;
        for(ulong c = 0; c < m_iWindow; c++)
        {
            ulong k = shift + c;
            m[r, c] = (k < total ? input_data.At((uint)k) : 0);
        }
    }
}
```

Then, we will add the *bias* vector, which includes a single column of ones, to the resulting matrix. We multiply the resulting matrix by the transposed weight matrix.

```
//--- add a bias column
if(!m.Resize(m.Rows(), m_iWindow + 1) ||
    !m.Col(VECTOR::Ones(m_iNeurons), m_iWindow))
    return false;
//--- Calculate the weighted sum of elements of the input window
m_cOutputs.m_mMatrix = m_cWeights.m_mMatrix.MatMul(m.Transpose());
}
```

Finally, we call the *Activation* method of the class of the activation function and terminate the method.

```
else
{
    //--- The multi-threaded calculation block will be added in the next chapter
    return false;
}
if(!m_cActivation.Activation(m_cOutputs))
    return false;
//--- Successful completion of the method
return true;
}
```

After completing work on the feed-forward pass, we will move on to working on the backpropagation pass. Unlike the pooling layer, the convolutional layer contains a weight matrix. Therefore, to organize the pass, we need a full set of methods.

A little ahead, I will say that the weight matrix update method from the base class is perfectly suitable. However, since we inherited not directly from the *CNeuronBase* class but from a pooling layer *CNeuronProof*, in which the method was replaced by a stub, we will have to forcefully turn to the base class method.


```

bool CNeuronConv::UpdateWeights(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda)
{
    return CNeuronBase::UpdateWeights(batch_size, learningRate, Beta, Lambda);
}

```

But let's return to the logical chain of the backpropagation algorithm and take a look at the method for distributing the gradient through the hidden layer, *CNeuronConv::CalcHiddenGradient*.

If you look at the influence of the elements of the initial data on the elements of the results, you will notice a dependence. Each element of the resulting vector analyzes a block of data from the initial data vector in the size of the specified window. Similarly, each element of the initial data affects the value of elements in the result vector within a certain influence window. The size of this window depends on the step with which the input window moves across the source data array. With a step equal to one, both windows are equal. However, as the step increases, the size of the influence window decreases. Consequently, to propagate the error gradient, we need to collect error gradients from elements of the subsequent layer within the influence window.

I propose to look at the practical implementation of this method. We continue working with the virtual methods of the parent classes. In the parameters, the method receives a pointer to the object of the previous layer. Following the same pattern as with other methods, we start with a data validation block at the beginning of the method. Here, we validate the received pointer in the parameters and check for the presence of valid objects for output value buffers and error gradients of the previous layer. We also check for the presence of the error gradient buffer and weight matrix of the current layer.

```

bool CNeuronConv::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    //--- control block
    if(!prevLayer || !prevLayer.GetOutputs() || !prevLayer.GetGradients() ||
        !m_cGradients || !m_cWeights)
        return false;
}

```

After successfully passing the control block, we will adjust the error gradient by the derivative of the activation function of the current layer.

```

//--- adjusting error gradients to the derivative of the activation function
if(m_cActivation)
{
    if(!m_cActivation.Derivative(m_cGradients))
        return false;
}

```

Next comes the branching of the algorithm depending on the computing device used. We are currently looking at the MQL5 branch.

The backpropagation method is the mirror of the forward pass method. During the feed-forward pass, we first transfer the input data to a local matrix and then multiply it by the weight matrix. During the backpropagation pass, we will first reformat the error gradient matrix received from the previous layer into the required format and then multiply it by the weight matrix.

```

//--- branching of the algorithm depending on the execution device
CBufferType* input_gradient = prevLayer.GetGradients();
if(!m_cOpenCL)
{

```

4. Basic types of neural layers

```
MATRIX g = m_cGradients.m_mMatrix;
if(!g.Reshape(m_iWindowOut, m_iNeurons))
    return false;
g = g.Transpose();
g = g.MatMul(m_cWeights.m_mMatrix);
if(!g.Resize(m_iNeurons, m_iWindow))
    return false;
```

As a result of matrix multiplication, we obtain a matrix of gradients for the previous layer. However, the process becomes more complex due to the presence of the analyzed window and its step. If they are equal, we just need to reformat the matrix and copy its value to the buffer of the previous layer. But if the size of the analyzed window of the source data is not equal to its step, then we will need to organize a loop system for copying and summing gradients. Indeed, in this case, one neuron of the source data influences several neurons of the results of each filter.

```
if(m_iWindow == m_iStep && input_gradient.Total() == (m_iNeurons * m_iWindow))
{
    if(!g.Reshape(input_gradient.Rows(), input_gradient.Cols()))
        return false;
    input_gradient.m_mMatrix = g;
}
else
{
    input_gradient.m_mMatrix.Fill(0);
    ulong total = input_gradient.Total();
    for(ulong r = 0; r < m_iNeurons; r++)
    {
        ulong shift = r * m_iStep;
        for(ulong c = 0; c < m_iWindow; c++)
        {
            ulong k = shift + c;
            if(k >= total)
                break;
            if(!input_gradient.m_mMatrix.Flat(k,
                input_gradient.m_mMatrix.Flat(k) + g[r, c]))
                return false;
        }
    }
}
```

After completing the loop iterations, we exit the method with a positive result.

```
else
{
    //--- The multi-threaded calculation block will be added in the next chapter
    return false;
}
//--- Successful completion of the method
return true;
}
```

After distributing the gradient through the hidden layer, it's time to calculate the error gradient on the elements of the weight matrix. After all, it is the weights that we will select for optimal operation of the neural network. All the work on propagating the error gradient is necessary only to determine the direction and magnitude of the weight adjustments. This approach makes selecting the optimal weight matrix directed and controllable.

Work on distributing the error gradient over the elements of the weight matrix is implemented in the *CalcDeltaWeights* method. This method is also virtual and is overridden in each class. In the parameters, the method receives a pointer to the object of the previous layer. At the beginning of the method, we immediately check the correctness of the received pointer and the presence of operational data buffers in the current and previous neural layers. To calculate the gradient on the weight matrix, we will need a buffer for incoming gradients, a buffer for input data (results from the previous layer), and a buffer to store the obtained results (*m_cDeltaWeights*). Let me remind you that our algorithm includes gradient distribution at each iteration of the backward pass, and the weight matrix update is triggered by a request from an external program. Therefore, in the *m_cDeltaWeights* buffer, we will accumulate the error gradient value. During the update, we will divide the accumulated value by the number of completed iterations. Thus, we obtain the average error for each weight.

```
bool CNeuronConv::CalcDeltaWeights(CNeuronBase *prevLayer)
{
    //--- control block
    if(!prevLayer || !prevLayer.GetOutputs() || !m_cGradients || !m_cDeltaWeights)
        return false;
```

To simplify access to the data buffer of the previous layer, we will save the pointer to the object in a local variable.

Next, we divide the algorithm into two logical threads of operations depending on the computational device in use.

```
//--- branching of the algorithm depending on the execution device
CBufferType *input_data = prevLayer.GetOutputs();
if(!m_cOpenCL)
{
```

We will discuss the implementation of the OpenCL algorithm in the next chapter. Now we will focus on the implementation using MQL5.

We have a two-dimensional weight matrix, in which one dimension represents the filters of our layer. Each row in the weight matrix is a separate filter. Therefore, the number of rows in the weight matrix is equal to the number of filters used. The second dimension (columns) of the matrix represents the elements of our filter, and their number is equal to the size of the input window plus *bias*.

However, since the filter window moves across the input data array, each element of the filter affects the result of all elements in the vector of the current layer results. Therefore, for each filter element, we need to collect error gradients from all elements of the result vector, which are stored in the *m_cGradients* buffer. Vector operations will help us with this. But first, let me remind you that during the forward pass, we transformed the vector of the original data. Let's repeat this process.

```
MATRIX inp;
uint input_total = input_data.Total();
if(m_iWindow == m_iStep && input_total == (m_iNeurons * m_iWindow))
{
    inp = input_data.m_mMatrix;
```

```

        if(!inp.Reshape(m_iNeurons, m_iWindow))
            return false;
    }
    else
    {
        if(!inp.Init(m_iNeurons, m_iWindow))
            return false;
        for(ulong r = 0; r < m_iNeurons; r++)
        {
            ulong shift = r * m_iStep;
            for(ulong c = 0; c < m_iWindow; c++)
            {
                ulong k = shift + c;
                inp[r, c] = (k < input_total ? input_data.At((uint)k) : 0);
            }
        }
    }
    //--- add a bias column
    if(!inp.Resize(inp.Rows(), m_iWindow + 1) ||
        !inp.Col(VECTOR::Ones(m_iNeurons), m_iWindow))
        return false;

```

Next, we will directly collect error gradients for filter elements. Similar to the fully connected layer, the weight gradient in the convolutional layer is equal to the product of the neuron error gradient and the value of the corresponding element of the input data. In terms of matrix operations, all we need to do is multiply the gradient matrix before the activation function by the reformatted matrix of input data.

```

    MATRIX g = m_cGradients.m_mMatrix;
    if(!g.Reshape(m_iWindowOut, m_iNeurons))
        return false;
    i    m_cDeltaWeights.m_mMatrix += g.MatMul(inp);
}

```

We will add the obtained result to the previously accumulated error gradients in the *m_cDeltaWeights* matrix.

```

    else
    {
        //--- The multi-threaded calculation block will be added in the next chapter
        return false;
    }
    //--- Successful completion of the method
    return true;
}

```

We will become familiar with the algorithm for implementing multi-threaded computations in the next chapter, and at this stage, we exit the method with a positive result.

We've already discussed the weight update method earlier. We still need to create methods for working with files because we should have the ability to load and use a previously trained neural network. And here, we will also use the previously created groundwork. We have already created similar methods for two parent classes: the base class of the neural layer *CNeuronBase*, and the pooling layer *CNeuronProof*. Pooling layer methods are greatly simplified since it does not contain a matrix of weights

and objects for its training. Therefore, we will use the base class method and force it to be called from the *CNeuronConv::Save* method. This approach will help us eliminate unnecessary controls since they are already implemented in the parent class method. We just have to check the result of the method. But we need more than that because the pooling layer introduces new variables. Therefore, after executing the parent class method, we will add the missing parameters to the file.

```
bool CNeuronConv::Save(const int file_handle)
{
    //--- call the method of the parent class
    if(!CNeuronBase::Save(file_handle))
        return false;
    //--- save constant values
    if(FileWriteInteger(file_handle, (int)m_iWindow) <= 0)
        return false;
    if(FileWriteInteger(file_handle, (int)m_iStep) <= 0)
        return false;
    if(FileWriteInteger(file_handle, (int)m_iWindowOut) <= 0)
        return false;
    if(FileWriteInteger(file_handle, (int)m_iNeurons) <= 0)
        return false;
    if(FileWriteInteger(file_handle, (int)m_bTransposedOutput) <= 0)
        return false;
    //---
    return true;
}
```

The data loading is organized on the same principle. First, we need to read the data from the file in the same order in which it was written there. Hence, we will first call the method of the parent class. In it, all the controls are already implemented, and the sequence of data loading is observed. We only need to check the result returned by the parent class method, and after successful execution, read additional parameters from the file in the same sequence in which they were saved.

```
bool CNeuronConv::Load(const int file_handle)
{
    //--- calling the method of the parent class
    if(!CNeuronBase::Load(file_handle))
        return false;
    //--- reading the values of constants
    m_iWindow = (uint)FileReadInteger(file_handle);
    m_iStep = (uint)FileReadInteger(file_handle);
    m_iWindowOut = (uint)FileReadInteger(file_handle);
    m_iNeurons = (uint)FileReadInteger(file_handle);
    m_eActivation = -1;
    //---
    if(!m_cOutputs.Reshape(m_iWindowOut, m_iNeurons))
        return false;
    if(!m_cGradients.Reshape(m_iWindowOut, m_iNeurons))
        return false;
    //---
    return true;
}
```

In this section, we created two new types of neural layers: pooling and convolutional. In the next section, we will further enhance their functionality with the ability to use the *OpenCL* for organizing parallel computations using multi-threading technologies. Then, in the comparative testing block, we will assemble a small neural network and compare the performance of the new architectural solution with the previously obtained testing results of fully connected neural networks.

4.1.3 Organizing parallel computing in convolutional networks using OpenCL

In the previous section, we already created classes for two new types of neural layers. These are the convolutional and pooling layers. These types of layers are key in the architecture of convolutional neural networks. By alternating convolutional and pooling layers, we can create a model that searches for the key components of the desired object in the array of source data while simultaneously reducing the size of the processed information without sacrificing the overall performance of the model. This approach also helps filter out noise from the source data.

Reducing the information volume leads to a reduction in the cost of processing it. Furthermore, we can also parallelize computations in the convolutional and pooling layers using the technology of multi-threaded calculations in OpenCL. This will help reduce the time required for calculations while maintaining the overall operation volume, making the training and operation of the neural network much faster.

To organize multi-threaded operations using OpenCL, we need to perform two blocks of operations:

- Write additional kernels in the previously created OpenCL program ([opencl_program.cl](#)).
- Organize the process of interaction with the OpenCL context on the side of the main program.

Before organizing the transfer of data from the main program to the OpenCL context, it is necessary to understand when and what data will be needed. Therefore, we will begin our work by making changes to the OpenCL program.

Pooling layer

The creation of kernels in the OpenCL program and the construction of classes in the main program will start with the implementation of methods for the pooling layer. Feed-forward operations will be implemented in the *ProofFeedForward* kernel. We will transfer two data buffers from the main program to the kernel:

- *inputs*: a vector of input data
- *outputs*: a vector for writing results

To prevent an array out-of-bounds error, we will pass the size of the *inputs_total* initial data vector to the kernel in the parameters.

Let me remind you that in the convolutional neural networks algorithm, the pooling layer follows the convolutional layer of neurons. In turn, the convolutional layer includes several filters. Therefore, when receiving the results of the work of multiple filters from the convolutional layer in a single buffer, the pooling layer should process each filter separately. Therefore, to logically divide the common buffer of results of the convolutional layer by filters, the kernel will be given the size of the output vector of one filter *input_neurons*.

4. Basic types of neural layers

In the kernel parameters, we specify the window size for analyzing the initial data (*window*), the step for moving the window (*step*), the number of filters (*window_out*), and the activation function (*activation*).

```
__kernel void ProofFeedForward(__global double *inputs,
                               __global double *outputs,
                               int inputs_total,
                               int input_neurons,
                               int window,
                               int step,
                               int activation)
```

We will run this kernel in a two-dimensional task space. Thus, in each kernel, we will process one element of the results array in one filter. The number of the processed element will be determined by the thread identifier in dimension with index 0. Therefore, the total number of threads will tell us the number of elements in the output of one filter (*neurons*). Using this data, we will determine the offsets to the beginning of the window of analyzed data within the filter array of the initial data (*shift*).

```
{
    const int n = get_global_id(0);
    const int w = get_global_id(1);
    const int neurons = get_global_size(0);
    const int window_out = get_global_size(1);
    int shift = n * step;
```

The second dimension with index 1 will indicate the index of the analyzed filter. Accordingly, we will determine the shift in the arrays of initial data (*shift_inp*) and results (*out*) before the beginning of the processed filter. Don't forget to check for any out-of-range errors within the result array.

Let's prepare a variable to store intermediate values of the current element of the result vector (*s*).

```
int out = w * neurons + n;
int shift_inp = w * input_neurons;
TYPE s = 0;
TYPE k = (TYPE)1 / (TYPE>window;
TYPE4 k4 = (TYPE4)(k);
```

The values in the pooling layer will be computed in a nested array. In it, we will iterate through the elements of the input data that fall within the analyzed window and assemble the resulting value according to the activation formula.

Let me remind you that in our implementation, the pooling layer can receive one of two activation functions:

- Average pooling which involves taking the arithmetic mean of the elements within the input data window.
- Max pooling which involves selecting the maximum element within the input data window.

When calculating the arithmetic mean, we will not collect the sum of all elements and then divide by the size of the analyzed window. On the contrary, each element is first divided by the size of the window, and then the resulting quotients are summed up. This will allow us to get the final result in the body of the loop, eliminating the division operation behind the loop. The implementation of the division operation behind the loop is not critical, but only if it concerns any variants of operations in the loop. In our case, division is necessary only in the case of the arithmetic mean. When using *Max pooling*, the

4. Basic types of neural layers

division is redundant, and for correct operation, we would need an additional check of the activation function. By moving the division inside the loop, we eliminate the need for an additional check for the activation function and only apply it when calculating the actual value.

Please note that we use vector operations with *TYPE4* data type to speed up the process. Consequently, the step of the loop through the elements of the window is equal to four.

```
for(int i = 0; i < window; i += 4)
    switch(activation)
    {
        case 0:
            s += dot(ToVect4(inputs, i, 1, min(shift_inp+input_neurons,inputs_total),
                shift_inp + shift), k4);
            break;
        case 1:
            s = Max4(ToVect4(inputs, i, 1, min(shift_inp+input_neurons,inputs_total),
                shift_inp + shift), s);
            break;
        default:
            break;
    }
    outputs[out] = s;
}
```

After exiting the loop that iterates over the elements of the analyzed window, we will save the obtained value into the corresponding element of the result vector and exit the kernel.

We have examined the feed-forward kernel and can now proceed to build the algorithm for the backpropagation pass. As discussed earlier in the context of building the algorithm using MQL5, in the pooling layer, the backpropagation pass algorithm involves simply propagating the error gradient through the hidden layer. Therefore, the process of constructing the backpropagation pass will consist of writing the *ProofCalcHiddenGradient* gradient propagation kernel algorithm.

The new kernel will communicate with the external program through four data buffers:

- *inputs*: buffer for the results of the preceding layer
- *gradient_inputs*: buffer for the gradients of the preceding layer (in this case, it is used to record the results of the kernel operation)
- *outputs*: buffer for the results of the forward pass of the current layer
- *gradients*: buffer for the gradients at the results level of the current layer

Buffer size control will be organized using the *inputs_total* and *outputs_total* parameters. The names of the parameters correspond to the buffers whose sizes they store.

It is important to note that, unlike a fully connected layer, neurons in the pooling layer have limited connections to neurons in the previous layer. We will define connection zones using the *window* and *step* parameters. You can see that parameters of the same name were declared in the forward pass kernel. We have also retained their functional significance.

Let's add parameters for the number of elements per filter output and the activation function being used.

```
__kernel void ProofCalcHiddenGradient(__global TYPE *inputs,
                                     __global TYPE *gradient_inputs,
```



```

__global__ TYPE *outputs,
__global__ TYPE *gradients,
int inputs_total,
int outputs_total,
int window,
int step,
int neurons,
int activation)

```

When organizing multi-threaded computations, it's important to consider the issue of concurrent attempts to write to the same buffer elements from different threads. Therefore, the most suitable algorithms are those in which each thread is provided with its own objects for writing data, and these objects do not intersect with objects being written to by other threads.

Following the logic mentioned above, we will create an algorithm in which each thread will collect gradients and write them to a separate element of the gradient buffer of the previous layer. It should be noted that one difference in this approach compared to the one we adopted in the MQL5 implementation is as follows. When using the *Max pooling* activation function, if there are two or more elements with values equal to the maximum, the gradient will be fully transferred to all such elements. In contrast, in the implementation of the main program, we passed the gradient to only one element. Considering the use of variables and their precision, we assess the risk of encountering such a situation as minimal and accept it.

At the beginning of the kernel body, let's determine the ordinal number of the required element and the filter by stream identifiers. The total number of threads will give us the number of elements of one filter in the input data buffer (*input_neurons*) and the number of filters (*window_out*). Based on this data, we determine the first (*start*) and last (*stop*) elements of the resulting vector, which are affected by the processed element. When defining the influence zone, we need to keep in mind the limitations of the data buffer dimension for each filter. Therefore, the first element cannot be less than 0, and the last element cannot be greater than the number of elements in one filter (*neurons*).

```

{
    const int n = get_global_id(0);
    const int w = get_global_id(1);
    const int input_neurons = get_global_size(0);
    const int window_out = get_global_size(1);
    //---
    int start = n - window + step;
    start = max((start - start % step) / step, 0);
    int stop = min((n - n % step) / step + 1, neurons);

```

Next, we determine the offset of the analyzed element in the common initial data buffer. At the same time, do not forget to check for going beyond the array of initial data.

After that, we will prepare the necessary internal variables. First of all, this is a variable for collecting intermediate values of the gradient (*grad*) and the value of the current element in the source data buffer (*inp*).

The creation of the last condition is because when using *Max pooling*, we will need to constantly compare the value of an element in the source data with the value from the results buffer. For technical reasons, accessing internal variables is much faster than accessing elements of the global array buffer. This is related to the storage location of the data. Internal variables are stored in private memory, while buffers are stored in global memory. The size of the private memory is small, and we cannot copy

the entire array there, but accessing it takes minimal time. The size of the global memory is much larger, but the access time to it is significantly longer. To reduce the overall running time of the program, we will move a frequently used value from the global to the private memory of the OpenCL context.

```
TYPE grad = 0;
int shift_inp = w * input_neurons + n;
if(shift_inp >= inputs_total)
    return;
TYPE inp = inputs[shift_inp];
```

Next, we will organize a nested loop in which we will iterate over the elements that fall within the influence zone of the analyzed element of the input data. Inside the loop, we will first determine the offset of the processed element in the gradient error buffer. We will immediately check if the error gradients array falls within the boundaries. Then we will transfer the gradient in accordance with the activation function used.

For *Average pooling*, we simply divide the value of the error gradient by the size of the input data window and add the resulting value to the accumulated error gradient of the analyzed source data element. Please note that we will divide the error gradient by the size of the input data window, and not by the zone of influence. Indeed, the error obtained during the feed-forward pass is influenced by all the elements of the input data that affect the specific value.

In the case of *Max pooling*, we will first compare the value of the corresponding elements at the output and input of the neural layer. Only if they match will we transmit the error gradient in full.

After exiting the loop, we will save the computed gradient value in the gradient error buffer of the previous layer and conclude the execution of the kernel.

```
for(int o = start; o < stop; o ++){
    {
        int shift_g = w * neurons + o;
        if(shift_g >= outputs_total)
            break;
        switch(activation)
        {
            case 0:
                grad += gradients[shift_g] / (TYPE>window;
                break;
            case 1:
                grad += (outputs[shift_g] == inp ? gradients[shift_g] : 0);
                break;
            default:
                break;
        }
    }
    gradient_inputs[shift_inp] = grad;
}
```

The above two kernels cover the forward and backward pass processes in the pooling layer. Now we can move on to working with the convolutional layer.

Convolutional layer

Convolutional layer For the convolutional layer, we also have to implement forward and backward pass algorithms. Similarly to the kernels discussed earlier, the forward pass algorithm will be described in the *ConvolutionFeedForward* kernel. A convolutional layer, like a fully connected one, has a weight matrix and an activation function. Therefore, to communicate with the main program, we need four data buffers:

- *inputs*: input data buffer
- *weights*: matrix of weights
- *sums*: vector of weighted sums of the original data before the activation function
- *outputs*: vector of results

In addition to buffers, for the proper functioning of the new kernel, the following parameters will be required:

- *inputs_total*: size of the input data array
- *window*: size of the analyzed window of the source data
- *step*: step of the source data window
- *window_out*: number of filters in the layer

```
__kernel void ConvolutionFeedForward(__global TYPE *inputs,
                                     __global TYPE *weights,
                                     __global TYPE *outputs,
                                     int inputs_total,
                                     int window,
                                     int step,
                                     int window_out)
```

Building the algorithm of the kernel itself is similar to constructing a similar kernel for a fully connected neuron. Just like in the fully connected layer, the number of threads will be tied to the number of elements in the output buffer. However, considering the specific nature of the convolutional layer's operation, we will not be guided by the total number of elements in the buffer, but by the number of elements in the results buffer of a single filter. In this case, the results of the n -th element of all filters will be calculated in one thread.

At the beginning of the kernel, we will carry out preparatory work. We will determine the index of the processed element in the filter results buffer based on the thread number. The total number of threads will give us the number of elements in the output of each filter. From the obtained data and information from the kernel parameters, we will calculate the offset to the beginning of the analyzed window in the source data buffer and the size of the weight matrix being used.

```
{
    const int n = get_global_id(0);
    const int neurons = get_global_size(0);
    const int weights_total = (window + 1) * window_out;
    int shift = n * step;
```

Since we decided to process all the filters sequentially in one thread, the next thing we do is organize a filter iteration loop. Inside the loop, we determine the offset to the processed element in the general result buffer and the offset in the weight matrix. At this point, we will also check for any out-of-bounds

4. Basic types of neural layers

access to the weight matrix and prepare an internal variable for collecting the resulting value. We will initialize the variable with the *bias* element.

```
for(int w = 0; w < window_out; w++)
{
    int out = (transposed_out == 1 ? w + n * window_out : w * neurons + n);
    int shift_weights = w * (window + 1) ;
    if((shift_weights + window) >= weights_total)
        break;
    TYPE s = weights[shift_weights + window];
```

We will directly calculate the weighted sum of the analyzed input data window in a nested loop. Inside this loop, we will iterate through the elements of the analyzed window of input data and multiply them by the corresponding weight. To reduce the time spent on execution, we use vector operations. At the same time, do not forget to increase the size of the cycle step to the size of the used vector variables.

```
    for(int i = 0; i < window; i += 4)
        s += dot(ToVect4(inputs, i, 1, inputs_total, shift),
                ToVect4(weights, i, 1, shift_weights + window, shift_weights));
    outputs[out] = s;
}
```

After collecting the weighted sum, we write the resulting value to the result buffer.

Next, we move on to creating kernels for the backward pass process. Unlike the pooling layer, the convolutional layer contains a weight matrix. Therefore, we will need to create more than one kernel, as in a similar process of a fully connected layer.

We will start building the process as before, following the algorithm of the backpropagation pass. We will fully apply the adjustments of the gradient based on the derivative of the activation function, just as we did for the fully connected layer. Let's start working on the convolutional layer by creating a gradient propagation kernel through the *ConvolutionCalcHiddenGradient* layer.

In this case, propagating the gradient to the lower layer does not depend on the input data and the results of the forward pass. Therefore, for our kernel to work, we will give it three data buffers:

- *gradient_inputs*: buffer for the error gradients of the preceding layer (in this case, the result buffer)
- *weights*: weight matrix
- *gradients*: buffer for the error gradients at the input of the current layer

In addition to data buffers, a number of parameters are required for the correct operation of the kernel:

- *outputs_total*: total number of elements in the result buffer (gradients at the output of the current neural layer);
- *window*: size of the input data window (the number of input data elements analyzed by one neuron of the current layer);
- *step*: step of moving the window along the array of initial data;
- *window_out*: number of filters in the current convolutional layer;
- *neurons*: number of elements at the output of one filter.

```
__kernel void ConvolutionCalcHiddenGradient(__global TYPE *gradient_inputs,
```

```

__global__ TYPE *weights,
__global__ TYPE *gradients,
int window,
int step,
int window_out,
int neurons)

```

The kernel will be launched in a multi-threaded mode with the number of threads equal to the number of elements in the gradient error buffer of the previous layer, which is also equal to the number of elements in the input data buffer.

As usual, at the beginning of the kernel, we determine the ordinal number of the element being processed by the number of the current thread and the number of elements in the gradient buffer of the previous layer by the total number of running threads. Additionally, we calculate the size of the weight matrix based on the size of the input data window and the number of filters in the current convolutional layer.

```

{
    const int n = get_global_id(0);
    const int inputs_total = get_global_size(0);
    int weights_total = (window + 1) * window_out;

```

Continuing the preparatory work, let's determine the zone of influence of the current element in the result buffer of one filter and prepare an internal variable to record the intermediate results of the accumulation of the error gradient for the processed element.

```

    TYPE grad = 0;
    int w_start = n % step;
    int r_start = max((n - window + step) / step, 0);
    int total = (window - w_start + step - 1) / step;
    total = min((n + step) / step, total);

```

Let me remind you that when creating the convolution layer class in the main program, we decided to consider the array of initial data as a single whole and apply all filters to the total amount of data. Therefore, each element of the input data affects the results of all filters. This means that we have to collect the error gradient on each element of the initial data from all filters. Therefore, to collect error gradients, we need a system of nested loops with iteration of filters and elements of each filter.

The outer loop iterates over the elements of the error gradient vector at the output of the current neural layer. In it, we will determine the offset to a specific element in the gradient vector and immediately check for going beyond the filter size.

```

for(int i = 0; i < total; i++)
{
    int row = r_start + i;
    if(row >= neurons)
        break;

```

In the body of the nested loop, we will first determine the offset in the gradient buffer of the error at the output of the current layer and the weight matrix. Then, we will add the product of the values of these elements to the previously accumulated error gradient for the analyzed element of the original data.

```

    for(int wo = 0; wo < window_out; wo++)
    {

```

4. Basic types of neural layers

```
int shift_g = (transposed_out == 1 ? row * window_out + wo :  
               row + wo * neurons);  
int shift_w = w_start + (total - i - 1) * step + wo * (window + 1);  
grad += gradients[shift_g] * weights[shift_w];  
}  
}  
gradient_inputs[n] = grad;  
}
```

After completion of all iterations and exiting from the block of two nested loops, the value of the accumulated gradient is stored in the error gradient buffer of the previous layer.

The distribution of the error gradient through the hidden layers of the neural network, in accordance with the algorithm of the error backward pass method, is followed by the transfer of the error gradient to weights. To perform this functionality, we create the *ConvolutionCalcDeltaWeights* kernel.

For the correct operation of the kernel, the use of 3 data buffers will be required:

- *inputs*: input data buffer
- *delta_weights*: buffer for the accumulated error gradients of the weight matrix (in this case, the results buffer)
- *gradients*: buffer for the error gradients of the current layer (at the results level)

The gradient buffer contains the values of the error gradients already corrected for the derivative of the activation function. This procedure is performed before passing the error gradient to the previous layer. Therefore, adjusting for the derivative of the activation function at this stage will be unnecessary.

In addition to the data buffers, we need to introduce a few parameters in order to build the algorithm correctly:

- *inputs_total*: total number of elements in the result buffer and, respectively, the error gradient buffer
- *step*: step of moving the analyzed data window along the source data array
- *neurons*: number of elements at the output of one filter

```
__kernel void ConvolutionCalcDeltaWeights(__global TYPE *inputs,  
                                         __global TYPE *delta_weights,  
                                         __global TYPE *gradients,  
                                         int inputs_total,  
                                         int step,  
                                         int neurons)
```

It can be noticed that among the parameters, there are no variables to indicate the size of the window for the analyzed data and the number of filters in the current convolutional layer. This is due to a change in the approach to creating threads for operations. This kernel will collect error gradients at the level of the weight matrix, so it is quite logical to run the kernel for each weight. Furthermore, the weight matrix is represented as a two-dimensional table, where each row corresponds to a separate filter, and the elements within each row are the weights of the corresponding filter.

The OpenCL technology allows threads to be launched in two-dimensional space, with two indices for each thread. Let's use this property and create threads for this kernel in two dimensions. In the first dimension, the number of threads will be equal to the number of weights in one filter. In the second dimension, the number of threads will correspond to the number of filters used.

In the body of the kernel, we will determine the position of the analyzed element in the weight matrix and its dimensions. It should be recalled here that each filter has a *bias* weight, so the size of the analyzed data window will be one element less than the number of threads in the first dimension (the dimension with index 0).

Right there, we will determine the position of the analyzed element in the one-dimensional buffer of the weight matrix and the offset to the beginning of the corresponding filter in the error gradient buffer. And of course, let's prepare a variable to store intermediate values of the accumulated error gradient.

```
{
    const int inp_w = get_global_id(0);
    const int w = get_global_id(1);
    const int window = get_global_size(0) - 1;
    const int window_out = get_global_size(1);
    //---
    int shift_delt = w * (window + 1) + inp_w;
    TYPE value = 0;
```

Next comes the process of directly calculating the error gradient. Here we must remember that for the *bias* element, there are no corresponding elements in the source data buffer. Therefore, the gradient will be transferred to this element in full. In order not to check at each iteration of the loop, we will do it once before starting the loop. In the loop, we will iterate through the elements of the error gradient buffer and the original data, while the element of the weight matrix remains unchanged.

Thus, first, we check whether the current element of the weight matrix is a *bias*, and then we organize a loop to iterate through all the error gradient elements of the corresponding filter. Inside the loop, we will sum up the error gradient adjusted for the corresponding value of the initial data buffer.

After exiting the loop, add the obtained value to the previously accumulated error gradient for the analyzed element of the weight matrix. Let me remind you that we will not update the weight matrix at each iteration of the backward pass. We only accumulate the error gradient. The weight matrix is updated by a command from the main program after processing the data package installed by the user.

```
if(inp_w == window)
{
    for(int n = 0; n < neurons; n++)
        value += gradients[w * neurons + n];
}
else
    for(int n = 0; n < neurons; n++)
    {
        int shift_inp = n * step + inp_w;
        if(shift_inp >= inputs_total)
            break;
        value += inputs[shift_inp] * gradients[w * neurons + n];
    }
    delta_weights[shift_delt] += value;
}
```

After distributing the error gradient to the weight matrix through the backpropagation algorithm, its update is provided. The weights are adjusted towards the anti-gradient. As mentioned before while creating the convolutional layer using MQL5, the previously established process for the fully connected

layer fully meets the requirements for working with convolutional layers as well. Therefore, we will not create separate kernels and blocks of the main program but will use the previously created solution.

Implementing functionality on the side of the main program

After supplementing the OpenCL program with new kernels, we have to embed code blocks into the main program to organize the process of data exchange and launch kernels for execution at the right time and with the right amount of information. Let's take a closer look at how this can be implemented.

As a reminder, when building a fully connected neural layer, we started [similar work](#) by declaring constants. Now we will do the same: we will declare constants for calling each kernel.

```
#define def_k_ProofFeedForward      21
#define def_k_ProofHiddenGradients  22
#define def_k_ConvolutionFeedForward 23
#define def_k_ConvolutionHiddenGradients 24
#define def_k_ConvolutionDeltaWeights 25
```

We will also declare parameter constants for each kernel. The constants of the parameters must strictly correspond to the ordinal number of the parameter in the *OpenCL* program kernel. Parameter numbering starts from zero.

```
//--- feed-forward pass of the pooling layer
#define def_prff_inputs      0
#define def_prff_outputs     1
#define def_prff_inputs_total 2
#define def_prff_input_neurons 3
#define def_prff_window      4
#define def_prff_step        5
#define def_prff_activation  6

//--- gradient distribution through the pooling layer
#define def_prhgr_inputs      0
#define def_prhgr_gradient_inputs 1
#define def_prhgr_outputs     2
#define def_prhgr_gradients   3
#define def_prhgr_inputs_total 4
#define def_prhgr_outputs_total 5
#define def_prhgr_window      6
#define def_prhgr_step        7
#define def_prhgr_neurons     8
#define def_prff_activation  9

//--- feed-forward pass of the convolutional layer
#define def_cff_inputs      0
#define def_cff_weights     1
#define def_cff_outputs     2
#define def_cff_inputs_total 3
#define def_cff_window      4
#define def_cff_step        5
#define def_cff_window_out  6
```


4. Basic types of neural layers

```
//--- gradient distribution through the convolutional layer
#define def_convhgr_gradient_inputs    0
#define def_convhgr_weights           1
#define def_convhgr_gradients         2
#define def_convhgr_window            3
#define def_convhgr_step              4
#define def_convhgr_window_out        5
#define def_convhgr_neurons           6

//--- distribution of the gradient to the weight matrix of the convolutional layer
#define def_convdelt_inputs            0
#define def_convdelt_delta_weights    1
#define def_convdelt_gradients        2
#define def_convdelt_inputs_total     3
#define def_convdelt_step             4
#define def_convdelt_neurons          5
```

After declaring the constants, we need to update the list of used kernels from the OpenCL program. Let me remind you that this work is carried out in the [CNet::InitOpenCL](#) method. Here we need to change the number of used kernels to 26.

```
if(!m_cOpenCL.SetKernelsCount(26))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}
```

Let's create entry points for new kernels.

```
if(!m_cOpenCL.KernelCreate(def_k_ProofFeedForward, "ProofFeedForward"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}

if(!m_cOpenCL.KernelCreate(def_k_ProofHiddenGradients,
                           "ProofCalcHiddenGradient"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}

if(!m_cOpenCL.KernelCreate(def_k_ConvolutionFeedForward,
                           "ConvolutionFeedForward"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}

if(!m_cOpenCL.KernelCreate(def_k_ConvolutionHiddenGradients,
```

```

        "ConvolutionCalcHiddenGradient"))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        return false;
    }

    if(!m_cOpenCL.KernelCreate(def_k_ConvolutionDeltaWeights,
        "ConcolutionCalcDeltaWeights"))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        return false;
    }

```

Further work will continue directly in the relevant methods. Remember that during the construction of classes, we implemented branching in many methods depending on the device used for executing operations. We have already written the MQL5 part. Now we will describe the algorithm for working with the OpenCL context.

We will supplement the methods in the same sequence in which we created them earlier. Let's start this work with the feed-forward method of the pooling layer *CNeuronProof::FeedForward*. To work correctly, this method uses two data buffers: initial data and results. At the beginning of the block, check for the presence of the specified buffers in the OpenCL context. The presence of a buffer handle will indicate a previously passed buffer to the OpenCL context.

```

bool CNeuronProof::FeedForward(CNeuronBase *prevLayer)
{
    //--- Control block
    if(!prevLayer || !m_cOutputs ||
        !prevLayer.GetOutputs())
        return false;
    CBufferType *input_data = prevLayer.GetOutputs();
    //--- Algorithm branching depending on the operating device
    if(!m_cOpenCL)
    {
        // The MQL5 block is missing here
    }
    else // Block of operations with OpenCL
    {
        //--- check the availability of buffers in the OpenCL context
        if(input_data.GetIndex() < 0)
            return false;
        if(m_cOutputs.GetIndex() < 0)
            return false;

```

If there is data in the *OpenCL* context, we will pass pointers to the data buffers and parameters necessary for its operation to the kernel. At each step, we also check the result of the operations. This is crucial because launching a kernel with incomplete information can lead to a critical error and the halt of the entire program.

```

        //--- Send parameters to the kernel
        if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofFeedForward, def_prff_inputs,

```

```

                                                                    input_data.GetIndex()))
    return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofFeedForward, def_prff_outputs,
                                                                    m_cOutputs.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_ProofFeedForward, def_prff_inputs_total,
                                                                    input_data.Total()))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_ProofFeedForward, def_prff_window,
                                                                    m_iWindow))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_ProofFeedForward, def_prff_step, m_iStep))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_ProofFeedForward, def_prff_activation,
                                                                    (int)m_eActivation))
        return false;
    uint input_neurons = (input_data.Total()+m_iWindowOut-1) / m_iWindowOut;
    if(!m_cOpenCL.SetArgument(def_k_ProofFeedForward, def_prff_input_neurons,
                                                                    input_neurons))
        return false;

```

Once all the necessary information is passed to the kernel, we then need to specify the number of threads for kernel execution and the initial offset in the task space. After that, we initiate the execution of the kernel and complete the method.

```

    //--- Queuing up the kernel for execution
    uint off_set[] = {0, 0};
    uint NDRange[] = {m_iNeurons, m_iWindowOut};
    if(!m_cOpenCL.Execute(def_k_ProofFeedForward, 2, off_set, NDRange))
        return false;
}
//---
return true;
}

```

After adding the code for the *CNeuronProof::FeedForward* forward pass method of the pooling layer, let's do the same work in the *CNeuronProof::CalcHiddenGradient* backward pass method. Unlike the forward pass, the error gradient distribution kernel through the pooling layer uses four data buffers:

- initial data
- feed-forward results
- error gradients at the output of the neural layer
- error gradients at the source data level (the result buffer in this case).

The first two buffers are used to determine which elements to employ when using *Max pooling*.

Therefore, we have to load all four buffers into the memory of the *OpenCL* context.

```

bool CNeuronProof::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    //--- Control block
    if(!prevLayer || !m_cOutputs ||
        !m_cGradients || !prevLayer.GetOutputs() ||

```

```

        !prevLayer.GetGradients())
    return false;
CBufferType *input_data = prevLayer.GetOutputs();
CBufferType *input_gradient = prevLayer.GetGradients();
if(!input_gradient.BufferInit(input_data.Rows(), input_data.Cols(), 0))
    return false;
//--- Algorithm branching depending on the operating device
if(!m_cOpenCL)
{
    // The MQL5 block is missing here
}
else // Block of operations with OpenCL
{
    //--- check for buffers in the OpenCL context
    if(input_data.GetIndex() < 0)
        return false;
    if(m_cOutputs.GetIndex() < 0)
        return false;
    if(input_gradient.GetIndex() < 0)
        return false;
    if(m_cGradients.GetIndex() < 0)
        return false;

```

If there is data in the memory of the *OpenCL* context, we will pass pointers to buffers and necessary constants to the kernel parameters. At the same time, do not forget to control the results of the operations.

```

//--- Send parameters to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofHiddenGradients,
                                def_prhgr_inputs, input_data.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofHiddenGradients,
                                def_prhgr_outputs, m_cOutputs.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofHiddenGradients,
                                def_prhgr_gradients, m_cGradients.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofHiddenGradients,
                                def_prhgr_gradient_inputs, input_gradient.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                           def_prhgr_inputs_total, input_data.Total()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                           def_prhgr_window, m_iWindow))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                           def_prhgr_step, m_iStep))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                           def_prhgr_activation, (int)m_eActivation))
    return false;

```

4. Basic types of neural layers

```
if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                          def_prhgr_neurons, m_iNeurons))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                          def_prhgr_outputs_total, m_cOutputs.Total()))
    return false;
```

Then we specify the number of threads to run the kernel and the offset in the task area. After that, we will put the kernel in the execution queue.

Please note that when launching the forward pass kernel, the number of threads is equal to the number of elements at the output of one filter in the pooling layer. When running a backward pass kernel, the number of threads is equal to the number of elements in one filter of the previous neural layer.

```
//--- Queuing up the kernel for execution
uint input_neurons = (input_data.Total() + m_iWindowOut - 1) / m_iWindowOut;
uint off_set[] = {0, 0};
uint NDRange[] = {input_neurons, m_iWindowOut};
if(!m_cOpenCL.Execute(def_k_ProofHiddenGradients, 2, off_set, NDRange))
    return false;
}
//---
return true;
}
```

This completes the work with the pooling layer class. We move on to do a similar job with the *CNeuronConv* convolutional layer class.

The convolutional neural layer, unlike the pooling layer, has a weight matrix and an activation function. Therefore, it will require the use of more buffers for its operation. The *CNeuronConv::FeedForward* forward pass method of the convolutional layer requires transferring 4 buffers to the *OpenCL* context memory:

- initial data
- weight matrix
- additional activation function buffer (used for *Swish* activation function)
- results buffer

Let's start working in the *CNeuronConv::FeedForward* forward pass method by checking the availability of buffers in use in the context of *OpenCL*.

```
bool CNeuronConv::FeedForward(CNeuronBase *prevLayer)
{
//--- control block
if(!prevLayer || !m_cOutputs || !m_cWeights || !prevLayer.GetOutputs())
    return false;
CBufferType *input_data = prevLayer.GetOutputs();
ulong total = input_data.Total();
//--- algorithm branching depending on the operating device
if(!m_cOpenCL)
{
    // The MQL5 block is missing here
}
```

```

else
{
    //--- checking data buffers
    if(input_data.GetIndex() < 0)
        return false;
    if(m_cWeights.GetIndex() < 0)
        return false;
    if(m_cOutputs.GetIndex() < 0)
        return false;
}

```

Then we need to pass buffer pointers to the corresponding kernel. In addition, in the kernel parameters, we will pass some constants necessary for the correct operation of the algorithm. Among the passed parameters will be the size of the analyzed window, the window step and the number of filters. At each step, we control the process of performing operations.

```

//--- pass arguments to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionFeedForward,
                                def_cff_inputs, input_data.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionFeedForward,
                                def_cff_weights, m_cWeights.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionFeedForward,
                                def_cff_outputs, m_cOutputs.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ConvolutionFeedForward,
                           def_cff_inputs_total, input_data.Total()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ConvolutionFeedForward,
                           def_cff_window, m_iWindow))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ConvolutionFeedForward,
                           def_cff_step, m_iStep))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ConvolutionFeedForward,
                           def_cff_window_out, m_iWindowOut))
    return false;

```

After passing all the necessary data to the kernel, we specify the number of threads to start and initiate its queuing.

```

//--- put the kernel in the execution queue
int off_set[] = {0};
int NDRange[] = {(int)m_iNeurons};
if(!m_cOpenCL.Execute(def_k_ConvolutionFeedForward, 1, off_set, NDRange))
    return false;
}
if(!m_cActivation.Activation(m_cOutputs))
    return false;
//---
return true;
}

```

Finally, we call the activation function and exit the method.

That's all for the feed-forward pass. Let's proceed to the backpropagation pass in the convolutional neural layer. As you remember, the backpropagation pass includes three sub-processes:

- Distributing the error gradient over the neural network from the result to the initial data.
- Distributing the error gradient to the weight matrix of each neural layer.
- Adjusting the weight matrix towards the anti-gradient.

From the methods already implemented using MQL5, we know that no new method was created for the last sub-process. Instead, it is suggested to use the ready-made method of the fully connected neural layer, where we have already implemented multi-threaded computations using OpenCL tools. Therefore, at this stage, we have to refine only the methods of the first two sub-processes.

The *CNeuronConv::CalcHiddenGradient* method is responsible for distributing the error gradient across the convolutional layer. Correct execution of the algorithm of this method requires the presence of three data buffers:

- Buffer for error gradients at the output of the neural layer (obtained from the next layer in the process of executing a similar method).
- Weight matrix.
- Buffer for error gradients at the input data level (in this case, it acts as a buffer for the results of the method).

Therefore, at the beginning of the block of work with the OpenCL technology, we check the presence of the necessary buffers in the context memory.

```
bool CNeuronConv::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    //--- control block
    if(!prevLayer || !prevLayer.GetOutputs() || !prevLayer.GetGradients() ||
        !m_cGradients || !m_cWeights)
        return false;
    //--- adjust error gradients to the derivative of the activation function
    if(m_cActivation)
    {
        if(!m_cActivation.Derivative(m_cGradients))
            return false;
    }
    //--- algorithm branching depending on the operating device
    CBufferType* input_gradient = prevLayer.GetGradients();
    if(!m_cOpenCL)
    {
        //The MQL5 block is missing here
    }

    else // Block for working with OpenCL
    {
        //--- checking data buffers
        if(m_cWeights.GetIndex() < 0)
            return false;
        if(input_gradient.GetIndex() < 0)
            return false;
    }
}
```

4. Basic types of neural layers

```
if(m_cGradients.GetIndex() < 0)
    return false;
```

The next step is to pass the necessary data to the kernel parameters. Among them are pointers to the data buffers used.

```
//--- pass arguments to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionHiddenGradients,
                                def_convhgr_gradient_inputs, input_gradient.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionHiddenGradients,
                                def_convhgr_weights, m_cWeights.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionHiddenGradients,
                                def_convhgr_gradients, m_cGradients.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ConvolutionHiddenGradients,
                           def_convhgr_neurons, m_iNeurons))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ConvolutionHiddenGradients,
                           def_convhgr_window, m_iWindow))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ConvolutionHiddenGradients,
                           def_convhgr_step, m_iStep))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ConvolutionHiddenGradients,
                           def_convhgr_window_out, m_iWindowOut))
    return false;
```

Next, we will specify the number of threads equal to the number of elements in the source data buffer and enqueue the kernel for execution.

```
//--- put the kernel in the execution queue
int NDRange[] = {(int)input_gradient.Total()};
int off_set[] = {0};
if(!m_cOpenCL.Execute(def_k_ConvolutionHiddenGradients, 1, off_set, NDRange))
    return false;
}
//---
return true;
}
```

To complete the work on the backpropagation pass methods in the convolutional network, we need to make similar changes to the method for distributing the error gradient to the weight matrix *CNeuronConv::CalcDeltaWeights*, taking into account the specifics of this method.

The algorithm of the error gradient distribution method to the weight matrix requires the presence of three buffers:

- Error gradient at the output level of the neural layer.
- Initial data buffer.
- Buffer for accumulating error gradients at the weight matrix level.

Let's check the presence of the specified buffers in the memory of the OpenCL context. Let me remind you that we proceed from the assumption that there is enough video memory to store the entire model. If the model does not completely fit in the memory of your OpenCL device, then you will need to load the necessary data into the context memory before launching each kernel. After the completion of the kernel, free up memory to load the next batch of data.

```
bool CNeuronConv::CalcDeltaWeights(CNeuronBase *prevLayer)
{
    //--- control block
    if(!prevLayer || !prevLayer.GetOutputs() || !m_cGradients || !m_cDeltaWeights)
        return false;
    //--- algorithm branching depending on the operating device
    CBufferType *input_data = prevLayer.GetOutputs();
    if(!m_cOpenCL)
    {
        // The MQL5 block is missing here
    }
    else // Block for working with OpenCL
    {
        //--- checking data buffers
        if(m_cGradients.GetIndex() < 0)
            return false;
        if(m_cDeltaWeights.GetIndex() < 0)
            return false;
        if(input_data.GetIndex() < 0)
            return false;
    }
}
```

Then we pass the necessary parameters to the kernel corresponding to our sub-process. Let me remind you that it is very important to observe the correspondence of the specified kernel ID, parameter ID and the specified value, and we also control the process of performing operations at each step.

```
//--- pass arguments to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionDeltaWeights,
                                def_convdel_t_delta_weights, m_cDeltaWeights.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionDeltaWeights,
                                def_convdel_t_inputs, input_data.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionDeltaWeights,
                                def_convdel_t_gradients, m_cGradients.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ConvolutionDeltaWeights,
                           def_convdel_t_inputs_total, input_data.Total()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ConvolutionDeltaWeights,
                           def_convdel_t_neurons, m_iNeurons))
    return false;
if(!m_cOpenCL.SetArgument(def_k_ConvolutionDeltaWeights,
                           def_convdel_t_step, m_iStep))
    return false;
```

When all the necessary information is transferred to the kernel, we specify the number of threads. In this case, we decided to use a two-dimensional thread distribution:

- by the number of filters
- by the number of weights in one filter

To do this, we specify two parameters in the *NDRange* array. Each parameter specifies the size of the corresponding task area. We send the kernel to the execution queue.

```

    ///--- put the kernel in the execution queue
    uint NDRange[] = {m_iWindow + 1, m_iWindowOut};
    uint off_set[] = {0, 0};
    if(!m_cOpenCL.Execute(def_k_ConvolutionDeltaWeights, 2, off_set, NDRange))
        return false;
}
///---
return true;
}

```

Now we have already created three types of fully functional neural layers for our neural network builder and can compare their effectiveness in solving a practical problem. I suggest doing some experiments in the next chapter. But before proceeding to the "field tests", we still have to check the correctness of the methods for transferring gradients.

4.1.4 Implementing a convolutional model in Python

We will implement the convolutional models in the Python language using the tools provided by the *Keras* library from *TensorFlow*. This library offers several options for convolutional layers. First of all, these are the basic versions of convolutional layers:

- *Conv1D*
- *Conv2D*
- *Conv3D*

From the names of the objects representing convolutional layers, it can be inferred that they are intended for processing input data of various dimensions.

The *Conv1D* class objects create the core of the convolution that collapses with the original data in one dimension to create an output tensor. It is important to understand and not to get confused. The initial data is convoluted in one dimension, but the initial data supplied to the neural layer input must be in the form of a three-dimensional tensor. The first dimension determines the size of the package of the data (*batch size*) being processed. The second is measuring convolution. The third dimension contains the initial data for convolution.

As a result of data processing, the layer also returns a 3D tensor. The first dimension remains the same; it is equal to the size of the data package being processed. The second dimension varies depending on the specified convolution parameters. The third dimension will be equal to the specified number of filters used.

It should be understood that each filter applies to all initial data. At one time, the initial data is processed in the size of the third dimension multiplied by the size of the convolution window. This is a slight difference from our implementation of the convolutional layer in MQL5. There, we defined the

convolution window as the number of elements, while here, the convolution window determines the number of elements in the second dimension of the three-dimensional tensor of input data.

One filter returns one value for each convolution window. Since the entire third dimension is involved in the convolution process, we get one element from each filter. As a result, the size of the third dimension of the output tensor changes by the number of filters used.

Like a fully connected layer, the convolutional layer class offers a fairly wide range of parameters for fine-tuning the operation. Let's take a look at them.

- **filters** – the number of filters used in the bundle.
- **kernel_size** – one-dimensional convolution window size.
- **strides** – the size of the convolution step.
- **padding** – one of the following values is allowed: "valid", "same" or "causal" (case-insensitive); "valid" means no indentation; "same" causes the input data to be evenly filled with zeros to obtain an output size equal to the input size; "causal" leads to the emergence of causal (extended) changes, for example, *output* [t] does not depend from *input* [t+1:]. It's useful when modeling temporal data, where the model must not violate the temporal order.
- **data_format** – one of the following values is allowed: "channels_last" or "channels_first"; determines which dimension of the input tensor contains data for convolution; the default is "channels_last".
- **dilation_rate** – used for advanced convolution and determines the expansion rate.
- **groups** – the number of groups into which the input is divided along the channel axis; each group is collapsed separately using filters, and the output is a combination of all results along the channel axis.
- **activation** – activation function.
- **use_bias** – indicates whether to use a bias vector.
- **kernel_initializer** – sets a method for initializing the weight matrix.
- **bias_initializer** – sets the method for initializing the displacement vector.
- **kernel_regularizer** – indicates a method for regularizing the weight matrix.
- **bias_regularizer** – indicates a method for regularizing the displacement vector.
- **activity_regularizer** – indicates a method for regularizing results.
- **kernel_constraint** – specifies the restriction function for the weight matrix.
- **bias_constraint** – specifies the constraint function for the displacement vector.

For timeseries, it is usually suggested to use a one-dimensional Conv1D convolution. Convolution is carried out by time intervals. At the same time, each filter checks for its own pattern in a specific time interval. In relation to solving our problem, filters will assess the status of all indicators used within the number of candles specified by the *strides* parameter. The process of convolution, like the neurons of a fully connected layer, does not assess the mutual influence of individual components of the initial data. It only assesses the similarity of the initial data with a given pattern. Of course, we don't explicitly define these patterns when constructing the neural network. We select them during the training process. However, it is assumed that during practical application, these patterns will remain static between retraining periods.

True, convolutional layers are more resistant to various distortions of the initial data due to the fact that small individual blocks are studied meticulously. However, it may be necessary to study the

patterns of individual indicators. To solve this problem, we may need to use convolutional layers of a different dimension.

For example, Conv2D objects operate with convolutions with input data in two dimensions. At the same time, it should be understood that the difference between one-dimensional and two-dimensional convolutional layers goes beyond just their names. Objects in a two-dimensional convolutional layer expect a four-dimensional tensor at the input. By analogy with the *Conv1D* tensor, the first dimension determines the batch size, the second and third dimensions determine the convolution dimensions and the fourth dimension contains the initial data for convolution. Here arises a valid question: where do we obtain the data for another dimension? How do we divide our initial data set into four dimensions? We need to translate our raw data from a flat table to a voluminous table. The simplest solution is on the surface. We say that the depth of the table of the initial data is 1. Before declaring the two-dimensional neural layer, let's change the dimensionality of the tensor input to the Conv2D convolutional layer to a four-dimensional one by specifying a size of 1 for the fourth dimension.

Note that since the fourth dimension is 1, the length of the input data vector for convolution is 1. Therefore, for the convolution process to be effective, the convolution window needs to be greater than 1 in at least one dimension.

We will not dwell too much on the parameters of the Conv2D convolutional layer, since they are identical to the parameters of a one-dimensional array. The only differences are in the *kernel_size*, *strides* and *dilation_rate* parameters, which, in addition to a scalar value, can take a vector of two elements. Each element of such a vector contains parameter values for the corresponding dimension. At the same time, these parameters can take scalar values. In this case, the specified value will be used for both dimensions.

For more complex architectural solutions for neural networks, it may be necessary to use Conv3D 3D convolutional layers. Their usage can be justified, for example, in building arbitrage trading systems, where a separate dimension might be needed to segregate input data by instruments.

Just like in the case of a two-dimensional convolutional layer, using three-dimensional space requires increasing the dimensionality of the input data. A five-dimensional tensor is expected at the Conv3D input.

The parameters of the Conv3D class, however, are inherited from the aforementioned classes with minimal changes. The only difference is in the size of the vectors of the convolution window and its pitch.

Attention should be paid to another feature of the convolution process. When performing operations, it is possible to both reduce the size of the data tensor (data compression) and increase it. The first approach is useful when dealing with large datasets, where it's necessary to extract a specific component from the overall volume of input information. This is frequently employed in computer vision tasks, where in high-resolution images, each pixel represents an individual value within the overall tensor of input data.

The second approach, increasing the dimensionality, can be beneficial when there's an insufficient amount of input data. In such cases, a small volume of input data needs to be split into separate components while searching for non-obvious dependencies.

It should be noted that this is not a complete list of convolutional layers offered by the Keras library. But it is beyond the scope of this book to describe all the library's features. You can always check them out on the library [website](#). There you can also find the latest version of the library and instructions for installing and using it.

Just like convolutional layers, the Keras library offers several class options for a pooling layer. Among them are:

- **AvgPool1D** – one-dimensional data averaging.
- **AvgPool2D** – two-dimensional data averaging.
- **AvgPool3D** – three-dimensional data averaging.
- **MaxPool1D** – one-dimensional extraction of the maximum value.
- **MaxPool2D** – two-dimensional extraction of the maximum value.
- **MaxPool3D** – three-dimensional extraction of the maximum value.

All of these pooling layers have the same set of parameters:

- *pool_size* – an integer number or vector of integers, determines the window size.
- *strides* – an integer number or vector of integers, determines the window pitch.
- *padding* – means one of the following values is allowed: "valid", "same" or "causal" (case-insensitive); "valid" means no indentation; "same" – causes the source data to be evenly filled with zeros to obtain an output size equal to the input size.
- *data_format* – one of the following values is allowed: "channels_last" or "channels_first"; determines which dimension of the input tensor contains data for convolution; the default is "channels_last".

We will also implement convolutional neural network models in our [template](#). Just like when testing perceptron models, we will create three neural network models with different architectures and compare the results of their training. Therefore, for implementation, we will take the previously created file [perceptron.py](#) and create a copy of it called *convolution.py*. In this created file, we will replace the model declaration blocks.

First, we will create a perceptron with three hidden layers and weight matrix regularization. It will serve as a basis for comparing the performance of convolutional neural networks to the results of training a fully connected perceptron.

```
# Create a perceptron model with three hidden layers and regularization
model1 = keras.Sequential([keras.Input(shape=inputs),
                            keras.layers.Dense(40, activation=tf.nn.swish,
                            kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                            keras.layers.Dense(40, activation=tf.nn.swish,
                            kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                            keras.layers.Dense(40, activation=tf.nn.swish,
                            kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                            keras.layers.Dense(targets, activation=tf.nn.tanh)
                            ])
```

This model has 9802 parameters. The screenshot below shows the structure of the neural network we created. In the first column of the table, the name and type of the neural layer are indicated, while in the second column, the tensor dimensionality of the results for each layer is specified. Note that the first dimension is not set; *None* is specified instead of the size. This means that this dimension is not strictly defined and can be of variable length. This dimension is set by the *batch size* of the data patch. The third column shows the number of parameters in the weight matrix for each layer.

In the second model, we will insert a one-dimensional *Conv1D* convolution layer with 8 filters immediately after the initial data, and specify the convolution window and step as 1. Such a layer will

roll up all specified indicators within a single candlestick. In doing so, let's not forget to change the dimensionality of the input data tensor from two-dimensional to three-dimensional.

Note that although we're transferring data to a 3D tensor, we specify two dimensions in the *Reshape* layer parameters. This is due to the fact that the first dimension of the tensor is variable and is set by the *batch size* of the input data batch.

Layer (type)	Output Shape	Param #
=====		
dense (Dense)	(None, 40)	6440
dense_1 (Dense)	(None, 40)	1640
dense_2 (Dense)	(None, 40)	1640
dense_3 (Dense)	(None, 2)	82
=====		
Total params: 9,802		

Perceptron structure

And one more thing. The dimensional vector passed in the parameters of the *Reshape* class contains `-1` in the first dimension. This tells the class to independently calculate the size of this dimension based on the size of the original data tensor and the specified dimensions of other dimensions.

```
# Add a 1D convolutional layer to the model
model2 = keras.Sequential([keras.Input(shape=inputs),
                           # Reformat the tensor into three-dimensional.
                           # We indicate 2 dimensions, because The 3rd dimension is determined by the packet
                           keras.layers.Reshape((-1,4)),
                           # Convolutional layer with 8 filters
                           keras.layers.Conv1D(8,1,1,activation=tf.nn.swish,
                           kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
```

Behind the convolutional layer, we will place a one-dimensional subsample layer with a choice of the maximum *MaxPool1D* value. As mentioned above, the convolutional layer operates with three-dimensional tensors. At the same time, the subsequent fully connected layers work with two-dimensional tensors. Therefore, for the proper functioning of fully connected layers, we need to return the data to a two-dimensional dimensionality. To do this, we will use the neural layer of the *Flatten* class.

```
# Pooling layer
keras.layers.MaxPooling1D(2, strides=1),
# Reformat the tensor into a two-dimensional one for fully connecte
keras.layers.Flatten(),
keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
keras.layers.Dense(40, activation=tf.nn.swish,
```

4. Basic types of neural layers

```
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
keras.layers.Dense(targets, activation=tf.nn.tanh)
])
```

Note: In the initial data, each candlestick is described by four values. The use of eight filters increases the dimensionality of the processed tensor. As a result, the model with a one-dimensional convolutional layer already contains 15,922 parameters.

In the third model, we will replace a 1 one-dimensional convolution layer with a two-dimensional one. As a result, we will change the pooling layer and the data dimension. As mentioned above, we will set the fourth dimension to 1. We can now control the size of the convolution window in two dimensions: time and indicator. Since we would like to assess different patterns in the readings of each individual indicator, we will specify the size of the convolution window in the first temporal dimension as 3 (evaluating patterns from 3 consecutive candlesticks), and the size of the window in the second dimension of indicators as 1. This will allow us to identify patterns in the movement of each indicator separately. The pitch of the convolution window in both directions will be set to 1.

Layer (type)	Output Shape	Param #
=====		
reshape (Reshape)	(None, 40, 4)	0
conv1d (Conv1D)	(None, 40, 8)	40
max_pooling1d (MaxPooling1D)	(None, 39, 8)	0
)		
flatten (Flatten)	(None, 312)	0
dense_4 (Dense)	(None, 40)	12520
dense_5 (Dense)	(None, 40)	1640
dense_6 (Dense)	(None, 40)	1640
dense_7 (Dense)	(None, 2)	82
=====		
Total params: 15,922		

Neural network structure with a one-dimensional convolutional layer

With these parameters, the first dimension (time dimension) will decrease by two elements as a result of the convolution operations. The second dimension (dimension of indicators) will remain unchanged since the convolution window and its pitch in this dimension are 1. At the same time, we will increase the third dimension, and it will become equal to the number of filters. Let me remind you that before the convolution operation, the third dimension was equal to 1. As a result of all iterations, the number of network parameters increased to 50,794. The structure of the new neural network is presented below. As you can see, the convolution layer has only 32 parameters. Such an increase in the number

of network parameters is due to the enlargement of the tensor size after the convolution operation for the reasons mentioned above. This can be seen from the number of parameters in the first fully connected layer after convolution.

```
# Replace the convolutional layer in the model with a two-dimensional one
model3 = keras.Sequential([keras.Input(shape=inputs),
    # Reformat the tensor into a four-dimensional one.
    #We indicate 3 dimensions, because... The 4th dimension is determined by the
    keras.layers.Reshape((-1,4,1)),
    # Convolutional layer with 8 filters
    keras.layers.Conv2D(8,(3,1),1,activation=tf.nn.swish,
    kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    # Pooling layer
    keras.layers.MaxPooling2D((2,1),strides=1),
    # Reformat the tensor into a two-dimensional one for fully connec
    keras.layers.Flatten(),
    keras.layers.Dense(40, activation=tf.nn.swish,
    kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(40, activation=tf.nn.swish,
    kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(40, activation=tf.nn.swish,
    kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(targets, activation=tf.nn.tanh)
])
```


Layer (type)	Output Shape	Param #
=====		
reshape_1 (Reshape)	(None, 40, 4, 1)	0
conv2d (Conv2D)	(None, 38, 4, 8)	32
max_pooling2d (MaxPooling2D)	(None, 37, 4, 8)	0
flatten_1 (Flatten)	(None, 1184)	0
dense_8 (Dense)	(None, 40)	47400
dense_9 (Dense)	(None, 40)	1640
dense_10 (Dense)	(None, 40)	1640
dense_11 (Dense)	(None, 2)	82
=====		
Total params: 50,794		

Neural network structure with a two-dimensional convolutional layer

The rest of our script will remain unchanged. We will learn more about the script results in the next section.

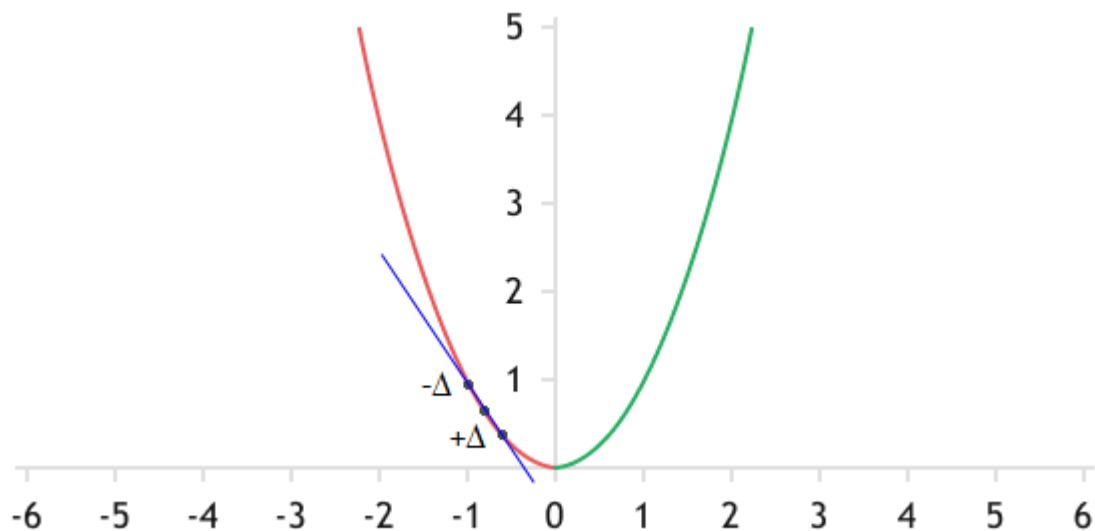
4.1.5 Practical testing of convolutional models

Look at the amount of work we have already completed. This is something to be proud of. We have implemented three types of neural layers, which already allow us to solve some practical problems. Using those, we can create fully connected perceptrons of different complexity. Or we can create convolutional neural network models and compare the performance of the two models on the same set of source data.

Before assessing the practical capabilities of different neural network models, you should verify the correctness of the methods for error gradient propagation through the convolutional neural network. We have already performed such a procedure for fully connected neural layers in the section [“Checking the correctness of the gradient distribution”](#).

Let me remind you of the essence of the procedure. The error gradient is a number that determines the slope of the tangent line to the function graph at the current point. It demonstrates how the value of a function will change when the parameter changes.

4. Basic types of neural layers



In geometry terms, the gradient is the slope of the tangent to the graph of the function at the current point

Of course, we are dealing with non-linear functions, and analytically computed error gradients provide only an approximate value. But when using a sufficiently small parameter change step, such an error becomes minimal.

Moreover, we can always determine how the function value changes when we experimentally alter a single parameter: we can take our function, change only one parameter, and calculate the new value. The difference between the two values of the function will show the influence of the analyzed parameter on the overall result at the current point.

Certainly, as the number of parameters increases, so do the costs of evaluating the influence of each parameter on the overall result. Therefore, neglecting a small error, everyone uses the analytical method to determine the error gradient. At the same time, by using the experimental method, we can assess the accuracy of our implemented analytical algorithm and adjust its operation if necessary.

When comparing the results of analytical and experimental methods for determining the error gradients, one point should be taken into account. To draw a straight line in a plane, two points are required. But if we draw a straight line through the current and new point, then such a straight line will not be tangent to the graph of the function at the current point. Most likely it will be tangent at some point between the current and future position. Therefore, to construct a tangent to the graph of a function at the current point, you will need to increase and decrease the current value of the indicator by the same small amount and calculate the function's value at both points. Then the line will touch the function at the point we need, and the effect of the parameter on the value of the function will be its average between two deviations.

When analyzing error gradient deviations between methods in a fully connected layer, we created the script [check_gradient_percp.mq5](#). Let's make a copy of the script named [check_gradient_conv.mq5](#). In the resulting copy, we will only change the *CreateNet* function. In it, after the input data layer, we will add one convolutional layer and one pooling layer.

```
bool CreateNet(CNet &net)
{
    CArrayObj *layers = new CArrayObj();
    if(!layers)
    {
```

4. Basic types of neural layers

```
        PrintFormat("Error creating CArrayObj: %d", GetLastError());
        return false;
    }
//--- The layer of source data
CLayerDescription *descr = new CLayerDescription();
if(!descr)
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    delete layers;
    return false;
}
descr.type = defNeuronBase;
int prev_count = descr.count = BarsToLine;
descr.window = 0;
descr.activation = AF_NONE;
descr.optimization = None;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete layers;
    delete descr;
    return false;
}
```

The convolutional layer will consist of two filters. The size of the convolution window is equal to two, its step is set to one. The activation function is *Swish*. The optimization method doesn't matter, as at this stage we won't be training the neural network. The size of one filter is recalculated based on the size of the previous layer and the convolution parameters.

```
//--- Convolutional layer
descr = new CLayerDescription();
if(!descr)
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    delete layers;
    return false;
}
descr.type = defNeuronConv;
int m_iWindow = descr.window = 2;
int prev_wind_out = descr.window_out = 2;
int m_iStep = descr.step = 1;
prev_count=descr.count=(prev_count-descr.window+2*descr.step-1)/descr.step;
descr.activation = AF_SWISH;
descr.optimization = Adam;
descr.activation_params[0] = 1;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete layers;
    delete descr;
    return false;
}
```

After the convolutional neural layer, we place the pooling layer. For it, we will specify the windows equal to two and the step of one. We will specify the activation function *AF_AVERAGE_POOLING*, which corresponds to the calculation of the average value for each source data window.

```
//--- Pooling layer
descr = new CLayerDescription();
if(!descr)
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    delete layers;
    return false;
}
descr.type = defNeuronProof;
descr.window = 2;
descr.window_out = prev_wind_out;
descr.step = 1;
descr.count = (prev_count - descr.window + 2 * descr.step - 1) / descr.step;
descr.activation = (ENUM_ACTIVATION_FUNCTION)AF_AVERAGE_POOLING;
descr.optimization = None;
descr.activation_params[0] = 1;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete layers;
    delete descr;
    return false;
}
```

Further, the script code remains unchanged.

We launch the prepared script in two modes: using *OpenCL* technology and without it. As a result of testing, we obtained quite decent results. In both cases, we received deviations in the 11th decimal place.

Use OpenCL true
Delta at input gradient between methods 1.48910e-11
Delta at weights gradient between methods -1.51489e-11
Use OpenCL false
Delta at input gradient between methods 3.91087e-11
Delta at weights gradient between methods -1.36456e-12

The results of checking deviations of the error gradient between the analytical and experimental methods of determination

Now that we are confident in the correctness of our neural layer classes, we can proceed to the construction and training of convolutional neural networks. First, we need to decide how we want to use convolution.

In our dataset, each candlestick is represented by several indicators. In particular, when creating a **training sample** for each candlestick, we identified four indicators:

- RSI

- MACD histogram
- MACD signal line
- Deviation between the signal line and MACD histogram

Each of you can conduct a series of tests yourselves and determine your approach to using convolutional models. To me, the most obvious are two use cases.

1. We can use convolution to determine certain patterns from indicator values at the level of each candlestick. In this version, we define the number of patterns to be searched for as the number of convolution filters. At the output of the convolutional layer, we get the degree of similarity of each candlestick with the desired patterns.
2. It should be remembered that a fully connected neural layer is a linear function. Only the activation feature adds non-linearity. Therefore, in general, neurons do not evaluate dependencies between elements of the source data, but instead learn to recognize patterns from the set of source data. Hence, each neuron evaluates the current pattern independently of past data.

But when analyzing time series, sometimes the dynamics of the change in an indicator is more important than its absolute value. We can use convolution to determine patterns in the dynamics of indicators. To do this, we need to slightly rearrange the indicators, lining up the values of each indicator in a row. For example, we can start by arranging all the RSI indicator values in the source data buffer. Then, we can place all the elements of the MACD histogram sequence, followed by the data of the signal line. We will complete the buffer with deviation data between the signal line and the MACD histogram. Of course, it would be more visual to arrange the data in a tabular form, where each row would represent the values of a separate indicator. But, unfortunately, only one-dimensional buffers are used in the OpenCL context. Therefore, we will use virtual partitioning of the buffer into blocks.

After arranging each indicator into a separate row, we can use convolution to identify patterns in sequential values of a single indicator. By doing so, we are essentially identifying trends within the analyzed data window. The number of convolutional layer filters will determine the number of trends to be recognized by the model.

Testing convolution within a single candlestick

To test the operation of convolutional neural network models, let's create a copy of the `perceptron_test.mq5` script [perceptron_test.mq5](#) with the name `convolution_test.mq5`. At the beginning of the script, as before, we specify the parameters for script operation.

As with checking the correctness of the gradient distribution, we only need to change the function for describing the architecture of the `CreateLayersDesc` model. In it, after the layer of initial data, we add convolutional and pooling neural layers.

```
bool CreateLayersDesc(CArrayObj &layers)
{
    CLayerDescription *descr;
    //-- create source data layer
    if(!(descr = new CLayerDescription()))
    {
        PrintFormat("Error creating CLayerDescription: %d", GetLastError());
        return false;
    }
    descr.type          = defNeuronBase;
    descr.count         = NeuronsToBar * BarsToLine;
    descr.window        = 0;
```

4. Basic types of neural layers

```
descr.activation = AF_NONE;
descr.optimization = None;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    return false;
}
```

Please note that for the convolutional layer, in the *count* field of the layer description object, we indicate not the total number of neurons, but the number of elements in one filter. In the *window_out* field, we will specify the number of filters to use. In the *window* and *step* fields, we will specify the number of elements per bar. With these parameters, we will obtain non-overlapping convolution, and each filter will compare the state of indicators at each bar with a certain pattern. The activation function is set to *Swish*, and the optimization method is set to *Adam*. We will use this optimization method for all subsequent layers. Except, of course, the pooling, which does not contain a weight matrix.

```
//--- Convolutional layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
descr.type = defNeuronConv;
descr.count = BarsToLine;
descr.window = NeuronsToBar;
descr.window_out = 8;
descr.step = NeuronsToBar;
descr.activation = AF_SWISH;
descr.optimization = Adam;
descr.activation_params[0] = 1;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

The convolutional layer is followed by the pooling layer. In this implementation, I used *Max Pooling*, i.e., selecting the maximum element within the input window. We are using a sliding window of two elements with a step of one element. With this set of parameters, the number of elements in one filter will decrease by one. We do not use the activation function for this layer. The number of filters is equal to the same parameter of the previous layer.

```
//--- Sub-sample layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
descr.type = defNeuronProof;
descr.count = BarsToLine - 1;
descr.window = 2;
```

4. Basic types of neural layers

```
descr.window_out = 8;
descr.step = 1;
descr.activation = (ENUM_ACTIVATION_FUNCTION)AF_MAX_POOLING;
descr.optimization = None;
descr.activation_params[0] = 0;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

Next comes an array of hidden fully connected layers. We will create them in a loop with the same parameters. The number of hidden layers to be created is specified in the script parameters. All hidden layers will have the same number of elements, which is specified in the script parameters. We will use the activation function *Swish*, and the weight matrix parameter optimization method, *Adam*, as we did for the convolutional layer.

```
//--- Block of hidden fully connected layers
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
descr.type = defNeuronBase;
descr.count = HiddenLayer;
descr.activation = AF_SWISH;
descr.optimization = Adam;
descr.activation_params[0] = 1;
for(int i = 0; i < HiddenLayers; i++)
    if(!layers.Add(descr))
    {
        PrintFormat("Error adding layer: %d", GetLastError());
        return false;
    }
```

At the end of the neural network initialization function, we will specify the parameters of the output layer. It will, just like in the previously created perceptron models, contain two elements with a linear activation function. We will use the *Adam* optimization method which is used by all other neural layers.

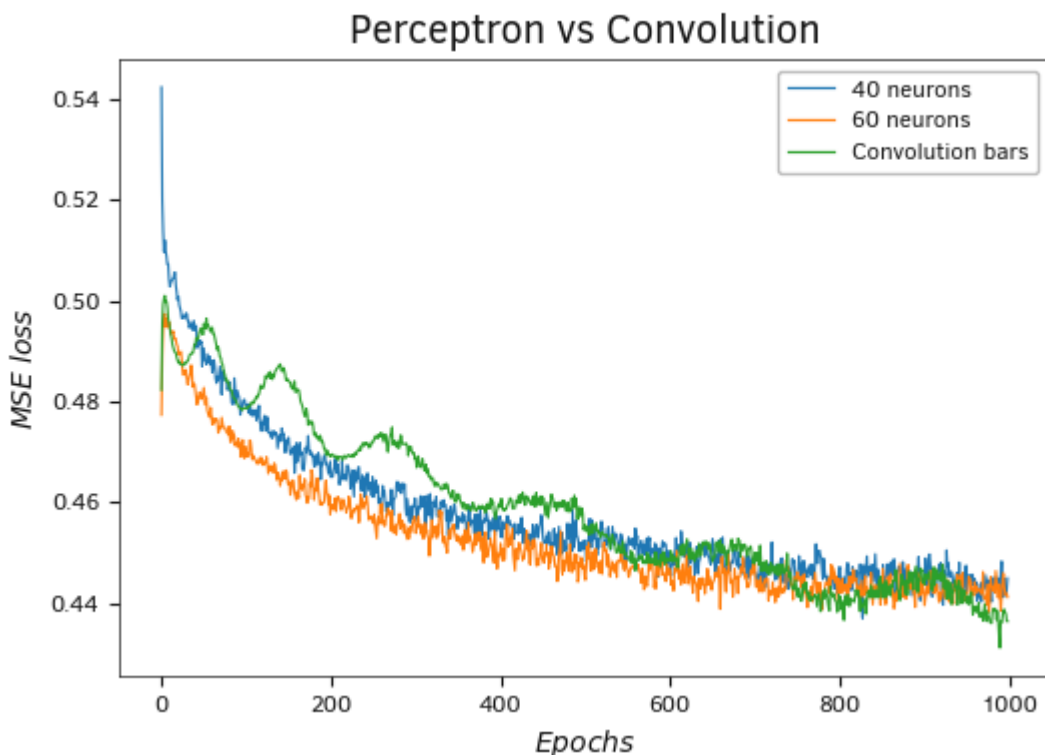
```
//--- Results Layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
descr.type = defNeuronBase;
descr.count = 2;
descr.activation = AF_LINEAR;
descr.optimization = Adam;
descr.activation_params[0] = 1;
if(!layers.Add(descr))
{
```

4. Basic types of neural layers

```
PrintFormat("Error adding layer: %d", GetLastError());  
return false;  
}  
return true;  
}
```

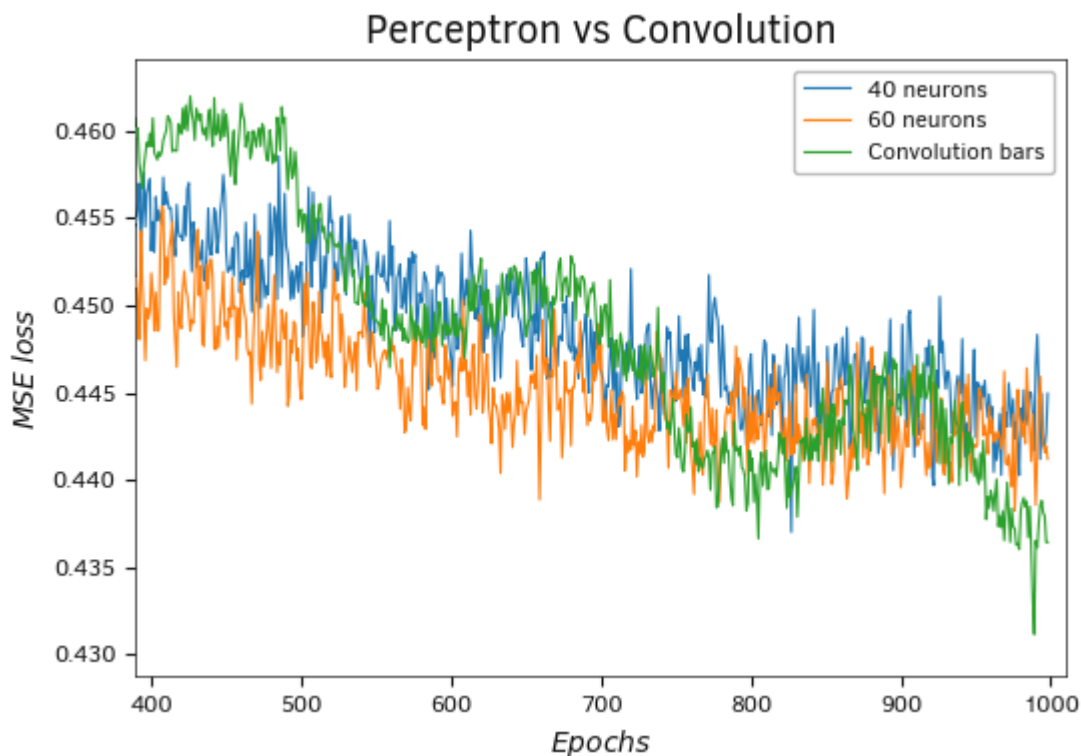
The rest of the script code remains unchanged.

According to the testing results, the convolutional neural network model exhibited a less smooth graph. On it, we observe a wave-like decrease in the value of the error function. But at the same time, after 1000 iterations of updating the weight matrix, we got a lower value of the loss function.



A comparative graph of the training progress of the perceptron and the convolutional neural network.

As the scale increases, we can also notice a tendency for the value of the loss function to potentially decrease with continued learning.



A comparative graph of the training progress of the perceptron and the convolutional neural network.

Testing of sliding window convolution by indicator values

Testing convolution with a sliding window on indicator values To experiment with finding patterns in the dynamics of indicator values, we need to make some modifications to the previously created script *convolution_test.mq5*. Let's create its copy with the name *convolution_test2.mq5*. We will make the first changes to the declaration of the convolutional layer. This time, we are creating a layer with a convolution window of three elements and a step of one element. With these parameters, the number of elements in one filter will be two less than the previous layer, but the total number of elements in the output buffer will increase by a factor equal to the number of filters used. The activation function and the optimization method remain unchanged.

```
//--- Convolutional layer
int prev_count = descr.count;
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
descr.type = defNeuronConv;
prev_count = descr.count = prev_count - 2;
descr.window = 3;
descr.window_out = 8;
descr.step = 1;
descr.activation = AF_SWISH;
descr.optimization = Adam;
descr.activation_params[0] = 1;
```

4. Basic types of neural layers

```
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

In the pooling layer, the changes affected only the number of elements in one filter.

```
//--- Pooling layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
descr.type = defNeuronProof;
descr.count = prev_count - 1;
descr.window = 2;
descr.window_out = 8;
descr.step = 1;
descr.activation = (ENUM_ACTIVATION_FUNCTION)AF_MAX_POOLING;
descr.optimization = None;
descr.activation_params[0] = 0;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

As mentioned before, for this test, we need to modify the sequence of source data being fed into the neural network. Therefore, we needed to make changes to the function of loading the training sample from the *LoadTrainingData* file.

As before, at the beginning of the function, we perform preparatory work. We declare instances of the necessary local objects and open the training dataset file for reading. The file name and path are specified in the function parameters. Let me remind you that the file with the training sample must be located within the sandbox of your terminal.

The result of the file opening procedure is checked by the received handle.

```
bool LoadTrainingData(string path, CArrayObj &data, CArrayObj &result)
{
    CBufferType *pattern;
    CBufferType *target;
    //--- open the file with the training sample
    int handle = FileOpen(path, FILE_READ | FILE_CSV | FILE_ANSI | FILE_SHARE_READ,
                                                                    ",", CP_UTF8);

    if(handle == INVALID_HANDLE)
    {
        PrintFormat("Error opening study data file: %d", GetLastError());
        return false;
    }
}
```

4. Basic types of neural layers

```
//--- display the progress of training data loading in the chart comment
uint next_comment_time = 0;
uint OutputTimeout = 250; // no more than once in 250 milliseconds
```

After successfully opening the training sample file for reading, we start the loop of direct data loading. We will repeat the loop iterations until the file is finished. During each iteration, we will check if a command to close the program has been received before proceeding.

Inside the loop body, we will first prepare new instances of objects for loading patterns and target results.

```
//--- organize the loop to load training sample
while(!FileIsEnding(handle) && !IsStopped())
{
    if(!(pattern = new CBufferType()))
    {
        PrintFormat("Error creating Pattern data array: %d", GetLastError());
        return false;
    }
    if(!pattern.BufferInit(NeuronsToBar, BarsToLine))
    {
        delete pattern;
        return false;
    }
    if(!(target = new CBufferType()))
    {
        PrintFormat("Error creating Pattern Target array: %d", GetLastError());
        delete pattern;
        return false;
    }
    if(!target.BufferInit(1, 2))
    {
        delete pattern;
        delete target;
        return false;
    }
}
```

We still use dynamic arrays to load data:

- **data**: array of source data patterns
- **result**: an array of patterns of target values for each pattern
- **pattern**: a buffer of elements of one pattern
- **target**: a buffer of target values of one pattern

However, to change the sequence of loaded data, we will first adjust the size of the *pattern* buffer matrix so that the first columns of the matrix correspond to the number of used indicators, and the rows correspond to the number of analyzed historical bars.

We create a system of nested loops. The outer loop has the number of iterations equal to the number of analyzed candles. The number of iterations in the inner loop is equal to the number of elements per candlestick. In the body of this looping system, we will write the initial data to the buffer matrix

pattern. Since the data in the training sample file is in chronological order, we will write them in the same order. But as we read, we will distribute the information in the corresponding rows and columns of the matrix.

```
for(int i = 0; i < BarsToLine; i++)
    for(int y = 0; y < NeuronsToBar; y++)
        pattern.m_mMatrix[y, i] = (TYPE)FileReadNumber(handle);
```

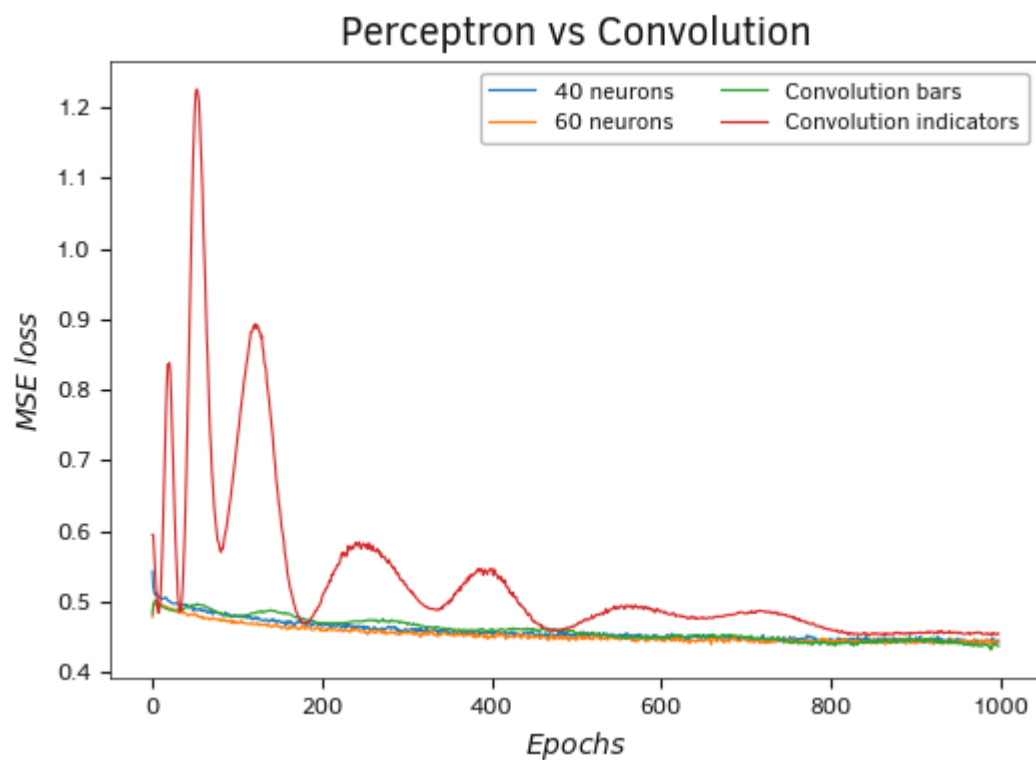
After completing the iterations of the loop system, we only need to reformat the resulting matrix.

```
if(!pattern.Reshape(1, BarsToLine * NeuronsToBar))
{
    delete pattern;
    delete target;
    return false;
}
```

The further process of loading the training sample has been moved without changes.

```
for(int i = 0; i < 2; i++)
    target.m_mMatrix[0, i] = (TYPE)FileReadNumber(handle);
if(!data.Add(pattern))
{
    PrintFormat("Error adding study data to array: %d", GetLastError());
    delete pattern;
    delete target;
    return false;
}
if(!result.Add(target))
{
    PrintFormat("Error adding study data to array: %d", GetLastError());
    delete target;
    return false;
}
//--- display loading progress in a comment on the chart (no more than 1 time i
if(next_comment_time < GetTickCount())
{
    Comment(StringFormat("Patterns loaded: %d", data.Total()));
    next_comment_time = GetTickCount() + OutputTimeout;
}
}
FileClose(handle);
return true;
}
```

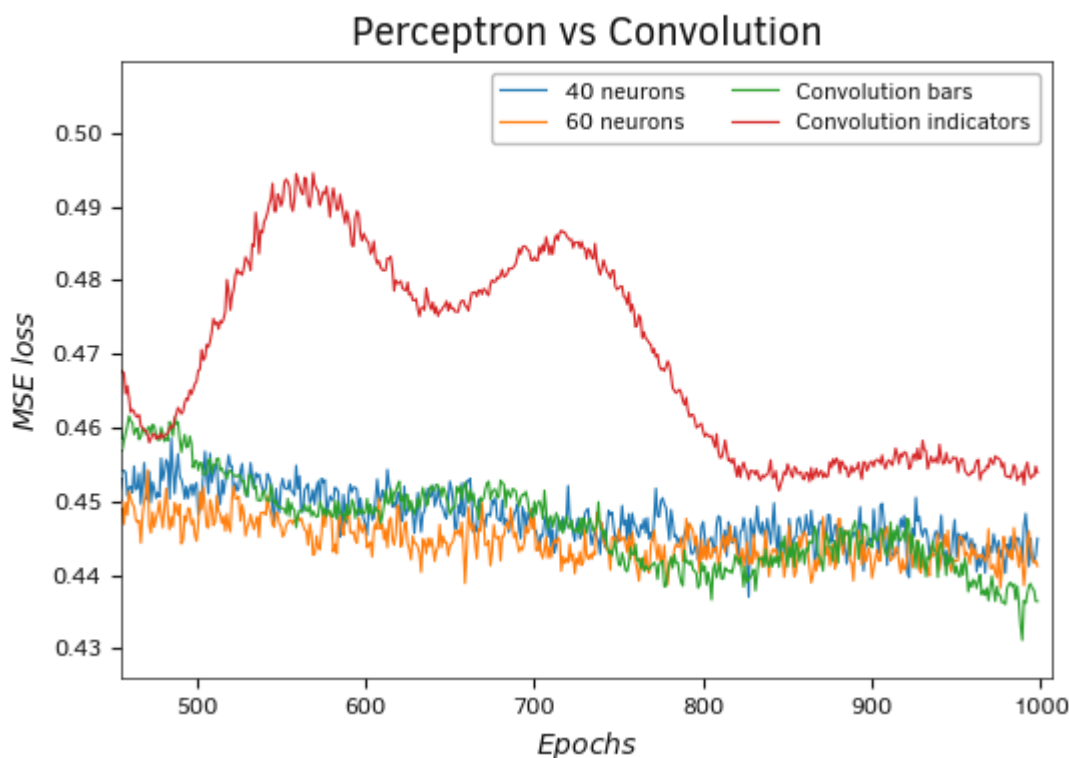
As a result of training, such a neural network demonstrated an even greater amplitude of waves in the dynamics of loss function values. During the learning process, the amplitude of the waves was reduced. Such behavior could indicate the use of an elevated learning rate, the effect of which was mitigated by the Adam training method. I'd like to remind you that this training method utilizes an algorithm of individualized adaptation of the learning rate for each element of the weight matrix.



Comparative training graph of a perceptron and two convolutional neural network models

But back to the results of our test. Unfortunately, in this case, our model changes did not produce the desired reduction in model error. On the contrary, it has even increased. Nevertheless, there is hope for improved results when the learning rate is reduced.

Increasing the scale of the graph confirms the above conclusions.



Comparative training graph of a perceptron and two convolutional neural network models

Combined model

We have examined the performance of two convolutional neural network models. In the first model, we carried out the convolution of indicator values within one candlestick. In the second model, we transposed (flipped) the original data and conducted a convolution in terms of indicator values. At the same time, in the first case, we carried out convolution of all indicators at once within one bar. I'd like to remind you that for each bar, we take four values from two indicators. In the second model, we used a convolution with a sliding window of three bars and a convolution window step of one bar. And here arises an obvious question: what results can be achieved if we combine both approaches? One more experiment will give us the answer to this question.

To conduct this test, we need to build another model. In practice, the creation of such a model did not take me much time. Let's discuss. In the first case, we took indicator values for each candlestick, while in the second case, we used three consecutive values (three consecutive bars) of each individual indicator. If we want to combine two approaches, then it would probably be logical to take all the values for three consecutive bars for convolution. In both approaches, we used a step of one bar. Therefore, we will keep this step.

To build such a model, we do not need to transpose the data. Therefore, we will build a new model based on the `convolution_test.mq5` script. First, we will create a copy of it called `convolution_test3.mq5`. In it, we will change the parameters of the convolutional layer. In the training sample, the data is in chronological order, so the convolution window of the full three bars will be equal to $3 * \text{NeuronsToBar}$. Then the step of the convolution window with the size of one bar will be equal to NeuronsToBar . With these parameters, the number of elements in one filter will be $\text{BarsToLine} - 2$. We leave the activation function and the parameter optimization method unchanged.

```
//--- Convolutional layer
if(!(descr = new CLayerDescription()))
```

4. Basic types of neural layers

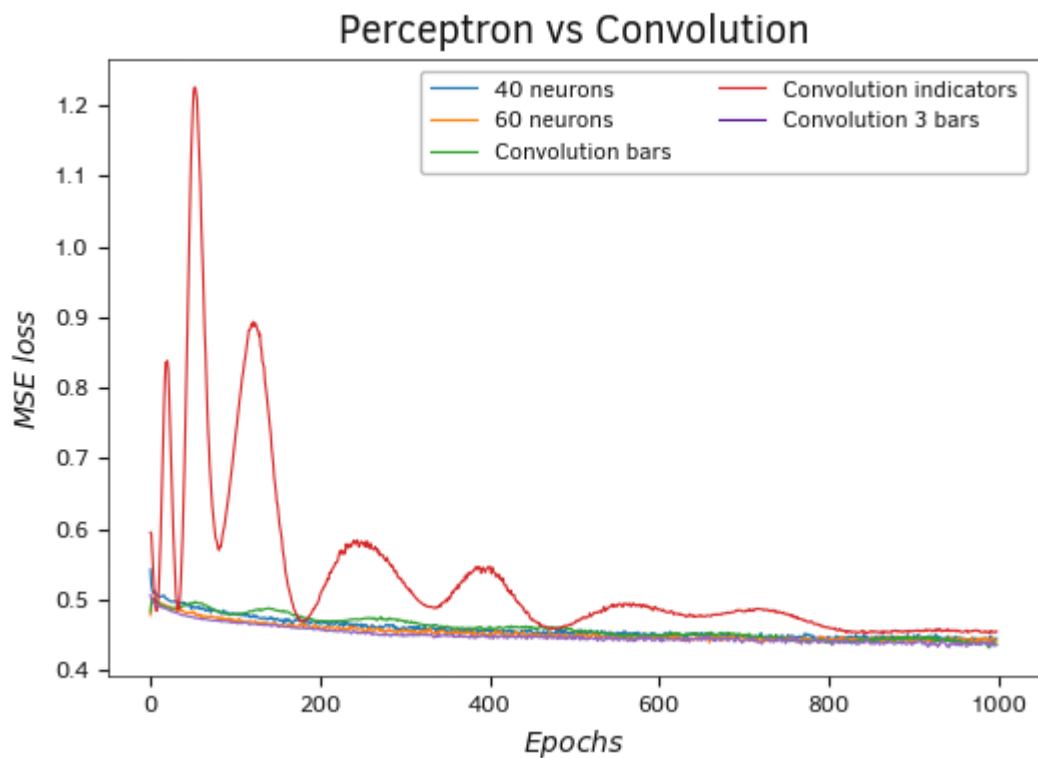
```
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
descr.type = defNeuronConv;
descr.count = BarsToLine - 2;
descr.window = 3 * NeuronsToBar;
descr.window_out = 8;
descr.step = NeuronsToBar;
descr.activation = AF_SWISH;
descr.optimization = Adam;
descr.activation_params[0] = 1;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

The changes made to the parameters of the convolutional layer required a slight adjustment of the parameters of the pooling layer. Here we have only made changes to the number of elements in one filter.

```
//--- Pooling layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
descr.type = defNeuronProof;
descr.count = BarsToLine - 3;
descr.window = 2;
descr.window_out = 8;
descr.step = 1;
descr.activation = (ENUM_ACTIVATION_FUNCTION)AF_MAX_POOLING;
descr.optimization = None;
descr.activation_params[0] = 0;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

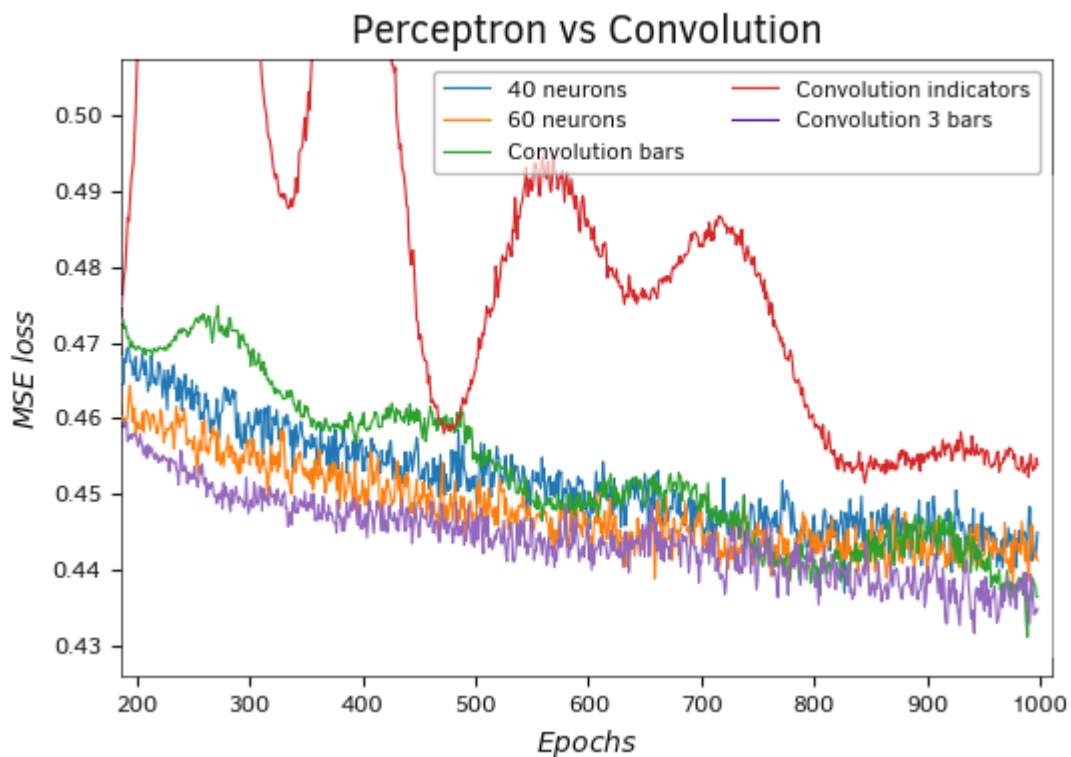
The rest of the script code remains unchanged.

The training results of the new model turned out to be better than all the previous ones. The graph of loss function dynamics without amplitude waves lies slightly below the graphs of all previously conducted tests.



Comparative training graph of a perceptron and three models of convolutional neural networks

Increasing the scale of the graph confirms the above conclusions.

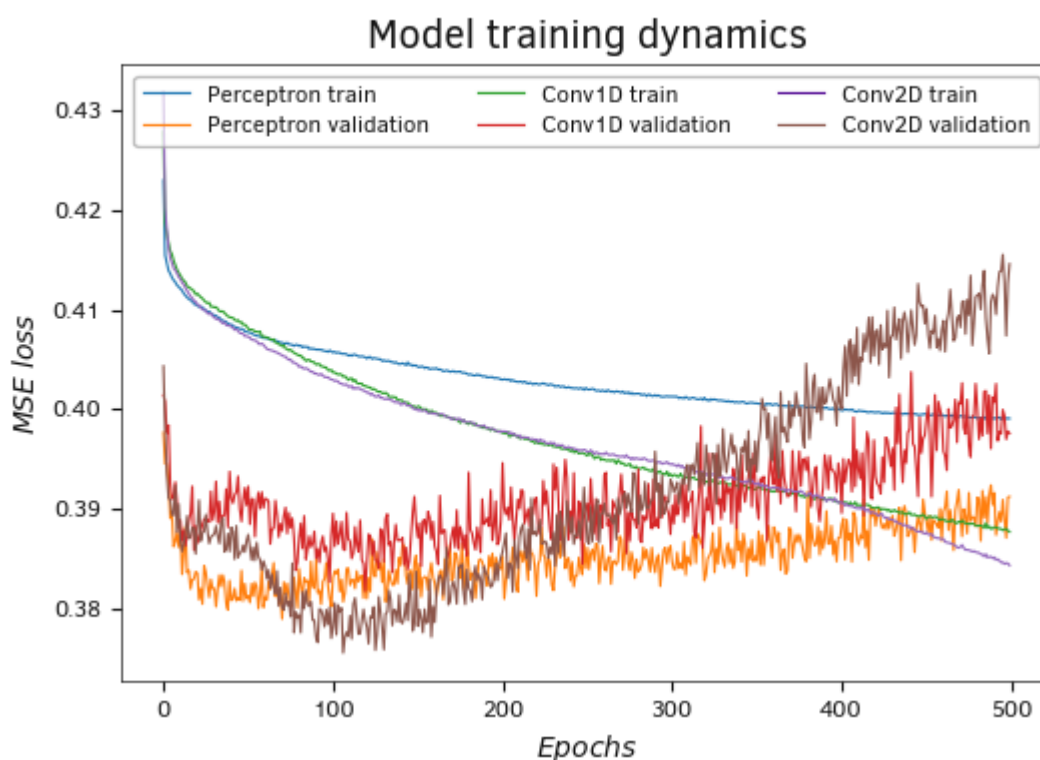


Comparative training graph of a perceptron and three models of convolutional neural networks

Testing Python models

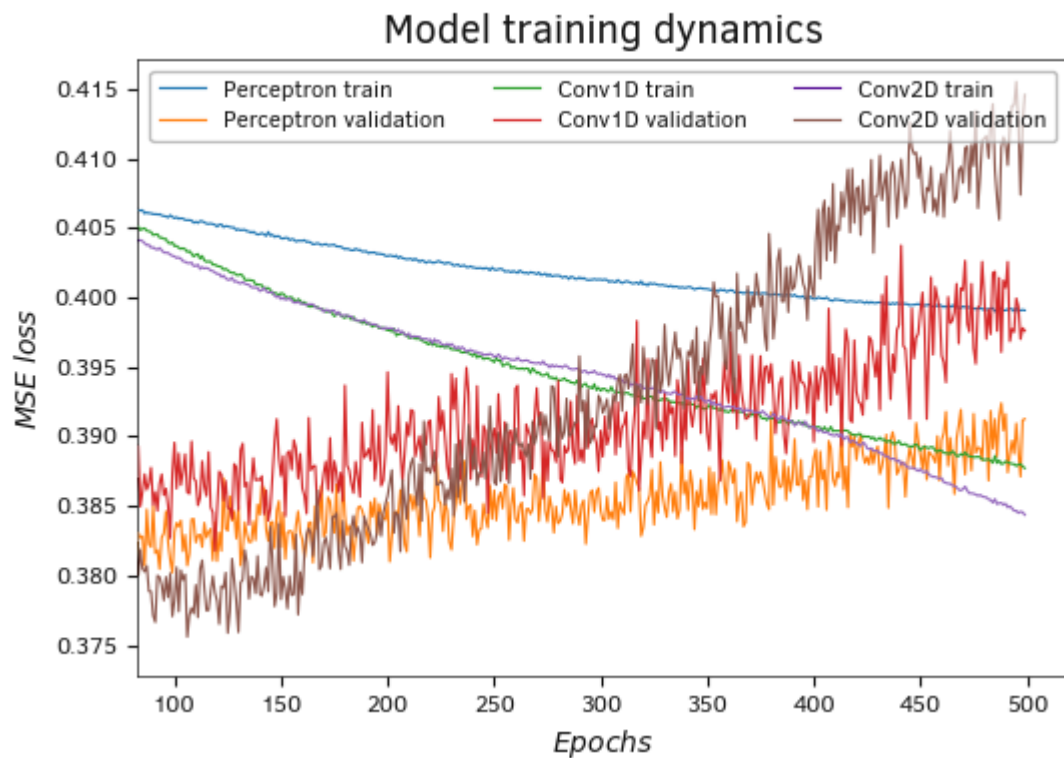
In the previous section, we created a script with three neural network models in Python. The first perceptron model has three hidden layers, the second one has an additional *Conv1D* convolutional neural layer before the perceptron model, and in the third model, the *Conv1D* convolutional layer is replaced with *Conv2D*. At the same time, the number of parameters increased in each subsequent model. Based on the logic of our work, the models we created in Python should replicate the experiments conducted earlier with the neural network models built using MQL5. Therefore, the test results were fully expected and fully confirmed the earlier conclusions. For us, this is an additional confirmation of the correct operation of our library written in the MQL5 language. So, we can use it in our future work. Moreover, obtaining similar results when testing models that were entirely created using different tools eliminates the randomness of the results and minimizes the likelihood of making errors in the model creation process.

Let's get back to the test results. During the training process, the model with the *Conv2D* convolution layer showed the best results in reducing the error, which fully confirms the results obtained above. A significant gap between the error dynamics graphs of the training and validation sets in the case of the perceptron could indicate the underfitting of the neural network.



Comparative training graph of a perceptron and 2 models of convolutional neural networks (Python)

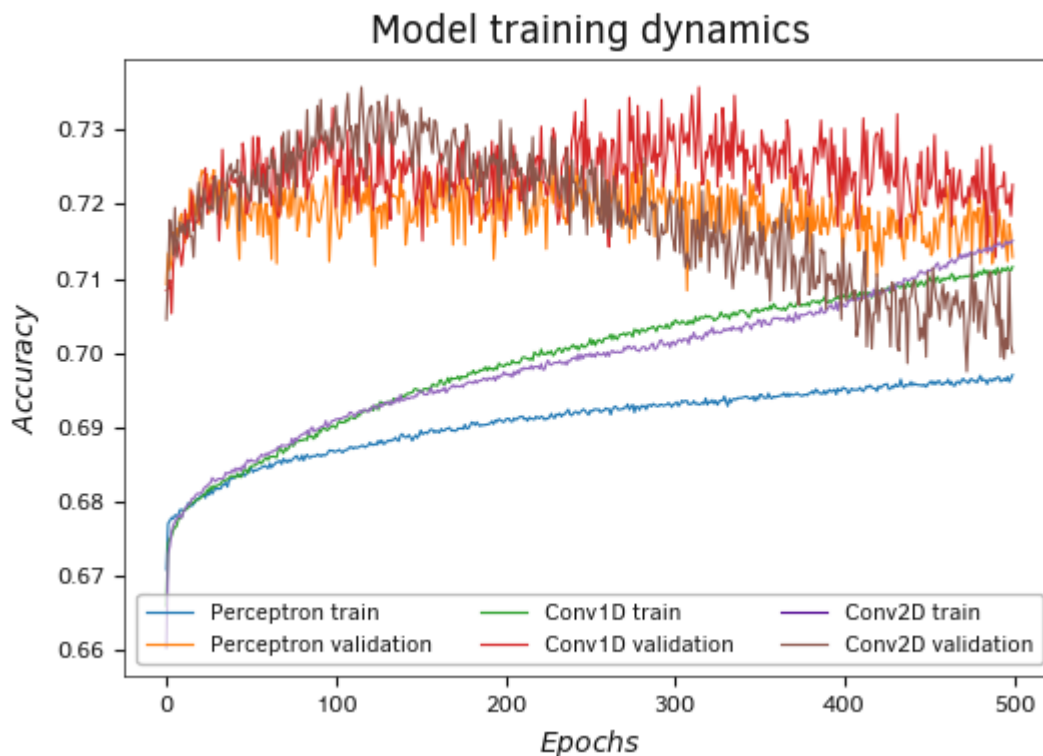
The error dynamics of the convolutional models are very close to each other. Their graphs are almost parallel. However, the model with the *Conv2D* convolutional layer shows less error throughout the training.



Comparative training graph of a perceptron and two convolutional neural network models (Python zoom)

On the validation set, the error graph of the *Conv2D* convolutional model first decreases, but after 100 epochs of training, there is an increase in the error. Along with a decrease in the error on the training set, this may indicate a tendency for models to overfit.

The graph of the *Accuracy* learning metric shows similar results.



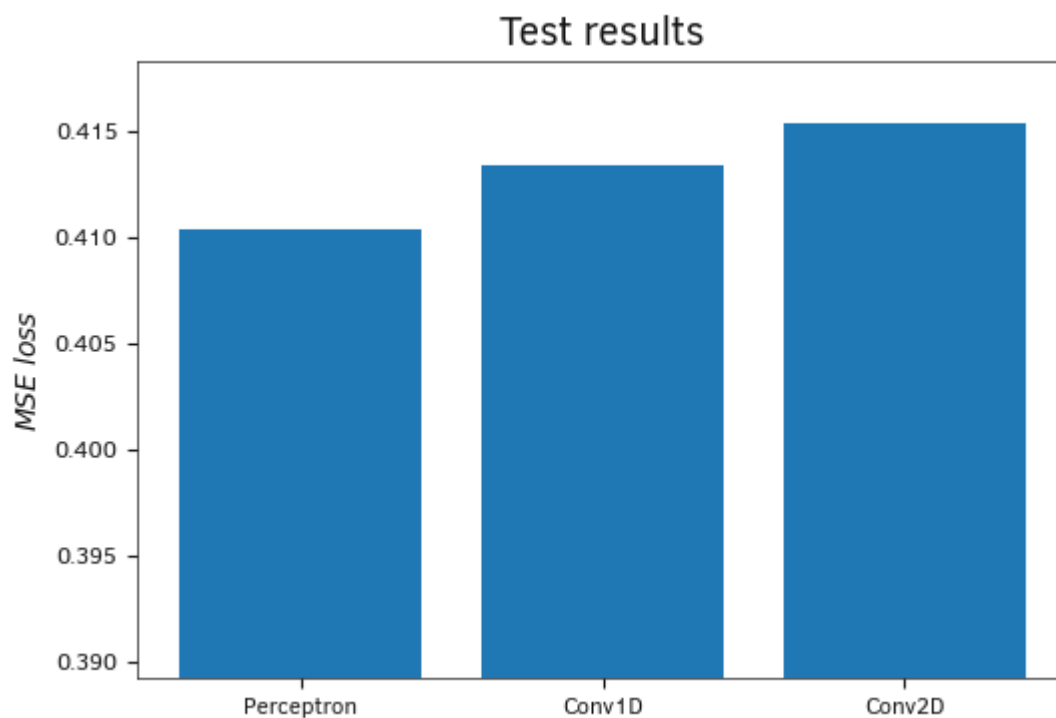
Comparative training graph of a perceptron and two convolutional neural network models (Python)

On the validation set, the graphs of all three models are closely intertwined in the range of 0.71–0.73. The graph shows the intersection of the training and validation sample graphs after 400.

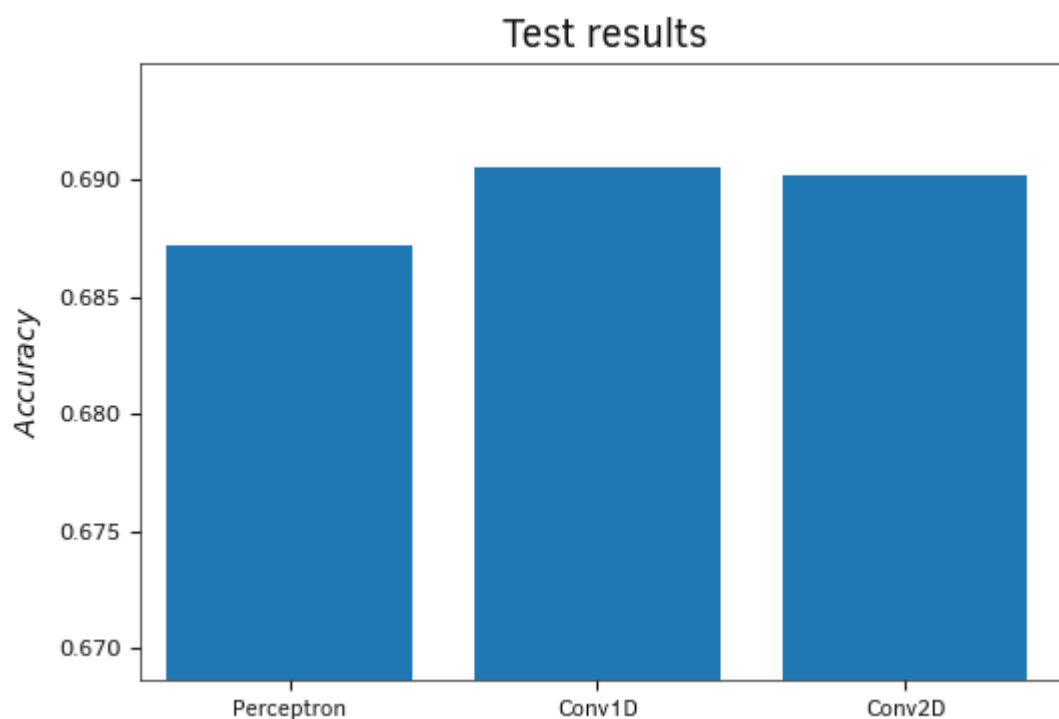
I would like to remind you that the validation dataset is significantly smaller than the training dataset; it consists of the last patterns without shuffling the overall dataset. Hence, there's a high likelihood that not all possible patterns will be included in the validation dataset. In addition, the validation set can be influenced by local trends.

Checking the performance of all three trained models on the test set showed quite similar, albeit slightly contradictory, results.

Testing the mean squared error of the models revealed that the convolutional model with the *Conv2D* convolution layer achieved the best results. This model analyzes patterns within a single indicator using a sliding window convolution. During training, it performed the best among the tested models. Certainly, the differences in the performance metrics are not very significant, and it can be considered that all models showed similar results.



Comparison of model errors on a test set



Comparison of the model accuracies on a test set

Comparison of the results by the *Accuracy* metric, in contrast to the just considered *MSE* graph, shows the best results for the Conv1D model. The model analyzes the patterns of each individual candlestick; the lowest result is for the perceptron. However, as with MSE, the gap between the results is small.

I suggest considering that all three models showed approximately equal results on the training dataset. The exact values of the metrics on the test sample are shown in the screenshot below.

Perceptron model
Test accuracy: 0.6872221827507019
Test loss: 0.41034841537475586
Conv1D model
Test accuracy: 0.6905249953269958
Test loss: 0.41335004568099976
Conv2D model
Test accuracy: 0.6902002692222595
Test loss: 0.4153774380683899

Conclusions

According to the results of the tests, we can say:

- The models built using MQL5 during training demonstrate results similar to models built using the *Keras* library in *Python*. This fact confirms the correctness of the library we are creating. We can confidently continue our work.
- In general, convolutional models contribute to improving the performance of the model on the same training dataset.
- Approaches to convolution of the initial data may be different, and the results of the model may depend on the chosen approach.
- Combining different approaches within one model does not always improve the results of the model.
- Don't be afraid to experiment. When creating your own model, try different architectures and various data processing approaches.

In our tests, we used only one convolutional and one pooling layer. This can be referred to as an approach to building simple models. The most successful convolutional models used for practical tasks often employ multiple sets of convolutional and pooling layers. At the same time, the dimension of the convolution window and the number of filters change in each set. Like I said, don't be afraid to experiment. Only by comparing the performance of different models will you be able to choose the best architecture for solving your task.

4.2 Recurrent neural networks

We have already discussed the multilayer perceptron and convolutional neural networks earlier. All of them operate with static data within the framework of Markov processes, where the subsequent state of the system depends only on its current state and is independent of the system's past states. Indeed, to compensate for this limitation, we fed the neural network not only the latest price data and indicator states but also historical data for the past few bars. However, the network itself did not memorize the processed data and the obtained results. During each new forward pass iteration into the neural network, we re-input the complete set of historical data, even if we had previously provided this data to the neural network. Essentially, with each forward pass, the neural network started with a "clean slate." The only memory such a neural network possesses is the weight matrix it learned during the training process.

Another drawback of using such neural networks for time series is that the position of a specific value/pattern in the data array has absolutely no impact on the outcome values. Let me remind you of the mathematical model of a neuron.

$$OUT = f\left(\sum_{i=1}^n w_i x_i\right)$$

Note that the sum of values is at the center of the entire model. Permuting the terms does not change the sum, so from a mathematical perspective, it is absolutely irrelevant whether a value appears at the beginning or at the end of the array. In practical usage of time series, however, it is quite common for the latest values to have a greater impact on the outcome compared to older values.

Now I suggest looking at *Recurrent Neural Networks*. They represent a special type of neural network designed to work with time sequences. The key feature of recurrent neurons is the transmission of their own state as input to themselves in the next iteration.

$$OUT_t = f\left(\sum_{i=1}^n w_i x_i^t + w_{out} OUT_{t-1}\right)$$

With each new input from the external environment, the neuron, along with the new data, essentially evaluates the outcome of its past performance as if reviewing its own history.

4.2.1 Description of architecture and implementation principles

Previously discussed types of neural networks operate with a predetermined volume of data. However, when working with price charts, it is difficult to determine the ideal size of the analyzed data. Different patterns may manifest over various time intervals, and these intervals are not always static, varying depending on the current market situation. Some events may be infrequent in the market but are likely to have a significant impact. Ideally, such an event should stay within the analyzed window. However, once it falls outside of it, the neural network no longer considers this event, even though the market may be reacting to it at that moment. Increasing the analyzed window leads to increased consumption of computational resources, requiring more time for training such a neural network. In practical real-world applications, more time will be needed for decision-making.

The use of recurrent neurons in neural networks has been proposed to address this issue in working with time series data. This involves attempting to implement short-term memory in neural networks, where the neuron input includes information about the current state of the system and its previous state. This approach is based on the assumption that the neuron output considers the influence of all factors, including its previous state, and passes all its knowledge to its future state on the next step. This is similar to human experience, where new actions are based on actions performed earlier. The duration of such memory and its impact on the current state of the neuron will depend on the weights.

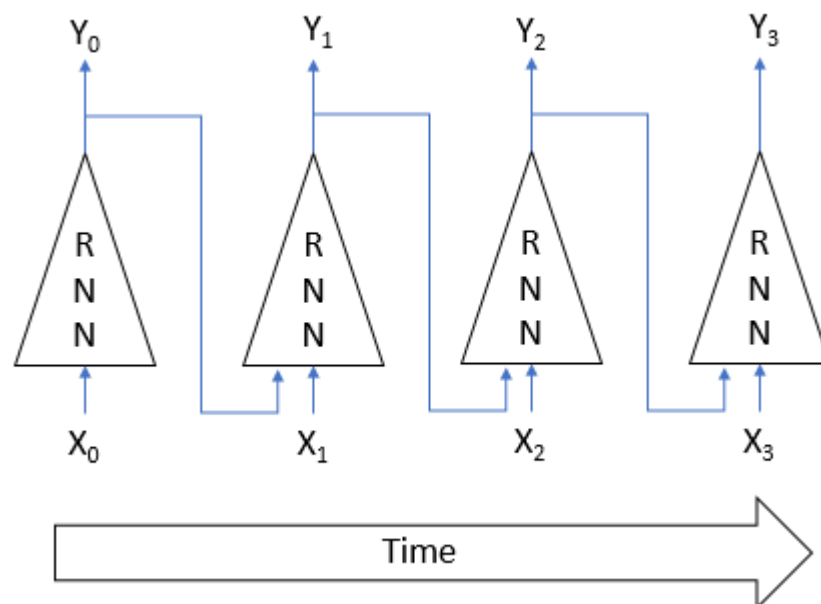
Any architectural solution for neurons can be used here, including the fully connected and convolutional layers we discussed earlier. We simply concatenate two tensors: one for the input data and one for the results of the previous iteration, and feed the resulting tensor into the neural layer. At the beginning of the neural network operation, when there is no tensor of results from the previous iteration yet, the missing elements are filled with zeros.

4. Basic types of neural layers



Recurrent neuron pattern

Training recurrent neural networks is done using the well-known method of backpropagation of errors. Similar to convolutional neural network training, the temporal nature of the process is unfolded into a multilayer perceptron. In such a perceptron, each time segment plays the role of a hidden layer. However, all layers of this perceptron use a single matrix of weights. Therefore, to adjust the weights, we take the sum of the gradients for all layers and count the delta of the weights once for the sum of all gradient layers.



Training algorithm of recurrent neural network

Unfortunately, such a simple solution is not free from drawbacks. This approach saves "memory" for a short time. The cyclical multiplication of the signal by a coefficient less than one, combined with the application of the neuron activation function, leads to a gradual attenuation of the signal as the number of such cycles increases. To solve this problem, Sepp Hochreiter and Jürgen Schmidhuber proposed the use of the *Long short-term memory* [LSTM architecture in 1997](#)). Today, the *LSTM* algorithm is

4. Basic types of neural layers

considered one of the best for solving classification and time series prediction problems, where significant events are separated over time and stretched over time intervals.

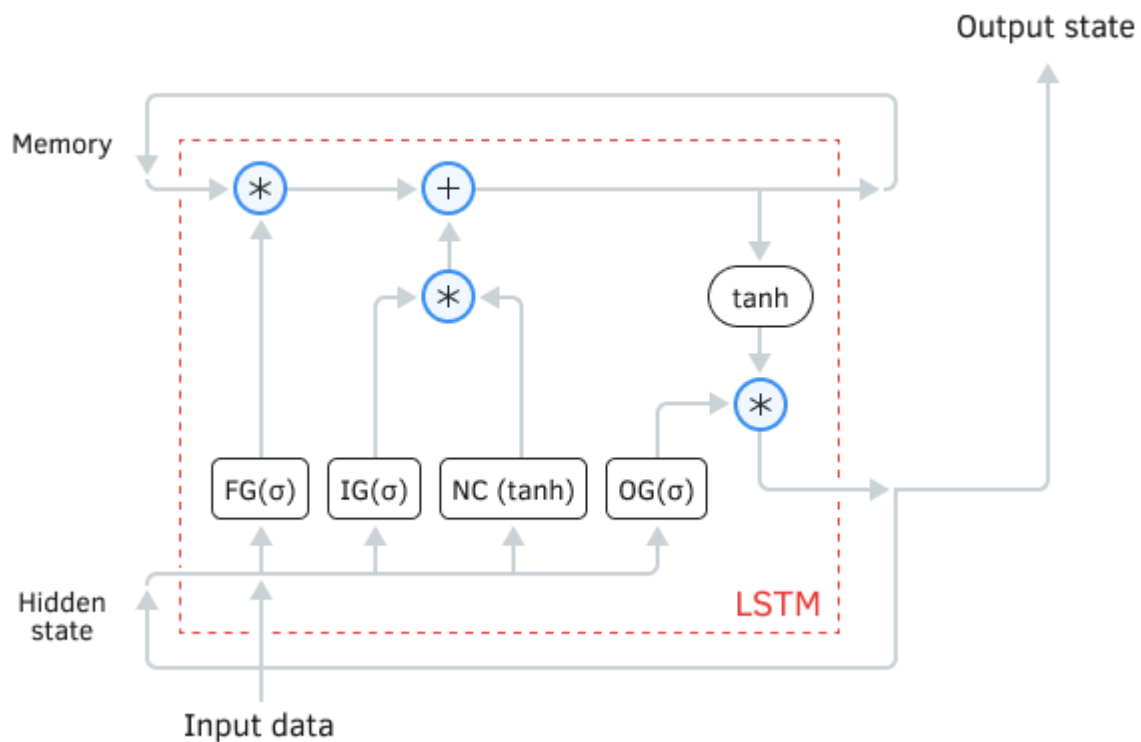
LSTM can hardly be called a neuron. Rather, it is already a neural network with three input data channels and three output data channels. Out of them, only two channels are used for data exchange with the surrounding world (one for input and one for output). The other four channels are locked in pairs for looping (*Memory* for memory and *Hidden state* for hidden state).

Within the *LSTM* block, there are two main information threads that are interconnected by four fully connected neural layers. All neural layers contain the same number of neurons, which is equal to the size of the output thread and the memory thread. Let's take a closer look at the algorithm.

The *Memory* data thread serves to store and transmit important information over time. Initially, it is initialized with zero values and filled during the neural network operation. One can compare it to a living person who is born without knowledge and learns throughout life.

The *Hidden state* thread is designed to transmit the system output state over time. The size of the data channel is equal to the data channel of the memory.

The *Input data* and *Output state* channels are designed to exchange information with the outside world.



LSTM Module Diagram

Three threads of data enter the algorithm:

- *Input data* describes the current state of the system.
- *Memory* and *Hidden state* are obtained from the previous state.

At the beginning of the algorithm, information from *Input data* and *Hidden state* are combined into a single data set, which is then fed to all four latent *LSTM* neural layers.

The first neural layer, the *Forget gate*, determines which information stored in memory can be forgotten and which should be remembered. It is organized as a fully connected neural layer with a sigmoid activation function. The number of neurons in the layer corresponds to the number of memory cells in the Memory thread. Each neuron in the layer receives a total array of *Input data* and *Hidden state* data at the input and outputs a number between 0 (completely forget) and 1 (save in memory). The element-wise product of the output from the neural layer with memory flow returns the corrected memory.

$$FG_t = \sigma(W_{FG}INP_t + U_{FG}HS_{t-1})$$

where:

- σ = activation logistic function
- W_{FG} = weights matrix for the input vector
- INP_t = input vector for the current iteration
- U_{FG} = hidden state weight matrix
- HS_{t-1} = hidden state vector from the previous iteration

In the next step, the algorithm determines which of the newly acquired information at this stage needs to be stored in memory. Two neural layers are used:

- *New Content*: a fully connected neural layer with hyperbolic tangent as an activation function normalizes the received information between -1 and 1 .

$$NC_t = \tanh(W_{NC}INP_t + U_{NC}HS_{t-1})$$

- *Input gate*: a fully connected neural layer with a sigmoid as an activation function. It is similar to the Forget gate and determines what new information to remember.

$$IG_t = \sigma(W_{IG}INP_t + U_{IG}HS_{t-1})$$

The use of the hyperbolic tangent as an activation function for the neural layer of new content allows the separation of the received information into positive and negative. The element-wise work of *New Content* and *Input gate* determines the importance of the information received and the extent to which it needs to be stored in memory.

The vector of values obtained as a result of operations is element-wise added to the current memory vector. This results in an updated memory state, which is subsequently transmitted to the input of the next iteration cycle.

$$M_t = FG_t \circ M_{t-1} + IG_t \circ NC_t$$

After updating the memory, we generate output thread values. To do this, normalize the current memory value using hyperbolic tangent. Similar to *Forget gate* and *Input gate*, let's compute *Output gate* (the output signal gate), which is also activated by the sigmoid function.

$$OG_t = \sigma(W_{OG}INP_t + U_{OG}HS_{t-1})$$

The element product of the two received data vectors gives an array of output that is produced from the LSTM to the outside world. The same data set will be passed to the next iteration cycle as a hidden state thread.

$$OUT_t = HS_t = OG_t \circ \tanh(M_t)$$

Since the introduction of the *LSTM* unit, there have appeared many different modifications to it. Some tried to make it "lighter" for faster information processing and training. Others, on the contrary, made it harder to try to get better results. The *GRU* (*Gated Recurrent Unit*) model introduced by Kyunghyun Cho and his team in September 2014 is considered to be one of the most successful variations. This solution can be considered a simplified version of the standard *LSTM* unit. In it, the *Forget gate* and the *Input gate* are combined into a single update gate. This eliminates the use of a separate memory thread. Only the Hidden state is used to transmit information through time.

At the beginning of the *GRU* algorithm, as in *LSTM*, the refresh and reset gates are defined. The mathematical formula for calculating values is similar to the definition of the gate values in *LSTM*.

$$UG_t = \sigma(W_{UG}INP_t + U_{UG}HS_{t-1})$$

$$RG_t = \sigma(W_{RG}INP_t + U_{RG}HS_{t-1})$$

Then the current memory state is updated. In this process, the hidden state from the previous iteration is first multiplied by the corresponding weight matrix and then element-wise multiplied by the value of the reset gate. The resulting vector is added from the product of the raw data to its weight matrix. The total vector is activated by a hyperbolic tangent.

$$HS'_t = \tanh(W_{HS'}INP_t + RG \circ U_{HS'}HS_{t-1})$$

In conclusion of the algorithm, the hidden state from the previous iteration is element-wise multiplied by the value of the update gate, while the current memory state is multiplied by the difference between one and the value of the update gate. The sum of these products is passed as the output from the block and as the hidden state for the next iteration.

$$HS_t = UG_t \circ HS_{t-1} + (1 - UG_t) \circ HS'_t$$

Thus, in the *GRU* model, the reset gate controls the rate of data forgetting. The update gate determines how much information to take from the previous state and how much of the new data.

4.2.2. Building an LSTM block in MQL5

To implement in our library, among all the options for architectural solutions of recurrent neurons, I have chosen the classical *LSTM* block. In my opinion, the presence of filters for new information and memory content in the form of gates will help minimize the influence of the noisy component of the signal. And a separate memory channel will help retain information for a longer period.

As before, to create a new type of neural layer, we will create a new class *CNeuronLSTM*. To maintain inheritance, the new class will be created based on our *CNeuronBase* neural layer base class.

```
class CNeuronLSTM : public CNeuronBase
{
public:
    CNeuronLSTM(void);
    ~CNeuronLSTM(void);
    //--- method of identifying the object
    virtual int Type(void) const { return(defNeuronLSTM); }
};
```

Since we apply the inheritance mechanism, our new class immediately possesses the basic functionality that was previously implemented in the parent class. Now we need to refine this functionality for the correct operation of our recurrent block. First, let's rewrite the virtual identification method.

As you know from the description of the *LSTM* block architecture presented in the previous chapter, we will need four fully connected layers for its proper operation. We'll declare them in the *protected* block of our class. And to maintain code readability, we will name them in accordance with the functionality laid out in the algorithm.

```
class CNeuronLSTM : public CNeuronBase
{
protected:
    CNeuronBase*      m_cForgetGate;
    CNeuronBase*      m_cInputGate;
    CNeuronBase*      m_cNewContent;
    CNeuronBase*      m_cOutputGate;
```

In addition to the created neural layers, the block algorithm uses memory streams and a hidden state. We will need separate buffers to store them. We will also need to use the chronology of internal neurons in our training. Therefore, to store such information, we will create dynamic arrays, which we will also declare in the *protected* block:

- *m_cMemorys* – memory state;
- *m_cHiddenStates* – hidden state;
- *m_cInputs* – concatenated array of raw data and hidden state;
- *m_cForgetGateOuts* – state of the forget gate;
- *m_cInputGateOuts* – state of the input gate;
- *m_cNewContentOuts* – new content;
- *m_cOutputGateOuts* – output gate state.

```
class CNeuronLSTM : public CNeuronBase
{
protected:
    ....
    CArrayObj*      m_cMemorys;
    CArrayObj*      m_cHiddenStates;
    CArrayObj*      m_cInputs;
    CArrayObj*      m_cForgetGateOuts;
    CArrayObj*      m_cInputGateOuts;
    CArrayObj*      m_cNewContentOuts;
    CArrayObj*      m_cOutputGateOuts;
```

Of course, in the process of operating a neural network, we cannot indefinitely accumulate a history of states, because our resources are finite. Therefore, we will need some kind of reference for understanding the buffer filling. If the buffer overflows above this limit, we will remove the oldest data and replace it with new. The depth of history for training the recurrent block will serve as such a reference for us. This parameter will be user-defined and stored in the *m_iDepth* variable.

```
class CNeuronLSTM : public CNeuronBase
{
protected:
    ....
```

```
int m_iDepth;
```

Continuing the discussion about declaring auxiliary variables for the class, there is another point to pay attention to. All four internal neural layers use the same input data which includes the concatenated tensor of the original data and the hidden state. The *CalcHiddenGradient* method of passing the gradient through the hidden layer of our base class is constructed so that it replaces the error gradient values in the buffer of the previous layer. However, we need to sum up the error gradient from all internal flows. Therefore, to accumulate the sum of the gradients, we will add another buffer *m_cInputGradient*.

```
class CNeuronLSTM : public CNeuronBase
{
protected:
    ....
    CBufferDouble* m_cInputGradient;
```

It seems we've sorted out the variables. Now let's start building the class methods. The first thing that the class starts with is the constructor *CNeuronLSTM::CNeuronLSTM*. In this method, we create instances of the objects used and set initial values for the internal variables.

```
CNeuronLSTM::CNeuronLSTM(void) : m_iDepth(2)
{
    m_cForgetGate = new CNeuronBase();
    m_cInputGate = new CNeuronBase();
    m_cNewContent = new CNeuronBase();
    m_cOutputGate = new CNeuronBase();
    m_cMemorys = new CArrayObj();
    m_cHiddenStates = new CArrayObj();
    m_cInputs = new CArrayObj();
    m_cForgetGateOuts = new CArrayObj();
    m_cInputGateOuts = new CArrayObj();
    m_cNewContentOuts = new CArrayObj();
    m_cOutputGateOuts = new CArrayObj();
    m_cInputGradient = new CBufferType();
}
```

We immediately create the destructor of the class *CNeuronLSTM::~~CNeuronLSTM*, in which the reverse operation takes place, that is, memory is released after the class has finished its work. Here it's important to ensure complete memory cleanup so that nothing is missed.

```
CNeuronLSTM::~~CNeuronLSTM(void)
{
    if(m_cForgetGate)
        delete m_cForgetGate;
    if(m_cInputGate)
        delete m_cInputGate;
    if(m_cNewContent)
        delete m_cNewContent;
    if(m_cOutputGate)
        delete m_cOutputGate;
    if(m_cMemorys)
        delete m_cMemorys;
    if(m_cHiddenStates)
```

```

        delete m_cHiddenStates;
    if(m_cInputs)
        delete m_cInputs;
    if(m_cForgetGateOuts)
        delete m_cForgetGateOuts;
    if(m_cInputGateOuts)
        delete m_cInputGateOuts;
    if(m_cNewContentOuts)
        delete m_cNewContentOuts;
    if(m_cOutputGateOuts)
        delete m_cOutputGateOuts;
    if(m_cInputGradient)
        delete m_cInputGradient;
}

```

Object initialization

Next, let's take a look at the method of initializing an instance of the *CNeuronLSTM::Init* class. It is in this method that all internal objects and variables are created and initialized, as well as the necessary foundation for the normal operation of the neural layer is prepared in accordance with the user-defined requirements. We created a similar virtual method in our base class for neural layers and constantly override it in each of our new classes.

```

class CNeuronLSTM    : public CNeuronBase
{
protected:
    ....
public:
                CNeuronLSTM(void);
                ~CNeuronLSTM(void);

    //---
    virtual bool    Init(const CLayerDescription *desc) override;

```

As you know, a similar method of the base class receives the description of the neural layer being created as parameters. So, our method in the parameters will get a pointer to an instance of the *CLayerDescription* class. Therefore, at the beginning of the method, we perform a check for the validity of the received pointer and the parameters set in it. First of all, the type of neural layer it specifies must match our class. Also, our LSTM block cannot be used as an input layer and must contain at least one neuron at the output.

```

bool CNeuronLSTM::Init(const CLayerDescription *desc)
{
    //--- Control block
    if(!desc || desc.type != Type() || desc.count <= 0 || desc.window == 0)
        return false;

```

The use of the *LSTM* block as a source data layer is simply a waste of resources. We create a large number of additional objects that will never be used since we write the information directly into the output buffer in the input layer.

Next, we have to initialize our internal neural layers. To do this, we will call the `sameInit` method of our objects. Therefore, we need to pass them the corresponding instance of the `CLayerDescription` class. We can't simply pass the object describing the recurrent block received from the user, as we need to create other objects. So, first, we will prepare a description of the objects to be created:

- All internal neural layers are fully connected. Hence, we create base class objects. Therefore, we will specify the `defNeuronBase` type in the `type` parameter.
- All of them take as input a single tensor, which is a combination of the original data vector and the hidden state. We get the size of the source data vector in the method parameters (`CLayerDescription.window` parameter). The size of the hidden state vector is equal to the size of the output buffer of the current layer. We also get this value in method parameters (`CLayerDescription.count` parameter). The sum of the two values will be written in the `window` parameter.
- If you look carefully at the *LSTM* block diagram in the previous section, you will be able to see that all internal information flows have the same size. The forget gate output vector is element-wise multiplied by the memory flow. This means their sizes are equal. Similarly, the input gate result vector is elementally multiplied by the new content layer result. Then this product is element-wise summed with the memory flow. Finally, everything is atomically multiplied by the output control gate. It becomes clear that all flows are equal to the size of the output buffer of the current block. So, to the `count` parameter, we will move the value of a similar element from the external parameters of the method.
- The activation function is defined by the architecture of the *LSTM* block. All gates are activated by a sigmoid and the new content layer by a hyperbolic tangent. Along with the activation function, we will specify its corresponding parameters.
- We will transfer the optimization method specified by the user.

```
//--- create a description for the inner neural layers
CLayerDescription *temp = new CLayerDescription();
if(!temp)
    return false;
temp.type = defNeuronBase;
temp.window = desc.window + desc.count;
temp.count = desc.count;
temp.activation = AF_SIGMOID;
temp.activation_params[0] = 1;
temp.activation_params[1] = 0;
temp.optimization = desc.optimization;
```

After preparing the description for the internal neural layers, we will return to our inheritance from the parent class. All the block parameters are hidden within the internal neural layers, so there is no need for us to keep an additional weight matrix in memory, nor the associated delta and momentum buffers. In addition, we do not plan to use the *CActivation* activation class object. Essentially, the functionality of the input layer from the base class is sufficient for us. To initialize the necessary objects and remove the excess ones, we will zero out the size of the input data in the description of the recurrent block and call the initialization method of the parent class.

```
//--- call the parent class initialization method
CLayerDescription *temp2=new CLayerDescription();
if(!temp2 || !temp2.Copy(desc))
    return false;
temp2.window = 0;
if(!CNeuronBase::Init(temp2))
```

```

        return false;
    delete temp2;

```

To obtain information from the user about the history depth for training the recurrent block, we will use the *window_out* element. We will save the received value in a specially prepared variable. We did not check this value at the beginning of the method in order not to block the operation of the neural network. Instead, we simply limited the lower bound of the stored value. Therefore, if the user forgets to specify a value or indicates an intentionally low value, the neural network will use the value that we have set.

```

    if(!InsertBuffer(m_cHiddenStates, m_cOutputs, false))
        return false;
    m_iDepth = (int)fmax(desc.window_out, 2);

```

Next, we move on to initializing our gate. The forget gate will be initialized first. Before calling the gate object initialization method, we need to verify the validity of the pointer to the object. If necessary, we will create a new instance of the object. If the attempt to create a new instance of the object is unsuccessful, we exit the method with the *false* result. If there is an actual object instance, we initialize the gate.

```

//--- initialize ForgetGate
if(!m_cForgetGate)
{
    if(!(m_cForgetGate = new CNeuronBase()))
        return false;
}
if(!m_cForgetGate.Init(temp))
    return false;
if(!InsertBuffer(m_cForgetGateOutputs, m_cForgetGate.GetOutputs(), false))
    return false;

```

Similar iterations are performed for the other two gates.

```

//--- initialize InputGate
if(!m_cInputGate)
{
    if(!(m_cInputGate = new CNeuronBase()))
        return false;
}
if(!m_cInputGate.Init(temp))
    return false;
if(!InsertBuffer(m_cInputGateOutputs, m_cInputGate.GetOutputs(), false))
    return false;

//--- initialize OutputGate
if(!m_cOutputGate)
{
    if(!(m_cOutputGate = new CNeuronBase()))
        return false;
}
if(!m_cOutputGate.Init(temp))
    return false;
if(!InsertBuffer(m_cOutputGateOutputs, m_cOutputGate.GetOutputs(), false))
    return false;

```

The new content layer will be initialized in the same way. We will only preliminarily change the type of the activation function in the layer description.

```
//--- initialize NewContent
if(!m_cNewContent)
{
    if(!(m_cNewContent = new CNeuronBase()))
        return false;
}
temp.activation = AF_TANH;
if(!m_cNewContent.Init(temp))
    return false;
if(!InsertBuffer(m_cNewContentOuts, m_cNewContent.GetOutputs(), false))
    return false;
```

After initializing the internal layers, we will move on to the other objects of our LSTM recurrent block. We initialize the gradient accumulation buffer. As in the case of neural layers, we first verify the validity of the object pointer. If necessary, we create a new instance of the class. Then we fill the entire buffer with zero values. We take the buffer size from the previously prepared description of the internal neural layers.

```
//--- initialize the InputGradient buffer
if(!m_cInputGradient)
{
    if(!(m_cInputGradient = new CBufferType()))
        return false;
}
if(!m_cInputGradient.BufferInit(1, temp.window, 0))
    return false;
delete temp;
```

It should be noted that after initializing the buffer for accumulating gradient values, we will no longer use the object describing the internal neural layers. Therefore, we can delete the unnecessary object.

In conclusion, all that remains is to create and fill with zero values the buffers for the memory flow and hidden state. Note that both buffers will be used on the first direct pass, and their absence will paralyze the entire neural network. A separate method *CreateBuffer* has been added to create these buffers, which we will consider later.

So, first, we create a memory buffer. We declare a temporary variable and call the *CreateBuffer* method. As a result of the method, we expect a pointer to the buffer object. Certainly, after obtaining a pointer, we check its validity. If an error occurs, we exit the method with the result of *false*.

Next, we check for the presence of existing objects in the memory stack. We are discussing the method of initializing a class instance, so we expect an empty stack to be present. If the stack, however, contains any information, we clear the stack and fill the created buffer with null values. After this, we place our buffer into the memory stack.

```
//--- initialize Memory
CBufferType *buffer = CreateBuffer(m_cMemorys);
if(!buffer)
    return false;
if(!InsertBuffer(m_cMemorys, buffer, false))
{
```


4. Basic types of neural layers

```
        delete buffer;
        return false;
    }
```

As a result of executing this code block within the method, we expect to obtain a memory stack containing a single null memory buffer. Please note that at the end of the block execution, we do not delete the buffer object, even though the variable scope does not extend beyond this method. The reason is, we operate object pointers here. By putting a pointer on the stack, we can always get it from there. Conversely, if we delete the object pointed to by the variable pointer in the stack, we will also end up with a pointer to a deleted object, along with all the resulting consequences. The object will actually be deleted either upon stack overflow or when attempting to close the entire instance of the class.

Repeat all iterations for the hidden state buffer.

```
//--- initialize HiddenStates
if(!(buffer = CreateBuffer(m_cHiddenStates)))
    return false;
if(!InsertBuffer(m_cHiddenStates, buffer, false))
{
    delete buffer;
    return false;
}
```

Lastly, we pass the current pointer to the OpenCL object to all internal objects and exit the method.

```
//---
SetOpenCL(m_cOpenCL);
//---
return true;
}
```

We have considered the algorithm of the class initialization method. However, as you may have noticed, during the execution of the algorithm, we used two methods of the class: *SetOpenCL* and *CreateBuffer*. The first method exists in the parent class, but for proper functionality, we will need to override it. The second method is new.

The *CreateBuffer* method in the initialization method was used to create a new buffer. Looking a bit ahead, we will use it in a broader context. As you know from the architecture of the *LSTM* recursive block we are building, we will need to extract the last hidden state and memory vectors from the stack on each feed-forward pass. We will also transfer this functionality to the *CreateBuffer* method.

Since we anticipate the method working with multiple stacks, we will pass a pointer to a specific stack as a parameter to the method. The result of the method execution will be a pointer to the desired buffer. We declare the method in the *protected* block of our class.

```
class CNeuronLSTM : public CNeuronBase
{
protected:
    ....
    CBufferDouble* CreateBuffer(CArrayObj * &array);
}
```

At the beginning of the method body, as usual, we check the received stack pointer. However, in case we receive an invalid pointer, we don't rush to exit the method with an error message. Instead, we try to create a new stack. Only if we can't create a new stack, we exit the method.

Remember, the code that invokes the method expects to receive not just the logical state of the method execution but a pointer to the buffer. Therefore, in case of an error, we return *NULL* instead of the expected pointer.

```
CBufferType *CNeuronLSTM::CreateBuffer(CArrayObj *&array)
{
    if(!array)
    {
        array = new CArrayObj();
        if(!array)
            return NULL;
    }
}
```

Next, we create a new buffer and immediately check the result.

```
CBufferType *buffer = new CBufferType();
if(!buffer)
    return NULL;
```

After successfully creating the buffer, we split the algorithm into two threads. In one case when there are no buffers on the stack, we fill the buffer we created with zero values. If there is already information on the stack, we copy the latest states to the buffer. Then we return a pointer to the buffer of the calling program.

```
if(array.Total() <= 0)
{
    if(!buffer.BufferInit(m_cOutputs.Rows(), m_cOutputs.Cols(), 0))
    {
        delete buffer;
        return NULL;
    }
}
else
{
    CBufferType *temp = array.At(0);
    if(!temp)
    {
        delete buffer;
        return NULL;
    }
    buffer.m_mMatrix = temp.m_mMatrix;
}
//---
if(m_cOpenCL)
{
    if(!buffer.BufferCreate(m_cOpenCL))
        delete buffer;
}
//---
return buffer;
}
```

Note that I'm referring to the latest data and, in doing so, I'm copying the buffer with index 0. This class implements reverse stack logic. For each new buffer, we will insert it at the beginning of the stack, pushing the older ones down, and when the stack is full, we will remove the last ones.

And second point: we don't take a pointer to an existing buffer, instead we create a new one. This is because we will change the contents of the buffer during the forward pass. In doing so, it's important for us to preserve the previous state. In the case of using a pointer to an old buffer, we will simply overwrite its values, effectively discarding the desired previous states.

The second method, *SetOpenCL*, is an overriding method of the parent class and has the same functionality of passing a pointer to the OpenCL context to all internal objects involved in the computation process. Similar to the method in the parent class, our method will receive a pointer to the OpenCL context as a parameter and will return a logical result indicating the readiness of the class to operate within the specified context.

```
class CNeuronLSTM      : public CNeuronBase
{
protected:
    ....
public:
    ....
    virtual bool        SetOpenCL(CMyOpenCL *opencl) override;
```

The algorithm of the method is quite simple. First, we call the method of the parent class and pass the resulting pointer to it. The validation of the received pointer correctness is already implemented in the parent class method. Therefore, we need not repeat it here.

Then, we pass the OpenCL context pointer stored in our class variable to all internal objects. The key point here is that the method of the parent class has verified the received pointer and has saved the corresponding pointer in a variable. To ensure that all objects operate within the same context, we propagate the processed pointer.

```
bool CNeuronLSTM::SetOpenCL(CMyOpenCL *opencl)
{
    //--- call the parent class method
    CNeuronBase::SetOpenCL(opencl);
    //--- call the relevant method for all internal layers
    m_cForgetGate.SetOpenCL(m_cOpenCL);
    m_cInputGate.SetOpenCL(m_cOpenCL);
    m_cOutputGate.SetOpenCL(m_cOpenCL);
    m_cNewContent.SetOpenCL(m_cOpenCL);
    m_cInputGradient.BufferCreate(m_cOpenCL);
    for(int i = 0; i < m_cMemorys.Total(); i++)
    {
        CBufferType *temp = m_cMemorys.At(i);
        temp.BufferCreate(m_cOpenCL);
    }
    for(int i = 0; i < m_cHiddenStates.Total(); i++)
    {
        CBufferType *temp = m_cHiddenStates.At(i);
        temp.BufferCreate(m_cOpenCL);
    }
    //---
```

4. Basic types of neural layers

```
    return(!m_cOpenCL);  
}
```

At this point, we can say that we have completed the work on the class initialization algorithm. We can now move on to the next phase, which is to create a feed-forward algorithm.

4.2.2.1 Feed-forward method for LSTM block

As always, we will create the feed-forward algorithm in the *FeedForward* method. The feed-forward pass method is one of the basic methods that is defined by a virtual method in the base class and is overridden in all inherited methods.

```
class CNeuronLSTM      : public CNeuronBase
{
protected:
    ....
public:
    ....
    virtual bool        FeedForward(CNeuronBase *prevLayer) override;
```

The *FeedForward* method receives a pointer to the previous neural layer as a parameter, which contains the initial data for the method operation. It returns a logical value indicating the execution status of the method operations.

At the beginning of the method, we check the validity of pointers to all objects that are critical for the method operations. If there is at least one invalid pointer, we exit the method with the result of *false*.

```
bool CNeuronLSTM::FeedForward(CNeuronBase *prevLayer)
{
    --- check the relevance of all objects
    if(!prevLayer || !prevLayer.GetOutputs() || !m_cOutputs ||
        !m_cForgetGate || !m_cInputGate || !m_cOutputGate ||
        !m_cNewContent)
        return false;
```

After successfully passing through the control block, we create stubs for new memory buffers and hidden states. To do this, we use the *CreateBuffer* method discussed above, remembering to control the result of the operations.

```
    --- prepare blanks for new buffers
    if(!m_cForgetGate.SetOutputs(CreateBuffer(m_cForgetGateOuts), false))
        return false;
    if(!m_cInputGate.SetOutputs(CreateBuffer(m_cInputGateOuts), false))
        return false;
    if(!m_cOutputGate.SetOutputs(CreateBuffer(m_cOutputGateOuts), false))
        return false;
    if(!m_cNewContent.SetOutputs(CreateBuffer(m_cNewContentOuts), false))
        return false;
    CBufferType *memory = CreateBuffer(m_cMemorys);
    if(!memory)
        return false;
    CBufferType *hidden = CreateBuffer(m_cHiddenStates);
    if(!hidden)
    {
        delete memory;
        return false;
    }
```

4. Basic types of neural layers

Next, we have to prepare the initial data for the correct operation of the internal layers. This procedure is not as simple as it may seem at first glance. The reason is that to call the feed-forward methods of our gates, we require not just a buffer but a neural layer. We cannot put a pointer to the previous layer obtained in the parameters, because it does not contain all the necessary information. It lacks the hidden state data necessary for the algorithm to function correctly. Therefore, we will need to create an empty neural layer and fill its output buffer with the necessary data.

But before creating a new neural layer, we verify the validity of the pointer to the stack storing the source data neural layers. If needed, we create a new one, as after conducting the feed-forward pass, we will need to store the created neural layer for subsequent neural network training. The check for the stack presence is performed before completing the entire loop of feed-forward operations, in order to save resources by avoiding unnecessary operations.

```
--- create a buffer for the source data
if(!m_cInputs)
{
    m_cInputs = new CArrayObj();
    if(!m_cInputs)
    {
        delete memory;
        delete hidden;
        return false;
    }
}
```

Please note that before exiting the method after an unsuccessful attempt to create a new stack, we will need to delete the objects created within the method, for which pointers are not passed to the global variables of the class.

Next, we create a new instance of the base neural layer object. And, as always, we check the result of the operation.

```
CNeuronBase *inputs = new CNeuronBase();
if(!inputs)
{
    delete memory;
    delete hidden;
    return false;
}
```

After successfully creating an instance of the base neural layer object, we need to create an object describing the structure of the neural layer for its initialization. That's what we'll proceed to do. We will create an instance of the *CLayerDescription* object and populate it with the necessary data. We will specify the type of neuron layer as *defNeuronBase*. The number of elements in the neural layer will be equal to the sum of the elements in the result buffers of the previous and current layers. Since we will directly populate the result buffer of the created layer from other sources, we set the window size for source data to 0.

```
CLayerDescription *desc = new CLayerDescription();
if(!desc)
{
    delete inputs;
    delete memory;
    delete hidden;
}
```

4. Basic types of neural layers

```
        return false;
    }
    desc.type = defNeuronBase;
    desc.count = (int)(prevLayer.GetOutputs().Total() + m_cOutputs.Total());
    desc.window = 0;
```

After creating the description of the neural layer, we proceed to its initialization. Upon successful completion of the operation, we delete the no-longer-needed layer description object.

```
if(!inputs.Init(desc))
{
    delete inputs;
    delete memory;
    delete hidden;
    delete desc;
    return false;
}
delete desc;
inputs.SetOpenCL(m_cOpenCL);
```

After this, we only need to fill the result buffer of the new layer with the necessary source data. To begin with, we get a pointer to the required buffer and verify its validity.

```
CBufferType *inputs_buffer = inputs.GetOutputs();
if(!inputs_buffer)
{
    delete inputs;
    delete memory;
    delete hidden;
    return false;
}
```

After that, we populate the buffer with the contents of the result buffers from the previous layer and the hidden state. We have moved the functionality of data transfer to a separate *Concatenate* method, which we will consider later.

```
if(!inputs_buffer.Concatenate(prevLayer.GetOutputs(), hidden,
                             prevLayer.Total(), hidden.Total()))
{
    delete inputs;
    delete memory;
    delete hidden;
    return false;
}
```

Now that we have completed the preparatory work, we can proceed directly to the feed-forward pass operations. We will start this process by calling the feed-forward pass methods of the internal neural layers. First, we will perform the forward pass for the forget gates. We simply call the method with the same name on the corresponding object. We will pass a pointer to the newly created instance of the neural layer for the source data as parameters to the method, and then we check the result of the operation execution.

```
--- perform a feed-forward pass of the internal neural layers
if(!m_cForgetGate.FeedForward(inputs))
```

4. Basic types of neural layers

```
{
    delete inputs;
    delete memory;
    delete hidden;
    return false;
}
```

We will repeat the operation for all internal layers.

```
if(!m_cInputGate.FeedForward(inputs))
{
    delete inputs;
    delete memory;
    delete hidden;
    return false;
}

if(!m_cOutputGate.FeedForward(inputs))
{
    delete inputs;
    delete memory;
    delete hidden;
    return false;
}

if(!m_cNewContent.FeedForward(inputs))
{
    delete inputs;
    delete memory;
    delete hidden;
    return false;
}
```

After successfully completing the feed-forward pass of all internal neural layers, the buffer of results for each object will store prepared information about the state of all gates and the normalized data of the new content. Now all we have to do is to combine all information flows according to the algorithm of the *LSTM* block. Before constructing this process, we need to organize the branching of the algorithm depending on the utilized device for computational operations: CPU using standard MQL5 tools or OpenCL context.

A reasonable question may arise: why are we separating transaction threads only now? Why didn't we utilize the power of multi-threaded operations when calculating gate state and new context? But believe me, these operations also utilized the technology of multi-threaded computations offered by OpenCL, although not as explicitly. For example, in the *CNeuronLSTM::SetOpenCL* method, we passed a pointer to the OpenCL context to all the internal neural layers, and just a few lines above, we called the feed-forward pass methods for each internal layer. And now take a look at the forward pass method of the parent class *CNeuronBase::FeedForward*, there is also thread division present there.

```
bool CNeuronBase::FeedForward(CNeuronBase *prevLayer)
{
    ....
    --- Branching the algorithm by the computing device
    if(!m_cOpenCL)
    {
```



```

.....
}
else
{
.....
}
//---
return false;
}

```

In other words, we have previously used ready-made methods of the base class of neural layers with ready-made functionality in both directions. We will now introduce additional operations that are unique to the *LSTM* block. Therefore, we need to split the thread of operations and organize the process for both technologies. Just as when building the previous classes, we will now go through the process of constructing the algorithm in MQL5. We will delve into the actual process organization within the context of OpenCL in the next chapter.

When performing operations using MQL5, we will first obtain pointers to the data buffers with the results of internal neural layers in local variables for ease of access. Then we will use the matrix operations of MQL5.

First, we multiply element-wise the *Memory* state by the *Forget Gate* values. We then multiply the normalized matrix of new content (*New Content*) by the *Input Gate*, step by step. The result is added to the updated memory state (*Memory*). In conclusion, we normalize the results of the operations performed above using the hyperbolic tangent function and then element-wise multiply them with the matrix of gate results (*Output Gate*). The result is written to the hidden state buffer (*Hidden*).

```

--- branching of the algorithm by the computing device
CBufferType *fg = m_cForgetGate.GetOutputs();
CBufferType *ig = m_cInputGate.GetOutputs();
CBufferType *og = m_cOutputGate.GetOutputs();
CBufferType *nc = m_cNewContent.GetOutputs();
if(!m_cOpenCL)
{
    memory.m_mMatrix *= fg.m_mMatrix;
    memory.m_mMatrix += ig.m_mMatrix * nc.m_mMatrix;
    hidden.m_mMatrix = MathTanh(memory.m_mMatrix) * og.m_mMatrix;
}

```

For the OpenCL context algorithm, we temporarily set an exit with a negative result, which will later be replaced by the correct code. This option will allow us to test the ready code and warn us about choosing an incorrect parameter.

```

else
{
    delete inputs;
    delete memory;
    delete hidden;
    return false;
}

```

After completing the loop that updates the full memory and hidden state of our *LSTM* block, we transfer the hidden state values to the result buffer.

4. Basic types of neural layers

```
--- copy the hidden state to the neural layer results buffer
m_cOutputs = hidden;
```

This could be the end of the feed-forward pass. However, we still need to save the current state for the subsequent training of our recurrent block. First, we save the initial data to the stack. As mentioned above, we insert new objects into the stack with an index of 0.

```
--- save the current state
if(!m_cInputs.Insert(inputs, 0))
{
    delete inputs;
    delete memory;
    delete hidden;
    return false;
}
```

After adding a new element, check the stack for overflow and remove excessive historical data. To perform this functionality, we create the *ClearBuffer* method. We will look at the algorithm of this method a little later.

```
ClearBuffer(m_cInputs);
```

Here it should be mentioned that we store the source data in the form of a neural layer. This allows us to solve two problems at once:

1. The feed-forward and backpropagation methods for the base neural layer require a pointer to the previous neural layer as input. Consequently, a single object can be used for both the feed-forward and backpropagation passes without any modifications to the base neural layer.
2. In one object, we store both the raw data buffer and the gradient buffer. We do not need to configure synchronization for buffer utilization.

In the remaining stacks, we will store buffers. Therefore, we will create an additional *InsertBuffer* method for the repetitive work of saving data to the stacks. We will take a look at the algorithm of the method a bit later, and for now, we will use it to copy information into the stacks. We will repeat the call of the specified method for each stack and the corresponding buffer.

```
if(!InsertBuffer(m_cForgetGateOuts, m_cForgetGate.GetOutputs(), false))
{
    delete memory;
    delete hidden;
    return false;
}

if(!InsertBuffer(m_cInputGateOuts, m_cInputGate.GetOutputs(), false))
{
    delete memory;
    delete hidden;
    return false;
}

if(!InsertBuffer(m_cOutputGateOuts, m_cOutputGate.GetOutputs(), false))
{
    delete memory;
    delete hidden;
    return false;
}
```

```

    }

    if(!InsertBuffer(m_cNewContentOuts, m_cNewContent.GetOutputs(), false))
    {
        delete memory;
        delete hidden;
        return false;
    }

```

Note that above, we saved the buffers of results from internal layers. These objects belong to the neural layer structure and will be deleted from memory together when the corresponding neural layer is deleted. Therefore, in the *InsertBuffer* method, we will not create a new instance of the buffer object and copy the data.

Here, it's crucial to have a clear understanding of the differences between a pointer to an object and the object itself. Every time we create an object, a certain amount of memory is allocated for it. The necessary information is recorded there. This is our object. A pointer to the object is saved to access it. It contains a reference to the memory area where the object is stored. Consequently, when accessing the object, we take the pointer, navigate to the desired memory location, and read the necessary information.

When we copy a pointer to an object, we don't create a new object, we just make a copy of the reference. Therefore, when someone makes changes to the content of the object, we will also see these changes by accessing the object through our pointer. Whether this is good or bad depends on the method of using the object. When we need synchronization of operations with an object from different sources, that's a good thing. Everyone will refer to the same object. This means there is no need to synchronize data in different storages. Moreover, a pointer requires fewer resources than creating a new object. But when we need to protect some data against changes, it is better to create a new object and copy the necessary information.

```

    if(!InsertBuffer(m_cMemorys, memory, false))
    {
        delete hidden;
        return false;
    }

    if(!InsertBuffer(m_cHiddenStates, hidden, false))
        return false;
    //---
    return true;
}

```

After successfully saving all the necessary information in the stack, we exit the method with a positive result.

Congratulations! We've reached the end of the forward pass method. It may not be the simplest, but I hope my comments have helped you understand its algorithm and the idea behind the process. However, we still have some open questions in the form of auxiliary methods.

As we considered the algorithm of the feed-forward pass method, we mentioned the *ClearBuffer* method. Here, everything is quite simple and straightforward. The method receives a pointer to the stack in the parameters. As always, at the beginning of the method, we check the validity of the received pointer. After successfully passing the pointer check, we verify the buffer size. If the size of the buffer exceeds the user-specified size, we delete the last elements. By doing so, we ensure that the

buffer size fits within the specified limits. As you can see, the whole code of the method fits literally into five lines.

```
void CNeuronLSTM::ClearBuffer(CArrayObj *buffer)
{
    if(!buffer)
        return;
    int total = buffer.Total();
    if(total > m_iDepth + 1)
        buffer.DeleteRange(m_iDepth + 1, total);
}
```

Then we discussed the *InsertBuffer* method that adds a buffer to the stack. This method has three parameters, the last of which has a default value and is not mandatory to specify when calling the method:

- *CArrayObj *array* – a pointer to the stack for adding a buffer.
- *CBufferType *element* – a pointer to the buffer to be added.
- *bool create_new* – a logical variable indicating the need to create a duplicate buffer. By default, a duplicate buffer is created.

As a result of the operations, the method returns a boolean value indicating the status of the operations.

As always, at the beginning of the method, we check if the obtained pointers are valid. Here, there is one nuance. First, we check the pointer to the buffer to be added to the stack. With an invalid pointer, we have nothing to add to the stack. Naturally, in such a situation, we exit the method with a negative result.

However, if the pointer to the stack turns out to be invalid, we will first attempt to create a new stack. Only after an unsuccessful attempt, we will exit the method with a negative result. But if we manage to create a new stack, we will continue working in the standard mode.

```
bool CNeuronLSTM::InsertBuffer(CArrayObj *&array,
                              CBufferType *element,
                              bool create_new = true)
{
    //--- control block
    if(!element)
        return false;
    if(!array)
    {
        array = new CArrayObj();
        if(!array)
            return false;
    }
}
```

Next, we split the algorithm into two separate branches depending on whether a duplicate buffer needs to be created. If a duplicate buffer is needed, we first create a new instance of the buffer object and immediately check the result of the operation using the obtained pointer to the object.

```
if(create_new)
{
    CBufferType *buffer = new CBufferType();
```

```

    if(!buffer)
        return false;

```

Then we transfer the contents of the source buffer to the new buffer. Only after that, we will add the pointer to the new buffer to the stack. Again, we add new elements to the stack with an index of 0.

```

    buffer.m_mMatrix = element.m_mMatrix;
    if(!array.Insert(buffer, 0))
    {
        delete buffer;
        return false;
    }
}

```

If we don't need to create a new instance of the buffer, then things are much simpler here. We simply add the pointer to the buffer received as a parameter to the stack.

```

else
{
    if(!array.Insert(element, 0))
    {
        delete element;
        return false;
    }
}

```

After adding a new element to the stack, we will check its size and remove excessive history. For this, we will use the *ClearBuffer* method.

```

    --- remove unnecessary history from the buffer
    ClearBuffer(array);
    //---
    return true;
}

```

After the operations are complete, we exit the method with a positive result.

We have thoroughly covered the feed-forward pass algorithm and the methods involved in it. Next, let's consider the backpropagation pass.

4.2.2.2 Backpropagation methods for LSTM block

The feed-forward pass represents the standard mode of operation of a neural network. However, before it can be used in real-life operations, we need to train our model. Recurrent neural networks are trained using the familiar backpropagation method with a slight addition. The reason is that, unlike the neural layer types we've discussed before, only recurrent layers use their own output as their input on future iterations. Also, they all have their own weights that need to be learned as well. In the learning process, we have to unfold the recurrent layers chronologically as a multilayer perceptron. The only difference is that all layers will use the same weight matrix. Precisely for this purpose, during the feed-forward pass, we kept a record of the state history of all objects. Now it's time to put them to good use.

We have three methods responsible for the backward pass in the base class of the neural layer:

- *CalcHiddenGradient* – a gradient distribution through a hidden layer.
- *CalcDeltaWeights* – a distribution of the gradient to the weighting matrix.
- *UpdateWeights* – the method of updating the weights.

```
class CNeuronLSTM      : public CNeuronBase
{
protected:
    ....
public:
    ....
    virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
    virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer)  override
                                                                { return true; }
    virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                    VECTOR &Beta, VECTOR &Lambda) override;
```

We have to redefine them.

First, we will override the *CalcHiddenGradient* method for distributing the gradient through the hidden layer. Here we will need to unwrap the entire historical chain and run the error gradient through all states. Additionally, let's not forget that besides distributing gradients within the LSTM block, we must also perform the second function of this method: propagating the gradient of the error back to the previous layer.

The method receives a pointer to the object of the previous layer and returns a boolean result indicating the success of the operations.

At the beginning of the method, we check all the objects used. We check both pointers to objects of the previous layer and internal objects received in the parameters.

```
bool CNeuronLSTM::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    //--- check the relevance of all objects
    if(!prevLayer || !prevLayer.GetGradients() ||
        !m_cGradients || !m_cForgetGate || !m_cForgetGateOuts ||
        !m_cInputGate || !m_cInputGateOuts || !m_cOutputGate ||
        !m_cOutputGateOuts || !m_cNewContent || !m_cNewContentOuts)
        return false;
```

Let's not forget that a backpropagation pass is only possible after a feed-forward pass. The foundation of source data for the backpropagation pass is established exactly during the feed-forward pass.

Therefore, the next step is to check for the presence of information in the memory stacks and hidden states. In addition, the stack filling indicates the depth of gradient propagation in the story.

```
//--- check the presence of forward pass data
int total = (int)fmin(m_cMemorys.Total(), m_cHiddenStates.Total()) - 1;
if(total <= 0)
    return false;
```

Continuing the preparatory work, let's create pointers to the result and gradient buffers of the internal layers. I think the need for pointers to gradient buffers is obvious. We will need to write error gradients to them, propagating them through the *LSTM* block. The need for result buffers, on the other hand, is not so obvious. As you know, every neuron has an activation function. Our inner layers are activated by the [logistic](#) function and by the [hyperbolic tangent](#). The error gradient obtained at the input of the neural layer must be adjusted to the derivative of the activation function. The derivative of the above activation functions can be easily recalculated based on the result of the function itself. Thus, we need the appropriate input data to perform a correct backpropagation pass. For the previously considered neural layers, such an issue was not raised because the correct data were written to the result buffer in a forward pass. In the case of a recurrent block, only the result of the last forward pass will be stored in the result buffer. To work out the depth of the history, we will have to overwrite the values of the result buffer with the values of the corresponding time step.

```
//--- make pointers to buffers of gradients and results of internal layers
CBufferType *fg_grad = m_cForgetGate.GetGradients();
if(!fg_grad)
    return false;
CBufferType *fg_out = m_cForgetGate.GetOutputs();
if(!fg_out)
    return false;

CBufferType *ig_grad = m_cInputGate.GetGradients();
if(!ig_grad)
    return false;
CBufferType *ig_out = m_cInputGate.GetOutputs();
if(!ig_out)
    return false;

CBufferType *og_grad = m_cOutputGate.GetGradients();
if(!og_grad)
    return false;
CBufferType *og_out = m_cOutputGate.GetOutputs();
if(!og_out)
    return false;

CBufferType *nc_grad = m_cNewContent.GetGradients();
if(!nc_grad)
    return false;
CBufferType *nc_out = m_cNewContent.GetOutputs();
if(!nc_out)
    return false;
```

At the end of the preparatory process, we will store the size of the internal thread buffers into a local variable.

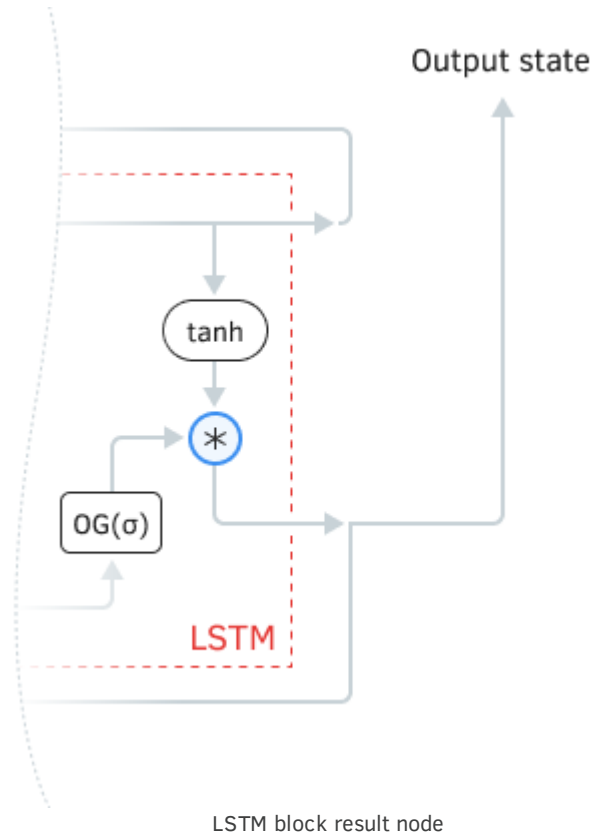
```
uint out_total = m_cOutputs.Total();
```

Next, we create a loop through historical data. The main operations of our method will be performed in the body of this loop. At the beginning of the loop, we will load information from the corresponding historical step in our stacks. Note that all buffers are loaded for the analyzed chronological step, while the memory buffer is taken from the preceding step. I will explain the reasons for this below.

```
//--- loop through the accumulated history
for(int i = 0; i < total; i++)
{
    //--- get pointers to buffers from the stack
    CBufferType *fg = m_cForgetGateOuts.At(i);
    if(!fg)
        return false;
    CBufferType *ig = m_cInputGateOuts.At(i);
    if(!ig)
        return false;
    CBufferType *og = m_cOutputGateOuts.At(i);
    if(!og)
        return false;
    CBufferType *nc = m_cNewContentOuts.At(i);
    if(!nc)
        return false;
    CBufferType *memory = m_cMemorys.At(i + 1);
    if(!memory)
        return false;
    CBufferType *hidden = m_cHiddenStates.At(i);
    if(!hidden)
        return false;
    CNeuronBase *inputs = m_cInputs.At(i);
    if(!inputs)
        return false;
}
```

Next, we have to distribute the error gradient received at the input of the *LSTM* block between the internal neural layers. This is where we build a new process. Following our class construction concept, we create a branching of the algorithm based on the execution device for mathematical operations.

The error gradient distribution is performed in reverse order of the forward flow of information. Hence, we will construct its propagation algorithm from output to input. Let's look at the result node of our *LSTM* block. During the feed-forward pass, the updated memory state is activated by the hyperbolic tangent and multiplied by the output gate state. Thus, we have two components affecting the result of the block: the memory value and the gate.



In order to reduce the error at the block output, we need to adjust the values of both components. To do this, we need to distribute the overall error gradient through a multiplication function that combines the two threads of information. That is, multiply the error gradient we know by the derivative of the function along each direction. We know from our high school math course that the derivative of the product of a constant over a variable is a constant. We apply the following approach: when determining the influence of one of the factors, we assume that all other components have constant values. Hence, we can write the following mathematical formulas.

$$OUT = OG * \tanh(Mem)$$

$$\frac{\partial Out}{\partial OG} = \tanh(Mem)$$

$$\frac{\partial Out}{\partial \tanh(Mem)} = OG$$

Then we can easily distribute the derivative in both directions using the following mathematical formulas.

$$Grad_{OG} = Grad_{out} * \tanh(Mem)$$

$$Grad_{\tanh(Mem)} = Grad_{out} * OG$$

We haven't created a separate buffer for the activated memory state. However, we can easily count it by re-activating the corresponding state or by dividing the hidden state by the output gate value. I chose the second path, and the entire algorithm for distributing the error gradient at this site is expressed in the following code.

```
//--- branching of the algorithm across the computing device
```

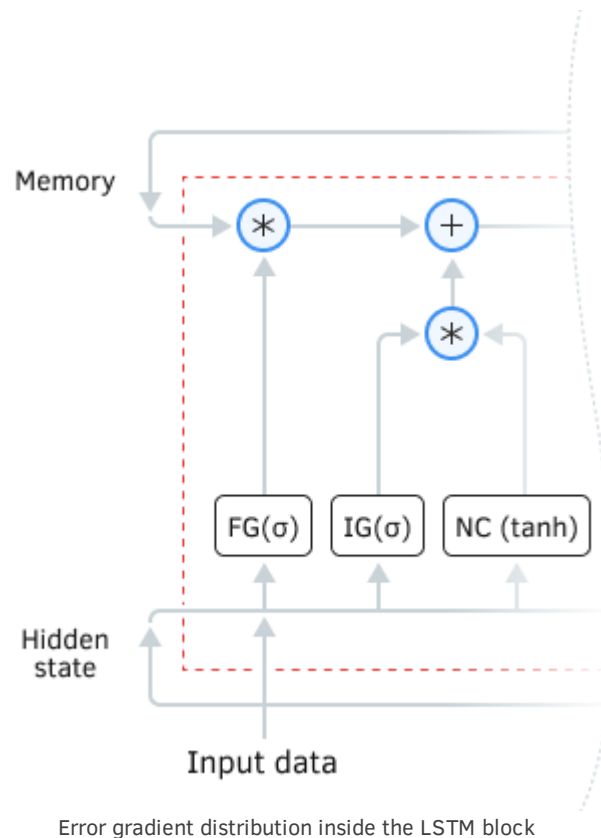
4. Basic types of neural layers

```
if(!m_cOpenCL)
{
    //--- calculate the gradient at the output of each internal layer
    MATRIX m = hidden.m_mMatrix / (og.m_mMatrix + 1e-8);
    //--- OutputGate gradient
    MATRIX grad = m_cGradients.m_mMatrix;
    og_grad.m_mMatrix = grad * m;
    //--- memory gradient
    grad *= og.m_mMatrix;
```

Before distributing the memory gradient to the rest of the internal layers, we must correct the resulting gradient by the derivative of the activation function.

```
//--- adjust the gradient to the derivative
grad *= MathPow(m, 2) * (-1) + 1;
```

We continue to distribute the error gradient between the internal layers. We need to distribute the error gradient from the memory flow to three more internal layers. Moving along the information flow inside the *LSTM* block in reverse, the first function we encounter is summation. The derivative of the sum is 1. Therefore, we pass the error gradient in both directions unchanged.



Next, in both directions, we encounter the product. The principles of propagating the gradient through the multiplication of two numbers have been explained in detail above, so there is no need to repeat them. I just want to remind you that, unlike all buffers from the stack, only the memory buffer was taken one step further back in history. I promised to clarify this point, and now is the most suitable time to do so. Take a look at the *LSTM* block diagram. To refresh memory, we multiply the output of the Forget gate by the memory state transferred from the previous iteration. Hence, to determine the error gradient at the output of the Forget gate, we need to multiply the error gradient in the memory

thread by the memory state of the previous iteration. It is the buffer of this state that we loaded at the start of the loop

The MQL5 code of the described operations is presented below.

```
//--- InputGate gradient
ig_grad.m_mMatrix = grad * nc.m_mMatrix;
//--- NewContent gradient
nc_grad.m_mMatrix = grad * ig.m_mMatrix;
//--- ForgetGates gradient
fg_grad.m_mMatrix = grad * memory.m_mMatrix;
}
```

This completes the thread separation block by computational operation unit, and we merge the threads of the algorithm. We set a stub for the OpenCL branch and move on.

```
else
{
    return false;
}
```

We have already discussed the need to use the historical states of the inner layer result buffers. Now we need to put this into practice and fill the result buffers with relevant historical data.

```
//--- copy the corresponding historical data to the buffers of the internal lay
if(!m_cForgetGate.SetOutputs(fg, false))
    return false;
if(!m_cInputGate.SetOutputs(ig, false))
    return false;
if(!m_cOutputGate.SetOutputs(og, false))
    return false;
if(!m_cNewContent.SetOutputs(nc, false))
    return false;
```

Next, we need to propagate the gradient from the output to the input of the internal neural layers. This functionality is easily implemented by the base class method. However, please note the following. All four internal neural layers use the same input data. We also need to put the error gradient together in the same buffer. The neural layer base class methods we developed earlier are constructed in such a way that they overwrite values. Therefore, we need to organize the process of summing the error gradients from each internal neural layer.

First, we'll run a gradient through the Forget gate. Recall that in order to transfer the source data to the internal neural layers, we created a base layer of source data and after performing forward pass operations, we stored a pointer to it in the source data stack. This type of object already contains buffers for writing data and error gradients. So, now we just take this pointer and pass it in the parameters of the *CNeuronBase::CalcHiddenGradient* method. After this, our base class method will execute and fill the error gradient buffer at the source data level for the forget gates. But it's only one gate, and we need to gather information from all of them. To avoid losing the computed error gradient when calling a similar method for other internal layers, we will copy the data into the *m_cInputGradient* buffer which we created in advance for accumulating error gradients.

```
//--- propagate a gradient through the inner layers
if(!m_cForgetGate.CalcHiddenGradient(inputs))
    return false;
```

4. Basic types of neural layers

```
if(!m_cInputGradient)
{
    m_cInputGradient = new CBufferType();
    if(!m_cInputGradient)
        return false;
    m_cInputGradient.m_mMatrix = inputs.GetGradients().m_mMatrix;
    m_cInputGradient.BufferCreate(m_cOpenCL);
}
else
{
    m_cInputGradient.Scaling(0);
    if(!m_cInputGradient.SumArray(inputs.GetGradients()))
        return false;
}
```

We repeat the operations for the remaining internal layers. However, now we add the new values of the error gradient to the previously obtained values.

```
if(!m_cInputGate.CalcHiddenGradient(inputs))
    return false;
if(!m_cInputGradient.SumArray(inputs.GetGradients()))
    return false;
if(!m_cOutputGate.CalcHiddenGradient(inputs))
    return false;
if(!m_cInputGradient.SumArray(inputs.GetGradients()))
    return false;
if(!m_cNewContent.CalcHiddenGradient(inputs))
    return false;
if(!inputs.GetGradients().SumArray(m_cInputGradient))
    return false;
```

Please note the following. While processing the first three internal layers we move values into the temporary buffer *m_cInputGradient*. However, while processing the last layer, we transfer the previously accumulated error gradient into the source data layer buffer. Thus, we keep the overall error gradient at the initial data layer along with the initial data itself in the same initial data layer. It will also be automatically saved in our stack. Recall what I wrote about objects and pointers to them.

Here comes an interesting moment. Remember, why we did all this? Propagation of the error gradient across all elements of the neural network is necessary to have a reference for determining the direction and extent of weight matrix element adjustments to reduce the overall error of our neural network performance. Consequently, as a result of the operations of this method, we must:

- Bring the error gradient to the previous layer, and
- Bring the error gradient to the weight matrices of the internal neural layers.

If we run the next iteration cycle in this state with new data for recalculating the error gradients of internal layers, we will simply replace the just-calculated values. However, we need to propagate the error gradients all the way to the weight matrices of the internal neural layers. Therefore, without waiting for a call from an external program, we call the *CNeuronBase::CalcDeltaWeights* method for all internal layers, which will recalculate the gradient at the weight matrix level.

```
//--- project the gradient onto the weight matrices of the internal layers
if(!m_cForgetGate.CalcDeltaWeights(inputs))
    return false;
```

4. Basic types of neural layers

```
if(!m_cInputGate.CalcDeltaWeights(inputs))
    return false;
if(!m_cOutputGate.CalcDeltaWeights(inputs))
    return false;
if(!m_cNewContent.CalcDeltaWeights(inputs))
    return false;
```

We pass the error gradient only from the current state to the previous neural layer. Historical data remains only for the internal user of the *LSTM* block. Therefore, we check the iteration index and only then pass the error gradient to the buffer of the previous layer. Do not forget that our error gradient buffer at the source data level contains more data than the buffer of the previous layer. This is because it also contains the error gradient of the hidden state. Hence, we will transfer only the necessary part of the data to the previous layer.

We transfer the remainder to the error gradient buffer of our *LSTM* block. Remember, at the beginning of the loop, it was from this buffer that we took the error gradient to propagate throughout the *LSTM* block? It's time to prepare the initial data for the next iteration of our loop through the chronological iterations of the feed-forward pass and error gradient propagation.

```
//--- if the gradient of the current state is calculated, then transfer it to the
//--- and write the hidden state gradient to the gradient buffer for a new iteration
if(!inputs.GetGradients().Split((i == 0 ? prevLayer.GetGradients() :
                                inputs.GetGradients()), m_cGradients,
                                prevLayer.GetOutputs().Total()))

    return false;
}
//---
return true;
}
```

After the successful execution of all iterations, we exit the method with a positive result.

We have gone through two of the most complex and intricate methods for constructing a recurrent *LSTM* block algorithm. The rest of it will be much easier. For example, the *CalcDeltaWeights* method. The functionality of this method involves passing the error gradient to the level of the weight matrix. The *LSTM* block does not have any separate weight matrix. All parameters are located within the nested neural layers, and we have already brought the error gradient to the level of their weight matrices in the previous method. Therefore, we rewrite the method with an empty stub with a positive result.

```
virtual bool CalcDeltaWeights(CNeuronBase *prevLayer) { return true; }
```

Another backward pass method, *UpdateWeights*, is a method for updating the weights matrix. The method is also inherited from the neural layer base class and overridden as needed. *LSTM* block unlike the previously discussed types of neural layers does not have a single weight matrix. Instead, internal neural layers with their own weight matrices are used. So we can't just use the method of the parent class and have to override it.

The *CNeuronLSTM::UpdateWeights* method from an external program receives the parameters required to execute the algorithm for updating the weight matrix and returns the logical value of the result of the method operations.

Even though the method parameters do not include any object pointers, we still set up control structures at the beginning of the method. Here, we check the validity of pointers to internal neural

layers and the value of the parameter indicating the depth of history analysis, which should be greater than 0.

```
bool CNeuronLSTM::UpdateWeights(int batch_size, TYPE learningRate, VECTOR &Beta,
                                VECTOR &Lambda)
{
    //--- check the state of objects
    if(!m_cForgetGate || !m_cInputGate || !m_cOutputGate ||
        !m_cNewContent || m_iDepth <= 0)
        return false;
}
```

Please note the *batch_size* parameter. This parameter indicates the number of backpropagation iterations between weight updates. It is tracked by an external program and passed to the method in parameters. For an external program and for the neural network types considered earlier, the number of feed-forward and backpropagation passes is equal, as each feed-forward pass is followed by a backpropagation pass, in which the deviation of the estimated neural network result from the expected result is determined and the error gradient is propagated throughout the neural network. In the case of a recurrent block, the situation is slightly different: for each feed-forward pass, a recurrent block undergoes multiple iterations of backward passes, determined by the depth of the analyzed history. Consequently, we must adjust the batch size received from the external program to the depth of the historical data.

```
int batch = batch_size * m_iDepth;
```

We can then use the methods to update the weight matrix by passing them the correct data in the parameters.

```
//--- update the weight matrices of the internal layers
if(!m_cForgetGate.UpdateWeights(batch, learningRate, Beta, Lambda))
    return false;
if(!m_cInputGate.UpdateWeights(batch, learningRate, Beta, Lambda))
    return false;
if(!m_cOutputGate.UpdateWeights(batch, learningRate, Beta, Lambda))
    return false;
if(!m_cNewContent.UpdateWeights(batch, learningRate, Beta, Lambda))
    return false;
//---
return true;
}
```

After successfully updating the weight matrices of all internal neural layers, we exit the method with a positive result.

This concludes our review of LSTM block backpropagation methods. We can move forward in building our system.

4.2.2.3 Saving and restoring the LSTM block

We have already looked at the methods for initializing the feed-forward and backpropagation passes of an LSTM block. This is enough for small-scale experiments but not enough for industrial use. One of the key requirements of practical application is the reusability of a once-trained neural network. We learned how to build and train our neural network. We can even get the results of applying it to real data. But we cannot yet save a trained *LSTM* block to restore it later from previously saved data. Two methods are provided in our neural layer classes to accomplish this functionality:

- *Save* saves the class.
- *Load* restores the class functions by previously saved data.

Before we start creating methods, let's look at the class structure of our LSTM block and determine which data we need to store and which we can simply initialize with initial values.

```
class CNeuronLSTM      : public CNeuronBase
{
protected:
    CNeuronBase*      m_cForgetGate;
    CNeuronBase*      m_cInputGate;
    CNeuronBase*      m_cNewContent;
    CNeuronBase*      m_cOutputGate;
    CArrayObj*        m_cMemorys;
    CArrayObj*        m_cHiddenStates;
    CArrayObj*        m_cInputs;
    CArrayObj*        m_cForgetGateOuts;
    CArrayObj*        m_cInputGateOuts;
    CArrayObj*        m_cNewContentOuts;
    CArrayObj*        m_cOutputGateOuts;
    CBufferType*      m_cInputGradient;
    int               m_iDepth;

    void              ClearBuffer(CArrayObj *buffer);
    bool              InsertBuffer(CArrayObj *array, CBufferType *element,
                                   bool create_new = true);
    CBufferType*      CreateBuffer(CArrayObj *array);

public:
    CNeuronLSTM(void);
    ~CNeuronLSTM(void);

    //---
    virtual bool      Init(const CLayerDescription *desc) override;
    virtual bool      SetOpenCL(CMyOpenCL *openc1) override;
    virtual bool      FeedForward(CNeuronBase *prevLayer) override;
    virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
    virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer) override
                                   { return true; }
    virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda) override;

    //---
    virtual int        GetDepth(void) const { return m_iDepth; }
    //--- methods for working with files
```

4. Basic types of neural layers

```
virtual bool      Save(const int file_handle) override;
virtual bool      Load(const int file_handle) override;
//--- method of object identification
virtual int       Type(void) override const { return(defNeuronLSTM); }
};
```

First of all, we understand that no constants or methods change during the class operation. Therefore, we will only store variables.

When declaring the class variables, the first variables we declared were those for storing pointers to the internal neural layers. Of course, it makes absolutely no sense to save pointers to class objects. However, we must save the contents of these objects because the trained weight matrices are stored in them.

Next, we declared pointers to stacks of chronological data. The stacks themselves, as well as their contents, are of no value to us when saving the data. Stacks are dynamic array objects that will be effortlessly recreated. Regarding their contents, the situation is as follows. For recurrent networks, the sequence of data and the absence of gaps are crucial. At the time when the data is saved, we do not understand when the data will be reused. Consequently, at the time of data loading, it is very likely that there are gaps between the current state of the analyzed system and the data at the time of saving. In such a situation, their use will not only be unhelpful but on the contrary will distort the results. Therefore, saving this data would only increase the amount of stored information without providing any benefit for later use.

The error gradient accumulation buffer *m_cInputGradient* is an auxiliary object for accumulating data and is overwritten with new data during each backpropagation pass. It does not contain information important for subsequent iterations and is not appropriate for saving.

The last global variable we declared is the depth of the analyzed chronological iterations: *m_iDepth*. It is a component of the architectural block design and is to be preserved.

After defining the scale of the work, we can proceed to its execution. First, we create the *CNeuronLSTM::Save* method to save the data. In the parameters, the method gets the handle of the file for saving the data. However, we will not organize a control unit to check the incoming parameters as usual. Instead of that, we will pass the received parameter to a similar method in the base class, where all the necessary controls are already implemented. Besides, earlier we analyzed only the variables declared in the class of the *LSTM* block but did not evaluate the need to preserve the contents of the parent class. However, we did this work when creating the method of saving data of the base class. Therefore, by calling the method of the parent class, we perform both functionalities in one line of code.

```
bool CNeuronLSTM::Save(const int file_handle)
{
//--- calling a method of the parent class
if(!CNeuronBase::Save(file_handle))
    return false;
```

After the successful execution of the parent class method, we save the value of the depth of the analyzed chronological iterations.

```
//--- saving the constants
if(FileWriteInteger(file_handle, m_iDepth) <= 0)
    return false;
```


After this, we only need to save the contents of the internal neural layers. For this purpose, we will also utilize the functionality of the underlying neural layer. We just need to call the save method for each of our internal layers, providing the file handle for writing data that we received as a parameter from the external program. At the same time, we will not forget to control the process of operations at each step.

```
//--- call the same method for all inner layers
if(!m_cForgetGate.Save(file_handle))
    return false;
if(!m_cInputGate.Save(file_handle))
    return false;
if(!m_cOutputGate.Save(file_handle))
    return false;
if(!m_cNewContent.Save(file_handle))
    return false;
//---
return true;
}
```

After successful completion of all operations, we will exit the method with a positive result.

Now I suggest looking at the whole code of the data-saving method again and evaluating how concise and readable it is. This effect is achieved through the use of object-oriented programming (*OOP*). Creating classes significantly reduces code and speeds up the work of the programmer, while using ready-made and tested libraries helps avoid many errors. Believe me, no matter how complex creating our library might seem, using it will make it easy and without significant effort for the programmer to create their own neural networks. Moreover, you don't need to be a highly qualified programmer to do it.

But I digress. We have created a method to save the data. Now, we need to build the process of restoring the functionality of our recurrent block from the saved data.

The data loading method *CNeuronLSTM::Load* is constructed in clear correspondence with the data saving method. The saved data must be loaded from the file in the same sequence, otherwise, we could encounter distorted data or loading errors.

In the parameters, the method gets the handle of the data file to load. Just like when saving data, instead of setting up a control block, we call the method of the parent class. It already implements all the necessary controls and data loading of the parent class.

```
bool CNeuronLSTM::Load(const int file_handle)
{
//--- call a method of the parent class
if(!CNeuronBase::Load(file_handle))
    return false;
}
```

Next, we load the depth of the analyzed chronological iterations and the contents of the internal neural layers from the file. We will also use the methods of the neural layer base class to perform the latter operations. And, as always, we will check the results of the operations.

But here, we need to pay attention to one significant detail. The method for saving the base neural layer *CNeuronBase::Save* begins with writing the type of object to be saved. We read its value in the neural network loading dispatcher method to determine the type of object to be created. Hence, in the neural layer loading method, we start reading the file from the next element. In this case, to maintain

the sequence of loading data from the file, we must first read the type of the next neural layer and only then call the loading method of the corresponding internal neural layer. Besides, this can be an additional point of control for loading the correct type of internal neural layer.

```
//--- read the constants
m_iDepth = FileReadInteger(file_handle);
//--- call the same method for all inner layers
if(FileReadInteger(file_handle) != defNeuronBase ||
    !m_cForgetGate.Load(file_handle))
    return false;
if(FileReadInteger(file_handle) != defNeuronBase ||
    !m_cInputGate.Load(file_handle))
    return false;
if(FileReadInteger(file_handle) != defNeuronBase ||
    !m_cOutputGate.Load(file_handle))
    return false;
if(FileReadInteger(file_handle) != defNeuronBase ||
    !m_cNewContent.Load(file_handle))
    return false;
```

After loading the data from the file, we need to initialize the remaining objects with the initial values. First, we initialize the memory stack and add a buffer with initial values to it. To do this, we will use the *CreateBuffer* method we already know. I'd like to remind you that this method only creates a buffer with zero values for an empty stack. Otherwise, the method will return the last buffer written. Therefore, before calling the method, we check the size of the stack: if the stack contains data, we clear the stack and set all buffer values to zero.

```
//--- initialize Memory
if(m_cMemorys.Total() > 0)
    m_cMemorys.Clear();
CBufferType *buffer = CreateBuffer(m_cMemorys);
if(!buffer)
    return false;
if(!m_cMemorys.Add(buffer))
    return false;
```

After all operations are completed, we will add the newly created buffer to the stack. Then we will repeat the same operations for the stack and the hidden state buffer.

```
//--- initialize HiddenStates
if(m_cHiddenStates.Total() > 0)
    m_cHiddenStates.Clear();
buffer = CreateBuffer(m_cHiddenStates);
if(!buffer)
    return false;
if(!m_cHiddenStates.Add(buffer))
    return false;
```

We built the forward pass method in such a way that it is not critical for us to create and initialize the other stacks now. However, we acknowledge that the data loading operation might be performed on a working neural network, where the stacks already hold some information. In such cases, using data from stacks created with different weights would be incorrect. Therefore, we will clear all previously created stacks.

```

//--- clear the rest of the stacks
    if(!m_cInputs)
        m_cInputs.Clear();
    if(!m_cForgetGateOuts)
        m_cForgetGateOuts.Clear();
    if(!m_cInputGateOuts)
        m_cInputGateOuts.Clear();
    if(!m_cNewContentOuts)
        m_cNewContentOuts.Clear();
    if(!m_cOutputGateOuts)
        m_cOutputGateOuts.Clear();
//---
    return true;
}

```

Once all operations of the method have been successfully executed, we terminate the method with a positive result.

At this point, we complete the construction of recurrent *LSTM* block by means of *MQL5* and move on to complementing the methods of our class with the ability to perform multi-threaded operations.

4.2.3 Organizing parallel computing in the *LSTM* block

In previous chapters, we looked at the implementation of an *LSTM* block using *MQL5*. However, a new stage in the development of neural networks came precisely with the development of parallel computing technologies. This is especially important for resource-intensive tasks such as recurrent neural networks. Therefore, it is especially important for us to add the ability to use multi-threaded parallel computing tools in the *LSTM* block class.

As mentioned when creating a block algorithm using *MQL5*, our class already has an implementation of multi-threaded calculations of individual blocks thanks to the use of objects of the previously discussed class of the base neural layer as gates in the *LSTM* block algorithm. Therefore, within the framework of this chapter, we only have to implement the missing part:

- Thread consolidating and processing data from internal neural layers within the forward pass.
- Propagating the error gradient from the output of the *LSTM* block to the internal neural layers within the backpropagation pass.

This gives us an understanding of the task. We already have an *MQL5* implementation of the process. This gives an understanding of the process and the algorithm for executing operations.

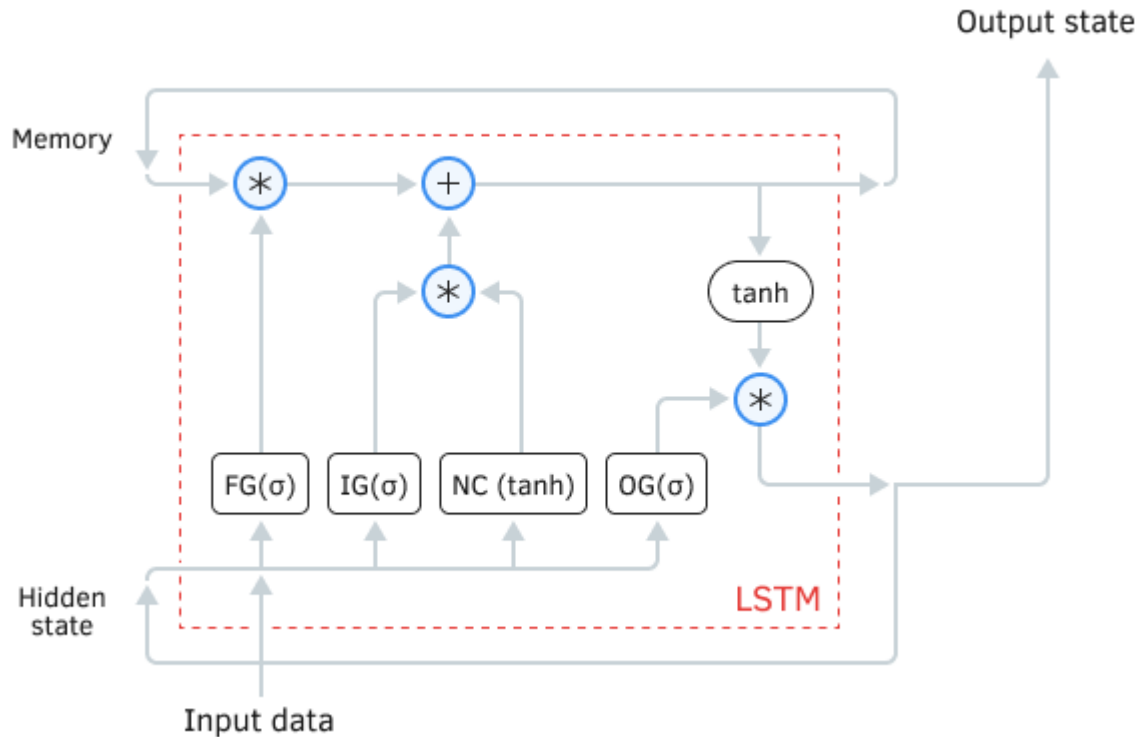
Therefore, we can proceed with the work. Let me remind you of the architecture for constructing a multi-threaded computing process. The actual execution of the computation process in parallel threads is carried out in an environment different from the main program – in the *OpenCL* context. To perform operations, three main components are required:

1. Program of performed operations.
2. Initial data for performing operations.
3. Process control commands (moment of program launch, number of threads created, etc.)

Let's look at the implementation of these points.

4.2.3.1 Making additions to the OpenCL program

The first item we indicate is the program of the operations being performed. This means that we need to augment our OpenCL program with new kernels to perform the additional operations we require. We collected all the code of the OpenCL program in the file [opencl_program.cl](#). Open this file and add two new kernels to it: *LSTMFeedForward* and *LSTMCalcHiddenGradient*. The names of the kernels correspond to the names of the methods of our classes. Therefore, it is easy to guess that the first will complement the feed-forward pass method, and the second will complement the error gradient backpropagation method.



Recurrent LSTM block diagram

We start with the feed-forward pass kernel *LSTMFeedForward*. In the parameters, this buffer will receive pointers to six data buffers (four source data buffers and two result buffers) and one constant:

- *forgetgate*: pointer to the forget gate buffer (source data)
- *inputgate*: pointer to the input gate buffer (source data)
- *outputgate*: pointer to the result gate buffer (source data)
- *newcontent*: pointer to the new content buffer (source data)
- *memory*: pointer to a memory stream (result buffer)
- *hiddenstate*: pointer to the hidden state stream (result buffer)
- *outputs_total*: number of elements in the data stream (constant)

```
__kernel void LSTMFeedForward(__global TYPE *forgetgate,
                              __global TYPE *inputgate,
                              __global TYPE *outputgate,
                              __global TYPE *newcontent,
                              __global TYPE *memory,
                              __global TYPE *hiddenstate,
```

```
int outputs_total)
```

At the beginning of the method, as before, we receive the thread index, which serves as a pointer to the data being processed. Also, we immediately determine the shift in the data buffers to access the data we need.

```
{
    const int n = get_global_id(0);
    const int shift = n * 4;
```

To improve the performance of our program, we will use vector arithmetic. Let's use vector variables of type *TYPE4*. Let me remind you that we use the *TYPE* macro substitution to quickly switch *the double* or *float* data type used, depending on the requirements for calculation accuracy and the OpenCL device used. But before we begin performing operations, we will transfer the data from our global data buffers to local vector variables.

```
TYPE4 fg = ToVect4(forgetgate, shift, 1, outputs_total, 0);
TYPE4 ig = ToVect4(inputgate, shift, 1, outputs_total, 0);
TYPE4 og = ToVect4(outputgate, shift, 1, outputs_total, 0);
TYPE4 nc = ToVect4(newcontent, shift, 1, outputs_total, 0);
TYPE4 mem = ToVect4(memory, shift, 1, outputs_total, 0);
```

Now, by analogy with the MQL5 program code, we will perform arithmetic operations to update the state of the memory stream. According to the algorithm of *the LSTM* block, we must first adjust the incoming memory stream to the value of the oblivion gate, and then add a new context, adjusted by the value of the input gate, to the resulting value. After completing the operations, we return the value of the updated memory stream back to the buffer.

```
TYPE4 temp = mem * fg;
temp += ig * nc;
D4ToArray(memory, temp, shift, 1, outputs_total, 0);
```

Next, we need to define a new hidden state flow value. It will also be supplied to the output of the *LSTM* block for transmission to the next neural layer. Here we need to first normalize the current memory state using the hyperbolic tangent function and then adjust the resulting value by the result gate value. The result of the operations is written to the data buffer.

```
temp = tanh(temp) * og;
D4ToArray(hiddenstate, temp, shift, 1, outputs_total, 0);
}
```

The operations of the feed-forward kernel are now completed. From the results of the work of the internal layers of our recurrent *LSTM* block, we updated the state of the memory stream and obtained values that will be provided at the output of the recurrent block.

In the second kernel *LSTMCalcHiddenGradient*, we need to perform the reverse operation, that is, carry out the error gradient in the opposite direction, from the output of the recurrent block to the output of each internal neural layer. The specific operation of the backpropagation kernel requires an increase in the number of used data buffers to 10:

- *outputs*: pointer to the result vector buffer (source data)
- *gradients*: pointer to the gradient vector buffer of the current layer (source data)
- *inputgate*: pointer to the input gate buffer (source data)
- *outputgate*: pointer to the result gate buffer (source data)

- *newcontent*: pointer to the new content buffer (source data)
- *memory*: pointer to a memory stream (source data)
- *fg_gradients*: pointer to the oblivion gate gradient buffer (result buffer)
- *ig_gradients*: pointer to the input gate gradient buffer (result buffer)
- *og_gradients*: pointer to the result gate gradient buffer (result buffer)
- *nc_gradients*: pointer to the new content gradient buffer (result buffer)
- *outputs_total*: number of elements in the data stream (constant)

```
__kernel void LSTMCalcHiddenGradient(__global TYPE *outputs_
                                     __global TYPE *gradients,
                                     __global TYPE *inputgate,
                                     __global TYPE *outputgate,
                                     __global TYPE *newcontent,
                                     __global TYPE *memory,
                                     __global TYPE *fg_gradients,
                                     __global TYPE *ig_gradients,
                                     __global TYPE *og_gradients,
                                     __global TYPE *nc_gradients,
                                     int outputs_total)
```

At the beginning of the kernel, we determine the thread ID and the offset in the data buffers to the values being processed.

```
{
    const int n = get_global_id(0);
    int shift = n * 4;
```

As in the forward pass kernel, we will use operations with vector variables of type *TYPE4*. Therefore, in the next step, we transfer the original data from global buffers to local vector variables.

```
TYPE4 out = ToVect4(outputs, shift, 1, outputs_total, 0);
TYPE4 grad = ToVect4(gradients, shift, 1, outputs_total, 0);
TYPE4 ig = ToVect4(inputgate, shift, 1, outputs_total, 0);
TYPE4 og = ToVect4(outputgate, shift, 1, outputs_total, 0);
TYPE4 nc = ToVect4(newcontent, shift, 1, outputs_total, 0);
TYPE4 mem = ToVect4(memory, shift, 1, outputs_total, 0);
```

After completing the preparatory operations, we proceed to execute the mathematical part of the kernel. [Formulas](#) for carrying out operations and their explanation are presented when describing the construction of a process using MQL5. Therefore, in this section, only the implementation of the process in OpenCL will be given.

When implementing this part in MQL5, we decided that it was inappropriate to create an additional data buffer to store the normalized value of the memory stream. In the kernel parameters, we received a pointer to a stream not of the current memory state, but of a recurrent block arriving at the input from the previous iteration of the forward pass. Therefore, before proceeding with the error gradient distribution operations, we need to find the value of the normalized state of the memory stream. We define it as the ratio of the result buffer value to the result gate value. To eliminate division by zero, we add a small constant in the denominator.

```
TYPE4 m = out / (og + 1.0e-37f);
```

4. Basic types of neural layers

Following the logic of the error gradient backpropagation algorithm, we first determine the error gradient at the output of the oblivion gate neural layer. To do this, we need to multiply the error gradient at the output of our *LSTM* block by the derivative of the product. In this case, it is equal to the value of the normalized memory state. We will immediately write the resulting value into the corresponding data buffer.

```
//--- OutputGate gradient
    TYPE4 temp = grad * m;
    D4ToArray(og_gradients, temp, shift, 1, outputs_total, 0);
```

Next, we must similarly determine the error gradient with another multiplier, which is the normalized memory state. That is, we multiply the error gradient at the output of our recurrent block by the state of the results gate.

Before continuing to propagate the gradient to the remaining neural layers, we need to pass it through the [hyperbolic tangent](#) function. In other words, we multiply the previously obtained value by the derivative of the hyperbolic tangent.

$$\frac{df(x)}{dx} = (1 + f(x))(1 - f(x)) = 1 - (f(x))^2$$

```
//--- Adjust the memory gradient to the derivative TANH
    grad = grad * og * (1 - pow(m, 2));
```

Now we only need to propagate the error gradient across the remaining internal layers. The algorithm will be the same for all neural layers. The only difference is in the buffer used as a derivative of the multiplication function. After determining the error gradient, we immediately write its value into the appropriate buffer.

```
//--- InputGate gradient
    temp = grad * nc;
    D4ToArray(ig_gradients, temp, shift, 1, outputs_total, 0);
//--- NewContent gradient
    temp = grad * ig;
    D4ToArray(nc_gradients, temp, shift, 1, outputs_total, 0);
//--- ForgetGates gradient
    temp = grad * mem;
    D4ToArray(fg_gradients, temp, shift, 1, outputs_total, 0);
}
```

After completing the operations, we exit the kernel.

Thus, we implemented the missing kernels to organize forward and backward passes as part of performing operations for a recurrent *LSTM* block. This completes the modification of the OpenCL program, and we move on to performing operations on the side of the main program.

4.2.3.2 Implementing functionality on the side of the main program

After making changes to the OpenCL program, we must do the second part of the work and organize the process on the side of the main program. The first thing we will do is create constants for working with kernels. Here we need to create constants to identify kernels and their parameters. We will add the specified constants to those previously created in the file [defines.mqh](#).

4. Basic types of neural layers

```
#define def_k_LSTMFeedForward      26
#define def_k_LSTMHiddenGradients  27

//--- LSTM Feed Forward
#define def_lstmff_forgetgate      0
#define def_lstmff_inputgate      1
#define def_lstmff_outputgate     2
#define def_lstmff_newcontent     3
#define def_lstmff_memory         4
#define def_lstmff_hiddenstate    5
#define def_lstmff_outputs_total  6
```

When adding constants, we follow the previously defined naming rules. All kernel constants begin with the prefix *def_k_*, and parameter constants contain the kernel abbreviation: *def_lstmff_* for feed-forward kernel parameters and *def_lstmhgr_* for gradient backpropagation kernel parameters.

```
//--- LSTM Hidden Gradients
#define def_lstmhgr_outputs        0
#define def_lstmhgr_gradients     1
#define def_lstmhgr_inputgate     2
#define def_lstmhgr_outputgate    3
#define def_lstmhgr_newcontent    4
#define def_lstmhgr_memory        5
#define def_lstmhgr_fg_gradients  6
#define def_lstmhgr_ig_gradients  7
#define def_lstmhgr_og_gradients  8
#define def_lstmhgr_nc_gradients  9
#define def_lstmhgr_outputs_total 10
```

We then go to the *neuronnet.mqh* file, which contains the code for our neural network class. In the *CNet::InitOpenCL* method, we need to change the number of used kernels and simultaneously open buffers.

```
if(!m_cOpenCL.SetKernelsCount(28))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}
if(!m_cOpenCL.SetBuffersCount(10))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}
```

Changing the last parameter is not critical since in our buffer creation method, we, if necessary, change the size of the array for storing buffer handles. However, using the standard *OpenCL.mqh* library, there is no such functionality. This may result in a runtime error.

Next, we declare the kernels for use within our program, while always controlling the process of operations.

```
if(!m_cOpenCL.KernelCreate(def_k_LSTMFeedForward, "LSTMFeedForward"))
```


4. Basic types of neural layers

```
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}

if(!m_cOpenCL.KernelCreate(def_k_LSTMHiddenGradients, "LSTMCalcHiddenGradient"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}
```

This completes the preparatory work, and we move on to making changes directly to the code of the executable methods of our recurrent *LSTM* block class.

According to the chronology of the execution of the algorithm of our neural network, we will be the first to make changes to the feed-forward method. In it, we first organize a check for the presence of data in the memory of the OpenCL context.

```
bool CNeuronLSTM::FeedForward(CNeuronBase *prevLayer)
{
    ....
    //--- Branching of the algorithm by the computing device
    CBufferType *fg = m_cForgetGate.GetOutputs();
    CBufferType *ig = m_cInputGate.GetOutputs();
    CBufferType *og = m_cOutputGate.GetOutputs();
    CBufferType *nc = m_cNewContent.GetOutputs();
    if(!m_cOpenCL)
    {
        // MQL5 Block is missing here
    }
    else // Block for working with OpenCL
    {
        //--- check buffers
        if(fg.GetIndex() < 0 || ig.GetIndex() < 0 || og.GetIndex() < 0 ||
           nc.GetIndex() < 0 || memory.GetIndex() < 0 || hidden.GetIndex() < 0)
            return false;
    }
```

We then pass pointers to the created buffers to our kernel parameters. Here we indicate the constants necessary for the correct execution of the program code. Again, we check the results of the operations.

```
        //--- pass parameters to the kernel
        if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_forgetgate,
            fg.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_inputgate,
            ig.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_newcontent,
            nc.GetIndex()))
            return false;
```

4. Basic types of neural layers

```
        if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_outputgate,
og.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_memory,
memory.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_hiddenstate,
hidden.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgument(def_k_LSTMFeedForward, def_lstmff_outputs_total,
m_cOutputs.Total()))
            return false;
```

This completes the preparatory work stage. Let's move on to launching the kernel to perform operations. First, let's determine the number of required threads. In the kernel body, we use vector operations and therefore the number of threads will be four times less than the size of the buffers.

We write the calculated number of threads into the *NDRange* array and indicate the zero offset in the data buffers in the *off_set* array. The kernel is added in the execution queue. If an error occurs when queuing the kernel, the *m_cOpenCL.Execute* function will return a *false* result, which we must check and process.

```
//--- launch the kernel
int NDRange[] = {(int)(m_cOutputs.Total() + 3) / 4};
int off_set[] = {0};
if(!m_cOpenCL.Execute(def_k_LSTMFeedForward, 1, off_set, NDRange))
    return false;
}
```

This completes the work on the *LSTM* feed-forward method. Let's move on to making additions to the backpropagation method.

As in the case of the feed-forward pass, we will begin work in the error gradient distribution method *CNeuronLSTM::CalcHiddenGradient* by checking the presence of source data in the OpenCL context memory.

```
bool CNeuronLSTM::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    ....
    //--- Branching the algorithm by the computing device
    if(!m_cOpenCL)
    {
        // MQL5 Block is missing here
    }

    else // Block for working with OpenCL
    {
        //--- check buffers
        if(hidden.GetIndex() < 0)
            return false;
        if(m_cGradients.GetIndex() < 0)
            return false;
        if(ig.GetIndex() < 0)
            return false;
```

```

if(og.GetIndex() < 0)
    return false;
if(nc.GetIndex() < 0)
    return false;
if(memory.GetIndex() < 0)
    return false;
if(fg_grad.GetIndex() < 0)
    return false;
if(ig_grad.GetIndex() < 0)
    return false;
if(og_grad.GetIndex() < 0)
    return false;
if(nc_grad.GetIndex() < 0)
    return false;

```

Next, we completely repeat the algorithm for working with OpenCL kernels on the side of the main program. After creating the necessary buffers in the OpenCL context memory, we pass the data buffer handles and variable values to the kernel parameters. And it is very important to monitor the execution of all process operations.

```

//--- pass parameters to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_fg_gradients, fg_grad.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_gradients, m_cGradients.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_ig_gradients, ig_grad.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_inputgate, ig.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_memory, memory.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_nc_gradients, nc_grad.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_newcontent, nc.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_og_gradients, og_grad.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_outputgate, og.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_outputs, hidden.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_LSTMHiddenGradients,

```

```

                                def_lstmhgr_outputs_total, m_cOutputs.Total()))
    return false;

```

This concludes the stage of preparatory work. We move on to the procedure for launching the kernel. First of all, here we write the number of threads to start in the *NDRange* array and the zero offset in the *off_set* array.

Thanks to the use of vector operations in the kernel body, we need four times fewer threads for the full cycle of operations. Therefore, before we write the value to the *NDRange* array, we need to calculate it.

After this, we will send our kernel to the execution queue.

```

//--- launch the kernel
int NDRange[] = { (int)(m_cOutputs.Total() + 3) / 4 };
int off_set[] = {0};
if(!m_cOpenCL.Execute(def_k_LSTMHiddenGradients, 1, off_set, NDRange))
    return false;
}

```

I might sound repetitive, but I want to stress the importance of checking the result of each operation. This is a crucial point since any error in performing the operation can both distort the entire result of our neural network and cause a critical error, resulting in the termination of the entire program.

With this, we have completed the work on the recurrent *LSTM* block class. We have organized the class to work in two environments:

- Implemented operation on the CPU using standard MQL5 tools.
- Created the ability to implement multi-threaded parallel calculations using OpenCL.

Now, we can evaluate the results of our work by creating and testing a recurrent neural network model.

4.2.4 Implementing recurrent models in Python

In the previous sections, we reviewed the principles of organizing a recurrent model architecture, and even built a recurrent neural layer using the *LSTM* block algorithm. Earlier, we used the *Keras* library for *TensorFlow* to build previous neural network models in Python. The same library offers a number of options for building recurrent neural layers. These include classes of basic recurrent neural layers as well as more complex models.

- *AbstractRNNCell* – abstract object representing an RNN cell
- *Bidirectional* – bidirectional shell for RNN
- *ConvLSTM1D* – 1D convolutional LSTM block
- *ConvLSTM2D* – 2D convolutional LSTM block
- *ConvLSTM3D* – 3D convolutional LSTM block
- *GRU* – recurrent block by Cho et al. (2014)
- *LSTM* – layer of long-term short-term memory by Hochreiter (1997)
- *RNN* – base class for the recurrent layer
- *SimpleRNN* – fully connected recurrent layer in which the output must be returned to the input

In the presented list, in addition to the basic recurrence layer class, you can find already familiar *LSTM* and *GRU* models. It is also possible to create bidirectional recurrent layers, which are most often used

in text translation tasks. The *ConvLSTM* model is built based on the architecture of the *LSTM* block but uses convolutional layers instead of fully connected layers as gates and a new content layer.

Additionally, there is an abstract recurrent cell class for creating custom architectural solutions for recurrent models.

We won't go deep into the *Keras* library API right now. We will use the *LSTM* block to create our test recurrent models. Exactly this kind of model we recreated using MQL5 and will be able to compare the performance of our models created in different programming languages.

The *LSTM* block class is designed to automatically choose between *CuDNN* or pure *TensorFlow* implementations based on available hardware and environment constraints, ensuring optimal performance.

Users have access to an excessive range of parameters for fine-tuning the recurrent block:

- *units* – dimensionality of the output space
- *activation* – activation function
- *recurrent_activation* – activation function for the recurrent step (gate)
- *use_bias* – flag of using an offset vector
- *kernel_initializer* – method to initialize the weights matrix for the new content layer
- *recurrent_initializer* – method to initialize the weight matrix for gates
- *bias_initializer* – initialization method for bias vector
- *kernel_regularizer* – function to regularize the weight matrix for the new content layer
- *recurrent_regularizer* – function to regularize the weight matrix for gates
- *bias_regularizer* – bias vector regularization function
- *activity_regularizer* – output layer regularization function
- *kernel_constraint* – function of constraints for the weight matrix of the new content layer
- *recurrent_constraint* – function of constraints for the weight matrix of gates
- *bias_constraint* – function of vector constraints
- *dropout* – floating-point number from 0 to 1, defining the share of elements to be dropped out during linear transformation of input data
- *recurrent_dropout* – floating-point number from 0 to 1, determining the share of elements to be dropped out during linear transformation of memory state
- *return_sequences* – boolean flag to specify whether to return the last result in the output sequence or the results of the whole sequence
- *return_state* – boolean flag to indicate whether to return the last state in addition to the output
- *go_backwards* – boolean flag to instruct the processing of the input sequence in the backward order and return the reverse sequence
- *stateful* – boolean flag to indicate the use of the last state for each sample with the *i* index in the batch as the initial state for the sample with the *i* index in the next batch
- *time_major* – the format of the input and output sequence tensor shapes
- *unroll* – boolean flag used to indicate whether to unroll the recurrent network or use a simple loop; unrolling can accelerate the training of the recurrent network, but it requires more memory

4. Basic types of neural layers

After acquainting ourselves with the control parameters of the *LSTM* layer class, we will proceed to the practical implementation of various models using the recurrence layer.

4.2.4.1 Building a test recurrent model in Python

To build test recurrent models in *Python*, we will use the previously developed template. Moreover, we will take the script file [convolution.py](#), which we used when testing convolutional models. Let's make a copy of it with the file name *lstm.py*. In the created copy, we leave the perceptron model and the best convolutional model, deleting the rest. This approach will allow us to compare the performance of the new models with the architectural solutions discussed earlier.

```
# Creating a perceptron model with three hidden layers and regularization
model1 = keras.Sequential([keras.Input(shape=inputs),
                            keras.layers.Dense(40, activation=tf.nn.swish,
                            kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                            keras.layers.Dense(40, activation=tf.nn.swish,
                            kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                            keras.layers.Dense(40, activation=tf.nn.swish,
                            kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                            keras.layers.Dense(targetts, activation=tf.nn.tanh)
                            ])

# Model with 2-dimensional convolutional layer
model3 = keras.Sequential([keras.Input(shape=inputs),
                            # Reformat the tensor to 4-dimensional.
                            # Specify 3 dimensions, because the 4th dimension is determined by the size of the
                            keras.layers.Reshape((-1,4,1)),
                            # Convolutional layer with 8 filters
                            keras.layers.Conv2D(8,(3,1),1,activation=tf.nn.swish,
                            kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                            # Pooling layer
                            keras.layers.MaxPooling2D((2,1),strides=1),
                            # Reformat the tensor to 2-dimensional for fully connected layers
                            keras.layers.Flatten(),
                            keras.layers.Dense(40, activation=tf.nn.swish,
                            kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                            keras.layers.Dense(40, activation=tf.nn.swish,
                            kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                            keras.layers.Dense(40, activation=tf.nn.swish,
                            kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                            keras.layers.Dense(targetts, activation=tf.nn.tanh)
                            ])
```

After that, we will create three new models using the recurrent *LSTM* block. Initially, we will take the convolutional neural network model and replace the convolutional and pooling layers with a single recurrent layer with 40 neurons at the output. Note that the input to the recurrent *LSTM* block should be a three-dimensional tensor of the format *[batch, timesteps, feature]*. Just like in the case of a convolutional layer, when specifying the dimensionality of a layer in the model, we don't explicitly mention the batch dimension, as its value is determined by the batch size of the input data.

```
# Add an LSTM block to the model
model2 = keras.Sequential([keras.Input(shape=inputs),
                            # Reformat the tensor to 3-dimensional.
                            # Specify 2 dimensions, because. The 3rd dimension is determined by the size of the p
                            keras.layers.Reshape((-1,4)),
                            # The LSTM block contains 40 elements and returns the result at each step
```

4. Basic types of neural layers

```
keras.layers.LSTM(40, return_sequences=False,  
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
```

In this model, we specified parameter *return_sequences=False* which instructs the recurrent layer to produce the result only after processing the full batch. In this version, our *LSTM* layer returns a two-dimensional tensor in the format *[batch, feature]*. In this case, the dimension of the *feature* measurement will be equal to the number of neurons that we specified during the creation of the recurrent layer. A tensor of the same dimension is required for the input of a fully connected neural layer. Therefore, we do not need additional reformatting of the data, and we can use a fully connected neural layer.

```
keras.layers.Dense(40, activation=tf.nn.swish,  
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),  
keras.layers.Dense(40, activation=tf.nn.swish,  
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),  
keras.layers.Dense(40, activation=tf.nn.swish,  
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),  
keras.layers.Dense(targets, activation=tf.nn.tanh)  
])
```

Layer (type)	Output Shape	Param #
reshape (Reshape)	(None, 40, 4)	0
lstm (LSTM)	(None, 40)	7200
dense_4 (Dense)	(None, 40)	1640
dense_5 (Dense)	(None, 40)	1640
dense_6 (Dense)	(None, 40)	1640
dense_7 (Dense)	(None, 2)	82
Total params: 12,202		

Structure of a recurrent model with four fully connected layers

In this implementation, we use the recurrent layer for preliminary data processing, while decision-making in the model is carried out by several fully connected perceptron layers that follow the recurrent layer. As a result, we got a model with 12,202 parameters.

We will compile all neural models with the same parameters. We use the *Adam* method for optimization and the standard deviation for the network error. We also add an additional metric *accuracy*.

```
model2.compile(optimizer='Adam',  
loss='mean_squared_error',  
metrics=['accuracy'])
```

We compiled earlier neural network models with the same parameters.

4. Basic types of neural layers

One more point should be noted. Recurrent models are sensitive to the sequence of the input signal being fed. Therefore, when training a neural network, unlike the previously discussed models, we cannot shuffle the input data. For this purpose, when we start training the model, we will specify the *False* for the *shuffle* parameter. The rest of the training parameters of the model remain unchanged.

```
history2 = model2.fit(train_data, train_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.01,
                      shuffle=False)
```

In the first model, we used a recurrent layer for preliminary data processing before using a fully connected perceptron for decision-making. However, it is also possible to use recurrent neural layers in their pure form, without subsequent utilization of fully connected layers. It is this implementation that I propose to consider as the second model. In this case, we simply replace all the fully connected layers with a single recurrent layer, and we set the size of the layer to match the desired output size of the neural network.

It's important to note that the recurrent neural layer requires a three-dimensional tensor as input, whereas we obtained a two-dimensional tensor at the output of the previous recurrent layer. Therefore, before passing information to the input of the next recurrent layer, we need to reshape the data. In this implementation, we set the last adjustment to be equal to two, while leaving the size of the temporal labels dimension for the model's calculation. We don't expect any data distortion from such reshaping, as we're grouping sequential data, essentially just enlarging the time interval. At the same time, the time interval between any two subsequent elements in the new time series remains constant.

```
# LSTM block model without fully connected layers
model4 = keras.Sequential([keras.Input(shape=inputs),
# Reformat the tensor to 3-dimensional.
# Specify 2 dimensions, because. The 3rd dimension is determined by the size of the p
    keras.layers.Reshape((-1,4)),

#2 Serial LSTM Units
#1st contains 40 elements
    keras.layers.LSTM(40,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5),
        return_sequences=False),
# 2nd produces the result instead of a fully connected layer
    keras.layers.Reshape((-1,2)),
    keras.layers.LSTM(targetts)
])
```

Now we have a neural network where the first recurrent layer performs preliminary data processing, and the second recurrent layer generates the output of the neural network. By eliminating the use of a perceptron, we've reduced the number of neural layers in the network and, consequently, the total number of parameters, which in the new model amounts to 7,240 parameters.

4. Basic types of neural layers

Layer (type)	Output Shape	Param #
reshape_2 (Reshape)	(None, 40, 4)	0
lstm_1 (LSTM)	(None, 40)	7200
reshape_3 (Reshape)	(None, 20, 2)	0
lstm_2 (LSTM)	(None, 2)	40
Total params: 7,240		

The structure of a recurrent neural network without the use of fully connected layers

We compile and train the model with the same parameters as all previous models.

```
model4.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])

history4 = model4.fit(train_data, train_target,
                     epochs=500, batch_size=1000,
                     callbacks=[callback],
                     verbose=2,
                     validation_split=0.01,
                     shuffle=False)
```

In the second recurrent model, to create the input tensor for the second LSTM layer, we reshaped the tensor of results from the previous layer. The *Keras* library gives us another option. In the first *LSTM* layer, we can specify the parameter *return_sequences=True*, which switches the recurrent layer to a mode that outputs results at each iteration. As a result of this action, at the output of the recurrent layer, we immediately obtain a three-dimensional tensor of the format *[batch, timesteps, feature]*. This will allow us to avoid reformatting the data before the second recurrent layer.

```
# LSTM model block without fully connected layers
model5 = keras.Sequential([keras.Input(shape=inputs),
# Reformat the tensor to 3-dimensional.
# Specify 2 dimensions, because. The 3rd dimension is determined by the size of the p
    keras.layers.Reshape((-1,4)),
# 2 Serial LSTM Units
#1st contains 40 items and returns the result at each step
    keras.layers.LSTM(40,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5),
        return_sequences=True),
# 2nd produces the result instead of a fully connected layer
    keras.layers.LSTM(targetts)
])
```

4. Basic types of neural layers

Layer (type)	Output Shape	Param #
=====		
reshape_4 (Reshape)	(None, 40, 4)	0
lstm_3 (LSTM)	(None, 40, 40)	7200
lstm_4 (LSTM)	(None, 2)	344
=====		
Total params: 7,544		

The structure of a recurrent neural network without the use of fully connected layers

As you can see, with this model construction, the dimensionality of the tensor at the output of the first recurrent layer has changed. As a result, the number of parameters in the second recurrent layer has slightly increased. This resulted in a total increase in parameters throughout the model, reaching 7,544 parameters. Nevertheless, this is still fewer parameters than the total number of parameters in the first recurrent model that used a perceptron for decision-making.

Let's supplement the plotting block with new models.

```
# Rendering model training results
plt.figure()
plt.plot(history1.history['loss'], label='Perceptron train')
plt.plot(history1.history['val_loss'], label='Perceptron validation')
plt.plot(history3.history['loss'], label='Conv2D train')
plt.plot(history3.history['val_loss'], label='Conv2D validation')
plt.plot(history2.history['loss'], label='LSTM train')
plt.plot(history2.history['val_loss'], label='LSTM validation')
plt.plot(history4.history['loss'], label='LSTM only train')
plt.plot(history4.history['val_loss'], label='LSTM only validation')
plt.plot(history5.history['loss'], label='LSTM sequences train')
plt.plot(history5.history['val_loss'], label='LSTM sequences validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics')
plt.legend(loc='upper right', ncol=2)

plt.figure()
plt.plot(history1.history['accuracy'], label='Perceptron train')
plt.plot(history1.history['val_accuracy'], label='Perceptron validation')
plt.plot(history3.history['accuracy'], label='Conv2D train')
plt.plot(history3.history['val_accuracy'], label='Conv2D validation')
plt.plot(history2.history['accuracy'], label='LSTM train')
plt.plot(history2.history['val_accuracy'], label='LSTM validation')
plt.plot(history4.history['accuracy'], label='LSTM only train')
plt.plot(history4.history['val_accuracy'], label='LSTM only validation')
plt.plot(history5.history['accuracy'], label='LSTM sequences train')
plt.plot(history5.history['val_accuracy'], label='LSTM sequences validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics')
```

```
plt.legend(loc='lower right', ncol=2)
```

Additionally, let's add the new models to the testing block to evaluate their performance on the test dataset and display the results.

```
# Check the results of models on a test sample
test_loss1, test_acc1 = model1.evaluate(test_data, test_target, verbose=2)
test_loss2, test_acc2 = model2.evaluate(test_data, test_target, verbose=2)
test_loss3, test_acc3 = model3.evaluate(test_data, test_target, verbose=2)
test_loss4, test_acc4 = model4.evaluate(test_data, test_target, verbose=2)
test_loss5, test_acc5 = model5.evaluate(test_data, test_target, verbose=2)

print('LSTM model')
print('Test accuracy:', test_acc2)
print('Test loss:', test_loss2)

print('LSTM only model')
print('Test accuracy:', test_acc4)
print('Test loss:', test_loss4)

print('LSTM sequences model')
print('Test accuracy:', test_acc5)
print('Test loss:', test_loss5)
```

In this section, we have prepared a Python script that creates a total of 5 neural network models:

- Fully connected perceptron
- Convolutional model
- 3 models of recurrent neural networks

Upon executing the script, we will conduct a brief training of all five models using a single dataset and then compare the performance of the trained models on a shared set of test data. This will give us the opportunity to compare the performance of various architectural solutions on real data. The test results will be provided in the next chapter.

4.2.5 Comparative testing of recurrent models

Finally we have reached the testing phase of recurrent models. Previously, we have already tested various fully connected perceptron models and several convolutional models. You may notice that in both sections devoted to testing models, there is a certain sequence of actions, that is, a specific testing algorithm. In this section, we will follow this sequence.

As with the testing of previous models, we will start by checking the correctness of the gradient distribution through our recurrent layer built in MQL5. To do this, we will create the *check_gradient_lstm.mq5* script based on previously created similar scripts for testing the correctness of the performance of previous models. Basically, we will make a copy of the script *check_gradient_conv.mq5* from the convolutional model testing section and make changes to match the new model.

The change we will make in the script is the block defining the model structure for testing. We will remove the convolutional and pooling layers from the model. Instead, our model will feature one recurrent layer.

```
//--- recurrent layer
```

```

if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    delete layers;
    return false;
}
descr.type = defNeuronLSTM;
descr.count = BarsToLine;
descr.window_out = 2;
descr.activation = AF_NONE;
descr.optimization = Adam;
descr.activation_params[0] = 1;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete layers;
    delete descr;
    return false;
}

```

The rest of the neural network building block remains unchanged.

When testing other architectural solutions of neural layer configurations, the modifications to the script provided above in terms of defining the neural network structure would be sufficient for conducting the test. But the *LSTM* recurrent block has its own peculiarities. First, it lacks a weight matrix in the conventional sense. Instead, its functionality is assigned to the weight matrices of the inner layers. It will be a little more difficult to organize access to them, but I do not see the point in doing this. For inner layers, we utilize a previously validated fully connected layer class, the proper functioning of which we are confident in. Therefore, there is no need for us to retest the functioning of the already validated algorithm for gradient error distribution to the weight matrix. At the same time, we have a question about the correctness of the new functionality for the distribution of the error gradient inside the *LSTM* block. I believe that to answer this question, it is sufficient to verify the propagation of the error gradient back to the level of the original data (input of the neural network). Hence, we are removing the gradient error correctness checking block at the weight matrix level from the script.

The second feature of the recurrent layer is the use of its results as input for the new iteration, which is the desired feature. We wanted the neural network to consider not only the current state of the external environment but also its previous states, which we pass as hidden states to the new iteration. While this approach yields a positive impact on the neural network performance, it does distort the data for testing the correctness of gradient error distribution. The reason is that our entire algorithm for testing the correctness of gradient error distribution is built on the principle of changing only one tested parameter while keeping other values of the external environment constant. However, with a recurrent layer, even when all parameters of the input data remain constant, we can obtain a different result due to changes in the hidden state. To exclude this influence, we temporarily need to add a memory buffer and hidden state clearing within the forward pass method of our recurrent *LSTM* block class *CNeuronLSTM::FeedForward*.

```

bool CNeuronLSTM::FeedForward(CNeuronBase *prevLayer)
{
    //--- Check the relevance of all objects
    ....
    //--- Prepare blanks for new memory and hidden state buffers
    CBufferDouble *memory = CreateBuffer(m_cMemorys);

```

```

    if(!memory)
        return false;
    CBufferDouble *hidden = CreateBuffer(m_cHiddenStates);
    if(!hidden)
    {
        delete memory;
        return false;
    }
    //--- Gradient check only
    memory.BufferInit(m_cOutputs.Total(), 0);
    hidden.BufferInit(m_cOutputs.Total(), 0);
    //--- The following is the code of the method without changes

```

Don't forget to remove or comment out these lines after running the gradient propagation test.

After making all the necessary adjustments, we will compile and initiate the test execution using the OpenCL multi-threaded computation technology and without it. The results obtained fully satisfy our requirements and we can continue testing the models further.

Use OpenCL true
Delta at input gradient between methods -1.94600e-11
Use OpenCL false
Delta at input gradient between methods -2.85605e-13

Correctness Test of Error Gradient Distribution via LSTM Block

We have obtained confirmations of the correctness of the algorithm we built for propagating gradient error through the recurrent *LSTM* block. Now we can proceed to the next stage of our tests. But once again, before starting work on conducting tests, we need to remove the above code for resetting memory buffers and hidden state from the code of the direct *CNeuronLSTM::FeedForward* method.

Script for testing recurrent models

Let's create the script *lstm_test.mq5* to test train the recurrent model. This script is created following the template of scripts used for similar testing of previous models.

At the beginning of the script, we declare external parameters to control the process of creating and training the neural network model. Almost all external parameters migrated from the script for testing convolutional models without changes.

```

//+-----+
//| External parameters for script operation |
//+-----+
// Name of the file with the training sample
input string   StudyFileName = "study_data.csv";
// Name of file for recording the error dynamics
input string   OutputFileName = "loss_study_lstm.csv";
// Number of historical bars in one pattern
input int      BarsToLine    = 40;
// Number of input layer neurons per 1 bar
input int      NeuronsToBar   = 4;
// Use OpenCL

```

4. Basic types of neural layers

```
input bool    UseOpenCL    = false;
// Packet size for updating the weights matrix
input int     BatchSize    = 10000;
// Learning rate
input double  LearningRate = 0.00003;
// Number of hidden layers
input int     HiddenLayers = 3;
// Number of neurons in one hidden layer
input int     HiddenLayer  = 40;
// Number of cycles of updating the weights matrix
input int     Epochs       = 1000;
```

In the *CreateLayersDesc* model architecture description function, we insert one *LSTM* block between the source data layer and the block of hidden layers. The size of the result buffer for this recurrent block will be equal to the number of analyzed neural layers. The depth of the analyzed history will be set to five iterations. The architecture of the *LSTM* block defines the activation functions for all its components, and the block itself does not have a top-level activation function. Consequently, in the description of the block architecture, we will specify the absence of an activation function. We will use Adam as a method of parameter optimization.

```
//--- recurrent layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
descr.type = defNeuronLSTM;
descr.count = BarsToLine;
descr.window_out = 5;
descr.activation = AF_NONE;
descr.optimization = Adam;
descr.activation_params[0] = 1;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

The process of creating a recurrent neural network model can be considered completed, since the rest of the function code has remained unchanged.

At this stage of script execution, we already have a constructed recurrent neural network model and the training dataset loaded into memory. Now we can say that we are ready to train the model. And here we will have a slight departure from the previously used template. The reason is that for training the fully connected perceptron and the convolutional neural network, we used random patterns from the overall training dataset. At the same time, we've mentioned multiple times that recurrent neural networks require strict adherence to the chronological sequence of inputted raw data. Therefore, we have to make small changes to the training function of the *NetworkFit* model.

We need a strict sequence of patterns when training a model. Therefore, we remove the generation of a random pattern for each iteration. Instead, we will randomly determine the start of the next data batch from the training dataset.

4. Basic types of neural layers

```
bool NetworkFit(CNet &net, const CArrayObj &data, const CArrayObj &target,
               VECTOR &loss_history)
{
    //--- training
    int patterns = data.Total();
    //--- loop through the eras
    for(int epoch = 0; epoch < Epochs; epoch++)
    {
        ulong ticks = GetTickCount64();
        //--- train in batches
        //--- select a random pattern
        int k = (int)((double)(MathRand() * MathRand()) / MathPow(32767.0, 2) *
                    (patterns - 10));

        k = fmax(k, 0);
    }
}
```

But there is a nuance here as well. During the feed-forward pass, the recurrent block takes into account the results of previous iterations to the depth of the analyzed history. For the sake of data comparability, we should fill the buffer with sequential data before training the model. Therefore, we extend the loop of each batch before updating the parameters by the number of iterations required to fill the buffer with the depth of the analyzed history. In this case, we will not call the backpropagation method until the buffer is full.

```
for(int i = 0; (i < (BatchSize + 10) && (k + i) < patterns); i++)
{
    //--- check to see if the training has stopped
    if(IsStopped())
    {
        Print("Network fitting stopped by user");
        return true;
    }
    if(!net.FeedForward(data.At(k + i)))
    {
        PrintFormat("Error in FeedForward: %d", GetLastError());
        return false;
    }
    if(i < 10)
        continue;
    if(!net.Backpropagation(target.At(k + i)))
    {
        PrintFormat("Error in Backpropagation: %d", GetLastError());
        return false;
    }
}

//--- reconfigure the network weights
net.UpdateWeights(BatchSize);
printf("Use OpenCL %s, epoch %d, time %.5f sec", (string)UseOpenCL, epoch,
      (GetTickCount64() - ticks) / 1000.0);

//--- report on a bygone era
TYPE loss = net.GetRecentAverageLoss();
Comment(StringFormat("Epoch %d, error %.5f", epoch, loss));
//--- remember the epoch error to save to file
loss_history[epoch] = loss;
```



```
    }  
    return true;  
}
```

The rest of the script code remained unchanged.

I hope that everything is clear with the algorithm and the principle of constructing the script, and we can proceed to the analysis of the results.

Testing the LSTM for the first time

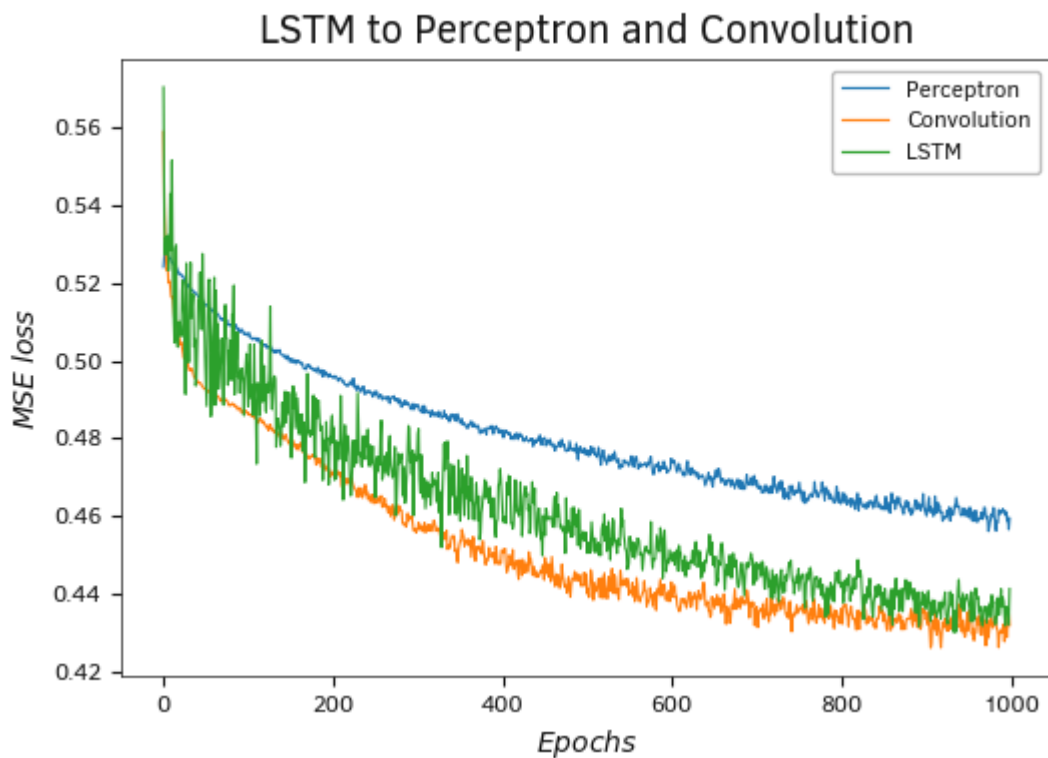
First, I created a model similar to the convolutional models I tested: one recurrent layer, three hidden fully connected layers, and one fully connected layer to display the results.

Based on the test results, it can be observed that using a recurrent layer alongside a convolutional layer for data preprocessing significantly improves the performance quality of the fully connected perceptron.

Let me remind you that in the perceptron model we used three hidden fully connected layers and one fully connected results layer. In the convolutional network model, we employed one convolutional layer, one pooling layer, three hidden fully connected layers, and one fully connected layer for output results. In the recurrent neural network model, we utilized one recurrent LSTM block, three hidden fully connected layers, and one fully connected layer for output results.

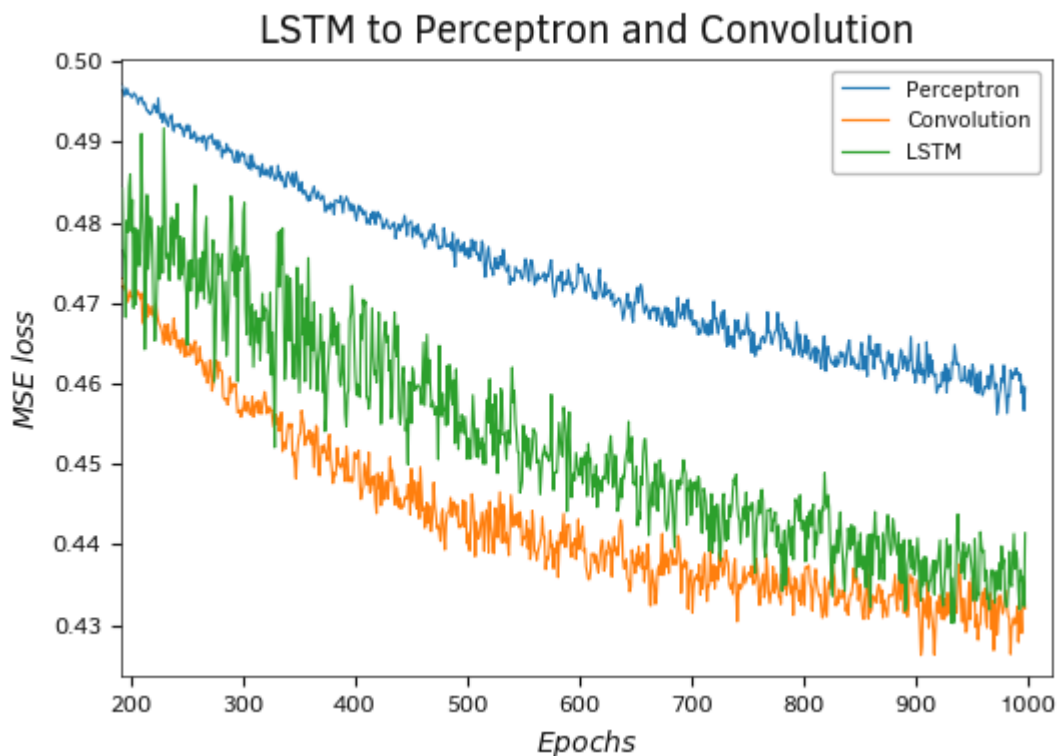
Essentially, in the convolutional and recurrent models, we introduced a convolutional or recurrent block before the previously tested perceptron for data preprocessing. The type of block used depends on the model.

As a result, we see an improvement in neural performance due to an additional layer of preliminary data processing.



Testing a Recurrent Neural Network Model

Comparing the convolutional and recurrent models, it can be observed that the error graph of the recurrent model exhibits larger noisy fluctuations. This may be due to the peculiarities of model training. To train the convolutional model, we used patterns randomly selected from the entire training set. This approach provides the most representative sample for each gradient error accumulation batch before updating the weights. At the same time, for training the recurrent model, we took patterns in chronological order. Consequently, the updating of weights and the recording of the model average error were done at different time intervals. This could not have gone unnoticed in the results, as each local time interval is subject to its own local trends.



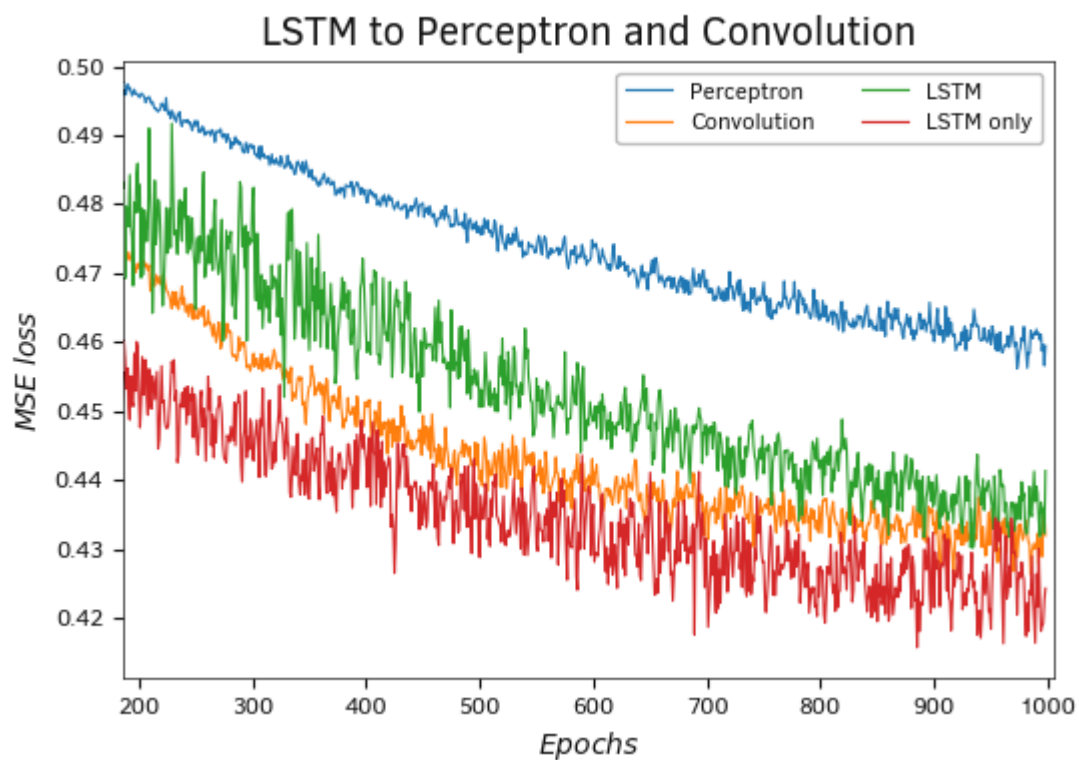
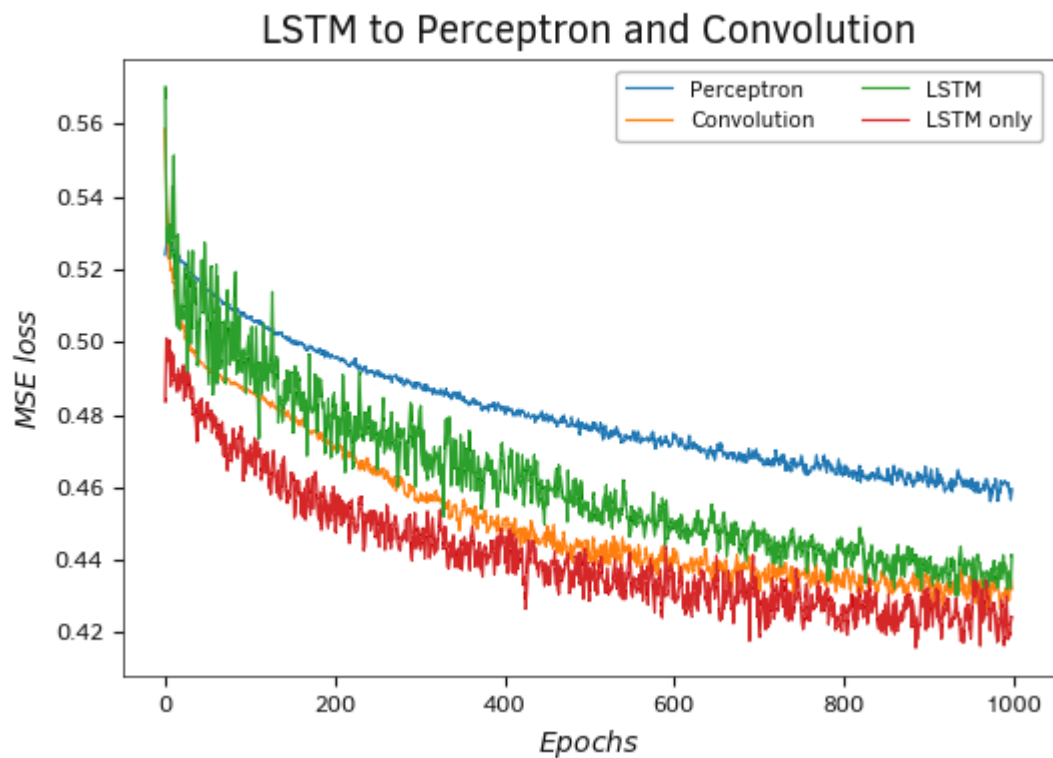
Testing a Recurrent Neural Network Model

Despite the large graph noise, the overall trend of the recursive model has a large tendency to reduce the error. True, throughout the training process, the model error value is slightly better for the convolutional model. But after 700 iterations of updating the weight matrix of the model, there is a noticeable trend towards a slowdown in the error reduction rate. This may indicate an approach to a minimum. At the same time, the recurrent model does not have such a trend. The recurrent model has a large number of parameters, and it takes more time to train. Potentially, it can improve the results in further training.

Second test of the LSTM mode

In the previous testing of the recurrent model, we used an LSTM layer to preprocess the initial data before a block of fully connected neural layers. But in practice, there is the possibility of using recurrent layers without additional preprocessing. To assess the impact of the fully connected layer block on the performance quality of the recurrent neural network, we conducted a second experiment using the same script. However, now we have specified 0 in the number of hidden layers parameter. Thus, we aim to compare the performance of two recurrent models and evaluate the necessity of using a block of fully connected neural layers for further data processing after the recurrent layer.

The test results show a very interesting trend. At the beginning of training, the recurrent model without hidden fully connected layers demonstrates a sharper drop in model error, surpassing all other models depicted on the graph. When you zoom in on the graph, you can see a clear advantage of the model without a block of hidden fully connected layers.



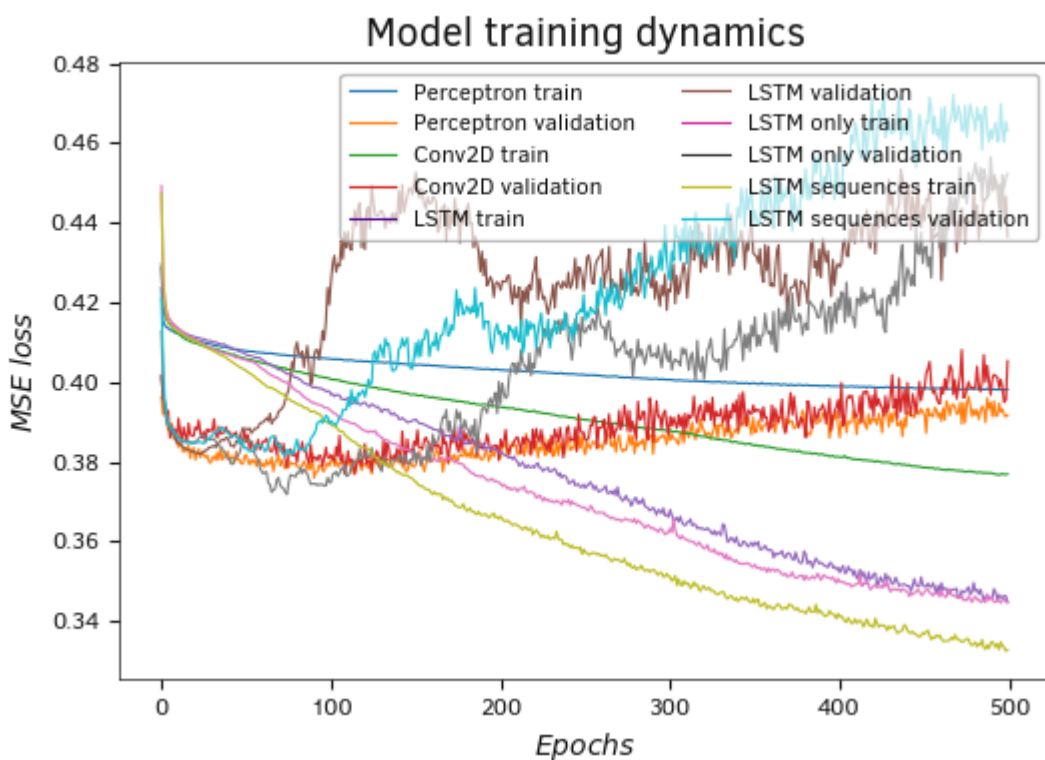
The results of the tests show the advantage of the operation of recurrent networks over the previously considered models. In this case, the use of recurrent layers yields results even without the additional processing of results by fully connected layers.

Here it must be noted that the evaluation of models was carried out only to solve a specific problem of working with time series. When solving other problems, it is possible to obtain absolutely opposite results. Therefore, when tackling your tasks, it is recommended to experiment with various architectural solutions for neural networks.

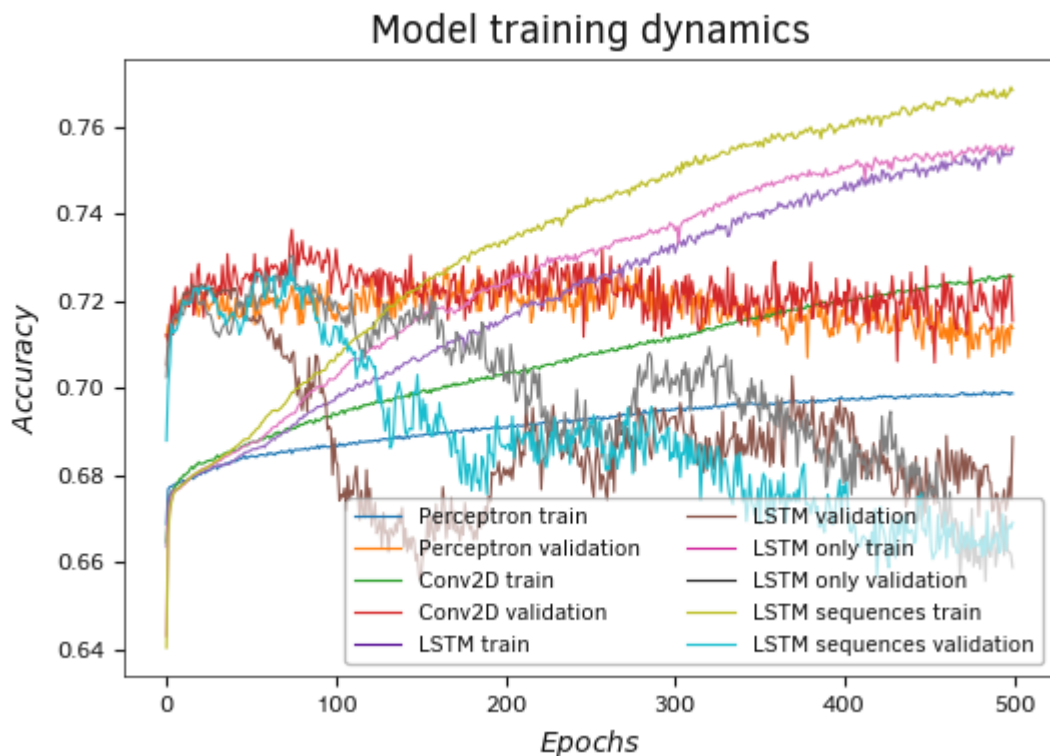
Results of testing recurrent models in Python

Earlier, we considered the implementation of a script with the construction of three recurrent models in Python. Now I propose to consider the results of test training of the constructed models.

The obtained testing results confirm the conclusions we made earlier based on the testing of models created using MQL5 tools. All three recurrent models are significantly superior to other models in terms of the quality of the neural network. In the graph depicting the change in error during the neural network training process, we can observe that the recurrent models already demonstrate lower error after 50 epochs of training compared to the fully connected perceptron and the convolutional model. With further training, superiority only grows. At the same time, one can also notice an increase in the error on the validation set, which indicates the tendency of the model to overfit.



Test training results for Python models

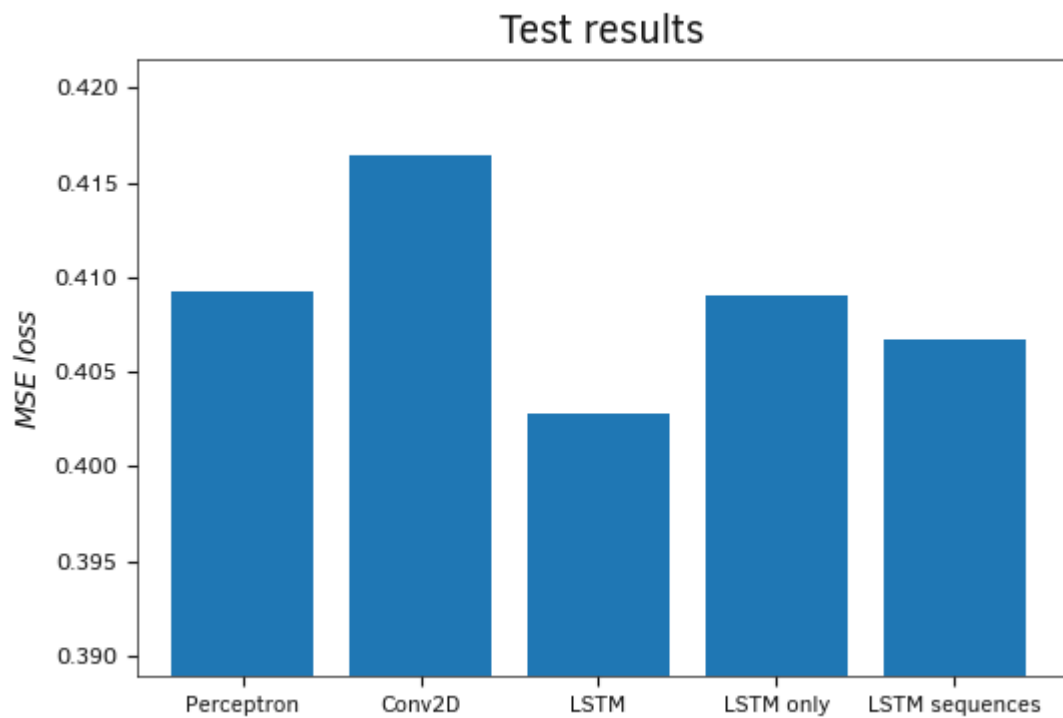


Test training results for Python models

Comparing the recurrent models with each other, you can see that the recurrent model in which the first recurrent layer returns values on each cycle is more prone to overfitting. It shows the smallest error of all models on the training set and the maximum error on the validation set. At the same time, the intersection of error curves on the testing and validation datasets for the mentioned model occurs around 130 epochs with an error value of approximately 0.385. The intersection of the graphs of the other two models is observed with an error level of about 0.395.

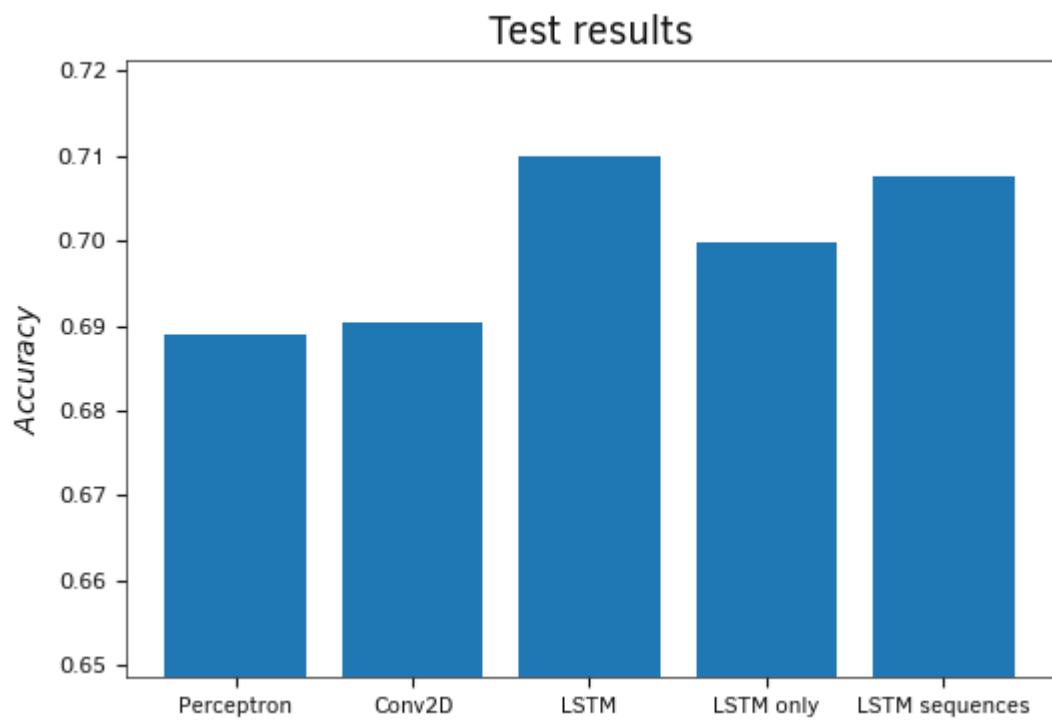
The graph of the dynamics of learning by the *Accuracy* metric fully confirms our conclusions made on the error graph.

On the test set, all trained models showed fairly close results. The deviation in both the root-mean-square error and the accuracy metric is minimal.



Testing Trained Python Models on a Test Set

While the picture is quite mixed in terms of MSE values, a clear superiority of the recurrent models is evident on the Accuracy metric graph.



Testing Trained Python Models on a Test Set

Based on the conducted tests, it can be concluded that when dealing with time series tasks, recurrent networks are capable of producing better results than the previously examined architectural solutions. At the same time, to solve such problems, we can consider various architectural solutions. Among these solutions, there could be neural networks consisting solely of recurrent layers, or mixed models that combine layers of different types.

Despite the fact that our Python language model test resulted in the victory of a recurrent model containing only recurrent neural layers, I recommend that when tackling your practical tasks, you always experiment with different models. Often, the best results come from the most unconventional architectural solutions.

5. Attention mechanisms

In the previous sections of the book, we have explored various architectures for organizing neural networks, including convolutional networks borrowed from image processing algorithms. We also learned about recurrent neural networks used to work with sequences where both the values themselves and their place in the original data set are important.

Fully connected and convolutional neural networks have a fixed input sequence size. Recurrent neural networks allow a slight extension of the analyzed sequence by transmitting hidden states from previous iterations. Nevertheless, their effectiveness also declines as consistency increases.

All the models discussed so far spend the same amount of resources analyzing the entire sequence. However, consider your behavior in a given situation. For example, even as you read this book, your gaze moves across letters, words, and lines, turning the pages in sequence. At the same time, you focus your attention on some specific component. Gradually reading the words written in the book, in your mind you assemble a mosaic of the logical chain embedded in the written words. And again, in your consciousness, there is always only a certain part of the overall content of the book.

Looking at a photograph of your loved ones, you first and foremost focus your attention on their portraits. Only then might you shift your gaze to the background elements of the photograph. At the same time, you focus your attention on photography. And the entire external environment surrounding you remains outside of your cognitive activity at that moment.

I want to show you that human consciousness does not evaluate the entire environment. It constantly picks out some details from it and shifts its attention to them. However, the neural network models we have discussed do not possess such capability.

Therefore, in 2014, in the field of machine translation, the first attention mechanism was proposed, which was designed to programmatically identify and highlight blocks of the source sentence (context) most relevant to the target translation word. This intuitive approach has greatly improved the quality of text translation by neural networks.

Analyzing the financial symbol candlestick chart, we identify trends and determine trading zones. That is, from the overall picture, we single out certain objects, focusing our attention specifically on them. It is intuitive to us that objects influence future price behavior to different degrees. To implement exactly this approach, the first proposed algorithm analyzed and identified dependencies between elements of the input and output sequences. The proposed algorithm was called a generalized attention mechanism. Initially, it was proposed for use in machine translation models using recurrent networks to address the long-term memory challenges in translating long sentences. This approach significantly outperformed the results of the previously considered recurrent neural networks based on *LSTM* blocks.

The classic machine translation model using recursive networks consists of two units, the Encoder and the Decoder. The first one encodes the input sequence in the source language into a context vector, and the second decodes the obtained context into a sequence of words in the target language. As the length of the input sequence increases, the influence of the first words on the final context of the sentence decreases, and as a result, the quality of the translation deteriorates. The use of *LSTM* blocks slightly enhanced the capabilities of the model, but they still remained limited.

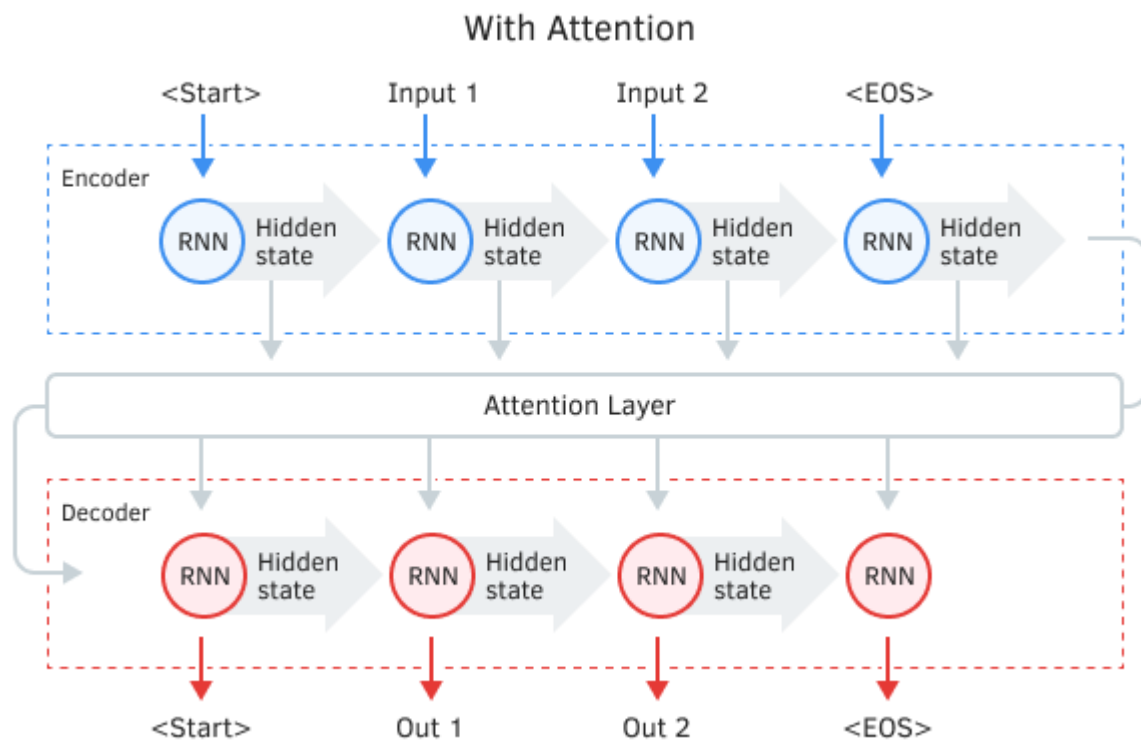


Authors of the basic attention mechanism proposed using an additional layer that would accumulate the hidden states of all recurrent blocks of the input sequence and then, during the decoding process, evaluate the influence of each element of the input sequence on the current word of the output sequence and suggest to the decoder the most relevant part of the context.

The algorithm for such a mechanism included the following iterations:

1. Creation of hidden states in the *Encoder* and their accumulation in the attention block.
2. Evaluation of pairwise dependencies between the hidden states of each element of the *Encoder* and the last hidden state of the *Decoder*.
3. The resulting estimates are combined into a single vector and normalized using the *Softmax* function.
4. Calculation of the context vector by multiplying all the hidden states of the *Encoder* by their corresponding alignment scores.
5. Decoding of the context vector and merging the resulting value with the previous *Decoder* state.

All iterations are repeated until the signal of the sentence end is received.



Encoder-Decoder with the attention mechanism

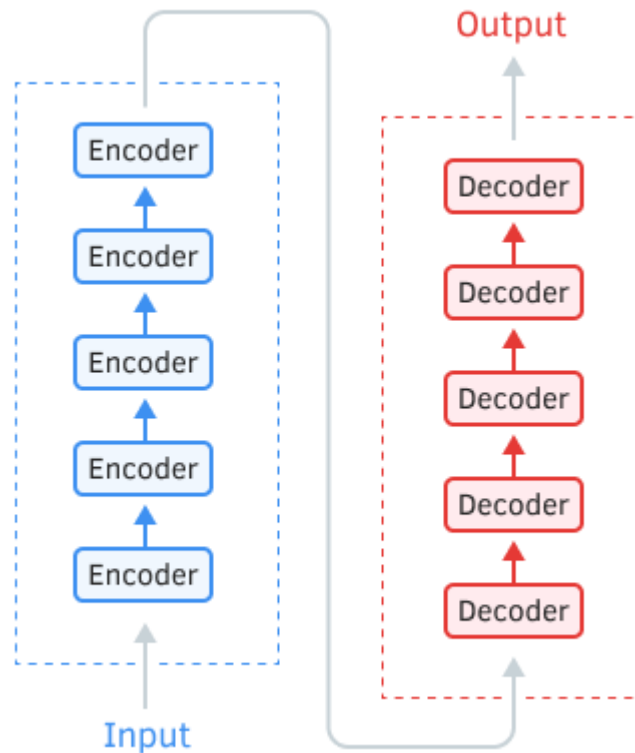
The proposed mechanism addressed the issue of the input sequence length limitation and enhanced the quality of machine translation using the recurrent neural network. As a result, it gained widespread popularity and various implementation variations. In particular, in August 2015, in their article [Effective Approaches to Attention-based Neural Machine Translation](#), Minh-Thang Luong presented their variation on the method of attention. The main differences of the new approach were the use of three functions to calculate the degree of dependencies and the point of using the attention mechanism in the Decoder.

5.1 Self-Attention

The models described above utilize recurrent blocks, the training of which incurs significant costs. In June 2017, in the article [Attention Is All You Need](#) the authors proposed a new neural network architecture called the Transformer, which eliminated the use of recurrent blocks and proposed a new *Self-Attention* algorithm. In contrast to the algorithm described above, the *Self-Attention* algorithm analyzes pairwise dependencies within the same sequence. The Transformer performed better on the tests, and today the model and its derivatives are used in many models, including *GPT-2* and *GPT-3*. We will consider the *Self-Attention* algorithm in more detail.

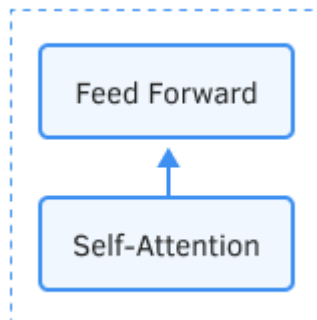
5.1.1 Description of architecture and implementation principles

The Transformer architecture is based on sequential *Encoder* and *Decoder* blocks with similar architectures. Each of the blocks includes several identical layers with different weight matrices.



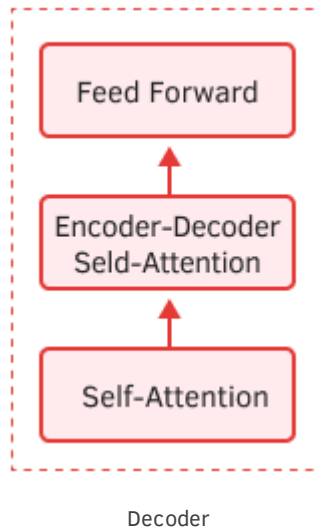
Transformer architecture

Each Encoder layer contains two internal layers: *Self-Attention* and *Feed Forward*. The *Feed Forward* layer includes two fully connected layers of neurons with a *ReLU* activation function on the internal layer. Each layer is applied to all elements of the sequence with the same weights, enabling simultaneous independent calculations for all sequence elements in parallel threads.

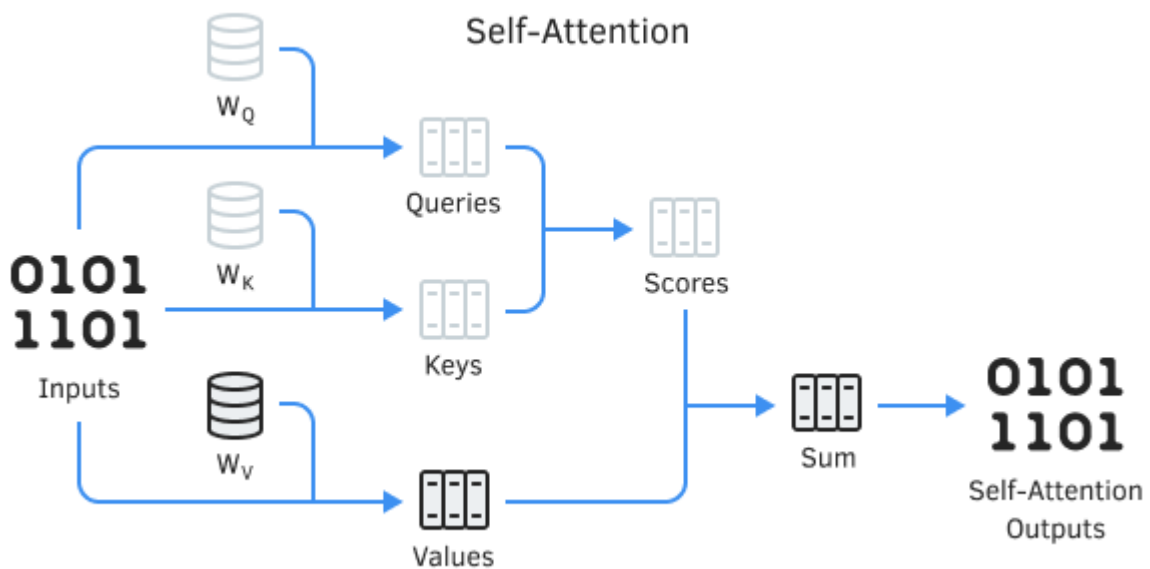


Encoder

The Decoder layer has a similar structure with an additional layer called *Self-Attention* which analyzes dependencies between the input and output sequences.



The *Self-Attention* mechanism includes several iterative actions applied to each element of the sequence.



1. First, we compute the *Query*, *Key*, and *Value* vectors. The mentioned vectors are obtained by multiplying each element of the sequence by the corresponding matrix, W_Q , W_K and W_V .
2. Next, we determine pairwise dependencies between elements of the sequence. To do this, we multiply the *Query* vector with the *Key* vectors of all elements of the sequence. This iteration is repeated for the *Query* vector of each element in the sequence. As a result of this iteration, we obtain a matrix called *Score* with a size of $N \times N$, where N is the sequence length.
3. The next step involves dividing the obtained values by the square root of the dimension of the *Key* vector and normalizing using the *Softmax* function with respect to each *Query*. Thus, we obtain coefficients representing the pairwise dependencies between the elements of the sequence.
4. By multiplying each *Value* vector by the corresponding attention coefficient, we obtain the adjusted value of the element. The goal of this iteration is to focus attention on relevant elements and reduce the influence of irrelevant values.

- Next, we summarize all the adjusted *Value* vectors for each element. The result of this operation will be the vector of output values for the *Self-Attention* layer.

The results of iterations for each layer are added to the input sequence and normalized.

$$\bar{a} = \frac{1}{h} \sum_{i=1}^h a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h} \sum_{i=1}^h (a - \bar{a})^2}}$$

For data normalization, we first determine the mean value of the entire sequence. Then, for each element, we calculate the quotient of its deviation from the mean divided by the standard deviation of the sequence.

5.1.2 Building Self-Attention with MQL5 tools

The presented *Self-Attention* architecture may seem rather difficult to understand and to implement after the first acquaintance. Let's not be pessimistic. We will try to break down the whole algorithm into small components. Then, with the implementation of each individual block, we will assemble the overall picture, and it will no longer be so complex to understand. At the same time, you will be amazed at how we manage the work and build a functional mechanism for our library.

Now let's get to work. To implement our *Self-Attention* layer, let's create a new *CNeuronAttention* class. As always, we will inherit from our base class of the neural layer, *CNeuronBase*.

```
class CNeuronAttention      : public CNeuronBase
{
public:
    CNeuronAttention(void);
    ~CNeuronAttention(void);

    //---
    virtual bool    Init(const CLayerDescription *desc) override;
    virtual bool    SetOpenCL(CMyOpenCL *opencl) override;
    virtual bool    FeedForward(CNeuronBase *prevLayer) override;
    virtual bool    CalcHiddenGradient(CNeuronBase *prevLayer) override;
    virtual bool    CalcDeltaWeights(CNeuronBase *prevLayer) override;
    virtual bool    UpdateWeights(int batch_size, TYPE learningRate,
                                  VECTOR &Beta, VECTOR &Lambda) override;

    //--- methods of working with files
    virtual bool    Save(const int file_handle) override;
    virtual bool    Load(const int file_handle) override;
    //---object identification method
    virtual int     Type(void) override const { return(defNeuronAttention); }
};
```

Let's consider the first action of the *Self-Attention* algorithm which is the computation of the *Query*, *Key*, and *Value* vectors. At the input, we get a tensor of raw data containing features for each bar of the analyzed sequence. Sequentially, we take the features of one candlestick and, by multiplying them with a weight matrix, obtain a vector. Then we take the features of the second candlestick and multiply

them by the same weight matrix so we get the second vector similar to the first one. Does this look similar to the convolution layer created earlier? Here, the length of the result vector is equal to the number of filters used in the convolution layer. Hence, to organize the above process, we declare three nested convolutional layers *CNeuronConv*. We use the appropriate layer names to make the code easier to read.

```
class CNeuronAttention : public CNeuronBase
{
protected:
    CNeuronConv    m_cQuerys;
    CNeuronConv    m_cKeys;
    CNeuronConv    m_cValues;
    .....
};
```

According to this algorithm, in the next step, we determine the *Score* matrix by multiplying the *Query* and *Key* matrices. To write the matrix data, we will create a data buffer as an object of the *CBufferType* class.

```
class CNeuronAttention : public CNeuronBase
{
protected:
    .....
    CBufferType    m_cScores;
    .....
};
```

After determining the *Score* dependency coefficient matrix, we will need to find the weighted values. To do this, we multiply the *Values* vectors by the corresponding values of the *Score* matrix. After additional processing, we obtain a tensor equal to the size of the initial data. We will talk about the reasons for the same size during the implementation process. Right now, let's just note for ourselves the need for a data warehouse. To collect data in the storage, we will need to set up a new process, so we require an object with easy access for writing data. In the future, we plan to pass data as input to the internal neural layer. So, the neural layer template of the raw data will be the most suitable for us. We use a basic neural layer with zero input window as the input data layer.

```
class CNeuronAttention : public CNeuronBase
{
protected:
    .....
    CNeuronBase    m_cAttentionOut;
    .....
};
```

Here, it's important to note the difference between the output of the *Self-Attention* algorithm and the output of the entire *CNeuronAttention* class. The first one is obtained after execution of the *Self-Attention* algorithm by adjusting the values of *Value* vectors. We save it to the instance of the object of the basic neural layer *m_cAttentionOut*. The second one is obtained after processing in the *Feed Forward* block. This one is saved to the result buffer of our class.

So, next, we need to organize the *Feed Forward* block. We will create it from two consecutive convolution layers. It may seem unusual to use a convolutional layer when the solution architecture is described as having fully connected layers. The situation here is similar to the first point of the algorithm when we determined the value of *Query*, *Key*, and *Value* vectors. Looking at the block within

the context of one element of the sequence, we can see two fully connected neural layers. However, when considering the entire time series, it becomes evident that the same weight matrix is applied sequentially to each element of the sequence. Furthermore, as the input data progresses sequentially, the results are laid out in the same order. Doesn't this resemble the operation of a convolutional layer? We just need to take the convolution layer and set the width of the source data window equal to the vector size of one sequence element. The step of the initial data window is set equal to the window width, and the number of filters used is determined by the size of the fully connected layer for one element of the sequence.

Thus, we add two convolution layers to organize the *Feed Forward* block.

```
class CNeuronAttention      : public CNeuronBase
{
protected:
    .....
    CNeuronConv      m_cFF1;
    CNeuronConv      m_cFF2;
    .....
};
```

We identified the objects we need to organize the *Self-Attention* mechanism in our class. To complete the picture, let's add a few more variables:

- *m_iWindow* – width of the initial data window (size of one sequence element vector)
- *m_iUnits* – the number of units in the sequence
- *m_iKeysSize* – width of the result vector size for *Query* and *Key*
- *m_dStd* – during normalization of the layer, we will divide the value by the standard deviation and will save the result to determine the derivative

Taking into account the standard set of functions for overriding, the class structure will have the following form.

```
class CNeuronAttention      : public CNeuronBase
{
protected:
    CNeuronConv      m_cQueryys;
    CNeuronConv      m_cKeys;
    CNeuronConv      m_cValues;
    CBufferType      m_cScores;
    int              m_cScoreGrad;
    int              m_cScoreTemp;
    CNeuronBase      m_cAttentionOut;
    CNeuronConv      m_cFF1;
    CNeuronConv      m_cFF2;
    /---
    int              m_iWindow;
    int              m_iUnits;
    int              m_iKeysSize;
    CBufferType      m_cStd;

public:
    CNeuronAttention(void);
    ~CNeuronAttention(void);
```

```

//---
virtual bool      Init(const CLayerDescription *desc) override;
virtual bool      SetOpenCL(CMyOpenCL *openc1) override;
virtual bool      FeedForward(CNeuronBase *prevLayer) override;
virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer) override;
virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda) override;
//--- methods for operations with files
virtual bool      Save(const int file_handle) override;
virtual bool      Load(const int file_handle) override;
//--- object identification method
virtual int       Type(void) override const { return(defNeuronAttention); }
};

```

In the class constructor, we only set the initial values of the variables.

Please note that in this class, we are using static objects rather than pointers to objects as we did previously. The lifetime of static objects, like variables, is equal to the lifetime of the object containing them. By using such objects, we avoid the need to create object instances during class initialization and to clean up memory when the class operation is completed. Also, we don't need to check the validity of the object pointer every time. This saves some time in performing each method. However, in this case, we cannot replace objects by copying only the object's pointer, which this property is actively used in our activation class and in recurrent networks (using the same object pointers when analyzing the entire depth of the history).

```

CNeuronAttention::CNeuronAttention(void) :    m_iWindow(1),
                                              m_iUnits(0),
                                              m_iKeysSize(1)
{
    m_cStd.BufferInit(1, 2, 1);
}

```

Since we use static objects, we leave the class destructor empty.

```

CNeuronAttention::~CNeuronAttention(void)
{
}

```

Method of class initialization

After creating the class constructor and destructor, we move on to overriding the main methods of the class. First, we will override the class initialization method *CNeuronAttention::Init*. The main task of this method is to prepare the class to perform its functionality with user-defined parameters. Like similar methods in other previously discussed classes, the method receives an instance of the *CLayerDescription* object as a parameter, in which the parameters of the initialized neural layer are specified. Therefore, in order to eliminate possible errors in future work, we organize the block of initial data verification. In this method, we will check for the presence of the minimum required parameters in the received data.

```

bool CNeuronAttention::Init(const CLayerDescription *desc)
{

```



```
//--- check the initial data
if(!desc || desc.type != Type() || desc.count <= 0 ||
    desc.window <= 0 || desc.window_out <= 0)
    return false;
```

After that, we will save the main parameters into specially prepared variables. Note the correlation between the parameters of the neural layer description class and their functional purpose:

- *CLayerDescription.window* – the size of the source data window, a vector of source data of one element of the sequence (in our case description of one bar)
- *CLayerDescription.count* – the number of elements in the sequence (the number of analyzed bars)
- *CLayerDescription.window_out* – size of the result vector for *Query* and *Key*

```
m_iWindow    = desc.window;
m_iUnits     = desc.count;
m_iKeysSize  = desc.window_out;
```

As before, we start initializing the object by calling a similar initialization method of the parent class. But there's a nuance here. We cannot simply transfer the resulting description of the neural layer. We will create a new instance of the neural layer description object and *CLayerDescription* and enter the corrected data into it.

In the *count* field, we specify the total number at the output of the layer, which is obtained by multiplying the *count* and *window* fields of this object.

Note that to obtain the total number of elements in the output of the neural layer, we multiply the number of elements in the sequence (number of bars analyzed) by the size of the source window (elements describing 1 bar), not the size of the results window. The reason is that we will use the results window size only for *Query* and *Key* tensors. The size of the result vector for the *Value* tensors and the second layer of the *Feed Forward* block will be equal to the size of the initial data window. This is done to align the dimensionality of the initial data and the results. The algorithm involves adding the tensors of the original data to the results of the *Self-Attention* block and then adding the tensors of the results of the *Feed Forward* and *Self-Attention* blocks, as well. Thus, as a result of tensor addition, the sequence at the output of our neural layer cannot be shorter than the initial data. And it doesn't make any sense to increase it. Therefore, we align the dimensions of the vectors.

In addition to changing the number of elements, we will also change the size of the output window, setting it to one. The size of the initial data window will be equal to zero. After that, we will call the parent class initialization method.

```
//--- calling the parent class initialization method
CLayerDescription *temp = new CLayerDescription();
if(!temp)
    return false;
temp.count = desc.count * desc.window;
temp.window_out = 1;
temp.window     = 0;
temp.optimization = desc.optimization;
temp.activation = desc.activation;
temp.activation_params = desc.activation_params;
temp.type = desc.type;
if(!CNeuronBase::Init(temp))
{
    delete temp;
```

```

    return false;
}

```

Such a parameter substitution will allow the running of the parent class initialization method in the neural layer mode of the source data. At the same time, no additional buffers will be created for the weight matrix, as well as the corresponding optimization method buffers. As with the LSTM block, this neural layer will not have a separate weight matrix. All weight factors will be stored in the inner neural layers.

We specify a similar architecture for the inner data collection layer of the *AttentionOut* attention block. We will simply change the type of neural layer and explicitly disable the activation function.

```

//--- initialize AttentionOut
temp.type = defNeuronBase;
temp.activation=AF_NONE;
if(!m_cAttentionOut.Init(temp))
{
    delete temp;
    return false;
}

```

Next, to initialize our internal neural layers, we need to create a description for them. We fill the previously created instance of the *CLayerDescription* class with the necessary data. Almost all of our internal neural layers are convolutional, so in the *type* parameter, we will specify *defNeuronConv*. The rest of the parameters are transferred without changes from the obtained external description.

```

//--- create a description for the internal neural layers
temp.type = defNeuronConv;
temp.window = desc.window;
temp.window_out = m_iKeysSize;
temp.step = desc.window;
temp.count = desc.count;

```

Next, we proceed to initialize the internal neural layers. We first initialize the convolution layer to define *Query* vectors using a pre-built description. Don't forget to check the results of the operations.

```

//--- initialize Querys
if(!m_cQuerys.Init(temp) || !m_cQuerys.SetTransposedOutput(true))
{
    delete temp;
    return false;
}

```

Note that we use the new *CNeuronConv::SetTransposedOutput* method after initializing the convolutional neural layer. The reasons for its appearance and its functionality will be discussed a bit later.

We initialize the *Keys* layer using a similar algorithm.

```

//--- initializing Keys
if(!m_cKeys.Init(temp) || !m_cKeys.SetTransposedOutput(true))
{
    delete temp;
    return false;
}

```

Next, initialize the *Values* layer. We use the above algorithm with a small addition. As mentioned earlier, when initializing this object, the result window is set equal to the input data window. Therefore, we make changes to the neural layer description object and call the initialization method. Let's check the result of the operations.

```
//--- initialize Values
temp.window_out = m_iWindow;
if(!m_cValues.Init(temp) || !m_cValues.SetTransposedOutput(true))
{
    delete temp;
    return false;
}
```

Next, we initialize the *Scores* coefficient matrix. According to the *Self-Attention* mechanism algorithm, this is a square matrix with a side length equal to the number of elements in the sequence. For us, it is the number of bars analyzed.

In the discussion of this algorithm, it's important to understand the difference between the *number of elements in the sequence* and the total *number of elements at the output of the neural layer*. If you translate this to analyzing a candlestick chart of a change in a stock instrument, then:

- The number of elements in a sequence is the number of bars to be analyzed.
- The length of a vector of one sequence element (input / output window) is the number of elements describing 1 bar.
- The total number of elements at the input/output of the neural layer is the product of the first two quantities.

Let's return to the initialization of the coefficient matrix buffer. For it, we have declared a data buffer. We will initialize it with zero values by setting the buffer size as a square matrix.

```
//--- initialize Scores
if(!m_cScores.BufferInit(temp.count, temp.count, 0))
{
    delete temp;
    return false;
}
```

The *Self-Attention* algorithm is followed by the base neural layer object for recording attention results, which we have already initialized above.

All we have to do is initialize the *Feed Forward* block. As mentioned, it will consist of two convolutional neural layers. According to the architecture proposed by the authors, in the first neural layer, the tensor of results is four times larger than the input data. In addition, the authors used the *ReLU* activation function in the first neuron layer. We'll replace it with *Swish*. We will make the specified changes to the description of the neural layer and proceed with its initialization.

```
//--- initialize FF1
temp.window_out *= 4;
temp.activation = AF_SWISH;
temp.activation_params[0] = 1;
temp.activation_params[1] = 0;
if(!m_cFF1.Init(temp) || !m_cFF1.SetTransposedOutput(true))
{
    delete temp;
}
```

```

        return false;
    }

```

To initialize the second neural layer in the *Feed Forward* block, we need to increase the size of the input data window and its stride. The size of the results window should be resized to match the size of the attention block results tensor. It will also correspond to the tensor size of the previous layer.

For the second neural layer in the *Feed Forward*, we will use the activation function specified by the user during the initialization of our class.

After making the necessary changes to the description object of the neural layer, we will use the algorithm discussed earlier to initialize the last internal neural layer.

```

//--- initialize FF2
temp.window = temp.window_out;
temp.window_out = temp.step;
temp.step = temp.window;
temp.activation = desc.activation;
temp.activation_params = desc.activation_params;
if(!m_cFF2.Init(temp) || !m_cFF2.SetTransposedOutput(true))
{
    delete temp;
    return false;
}
delete temp;

```

After initializing all internal neural layers, we delete the temporary neural layer description object. We don't need it anymore.

Now let's use a little trick. According to the algorithm, we obtain the result of the second neural layer operation in the result buffer of the *Feed Forward* block's second layer. To transfer the data to the subsequent neural layer, we need to transfer the data to the result buffer of our class. We will need additional time and resources at each iteration for the data copy operation. To avoid this, we can substitute pointers to objects. Remember that we discussed objects and pointers to them?

Initially, we delete the result buffer object of our class to avoid leaving unaccounted objects in memory. Then, in the variable used to store the pointer to the buffer object, we assign the pointer to a similar buffer in the second neural layer of the *Feed Forward* block. The same operation is performed for the gradient buffer.

```

//--- to avoid copying the buffers we swap them
if(m_cOutputs)
    delete m_cOutputs;
m_cOutputs = m_cFF2.GetOutputs();
if(m_cGradients)
    delete m_cGradients;
m_cGradients = m_cFF2.GetGradients();

```

Thanks to this simple trick, we have been able to avoid constant data copying between buffers and reduce the time required to perform operations within the class.

In conclusion, at the end of the initialization method, we call the *SetOpenCL* method to ensure that all our internal objects work in the same context. Now we exit the method with a positive result.

```

//--- pass a pointer to the object of work with OpenCL before all internal objects

```

```

        SetOpenCL(m_cOpenCL);
    //---
    return true;
}

```

The *SetOpenCL* method, called at the end of the initialization method, is designed to distribute the pointer to the *OpenCL* context work object among all internal objects. This is necessary to ensure that all objects work in the same space. This method was created as virtual in the base class of the neural layer. It is redefined in each new class as needed.

The algorithm of the method is quite simple, and we have already discussed it in all the previous classes. In the parameters, the method receives a pointer of the object of work with *OpenCL* context from an external program. We simply start by calling the method of the parent class and pass it the obtained pointer. The validation of the obtained pointer is already implemented in the parent class's method, so there is no need to repeat it here.

Then we pass the pointer to the *OpenCL* context to all internal objects stored in the variable of our class. The trick is that the method of the parent class checks the obtained pointer and stores the appropriate pointer in the variable. To ensure that all objects work in the same context, we propagate the already processed pointer.

```

bool CNeuronAttention::SetOpenCL(CMyOpenCL *opencl)
{
    CNeuronBase::SetOpenCL(opencl);
    m_cQuerys.SetOpenCL(m_cOpenCL);
    m_cKeys.SetOpenCL(m_cOpenCL);
    m_cValues.SetOpenCL(m_cOpenCL);
    m_cAttentionOut.SetOpenCL(m_cOpenCL);
    m_cFF1.SetOpenCL(m_cOpenCL);
    m_cFF2.SetOpenCL(m_cOpenCL);
    if(m_cOpenCL)
    {
        m_cScores.BufferCreate(m_cOpenCL);
        ulong size = sizeof(TYPE) * m_cScores.Total();
        m_cScoreGrad = m_cOpenCL.AddBuffer((uint)size, CL_MEM_READ_WRITE);
        m_cScoreTemp = m_cOpenCL.AddBuffer((uint)size, CL_MEM_READ_WRITE);
        m_cStd.BufferCreate(m_cOpenCL);
    }
    else
    {
        m_cScores.BufferFree();
        m_cStd.BufferFree();
    }
    //---
    return(!m_cOpenCL);
}

```

Going a bit ahead, I want to draw attention to the creation of the *m_cScoreGrad* and *m_cScoreTemp* buffers. They are used only in the *OpenCL* context for temporary data storage, so we did not create mirror objects for them in the main memory. Also, we will not use them to exchange data between the main program and the *OpenCL* context. In this case, we will create buffers in the *OpenCL* context, while on the side of the main program, we use only pointers to work with them. When disabling multi-threading technology, we immediately delete the mentioned buffers.

5. Attention mechanisms

After completing the initialization method of the class, we can proceed to override the functional methods of our class.

5.1.2.1 Self-Attention feed-forward method

We have already created the structure of a class organization for implementing the attention mechanism and even created an object initialization method. In this section, we will organize the forward pass process.

As you know, in the base class of the neural network, we have created a virtual method *CNeuronBase::FeedForward* which is responsible for organizing the feed-forward pass. In each new class, we override this method to organize the relevant process according to the algorithm of the implemented architectural solution. By doing so, we kind of personalize the method for each class. At the same time, the external program does not need to know anything about the organization of the process within the class. It doesn't even need to know the type of neural layer. It simply calls the *FeedForward* method of the next object and passes it a pointer to the previous layer of the neural network. In this way, we have shifted the functionality of dispatching and checking the required object type from our program to the system.

Let's go back to our *CNeuronAttention::FeedForward* method. Just like the method of the parent class, in parameters it receives a pointer to the object of the previous layer. This is consistent with the principles of method inheritance and overriding. Since we receive a pointer to an object, it is customary to begin the method with a block to check the validity of the received pointer. However, in this case, we will omit it. The reason is that the use of static internal objects allows us to refuse to check their pointers. Regarding the pointer to the previous neural layer, we will use it for the feed-forward pass of the internal convolutional neural layers *m_cQueryys*, *m_cKeys* and *m_cValues*. They already have the relevant controls and thus we do not need to duplicate them.

In accordance with the *Self-Attention* algorithm, we need to define the *Query*, *Key*, and *Value* vectors for each element of the sequence. As you remember, it was for this functionality that we created the first three convolutional layers. Therefore, to solve this problem, we just need to call the *FeedForward* methods for the named internal layers. With each call in the parameters, we pass a pointer to the previous neural layer obtained in the parameters of our *CNeuronAttention::FeedForward* method.

```
if(!m_cQueryys.FeedForward(prevLayer))
    return false;
if(!m_cKeys.FeedForward(prevLayer))
    return false;
if(!m_cValues.FeedForward(prevLayer))
    return false;
```

Next in the *Self-Attention* algorithm, we need to determine the dependency coefficients and fill in the Score matrix. At this point, it's essential to recall our paradigm of creating classes capable of running both on the CPU and using the GPU tools. Each time we build a new process, we create a branching of the algorithm depending on the computing device in use. This method will not be an exception, and we will continue to work in the same direction. Right now, we will create a similar branching of the process. We will start with the process using MQL5 tools and will return to the OpenCL branch a little later.

For convenience, we copy the *m_cQueryys* and *m_cKeys* matrices which contain the results of the convolutional layers.

```
//--- branching of the algorithm by the computing device
MATRIX out;
if(!m_cOpenCL)
{
    MATRIX queryys = m_cQueryys.GetOutputs().m_mMatrix;
```

```
MATRIX keys = m_cKeys.GetOutputs().m_mMatrix;
```

After completing the preparatory work, we need to "roll up our sleeves" and build a new process. The *Self-Attention* method involves line-wise normalization of the dependency matrix using the *Softmax* function.

$$\text{SoftMax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

The main feature of such normalization lies in obtaining a series of positive values that sum up to 1. Thus, by multiplying the normalized dependency coefficients with the values of *Value* vectors of the corresponding sequence elements and then summing up these vectors within one *Query*, we expect to obtain new vectors within the same range of values.

Let's look at the implementation of this process. First, we organize the process of calculating the dependency coefficients into the *Score* matrix. According to the *Self-Attention* algorithm, each element of the matrix represents the product of the *Query* and *Key* vectors. In this case, the matrix row indicates the position of the vector in the *Queries* matrix and its column indicates the position in the *Keys* matrix.

Here, it is important to carefully consider the choice of elements to be multiplied. Let's recall how we organized the output of the results to the buffer of the convolutional layer. To enable the operation of the pooling layer in the context of filters, we have organized the sequential output of filters. First, in the first row of the result buffer matrix, we output all the elements of the result of one filter. Then, in the next row, we write the elements of the next filter, and so on. This organization of the buffer is convenient for the transparent operation of the pooling layer within the filters. In this case, within the vector of one element of the sequence, we need to use one value from each filter. In other words, we need a transposed matrix.

Reorganizing the buffer data in such a way that the first elements of all filters come first, then the second elements of all filters, and so on, would require additional resources on each feed-forward pass. It would be much easier to organize a convenient record directly in the convolutional layer. However, this would disrupt the operation of both the pooling layer and subsequent convolutional layers when building convolutional models. Therefore, it was decided to introduce a flag into the operation of the convolutional layer to determine whether the values should be arranged in the result buffer. You may have already guessed this when I talked about the new *SetTransposedOutput* convolutional layer method when describing the initialization method. I promised to return to the description of the functionality of this method. Such a solution has helped us keep the structure of the feed-forward pass method transparent and avoid additional time and resource costs for data reorganization. Let's finish working with the feed-forward pass method, and then we can revisit the changes in the convolutional layer.

Taking into account the transposition of the convolutional layer results, to obtain the values of the matrix of dependency coefficients, we need to multiply the *Queries* matrix by the transposed matrix *Keys*. It sounds a little strange to transpose the work of the convolutional layer method and then transpose the *Keys* matrix. However, we will use the result of transposing the work of the convolutional layer more than once. Of course, with the help of the entered flag, we could transpose the work of the convolutional layer *m_cQueries*, and leave the *m_cKeys* layer unchanged. But in this case, there is a possibility of confusion with the matrix dimensions. This will make the code more difficult to read and understand. Therefore, I decided to unify the dimensions of the matrices used.

Please note that simultaneously with the calculation of the vector product, we will prepare data for normalization according to the *Softmax* formula above. For this purpose, we will immediately divide the obtained matrix by the square root of the *Key* vector size and take the exponent of the resulting value.

Then we will take the row-wise sum of the matrix values and divide the values by the resulting vector of the matrix *Scores*. MQL5 matrix operations do not allow you to divide a matrix by a vector. Therefore, we will organize a loop in which we will sequentially divide each row by the sum of its values.

```
//--- define Scores
MATRIX scores = MathExp(querys.MatMul(keys.Transpose()) / sqrt(m_iKeysSize));
//--- normalize Scores
VECTOR summs = scores.Sum(1);
for(int r = 0; r < m_iUnits; r++)
    if(!scores.Row(scores.Row(r) / summs[r], r))
        return false;
m_cScores.m_mMatrix = scores;
```

After normalizing the data in the matrix containing the dependencies coefficients of the elements in the sequence, we will transfer these values to our data buffer *buffer m_cScores*.

At this stage, we have computed and normalized the dependency coefficients between all elements of the sequence. Now, according to the algorithm of the Self-Attention method, we need to calculate the weighted sum of the *Values* vectors in terms of each *Query*. To do this, we just need to multiply the matrix of dependency coefficients by the matrix of results of the convolutional layer *m_cValues*. Again, it is precisely because of the transposition of the work of the convolutional layer that we do not transpose the matrix of the results of the *m_cValues* layer.

```
//--- the output of the attention block
MATRIX values = m_cValues.GetOutputs().m_mMatrix;
out = scores.MatMul(values);
```

The product of the matrices will give us the result of the *Self-Attention* mechanism. But we will go a little further and build the entire Encoder block of the transformer. According to his algorithm, the results of *Self-Attention* are added to the buffer of the original data. The obtained values are normalized within the neural layer. The following formulas are used to normalize the data.

$$\bar{a} = \frac{1}{h} \sum_{i=1}^h a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h} \sum_{i=1}^h (a - \bar{a})^2}}$$

To perform this operation, we will first bring the format of the results matrix of the *Self-Attention* block in accordance with the format of the matrix of the initial data and add the two matrices. The result is normalized in a specially selected *NormalizeBuffer* method.

```
//--- add to initial data and normalize
if(!out.Reshape(prevLayer.Rows(), prevLayer.Cols()))
    return false;
m_cAttentionOut.GetOutputs().m_mMatrix = out +
    prevLayer.GetOutputs().m_mMatrix;
```

```

        if(!NormalizeBuffer(m_cAttentionOut.GetOutputs(), GetPointer(m_cStd), 0))
            return false;
    }

```

With this, the first block of operations is completed. This concludes the section on dividing the algorithm based on the execution of mathematical operations. For the block of operations using OpenCL, we will temporarily set the return of an error value and come back to it later.

```

else // OpenCL block
{
    return false;
}

```

Let's continue working with the encoder algorithm and move on to the second block of operations. Here it is necessary to conduct the signal of each element of the sequence through two fully connected layers. As you remember, we decided to organize this work through two convolutional layers. At first glance, there is nothing complicated about it - we simply call the forward pass methods for each convolutional layer sequentially.

```

--- call the feed-forward methods of the Feed Forward block layers
if(!m_cFF1.FeedForward(GetPointer(m_cAttentionOut)))
    return false;
if(!m_cFF2.FeedForward(GetPointer(m_cFF1)))
    return false;

```

Here, correct operation is possible only due to the transposition of the buffer of the convolutional neural layers results. Only this approach allows the aligned operation on each individual element of the sequence.

After conducting a forward pass through two convolutional layers, just as after determining the attention results, it is necessary to propagate the obtained results to the data input into the first convolutional layer and normalize the resulting sums. We have already considered such a task above. Here we use the same algorithm, only the data buffers are different.

```

//--- add to the output of attention and normalize
if(!m_cOutputs.SumArray(m_cAttentionOut.GetOutputs()))
    return false;
//--- normalize
if(!NormalizeBuffer(m_cOutputs, GetPointer(m_cStd), 1))
    return false;
//---
return true;
}

```

It should be noted that thanks to the buffer substitution organized in the initialization method, we obtain the results of the second convolutional layer from the result buffer of the current layer. In the same buffer, we will save the results of data normalization.

After the completion of the operations, we exit the feed-forward method with a positive result.

Now let's take a look at the changes made to the convolutional layer class. First, we'll add a variable to store the flag of the *m_bTransposedOutput* output structure. This will be a Boolean flag indicating the need to transpose the result matrix for output to the buffer. By default, we will set the value to *false*, which means working in normal mode.

```

class CNeuronConv      : public CNeuronProof
{
protected:
    bool                m_bTransposedOutput;

public:
    bool                SetTransposedOutput(const bool value);
    ....
}

```

To control the value of the flag, let's create the *SetTransposedOutput* method. The functionality of the method is quite simple. We resize the result matrices and error gradients.

```

bool CNeuronConv::SetTransposedOutput(const bool value)
{
    m_bTransposedOutput = value;
    if(value)
    {
        if(!m_cOutputs.BufferInit(m_iNeurons, m_iWindowOut, 0))
            return false;
        if(!m_cGradients.BufferInit(m_iNeurons, m_iWindowOut, 0))
            return false;
    }
    else
    {
        if(!m_cOutputs.BufferInit(m_iWindowOut, m_iNeurons, 0))
            return false;
        if(!m_cGradients.BufferInit(m_iWindowOut, m_iNeurons, 0))
            return false;
    }
    //---
    return true;
}

```

However, as you understand, the presence of a flag and even a method that changes it will not affect the results of data output to the buffer. To do this, we have to make some changes to the forward pass method. We are not changing the algorithm or the calculation logic at all; our changes will only involve rearranging matrices when multiplying the input data by the weight matrix, depending on the state of the *m_bTransposedOutput* flag.

```

bool CNeuronConv::FeedForward(CNeuronBase *prevLayer)
{
    //--- control block
    ....
    //--- branching the algorithm depending on the execution device
    if(!m_cOpenCL)
    {
        ....
        //--- Calculating the weighted sum of the elements of the input window
        if(m_bTransposedOutput)
            m = m.MatMul(m_cWeights.m_mMatrix.Transpose());
        else
            m = m_cWeights.m_mMatrix.MatMul(m.Transpose());
    }
}

```

```

        m_cOutputs.m_mMatrix = m;
    }
    else // OpenCL block
    {
        ....
    }
//---
    if(!m_cActivation.Activation(m_cOutputs))
        return false;
//---
    return true;
}

```

After making changes to the feed-forward method, we need to make similar adjustments to the backpropagation methods because the error gradient should be propagated back to the point of error occurrence. Otherwise, the results of training the neural network will be unpredictable. First, we make changes to the gradient distribution method in the hidden layer *CNeuronConv::CalcHiddenGradient*.

```

bool CNeuronConv::CalcHiddenGradient(CNeuronBase *prevLayer)
{
//--- control block
    ....
//--- correction of error gradients to the derivative of the activation function
    ....
//--- branching the algorithm depending on the execution device
    CBufferType* input_gradient = prevLayer.GetGradients();
    if(!m_cOpenCL)
    {
        MATRIX g = m_cGradients.m_mMatrix;
        if(m_bTransposedOutput)
        {
            if(!g.Reshape(m_iNeurons, m_iWindowOut))
                return false;
        }
        else
        {
            if(!g.Reshape(m_iWindowOut, m_iNeurons))
                return false;
            g = g.Transpose();
        }
        ....
    }
    else // OpenCL block
    {
        ....
    }
//---
    return true;
}

```

Then we make the relevant changes in the *CNeuronConv::CalcDeltaWeights* method for distributing the gradient to the weight matrix level.

```

bool CNeuronConv::CalcDeltaWeights(CNeuronBase *prevLayer)
{
//--- control block
....
//--- branching the algorithm depending on the execution device
CBufferType *input_data = prevLayer.GetOutputs();
if(!m_cOpenCL)
{
....
//---
MATRIX g = m_cGradients.m_mMatrix;
if(m_bTransposedOutput)
{
    if(!g.Reshape(m_iNeurons, m_iWindowOut))
        return false;
    g = g.Transpose();
}
else
{
    if(!g.Reshape(m_iWindowOut, m_iNeurons))
        return false;
}
m_cDeltaWeights.m_mMatrix += g.MatMul(inp);
}
else // OpenCL block
{
....
}
//---
return true;
}

```

As you can see, the changes are not so crucial, but they provide enhanced flexibility in settings.

5.1.2.2 Self-Attention backpropagation methods

In the previous section, we discussed the feed-forward method in the *Encoder* block of the Transformer architectural solution. This block includes a *Self-Attention* mechanism, followed by processing by two fully connected neural layers. The peculiarity of the *Self-Attention* mechanism lies in determining the dependencies between elements of the sequence. Moreover, each element of the sequence is represented as a vector of properties of a fixed length. Each sequence element within one neural layer is processed by an *Encoder* block with one set of weighting factors. This allowed us to use previously developed convolutional layers to solve a number of problems. The organization of a forward pass is a very important part of the algorithm for the operation of neural networks. We use it both when training our neural network models, and during practical application. But neural network training is impossible without going back. So now we'll look at organizing the backward pass in our attention mechanism class.

Just to remind you, we have created our own class as a successor to the neural layer base class. Several methods are responsible for organizing the backward pass:

- *CNeuronBase::CalcOutputGradient*: method for calculating the error gradient of the result layer.
- *CNeuronBase::CalcHiddenGradient*: method for calculating the error gradient through a hidden layer.
- *CNeuronBase::CalcDeltaWeights*: method for calculating the error gradient to the level of the weight matrix.
- *CNeuronBase::UpdateWeights*: method for updating weights.

All methods were made virtual to allow overriding in descendant classes. In our class, we're not going to override only the first method.

We will work on the methods in accordance with the logic of the backward propagation of error gradient method. We will be the first to redefine the error gradient calculation method via the hidden *CNeuronAttention::CalcHiddenGradient* layer. Of the three redefined methods, this one is probably the most difficult to understand and organize. After all, it is in this method that we will need to repeat the entire path of the feed-forward pass, but in reverse order. At the same time, we will have to find derivatives of all operations used in the feed-forward pass.

In the method parameters, we get a pointer to an object in the previous layer, in whose buffer we will save the result of operations. Next, in the body of the method, we organize a block of checks on the relevance of pointers to objects. Here, I decided not to dwell on checking all objects but only checked those objects that are not verified when calling methods of internal classes. This decision was made in an attempt to avoid redundant validity checks of objects during the execution of method operations.

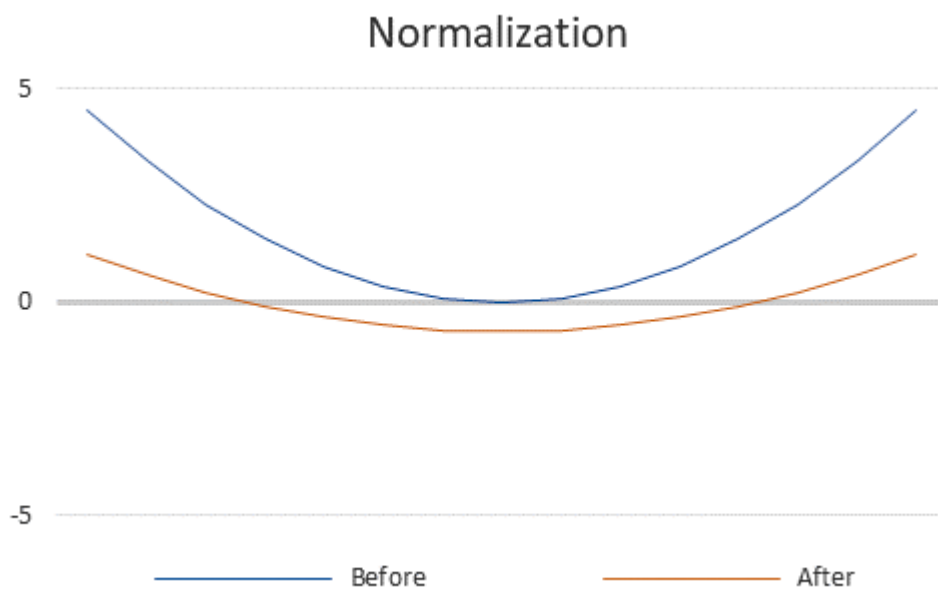
```
bool CNeuronAttention::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    //--- checking the relevance of all objects
    if(!m_cOutputs || !m_cGradients ||
        m_cOutputs.Total() != m_cGradients.Total())
        return false;
```

This is followed by the most interesting part in which the error gradient is propagated in the reverse order of the feed-forward algorithm. Let's look at the forward pass algorithm. It ends with the normalization of results, which is carried out using formulas.

$$\bar{a} = \frac{1}{h} \sum_{i=1}^h a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h} \sum_{i=1}^h (a - \bar{a})^2}}$$

What is the normalization process? This is the process of changing the statistical variables in a sampling and bringing it closer to some specified parameters. Most often this is the mean and standard deviation, as in our case. We equate the mean value to zero and reduce the standard deviation to one. As a result of this operation, the function graph is shifted and scaled, as shown in the figure.



Effect of normalization on the function graph

In essence, as part of the *Self-Attention* algorithm, the process of data normalization is used as a function of activating the neural layer. However, unlike the latter, it does not change the data structure.

But we are not going to dig into the details of calculating the derivative of the complex data normalization function now. We have implemented the process of correcting the error gradient as a separate method.

```
//--- adjust the gradient for normalization
if(!NormlizeBufferGradient(m_cOutputs, m_cGradients, GetPointer(m_cStd), 1))
    return false;
```

Next, we can use the *FeedForward* methods of our internal block layers and draw an error gradient to the internal layer for storing the results of the attention block.

```
//--- propagate a gradient through the layers of the Feed Forward block
if(!m_cFF2.CalcHiddenGradient(GetPointer(m_cFF1)))
    return false;
if(!m_cFF1.CalcHiddenGradient(GetPointer(m_cAttentionOut)))
```

```
return false;
```

In the feed-forward method, before normalizing the results layer, we added the values of two buffers (the results of the *FeedForward* and *Self-Attention* blocks). Therefore, the error gradient should also be propagated along both branches of the algorithm. So, let's add the two gradient buffers together. To facilitate access to the buffer of the internal *Self-Attention* results storage layer, we create a local pointer to objects.

```
CBufferType *attention_grad = m_cAttentionOut.GetGradients();
if(!attention_grad.SumArray(m_cGradients))
    return false;
```

Let's adjust the error gradient by the standard deviation.

```
//--- adjust the gradient for normalization
if(!NormlizeBufferGradient(m_cAttentionOut.GetOutputs(), attention_grad,
                           GetPointer(m_cStd), 0))

    return false;
```

After adding the two error gradient tensors, we need to distribute the error gradient between the internal layers *m_cQuerys*, *m_cKeys*, and *m_cValues*. When we passed forward, we fully recreated the data flow algorithm block from the specified neural layers to the Self-Attention results buffer. Therefore, we will also have to create a backpropagation process. As always, here we will create a branching of the algorithm depending on the computing device. We start with considering the process of algorithm creation using standard MQL5 tools and will get back to the implementation of the multi-threaded computing mechanism using OpenCL a little later.

```
//--- branching the algorithm by the computing device
if(!m_cOpenCL)
{
    MATRIX values, gradients;
```

At the beginning of the MQL5 block, we will create two matrices for storing intermediate data: *values* and *gradients*.

We will be the first to transfer the error gradient to the neural layer of *m_CValues* values. It was the values of the results buffer of this neural layer that we multiplied by the dependency coefficients of the *Score* matrix to determine the results of the *Self-Attention* block during a direct pass. Now we are performing the reverse operation. As we have already said, the derivative of the multiplication operation is equal to the second factor. In our case, these are the *Score* matrix coefficients.

The data tensors have the following dimensions:

- The *Score* matrix is square with a side equal to the number of elements in the sequence.
- The *m_CValues* and *m_CattentionOut* buffers of neural layers have the number of rows equal to the number of sequence elements and the number of elements in each row equal to the size of the vector describing one element of the sequence.

To prevent potential mismatches in matrix sizes, we will reshape the error gradient matrix to the required format.

```
if(attention_grad.GetData(gradients, false) < (int)m_cOutputs.Total())
    return false;
if(!gradients.Reshape(m_iUnits, m_iWindow))
    return false;
```


Each sequence element from *m_CValues* affects all elements of the *m_CattentionOut* sequence with the corresponding coefficient from the *m_cScores* matrix.

To organize the process of propagating the error gradient to the *m_CValues* neural layer buffer, we need to multiply the transposed *m_cScores* matrix of dependence coefficients by the *gradients* error matrix.

```
//--- gradient propagation to Values
m_cValues.GetGradients().m_mMatrix =
    m_cScores.m_mMatrix.Transpose().MatMul(gradients);
```

Next, we'll propagate the error gradients to *m_cQuery*s and *m_cKeys*. Both neural layers participated in creating the *m_cScores* matrix of dependence coefficients. Therefore, we first need to determine the error gradient on the matrix of dependence coefficients.

During the feed-forward pass, to obtain the *Self-Attention* result, we multiplied the *m_cScores* matrix by the results tensor of the neural layer *m_CValues*. We have already determined the error gradient for the neural layer. Now we need to propagate the error gradient along the second branch of the algorithm and distribute it to the values of the dependency coefficient matrix. Therefore, we will need to multiply the error gradient by the transposed results buffer of the *m_CValues* neural layer.

```
gradients = gradients.MatMul(values.Transpose());
```

Let me remind you that during a direct pass, the matrix values were normalized by the *Softmax* function as part of *Query* queries. The complexity of calculating this function and its derivative lies in the need to compute the entire normalization array at once. Unlike other functions, the derivative of a vector of values will be a matrix. This is due to the nature of the *Softmax* feature itself. A change in one element of the source data vector leads to a change in the entire sequence of the normalized result because the sum of all elements of the result vector is always equal to one. Therefore, in order to distribute the error gradient correctly, we need to work in the context of queries *Query*.

The mathematical formula for the derivative of the *Softmax* function is:

$$\frac{df(x_i)}{dx_i} = \begin{cases} i = j & f(x_i) * (1 - f(x_i)) \\ i \neq j & -f(x_j)f(x_i) \end{cases}$$

We'll use its matrix representation:

$$\frac{df(X)}{dX} = f(X)^T (E - f(X))$$

where *E* is a single square matrix with a size equal to the number of elements in the sequence.

The implementation of this approach is described below. In a loop, we determine the derivative of each individual row of the dependency coefficient matrix. After multiplying the resulting matrix by the gradient vector of the corresponding row, we get a vector of corrected error gradients. Let's not forget that before normalizing the dependency coefficient matrix *Score*, we divided its values by the square root of the dimension of the vector describing one element in the *Key* tensor. Accordingly, we will repeat this procedure for the error gradient as well. The logic of this operation is simple: dividing by a constant is equivalent to multiplying by the reciprocal of that constant, and the derivative of a multiplication operation is equal to its second multiplier.

$$\left(\frac{x}{const}\right)' = \left(\frac{1}{const}x\right)' = \frac{1}{const}$$

The result of the above operations will replace the analyzed row of the gradient matrix.

```
for(int r = 0; r < m_iUnits; r++)
{
    MATRIX ident = MATRIX::Identity(m_iUnits, m_iUnits);
    MATRIX ones = MATRIX::Ones(m_iUnits, 1);
    MATRIX result = MATRIX::Zeros(1, m_iUnits);
    if(!result.Row(m_cScores.m_mMatrix.Row(r), 0))
        return false;
    result = ones.MatMul(result);
    result = result.Transpose() * (ident - result);
    VECTOR temp = result.MatMul(gradients.Row(r));
    if(!gradients.Row(temp / sqrt(m_iKeysSize), r))
        return false;
}
```

After obtaining an adjusted error gradient for each individual dependency coefficient, we distribute it to the corresponding *Query* and *Key* tensor vectors. To this end, we will multiply the matrix of adjusted gradients of dependence coefficients by the opposite matrix.

```
m_cQuerys.GetGradients().m_mMatrix =
                                gradients.MatMul(m_cKeys.GetOutputs().m_mMatrix
m_cKeys.GetGradients().m_mMatrix =
                                gradients.Transpose().MatMul(m_cQuerys.GetOutputs().m_mMatrix
}

else // OpenCL block
{
    return false;
}
```

This completes the block for branching the algorithm by the computing device. In the OpenCL block, we will leave the return of a negative result for now and will come back to it a little later. Now let's move on with our error backpropagation algorithm. After obtaining the error gradient at the output of the internal neural layers, we need to propagate it back to the previous layer.

As you remember, in the feed-forward pass, the source data is used in four branches of the algorithm:

- At the input of the internal *m_Cquerys* layer
- At the input of the internal *m_CKeys* layer
- At the input of the internal *m_CValues* layer
- Added to the output of the *Self-Attention* block before the layer is normalized

Therefore, in the buffer for the error gradients of the previous layer, we should accumulate the error gradient from all 4 directions. The operating algorithm is similar to the previously constructed process of adding buffers in a recurrent *LSTM* block. However, we will not create a separate buffer to accumulate data; we will use the existing one instead. The error gradient at the output of the *Self-Attention* block has already been calculated in the neural layer buffer *m_CattentionOut*. This is where we will accumulate intermediate error gradients.

We will alternately call the method of transferring the gradient to the previous *CalcHiddenGradient* layer for each inner layer, giving it a pointer to the previous neural layer. After successfully executing

the method, we will add the obtained result to the previously accumulated error gradient in the gradient buffer of the *m_CattentionOut* neural layer.

```
//--- transfer the error gradient to the previous layer
if(!m_cValues.CalcHiddenGradient(prevLayer))
    return false;
if(!attention_grad.SumArray(prevLayer.GetGradients()))
    return false;
if(!m_cQuerys.CalcHiddenGradient(prevLayer))
    return false;
if(!attention_grad.SumArray(prevLayer.GetGradients()))
    return false;
if(!m_cKeys.CalcHiddenGradient(prevLayer))
    return false;
if(!prevLayer.GetGradients().SumArray(attention_grad))
    return false;
//---
return true;
}
```

Note that in the first two cases, we recorded the sum of two error gradient buffers into the internal neural layer buffer. In the last case, we saved the sum of the two buffers into the buffer for the error gradients of the previous neural layer. The reason is that the *CalcHiddenGradient* method of the internal neural layer overwrites the values in the gradient buffer of the neural layer specified in the parameters. So, we needed to accumulate intermediate gradients in a different buffer. However, at the end of the method, we need to propagate the error gradient to the previous layer. Therefore, during the last summation of the buffers, we immediately write the sum to the buffer of the previous neural layer, thereby avoiding unnecessary copying of data.

A method for correcting the error gradient for the *NormalizeBufferGradient* data normalization process was announced above. What is the normalization process and why is it difficult to determine the derivative of a function? At first glance, we subtract the arithmetic mean from each element of the normalized array and divide the resulting difference by the standard deviation.

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\sigma^2}}$$

If we were subtracting and dividing by constants, there would be no difficulties. When a constant is subtracted, the derivative does not change.

$$(f(a) - \text{const})' = (f(a))' - 0 = (f(a))'$$

The derivative of dividing by a constant is equal to the ratio of 1 to the constant.

$$\left(\frac{f(a)}{\text{const}}\right)' = \left(\frac{1}{\text{const}}f(a)\right)' = \frac{1}{\text{const}}(f(a))'$$

But the problem is that both the average ones are functions. When changing any single value in the input tensor, the values of the means change, and consequently, all the values in the output tensor of the normalization block are affected. This makes it much more difficult to calculate the derivative of the entire function. We will not present them now and will instead use the ready-made result.

$$\frac{\partial f(a_i)}{\partial \sigma^2} = \sum_{i=1}^m \frac{\partial f(a_i)}{\partial \hat{a}_i} * (a_i - \bar{a}) * \frac{-1}{2} (\sigma^2 + \varepsilon)^{-\frac{3}{2}}$$

$$\frac{\partial f(a_i)}{\partial \bar{a}} = \left(\sum_{i=1}^m \frac{\partial f(a_i)}{\partial \hat{a}_i} * \frac{-1}{\sqrt{\sigma^2 + \varepsilon}} \right) + \frac{\partial f(a_i)}{\partial \sigma^2} * \frac{\sum_{i=1}^m -2(a_i - \bar{a})}{m}$$

$$\frac{\partial f(a_i)}{\partial a_i} = \frac{\partial f(a_i)}{\partial \hat{a}_i} * \frac{1}{\sqrt{\sigma^2 + \varepsilon}} + \frac{\partial f(a_i)}{\partial \sigma^2} * \frac{2(a_i - \bar{a})}{m} + \frac{\partial f(a_i)}{\partial \bar{a}} * \frac{1}{m}$$

Let's implement the above formulas in code using *MQL5* matrix operations. In parameters, the method receives pointers to 3 data buffers:

- *output* – buffer with the results of normalizing feed-forward data
- *gradient* – error gradient buffer. It is used both for obtaining initial data and for recording results
- *std* – standard deviation buffer calculated during a forward pass.

As you can see, the parameters do not include the data buffer before normalization and the value of the arithmetic mean calculated during the forward pass. We simply replaced the difference between the non-normalized data and the arithmetic mean with the product of the normalized data and the standard deviation.

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\sigma^2}} \Rightarrow \hat{a} \sqrt{\sigma^2} = a - \bar{a}$$

Of course, we don't expect zero standard deviation. Let's add a check to prevent a critical error of division by zero.

```
bool CNeuronAttention::NormlizeBufferGradient(CBufferType *output,
                                              CBufferType *gradient,
                                              CBufferType *std,
                                              uint std_shift)
{
    //---
    if(!m_cOpenCL)
    {
        if(std.At(std_shift) <= 0)
            return true;
        MATRIX ScG = gradient.m_mMatrix / std.m_mMatrix[0, std_shift];
        MATRIX ScOut = output.m_mMatrix * std.m_mMatrix[0, std_shift];
        TYPE dSTD = (gradient.m_mMatrix * output.m_mMatrix / (-2 * MathPow(std.m_mMatrix[0, std_shift], 2)));
        TYPE dMean = -1 * ScG.Sum() - 2 * dSTD / (TYPE)output.Total() * ScOut.Sum();
        gradient.m_mMatrix = ScG + (ScOut * dSTD * 2 + dMean) / (TYPE)output.Total();
    }
    else // OpenCL block
    {
        return false;
    }
    //---
    return true;
}
```

```
}
```

In addition to the method of distributing the gradient through a hidden layer, the algorithm for the backward distribution of the error gradient in all previously considered neural layers is usually represented by two more methods:

- *CalcDeltaWeights* – method for calculating the error gradient to the level of the weight matrix
- *UpdateWeights* – method for updating weights

The *CNeuronAttention* class under consideration will be no exception. We will also use it to redefine these two methods. Their algorithm is straightforward: we will simply call the methods of all internal neural layers of the same name one by one, while constantly checking the results of the operations.

```
bool CNeuronAttention::CalcDeltaWeights(CNeuronBase *prevLayer)
{
    if(!m_cFF2.CalcDeltaWeights(GetPointer(m_cFF1)))
        return false;
    if(!m_cFF1.CalcDeltaWeights(GetPointer(m_cAttentionOut)))
        return false;
    if(!m_cQuerys.CalcDeltaWeights(prevLayer))
        return false;
    if(!m_cKeys.CalcDeltaWeights(prevLayer))
        return false;
    if(!m_cValues.CalcDeltaWeights(prevLayer))
        return false;
    //---
    return true;
}

bool CNeuronAttention::UpdateWeights(int batch_size, TYPE learningRate,
                                     VECTOR &Beta, VECTOR &Lambda)
{
    if(!m_cQuerys.UpdateWeights(batch_size, learningRate, Beta, Lambda))
        return false;
    if(!m_cKeys.UpdateWeights(batch_size, learningRate, Beta, Lambda))
        return false;
    if(!m_cValues.UpdateWeights(batch_size, learningRate, Beta, Lambda))
        return false;
    if(!m_cFF1.UpdateWeights(batch_size, learningRate, Beta, Lambda))
        return false;
    if(!m_cFF2.UpdateWeights(batch_size, learningRate, Beta, Lambda))
        return false;
    //---
    return true;
}
```

In this way, we have implemented three methods that make up the backpropagation algorithm for our attention block.

5.1.2.3 File operations

We have already built the feed-forward and backpropagation methods for our attention layer. We can add the layer to our model and train it, but we don't really want to retrain our model from scratch every time we want to use it. We need to be able to save a once-trained model to a file and, if necessary, load a ready-to-use neural network from the file. Two methods are responsible for working with files in our basic neural layer: *Save* and *Load*. To ensure the proper functioning of your new layer, you need to override the specified methods.

We perform a similar iteration when creating each new type of neural layer. Now we will follow the known path: we will focus on the structure of our class and determine what needs to be saved to a file, and which variables and objects we will simply create and initialize with initial values.

First of all, it is necessary to save the internal neural layers containing the weight matrices *m_cQuerys*, *m_cKeys*, *m_cValues*, *m_cFF1*, and *m_cFF2*. In addition, we need to save the values of the variables that define the architecture of the neural layer: *m_iWindow*, *m_iUnits*, and *m_iKeysSize*.

We do not need to save any information from the *m_cScores* buffer to the file, since it contains only intermediate data that is overwritten on each forward pass. Its size is easy to determine based on the number of elements in the sequence recorded in the *variable m_iUnits*.

The *m_cAttentionOut* inner layer does not contain the matrix weights, while its data, similarly to the data of the *m_cScores* buffer, are overwritten at each iteration of the forward and reverse passes. However, let's look at the situation from the other side. Recall the procedure for initializing the neural layer:

- Create a neural layer description object
- Fill in the neural layer description object with the necessary information
- Call the method that initializes the neural layer with the transfer of a description
- Delete the neural layer description object

At the same time, calling the save method for the base neural layer without weight matrices will write only 3 integers to the file, with a total size of 12 bytes. So, by sacrificing 12 bytes of disk space, we reduce our efforts in writing the initialization code for the neural layer in the data loading method.

```
class CNeuronAttention : public CNeuronBase
{
protected:
    CNeuronConv    m_cQuerys;
    CNeuronConv    m_cKeys;
    CNeuronConv    m_cValues;
    CBufferType    m_cScores;
    int            m_cScoreGrad;
    int            m_cScoreTemp;
    CNeuronBase    m_cAttentionOut;
    CNeuronConv    m_cFF1;
    CNeuronConv    m_cFF2;
    //---
    int            m_iWindow;
    int            m_iUnits;
    int            m_iKeysSize;
    CBufferType    m_cStd;
```

```

//---
virtual bool      NormlizeBuffer(CBufferType *buffer, CBufferType *std,
                                uint std_shift);

virtual bool      NormlizeBufferGradient(CBufferType *output,
                                CBufferType *gradient, CBufferType *std, uint std_shift);

public:
                                CNeuronAttention(void);
                                ~CNeuronAttention(void);

//---
virtual bool      Init(const CLayerDescription *desc) override;
virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
virtual bool      FeedForward(CNeuronBase *prevLayer) override;
virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer) override;
virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda) override;

//--- methods for working with files
virtual bool      Save(const int file_handle) override;
virtual bool      Load(const int file_handle) override;
//--- object identification method
virtual int      Type(void) override const { return(defNeuronAttention); }
};

```

Once we have decided on the objects to write data to the file, we can start working on our methods. Let's start with the *Save* method that writes data to the file. In the parameters, the method receives the handle of the file to write the data. However, we will not immediately check the received handle. Instead, we will call the analogous method of the parent class, where all checkpoints and the saving of inherited objects are already implemented. The result of the parent class method will indicate the result of the control block execution.

```

bool CNeuronAttention::Save(const int file_handle)
{
    if(!CNeuronBase::Save(file_handle))
        return false;
}

```

After executing the parent class method, we call the save method for internal objects one by one. At the same time, we check the results of the operations.

```

if(!m_cQuerys.Save(file_handle))
    return false;
if(!m_cKeys.Save(file_handle))
    return false;
if(!m_cValues.Save(file_handle))
    return false;
if(!m_cAttentionOut.Save(file_handle))
    return false;
if(!m_cFF1.Save(file_handle))
    return false;
if(!m_cFF2.Save(file_handle))
    return false;

```

After saving the data of internal objects, we'll save the values of variables that define the architecture of the neural layer. Quite obviously, we check the result of the operations.

```

    if(FileWriteInteger(file_handle, m_iUnits) <= 0)
        return false;
    if(FileWriteInteger(file_handle, m_iWindow) <= 0)
        return false;
    if(FileWriteInteger(file_handle, m_iKeysSize) <= 0)
        return false;
    //---
    return true;
}

```

After successfully saving all the necessary data, we complete the method with a positive result.

After creating a data writing method, we move on to work on the *Load* data reading method. In the parameters, the method receives the file handle to read the data. Just like in the case of writing data, we do not create a new control block in our method. Instead, we call the method of the parent class where all controls, reading of inherited objects, and variables are already implemented. Checking the result of the parent class method immediately informs us about both the completion of the control block and the loading of data from inherited objects and variables.

```

bool CNeuronAttention::Load(const int file_handle)
{
    if(!CNeuronBase::Load(file_handle))
        return false;
}

```

After successfully executing the data loading method of the parent class, we will sequentially read the data of internal objects. Recall that reading data from a file is carried out in strict accordance with the sequence of writing data. When writing data to a file, we first saved information from the *m_cQuery*s internal neural layer. Therefore, we will be loading data into this object first. However, don't forget about the nuance of loading internal neural layers: we first check the type of the loaded object and only then call the loading method for the corresponding object.

```

    if(FileReadInteger(file_handle) != defNeuronConv || !m_cQuerys.Load(file_handle))
        return false;

```

We repeat the same algorithm for all previously saved objects.

```

    if(FileReadInteger(file_handle) != defNeuronConv || !m_cKeys.Load(file_handle))
        return false;
    if(FileReadInteger(file_handle) != defNeuronConv || !m_cValues.Load(file_handle))
        return false;
    if(FileReadInteger(file_handle) != defNeuronBase ||
        !m_cAttentionOut.Load(file_handle))
        return false;
    if(FileReadInteger(file_handle) != defNeuronConv || !m_cFF1.Load(file_handle))
        return false;
    if(FileReadInteger(file_handle) != defNeuronConv || !m_cFF2.Load(file_handle))
        return false;

```

After loading the data of the internal neural layer objects, we read the values of the variables that determine the architecture of our attention neural layer from the file.

```

    m_iUnits = FileReadInteger(file_handle);

```



```
m_iWindow = FileReadInteger(file_handle);
m_iKeysSize = FileReadInteger(file_handle);
```

Then we need to initialize the *m_cScores* buffer of dependency coefficients with zero values. We do not change the size of the buffer beforehand, since the buffer initialization method provides for changing its size to the required level.

```
if(!m_cScores.BufferInit(m_iUnits, m_iUnits, 0))
    return false;
```

We have loaded all the data and initialized the objects. It is worth remembering that to avoid unnecessary data copying, we replaced the pointers to the result and gradient buffers of the internal layer *m_cFF2* and the attention layer itself. Without this substitution of pointers, all the work of our neural layer will be incorrect. But if for some reason we re-create the object of the *m_cFF2* inner layer, then new objects of buffers of the specified inner neural layer will be created. In this case, we need to perform such a substitution of pointers again. At the same time, if both variables contain pointers to the same object, then by deleting the object through one pointer, we will end up with an invalid pointer in the second variable. This is a tricky moment that you need to be careful with.

We will, of course, add buffer replacement, but we will first check the correspondence of the pointers.

```
if(m_cFF2.GetOutputs() != m_cOutputs)
{
    if(m_cOutputs)
        delete m_cOutputs;
    m_cOutputs = m_cFF2.GetOutputs();
}

if(m_cFF2.GetGradients() != m_cGradients)
{
    if(m_cGradients)
        delete m_cGradients;
    m_cGradients = m_cFF2.GetGradients();
}

//---
SetOpenCL(m_cOpenCL);
//---
return true;
}
```

After the successful completion of all operations, we exit the method with a positive result.

At this point, we can consider working on creating a neural layer of attention using the standard tools of the MQL5 language to be completed. In this version, we can insert a neural layer of attention into our model and check its performance. To make the most efficient use of the created class, we need to enhance its methods with multithreading capabilities.

5.1.3 Organizing parallel computing in the attention block

In the previous sections, we built a working attention block algorithm using standard MQL5 language capabilities. Now you can add an attention block to your model and test the quality of the *Self-Attention* mechanism. However, look at the block structure. In its operation, we used five internal layers and created an algorithm for transferring data between them in both the forward and backward

directions. It's also important to note that each element of the sequence, described by a value vector, is processed using shared weight matrices, but independently of each other. This allows us to easily distribute operations across parallel threads, enabling us to perform a full set of operations in shorter time intervals. And yes, from the beginning, we decided to create a library with the capability to use two technologies. By doing so, we provide users with the opportunity to independently test and choose the most suitable technology for their specific use case.

As before, we organize the parallel computing unit using OpenCL. To use this technology, we will need to complete two stages of work:

- Create an OpenCL program
- Make changes to the main program

We will add the OpenCL program code to the previously created file [opencl_program.cl](#). It is in this file that we collected all the kernels of the OpenCL program used in the work of the previous classes. To organize the operation of our attention class, we will need to create six kernels. In these kernels, we will need to organize the flow of information between the internal neural layers used in both the forward and backward directions.

First, we'll create the *AttentionFeedForward* kernel. Below is a brief recap on the sequence of operations during the feed-forward pass through the *Self-Attention* block:

1. The source data is fed into three internal convolutional neural layers: *m_cQuery*, *m_cKeys*, *m_cValues*.

$$Input * W_Q = Query$$

$$Input * W_K = Key$$

$$Input * W_V = Value$$

2. The *m_cQuery* and *m_cKeys* result tensors are multiplied to obtain the *m_cScores* dependency matrix.

$$Query * Key^T = S$$

3. The values of the *m_cScores* matrix are divided by the square root of the size of the description vector of one element of the *m_cKeys* sequence and normalized by the *Softmax* function in terms of rows (*m_cQuery* queries).

$$Softmax\left(\frac{S}{\sqrt{Key_{size}}}\right) = Score$$

4. The normalized matrix *m_cScores* is multiplied by the neural layer results tensor *m_cValues* to obtain the *Self-Attention* results.

$$Score * Value = Out_{Self-Attention}$$

5. The results of the *Self-Attention* block are added to the original data and normalized.

$$Normalize(Input + Out_{Self-Attention}) = Output$$

6. The obtained tensor serves as the input data for a block of two convolutional layers: *m_cFF1* and *m_cFF2*.

Points 1 and 6 are covered by using the previously discussed convolutional layer class, which already implements a multi-threaded computation block. So, we will need to implement the remaining points in a new kernel.

To organize the specified operations, we will need to pass six data buffers and two parameters to the kernel. To make the program code more readable, the names of buffers and variables will be aligned with the names of the corresponding matrices in the algorithm description.

```
__kernel void AttentionFeedForward(__global TYPE *querys,
                                   __global TYPE *keys,
                                   __global TYPE *scores,
                                   __global TYPE *values,
                                   __global TYPE *outputs,
                                   int window,
                                   int key_size)
{
```

As you may have noticed from the description of the *Self-Attention* algorithm, the primary analytical unit in this method is an element of the sequence, described by a value vector. For language models, this is usually a word. In the case of financial market analysis, we use a bar. It is precisely between these elements of the sequence that the coefficients of mutual dependencies are determined. Taking into account these coefficients, the values of the element description vectors are adjusted. Therefore, it is quite logical to divide the operations into threads based on the elements of the sequence.

Therefore, in the body of the kernel, the first thing we will do is determine the element of the sequence being analyzed based on the identifier of our thread. At the same time, the total number of running threads will indicate the number of elements in the sequence. Here, we will also immediately determine the offset in the query tensor and the dependency coefficient matrix to the first analyzed value.

```
const int q = get_global_id(0);
const int units = get_global_size(0);
int shift_query = key_size * q;
int shift_scores = units * q;
TYPE summ = 0;
```

To normalize data with the *Softmax* function, we need the sum of the exponents of all normalized values. To calculate it, we add a variable with an initial zero value.

After completing the preparatory work, we will determine the values of one vector from the dependency coefficient matrix, which is related to the calculations for the dependencies of the analyzed element of the sequence. For this, we create a loop with the number of iterations equal to the number of elements in the sequence. In the body of the loop, we will alternately multiply the *Query* vector of the analyzed sequence element with all vectors of the *Key* tensor. For each vector multiplication result, we will take an exponential value and write it into the corresponding element of the *Score matrix*. Of course, we will add the values of the vector to our accumulator sum of all vector values for subsequent normalization.

```
for(int s = 0; s < units; s++)
{
    TYPE score = 0;
    int shift_key = key_size * s;
    for(int k = 0; k < key_size; k++)
        score += querys[shift_query + k] * keys[shift_key + k];
    score = exp(score / sqrt((TYPE)key_size));
    summ += score;
    scores[shift_scores + s] = score;
}
```

After the loop completes, our variable *summ* will accumulate the sum of all elements of our vector from the dependency coefficients tensor. To complete the normalization of the given vector values, all we have to do is divide the value of each of its elements by the total sum of all the values of the vector.

```
for(int s = 0; s < units; s++)
    scores[shift_scores + s] /= summ;
```

In the analyzed vector, we obtained the coefficients of dependencies of the analyzed element of the sequence on the rest of its elements. The sum of all coefficients will be equal to one.

Next, according to the algorithm, we need to multiply each vector of the *Value* tensor by the corresponding element of the resulting vector of dependency coefficients. The resulting vectors need to be added up. The final vector of values will be the result of the *Self-Attention* block.

Before passing the data further, we need to add the obtained data to the tensor of input data and normalize them. In the body of the kernel, I propose focusing on determining the results of the *Self-Attention* block. It will be more efficient to perform matrix addition and data normalization separately across the entire neural layer.

Let's look at the implementation of such a solution. To avoid recalculating at each iteration, we first determine the offset in the tensors of the initial data and results. The tensors have the same dimension, so the offset will be the same for both cases. Then, we will set up a system of two nested loops: in the outer loop, we will iterate over the elements of the vector of the analyzed element of the sequence, and in the inner loop, we will perform the actual computation of the values for each element of the result vector. For this purpose, the number of iterations in the inner loop will be equal to the number of elements in the sequence. In the body of this loop, we will multiply the values of the *Value* tensor elements by the corresponding dependency coefficients from the *Score* matrix. We will accumulate the resulting products in the local variable *query*. After completing the iterations of the inner loop, we will write the result into the corresponding element of the result tensor.

```
shift_query = window * q;
for(int i = 0; i < window; i++)
{
    TYPE query = 0;
    for(int v = 0; v < units; v++)
        query += values>window * v + i] * scores[shift_scores + v];
    outputs[shift_query + i] = query;
}
```

With this, we will complete work on the first feed-forward kernel. The next step is to create a kernel for adding up two tensors. It is sometimes more economical to do such work using matrix operations on the side of the main program. The operation is straightforward, and the overhead of data transfer is unlikely to be justified. We now have the opposite situation. We organize the entire process on the OpenCL context side. All the information is already in the context memory, and to perform the operation on the main program side, we will need to copy the data. We do not need to transfer data if computations are performed within the context. Therefore, we have created a kernel called *Sum*, in which we simply add elements from two buffers with the same index and store the result in an element of the third buffer with the same index.

```
__kernel void Sum(__global TYPE *inputs1,
                  __global TYPE *inputs2,
                  __global TYPE *outputs)
{
```

```

    const int n = get_global_id(0);
    //---
    outputs[n] = inputs1[n] + inputs2[n];
}

```

The data normalization process has a more complex architecture. As you know, its process is expressed by the following mathematical formulas:

$$\bar{a} = \frac{1}{h} \sum_{i=1}^h a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h} \sum_{i=1}^h (a - \bar{a})^2}}$$

As you can notice, to calculate the normalized value of each element in the sequence, you need the arithmetic mean and the root mean square deviation of the entire sequence. To calculate them, we need to organize data transfer between individual threads. We will solve this problem in a way similar to the multi-threaded implementation of the *Softmax* activation function, that is, via an array in local memory. We will need to organize two summation blocks for values across the entire vector because before calculating the arithmetic mean, we cannot compute the variance. Furthermore, we cannot calculate the normalized value until we determine the variance.

The normalization process is organized in the *LayerNormalize* kernel. In the parameters, the kernel receives pointers to 3 buffers:

- Source data buffer
- Results buffer
- Buffer for recording standard deviation parameters

We needed the last standard deviation buffer to save and transmit data to the backpropagation kernel.

Additionally, we will pass two parameters to the kernel: the total number of elements in the buffer being normalized and the offset in the buffer for root mean square deviations. I would like to remind you that within one attention neural layer, we perform data normalization twice. Let's normalize the results of the *Self-Attention* and *FeedForward* blocks.

```

__kernel void LayerNormalize(__global TYPE* inputs,
                             __global TYPE* outputs,
                             __global TYPE* stds,
                             const int total,
                             const int std_shift)
{

```

In the kernel body, we define thread identifiers and initialize a local data array.

```

    uint i = (uint)get_global_id(0);
    uint l = (uint)get_local_id(0);
    uint ls = min((uint)get_local_size(0), (uint)LOCAL_SIZE);
    __local TYPE temp[LOCAL_SIZE];

```

First, we will determine the arithmetic mean of the buffer elements. To do this, we organize a loop in which each thread sums its values and stores the result in its own element of the local array. Since we are calculating the arithmetic mean of the entire buffer, we will divide the obtained value by the number of elements in the buffer.

```
uint count = 0;
do
{
    uint shift = count * ls + l;
    temp[l] = (count > 0 ? temp[l] : 0) + (shift < total ? inputs[shift] : 0);
    count++;
}
while((count * ls + l) < total);
temp[l] /= (TYPE)total;
barrier(CLK_LOCAL_MEM_FENCE);
```

We will synchronize the work of threads using the *barrier* function. Since the calculations of the threads do not overlap, we only need one barrier at the end of the block.

Next, we need to collect parts of the total amount into a single whole. We will organize another loop in which we will collect the arithmetic mean of the buffer into one element of the local array with index 0. The result will be saved in a local variable.

```
count = ls;
do
{
    count = (count + 1) / 2;
    temp[l] += (l < count ? temp[l + count] : 0);
    barrier(CLK_LOCAL_MEM_FENCE);
}
while(count > 1);
//---
TYPE mean = (TYPE) temp[0];
```

I would like to draw your attention once again to the arrangement of barriers. Here you need to pay special attention to the operation of the algorithm because all threads must reach each barrier. Moreover, the sequence of their visits must also be observed.

After determining the arithmetic mean, we repeat the loops and calculate the standard deviation.

```
count = 0;
do
{
    uint shift = count * ls + l;
    temp[l] = (count > 0 ? temp[l] : 0) + (shift < total ? (TYPE)pow(inputs[shift]
    count++;
}
while((count * ls + l) < total);
temp[l] /= (TYPE)total;
barrier(CLK_LOCAL_MEM_FENCE);

count = ls;
do
{
```

```

        count = (count + 1) / 2;
        temp[l] += (l < count ? temp[l + count] : 0);
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    while(count > 1);
//---
    TYPE std = (TYPE)sqrt(temp[0]);
    if(l == 0)
        stds[std_shift] = std;

```

We save the obtained standard deviation into a buffer. To avoid simultaneous writes by all threads, we will save the value in only one thread. To achieve this, we will perform a thread index check before the operation of writing a value to the buffer.

Now that we have calculated the averages, we can normalize the original data. It's important to note that the limitation of the workgroup size may not allow us to allocate a separate thread for each element of the input data buffer. Therefore, we will also implement data normalization in a loop.

```

count = 0;
while((count * ls + l) < total)
{
    uint shift = count * ls + l;
    outputs[shift] = (inputs[shift] - mean) / (std + 1e-37f);
    count++;
}
}

```

This concludes our work with feed-forward kernels. Continuing our work on making additions to the OpenCL program, we move on to building a reverse pass. Its algorithm completely mirrors the path taken above but in the reverse direction. In it, we have to propagate the error gradient from the output of the *Self-Attention* block to the internal neural layers *m_cQuery*, *m_cKeys*, *m_cValues*.

The simplest seems to be the calculation of the error gradient for the internal neural layer, *m_cValues*. Let me remind you that to obtain the result of the *Self-Attention* block, we multiplied the matrix of dependence coefficients *m_cScores* by the tensor of the results of the neural layer *m_cValues*. Therefore, to obtain the error gradient at the output level of the specified neural layer, we need to multiply the gradient obtained from previous operations by the derivative of the last operation. In this case, we have to multiply the matrix of dependency coefficients by the tensor of error gradients from previous operations.

$$G_{Value} = Score^T * G_{Self-Attention}$$

After determining the error gradient on the internal neural layer *m_cValues*, we need to distribute the error gradient to two more internal neural layers, *m_cQuery*s and *m_cKeys*. However, in order to bring the error gradient to the level of the specified neural layers, it is necessary to pass it through the matrix of dependency coefficients.

$$G_{Score} = G_{Self-Attention} * V^T$$

However, when implementing in MQL5, we do not create an additional buffer for error gradients at the level of the dependency coefficient matrix. But in OpenCL there is difficulty in allocating a dynamic array for recording intermediate data about the error gradient values at the dependency coefficient matrix level. Therefore, here we will create two temporary data buffers: the first for the error gradient

of the normalized data, and the second for the error gradients corrected by the derivative of the *Softmax* function.

Note that when we recalculate the error gradient to the level of the *m_cQuery*s and *m_cKeys* neural layers, the same elements of the dependency coefficient error gradient matrix are used in different operation threads. Therefore, we will divide the entire backpropagation algorithm within the attention layer into two blocks. In the first block, we will propagate the error gradient to the level of the internal neural layer of *m_cValues* value and the *m_cScores* coefficient matrix. In the second block, we will propagate the error gradient to two other neural layers: *m_cQuery*s and *m_cKeys*.

We implement the first block of operations in the *AttentionCalcScoreGradient* kernel. In the parameters of this kernel, we pass pointers to five data buffers and one parameter:

- *scores* – dependency coefficient matrix buffer
- *scores_temp* – buffer of error gradients at the level of the normalized dependency coefficient matrix
- *scores_grad* – buffer of error gradients at the level of the dependency coefficient matrix, adjusted to the derivative of the normalization function
- *values* – tensor buffer *Values* (buffer of neural layer results *m_cValues*)
- *values_grad* – error gradient tensor buffer at the level of results of the *m_cValues* neural layer
- *outputs_grad* is the buffer of error gradients at the output level of the *Self-Attention* block;
- *window* is the size of the description vector of one element of the sequence in the *Values* tensor.

Please note that the *scores_temp* and *scores_grad* buffers have no counterparts on the main program side. The reason is that we only need error gradients at the level of the dependency coefficient matrix to perform the operations of the current backward pass. However, OpenCL does not have the ability to create dynamic arrays. We created the specified buffers instead.

```
__kernel void AttentionCalcScoreGradient(__global TYPE *scores,
                                         __global TYPE *scores_grad,
                                         __global TYPE *values,
                                         __global TYPE *values_grad,
                                         __global TYPE *outputs_grad,
                                         __global TYPE *scores_temp,
                                         int window)
{
```

The feed-forward algorithm involves normalizing the dependency coefficient matrix *Score* with the *Softmax* function in the context of *Query* requests. So, after determining the error gradients at the coefficient matrix level, it is necessary to adjust these values based on the derivative of the data normalization operation. Therefore, it would be logical to divide the operations into threads in the same manner. Moreover, such a distribution of operations into threads would be entirely appropriate for propagating the error gradient to the level of values within the internal neural layer.

At the beginning of the kernel, we do a little preparatory work. We determine the serial number of the analyzed vector of values and rows of the matrix of dependency coefficients by the identification number of the thread. The total number of running threads will tell us the dimensions of the tensors. Let's immediately determine the offset in the data buffers to the first element of the analyzed vectors of values.

```
const int q = get_global_id(0);
const int units = get_global_size(0);
```



```
int shift_value = window * q;
int shift_score = units * q;
```

Next, we will propagate the error gradient to the level of the internal neural layer *m_cValues*. As mentioned above, to determine the error gradient, we need to multiply the transposed matrix of dependency coefficients by the gradient tensor at the output of the *Self-Attention* block.

Within the kernel, we will define the error gradient for only one vector of element description. As you know, with a feed-forward pass, each element of the sequence in the *Value* tensor leaves its mark in the formation of all elements of the sequence of results of the *Self-Attention* block. Consequently, each element of the *Value* tensor must receive its share of the error gradient from all elements of the results tensor of the *Self-Attention* block. The measure of influence will be the corresponding dependence coefficient from the *Score* matrix. Thus, each element of the sequence of the *Value* tensor corresponds to one column in the dependency coefficient matrix *Score*. This explains the use of the transposed *Score* matrix in the formula above.

To organize this process, we will create a system of two nested loops. The number of iterations in the first loop is equal to the size of the vector describing one element of the sequence in the *Value* tensor. It should be noted that the error gradient tensor at the output of the *Self-Attention* block has the same dimensions. In the nested loop with a number of iterations equal to the number of elements in the sequence, we will iterate over the values of the corresponding column of the dependency coefficient matrix *Score* and the gradient vector of errors at the level of the *Self-Attention* block results. In this case, we will multiply the corresponding elements and sum the resulting products into a private variable. After completing the iterations of the inner loop, copy the accumulated sum of products to the error gradient buffer of the internal convolutional layer *m_cValues*.

```
//--- Distributing the gradient on Values
for(int i = 0; i < window; i++)
{
    TYPE grad = 0;
    for(int g = 0; g < units; g++)
        grad += scores[units * g + q] * outputs_grad[window * g + i];
    values_grad[shift_value + i] = grad;
}
```

After the execution of the loop system the first part of our task, in which we propagate the error gradients to the internal neural layer *m_cValues*, can be considered complete.

The second part of our kernel is devoted to determining the error gradient at the level of the dependency coefficient matrix.

In the feed-forward pass, each element of the *Query* sequence is multiplied with all the elements of the *Key* sequence to form a single dependency coefficient matrix vector *Score*. Each such vector is normalized by the function *Softmax*. After that, we multiply it by the *Value* tensor. As a result of these operations, we obtain the corrected vector representation of one element of the sequence in the tensor of the *Self-Attention* block results. Thus, one element of the *Query* sequence interacts with all elements of the *Key* and *Value* tensors to form a vector describing one element of the result sequence. Therefore, to distribute the error gradient to a specific vector from the *Query* tensor, we need to take one corresponding error gradient vector of one element of the sequence at the level of the *Self-Attention* block and first multiply it by the transposed tensor of *Value*. Thus, we obtain an error vector at the level of the dependency coefficient matrix *Score*. Next, we need to adjust the resulting vector to the derivative of the *Softmax* function. It is this part of the error gradient distribution that we

implement in this kernel. To further propagate the error gradient to the level of the internal neural layers $m\_cQueries$ and m_cKeys , we will create another kernel a little later.

The error gradient distribution algorithm described above in matrix form can be represented as follows:

1. Error gradient at the Score matrix level.

$$G_{Score} = G_{Self-Attention} * V^T$$

2. Adjusting the error gradient to the derivative of the *Softmax* function.

$$G'_{Score} = S^T(E - S) \circ G_{Score}$$

Let's summarize the entire calculation into one formula:

$$G'_{Score} = S^T(E - S) \circ (G_{Self-Attention} V^T)$$

First, let's propagate the error gradient to the level of the dependency coefficient matrix *Score*. Since, thanks to the division of operations into parallel threads within the kernel, we will be determining the error gradient for only one row, to calculate this error gradient vector, we need to take the error gradient vector for one element of the sequence at the level of the *Self-Attention* block results and multiply it by the transposed tensor of the internal layer's results, $m\_cValues$. In practice, we will use the algorithm described above when calculating error gradients for the $m\_cValues$ layer. We will create a system of two nested loops. But this time, the number of iterations of the outer loop will be equal to the number of elements in the sequence. The nested loop will repeat its operations for the number of elements in the vector describing one element of the sequence. This difference is explained by the magnitude of the vector of results and is confirmed by the logic of the operations performed. Remember, with a forward pass, each element in the row of the dependency coefficient matrix corresponds to one vector describing the sequence element in the *Values* tensor.

```
//--- Gradient distribution on Score
for(int k = 0; k < units; k++)
{
    TYPE grad = 0;
    for(int i = 0; i < window; i++)
        grad += outputs_grad[shift_value + i] * values[window * k + i];
    scores_temp[shift_score + k] = grad;
}
```

After transferring the error gradient to the level of the dependency coefficient matrix, we need to adjust the obtained values using the derivative of the *Softmax* normalization function. Just like with the forward pass, when in order to obtain one normalized value it was necessary to process the entire vector of normalized values, to calculate one adjusted value we need to use all the elements of both vectors (error gradients at the level of the matrix of dependence coefficients and the normalized vector of coefficients itself).

The matrix expression of the process of adjusting for the derivative of the *Softmax* function is given above. For practical implementation, we will create a system of two nested loops. Both loops have the same number of iterations, which is equal to the size of the vector being normalized. In this case, it is equal to the number of elements in the sequence. When performing operations, it will be necessary to accumulate the sum of error gradients from each element of the normalized vector. To do this, we will create a private variable in the body of the outer loop *grad*. Besides, to reduce the number of accesses to global memory, we will store the repeated element in the private variable *score*. Let me remind you that accessing global memory is more time-consuming. So, by reducing the number of accesses to global memory buffers, we reduce the overall time spent on operations. In the body of the nested loop,

we will perform operations of multiplying elements and adding the resulting products into a previously created private variable *grad*.

Please note that we have replaced the identity matrix with the expression $(int)(i==k)$. The logical expression will give us the true value only on the diagonal of the matrix. Translating a boolean value into an integer will substitute 1 for true values and 0 for false values. Thus, such a short notation allows us to obtain the values of the identity matrix directly in the operation thread, without the need to first generate and save it.

```
//--- Adjust for the Softmax derivative
for(int k = 0; k < units; k++)
{
    TYPE grad = 0;
    TYPE score = scores[shift_score + k];
    for(int i = 0; i < units; i++)
        grad += scores[shift_score + i] *
                ((int)(i == k) - score) * scores_temp[shift_score + i];
    scores_grad[shift_score + k] = grad;
}
```

After completing the iterations of the loop system, we will obtain the error gradients at the level of the dependency coefficient matrix, adjusted for the derivative of the *Softmax* function.

With that, we conclude the first backpropagation kernel and move on to creating the second kernel *AttentionCalcHiddenGradient*, in which we will propagate the error gradient to the internal neural layers *m_cQuerys* and *m_cKeys*. To do this, in the kernel parameters we need to pass pointers to five data buffers and one constant:

- *querys* – buffer of results of the internal neural layer *m_cQuerys*
- *queries_grad* – buffer of error gradients of the internal neural layer *m_cQuerys*
- *keys* – buffer of results of the internal neural layer *m_cKeys*
- *keys_grad* – buffer of error gradients of the internal neural layer *m_cKeys*
- *scores_grad* – buffer of error gradients of dependency coefficient matrix *m_cScores*
- *key_size* – size of the key vector of one element

```
__kernel void AttentionCalcHiddenGradient(__global TYPE *querys,
                                         __global TYPE *queries_grad,
                                         __global TYPE *keys,
                                         __global TYPE *keys_grad,
                                         __global TYPE *scores_grad,
                                         int key_size)
{
```

Following the analogy with all the kernels discussed earlier, we will distribute the operations into threads in the context of a single element of the sequence. At the beginning of the kernel, we will perform preparatory work and determine the offsets in the data buffers to the first element of the vector of the analyzed element.

```
const int q = get_global_id(0);
const int units = get_global_size(0);
int shift_query = key_size * q;
int shift_score = units * q;
```

In the *AttentionCalcScoreGradient* kernel discussed above, we have already adjusted the error gradient of the dependency coefficient matrix to the derivative of the *Softmax* normalization function. However, during the feed-forward pass, before normalizing the matrix, we divided all its elements by the square root of the dimension of the key vector. Now we need to adjust the error gradient for the derivative of the mentioned operation. Similar to the multiplication operation, we will need to divide all the values of the error gradient buffer of the dependency coefficient matrix by the same constant.

$$\left(\frac{x}{\text{Constant}}\right)' = \left(\frac{1}{\text{Constant}}x\right)' = \frac{1}{\text{Constant}}$$

Let's determine the value of the constant and store it in a private variable.

```
//--- Distribute the gradient on Querys and Keys
const TYPE k = 1 / sqrt((TYPE)key_size);
```

This concludes the preparatory work. Now we can proceed directly to recalculating the error gradients. To obtain dependency coefficients, we multiplied two tensors (*Query* and *Keys*). We have already encountered derivatives of multiplication operations more than once. To obtain error gradients for one of the tensors, we need to multiply the error gradient tensor at the level of the dependency coefficient matrix by the second tensor. Since the *Query* and *Key* tensors have the same dimensions, we can calculate the error gradients for both tensors in the same loop system.

Let's create a system of two nested loops. The outer loop has a number of iterations equal to the size of the key vector of one sequence element. In the nested loop, we iterate through the vectors of the opposite tensor and the corresponding error gradients of the dependency coefficient matrix. Therefore, the number of its iterations will be equal to the number of elements in the analyzed sequence.

As a result, the number of iterations in the nested loop will be equal to the number of elements in the analyzed sequence. The results of these products will need to be summarized. To accumulate this amount, we will create two private variables *grad_q* and *grad_k* before declaring the nested loop.

Also, please note the following. To reduce the number of calculation operations, we will not add our previously calculated coefficient to adjust the error gradient to the products of the nested loop. We will use the mathematical properties of functions and take the constant factor out of brackets.

$$ka + kb + kc = k(a + b + c)$$

Thus, there is no need to multiply the value each time by a correction factor in the body of the nested loop. Instead, we can simply multiply the total amount once by the correction factor before writing it to the data buffer.

```
for(int i = 0; i < key_size; i++)
{
    TYPE grad_q = 0;
    TYPE grad_k = 0;
    for(int s = 0; s < units; s++)
    {
        grad_q += keys[key_size * s + i] * scores_grad[shift_score + s];
        grad_k += queries[key_size * s + i] * scores_grad[units * s + q];
    }
    queries_grad[shift_query + i] = grad_q * k;
    keys_grad[shift_query + i] = grad_k * k;
}
```

At the output of the loop system, we get error gradients for two nested internal neural layers *m_cQuery*s and *m_cKeys*. That is, the task of this kernel is solved. Considering the previously discussed *AttentionCalcScoreGradient* kernel, we have distributed the error gradient to all internal neural layers, and further distribution of the error gradient to the previous layer will be carried out using the well-tested methods of internal neural layers, as implemented by standard MQL5 means.

The backpropagation kernels discussed above bypassed the processes of adding result buffers and data normalization that we carried out during the feed-forward pass. The derivative of two functions is equal to the sum of the derivatives of these functions. So, for the operation of adding gradients, we can use a similar feed-forward kernel. We just need to specify the correct data buffers.

$$(f(x) + g(x))' = (f(x))' + (g(x))'$$

In the case of adjusting the error gradient to the data normalization function, we will have to create an additional kernel. Below is the error gradient correction formulas.

$$\frac{\partial f(a_i)}{\partial \sigma^2} = \sum_{i=1}^m \frac{\partial f(a_i)}{\partial \hat{a}_i} * (a_i - \bar{a}) * \frac{-1}{2} (\sigma^2 + \varepsilon)^{-\frac{3}{2}}$$

$$\frac{\partial f(a_i)}{\partial \bar{a}} = \left(\sum_{i=1}^m \frac{\partial f(a_i)}{\partial \hat{a}_i} * \frac{-1}{\sqrt{\sigma^2 + \varepsilon}} \right) + \left(\frac{\partial f(a_i)}{\partial \sigma^2} * \frac{-2 \sum_{i=1}^m (a_i - \bar{a})}{m} \right)$$

$$\frac{\partial f(a_i)}{\partial a_i} = \frac{\partial f(a_i)}{\partial \hat{a}_i} * \frac{1}{\sqrt{\sigma^2 + \varepsilon}} + \frac{\partial f(a_i)}{\partial \sigma^2} * \frac{2(a_i - \bar{a})}{m} + \frac{\partial f(a_i)}{\partial \bar{a}} * \frac{1}{m}$$

As you can see, in the formulas provided above, when calculating derivatives with respect to the means, the sum of values across the entire value buffer is used. However, unlike the forward pass, we have the ability to calculate all three sums in parallel.

In the kernel parameters, we pass pointers to four data buffers:

- *outputs* – buffer of forward pass normalization results
- *out_gradient* – buffer of gradients at the output of the normalization block
- *inp_gradient* – buffer for writing adjusted gradients
- *stds* – buffer of standard deviations calculated during the feed-forward pass

Also, in the parameters we will indicate the size of the buffers and the offset in the standard deviation buffer.

```
__kernel void LayerNormalizeGradient(__global TYPE* outputs,
                                     __global TYPE* out_gradient,
                                     __global TYPE* inp_gradient,
                                     __global TYPE* stds,
                                     const int total,
                                     const int std_shift)
{
    uint i = (uint)get_global_id(0);
    uint l = (uint)get_local_id(0);
```

In the kernel body we define thread identifiers and at the same time declare local data arrays. There will be three of them. In one, we will collect the derivative of the root mean square deviation, and the other two are intended for the terms in the derivative formula of the arithmetic mean.

```
uint ls = min((uint)get_local_size(0), (uint)LOCAL_SIZE);
__local TYPE dSTD[LOCAL_SIZE];
__local TYPE dMean1[LOCAL_SIZE];
__local TYPE dMean2[LOCAL_SIZE];
```

As with the feed-forward pass, each thread will first collect its share of the total.

```
uint count = 0;
do
{
    uint shift = count * ls + l;
    dSTD[l] = (count > 0 ? dSTD[l] : 0) -
        (shift < total ? out_gradient[shift] * outputs[shift] /
            (2 * (pow(stds[std_shift], (TYPE)2) + 1e-37f)) : 0);
    dMean1[l] = (count > 0 ? dMean1[l] : 0) -
        (shift < total ? out_gradient[shift] /
            (stds[std_shift] + 1e-37f) : 0);
    dMean2[l] = (count > 0 ? dMean2[l] : 0) -
        (shift < total ? 2 * outputs[shift] * stds[std_shift] /
            (TYPE)total : 0);
    count++;
}
while((count * ls + l) < total);
barrier(CLK_LOCAL_MEM_FENCE);
```

In the next loop, we will collect the sum in the first elements of the array.

```
count = ls;
do
{
    count = (count + 1) / 2;
    dSTD[l] += (l < count ? dSTD[l + count] : 0);
    dMean1[l] += (l < count ? dMean1[l + count] : 0);
    dMean2[l] += (l < count ? dMean2[l + count] : 0);
    barrier(CLK_LOCAL_MEM_FENCE);
}
while(count > 1);
//---
TYPE dstd = dSTD[0];
TYPE dmean = dMean1[0] + dstd * dMean2[0];
```

We will transfer the resulting values to private variables. When calculating the derivative of the arithmetic mean deviation, we multiply the value of the right term by the derivative of the standard deviation and add it to the left term.

At this stage, we have enough data to adjust the error gradient for each buffer element. Let's organize another loop, in the body of which this work will be performed.

```
//---
count = 0;
```

```

while((count * ls + l) < total)
{
    uint shift = count * ls + l;
    inp_gradient[shift] = out_gradient[shift] / (stds[std_shift] + 1e-32f) +
        (2 * dstd * outputs[shift] * stds[std_shift] + dmean) / total;
    count++;
}
}

```

This concludes our work with the OpenCL program. Now we need to proceed with the second part and set up the preparatory work for launching multi-threaded computations on the main program side.

First, let's add constants for working with kernels to the *defines.mqh* file. We need to add constants for identifying the kernels themselves and their variables. To name the constants, we use the previously agreed rules that apply to all constants within our project:

- All constants begin with the prefix *def*.
- Kernels begin with the prefix *def_k*.
- Parameter constants after the *def* prefix contain a pointer to the kernel.

```

#define def_k_AttentionFeedForward      28
#define def_k_AttentionScoreGradients  29
#define def_k_AttentionHiddenGradients 30
#define def_k_Sum                       31
#define def_k_LayerNormalize            32
#define def_k_LayerNormalizeGradient    33

//--- feed-forward pass of the attention block
#define def_attff_querys                 0
#define def_attff_keys                   1
#define def_attff_scores                  2
#define def_attff_values                  3
#define def_attff_outputs                 4
#define def_attff_window                  5
#define def_attff_key_size                6

//--- determine the gradient on the matrix of dependence coefficients of the attentio
#define def_attscr_scores                 0
#define def_attscr_scores_grad            1
#define def_attscr_values                 2
#define def_attscr_values_grad            3
#define def_attscr_outputs_grad           4
#define def_attscr_scores_temp            5
#define def_attscr_window                 6

//--- gradient distribution through the attention block
#define def_atthgr_querys                 0
#define def_atthgr_querys_grad            1
#define def_atthgr_keys                   2
#define def_atthgr_keys_grad              3
#define def_atthgr_scores_grad            4
#define def_atthgr_key_size               5

//--- sum of vectors

```

```

#define def_sum_inputs1          0
#define def_sum_inputs2          1
#define def_sum_outputs          2

//--- vector normalization
#define def_layernorm_inputs      0
#define def_layernorm_outputs     1
#define def_layernorm_std         2
#define def_layernorm_vector_size 3
#define def_layernorm_std_shift   4

//--- vector normalization gradient
#define def_layernormgr_outputs    0
#define def_layernormgr_out_grad   1
#define def_layernormgr_inp_grad   2
#define def_layernormgr_std        3
#define def_layernormgr_vector_size 4
#define def_layernormgr_std_shift   5

```

After that, we will need to add the declaration of the new kernels to the code of the main neural network dispatcher class. Like all previously created kernels, we will add the declaration of new kernels to the *CNet::InitOpenCL* method. In it, we will first update the total number of kernels used in the program.

```

if(!m_cOpenCL.SetKernelsCount(34))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}

```

After this, we will declare the kernels themselves.

```

if(!m_cOpenCL.KernelCreate(def_k_AttentionFeedForward,
                           "AttentionFeedForward"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}

if(!m_cOpenCL.KernelCreate(def_k_AttentionScoreGradients,
                           "AttentionCalcScoreGradient"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}

if(!m_cOpenCL.KernelCreate(def_k_AttentionHiddenGradients,
                           "AttentionCalcHiddenGradient"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
}

```



```

        return false;
    }

    if(!m_cOpenCL.KernelCreate(def_k_Sum, "Sum"))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        return false;
    }

    if(!m_cOpenCL.KernelCreate(def_k_LayerNormalize, "LayerNormalize"))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        return false;
    }

    if(!m_cOpenCL.KernelCreate(def_k_LayerNormalizeGradient
                               "LayerNormalizeGradient"))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        return false;
    }
}

```

Then we move on to the attention mechanism class *CNeuronAttention* and make changes to its methods in terms of working with OpenCL technology.

Let's first add the feed-forward pass method *CNeuronAttention::FeedForward*. In this method, we need to organize a procedure for calling the feed-forward kernel *AttentionFeedForward*. We have created similar processes multiple times. So, its algorithm is as follows:

1. Check the presence of data buffers in the OpenCL context.
2. Pass parameters to the kernel, including pointers to data buffers.
3. Queue the kernel to perform operations.

While doing so, we must ensure proper control of the operations to avoid potential critical errors during the program execution.

```

bool CNeuronAttention::FeedForward(CNeuronBase *prevLayer)
{
    //--- calculation of vectors Query, Key, Value
    .....
    //--- Branching the algorithm on the computing device
    MATRIX out;
    if(!m_cOpenCL)
    {
        // MQL5 block
        .....
    }
    else // OpenCL block
    {
        //--- checking data buffers
        if(m_cQuerys.GetOutputs().GetIndex() < 0)

```

```

        return false;
    if(m_cKeys.GetOutputs().GetIndex() < 0)
        return false;
    if(m_cValues.GetOutputs().GetIndex() < 0)
        return false;
    if(m_cScores.GetIndex() < 0)
        return false;
    if(m_cAttentionOut.GetOutputs().GetIndex() < 0)
        return false;

```

With all the necessary buffers in the *OpenCL* context, we will set up the transfer of pointers to them as kernel parameters.

```

//--- pass parameters to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward, def_attff_keys,
                                m_cKeys.GetOutputs().GetIndex()))

    return false;

if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward, def_attff_outputs,
                                m_cAttentionOut.GetOutputs().GetIndex()))

    return false;

if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward, def_attff_queries,
                                m_cQueries.GetOutputs().GetIndex()))

    return false;

if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward, def_attff_scores,
                                m_cScores.GetIndex()))

    return false;

if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward, def_attff_values,
                                m_cValues.GetOutputs().GetIndex()))

    return false;

if(!m_cOpenCL.SetArgument(def_k_AttentionFeedForward, def_attff_key_size,
                           m_iKeysSize))

    return false;

if(!m_cOpenCL.SetArgument(def_k_AttentionFeedForward, def_attff_window,
                           m_iWindow))

    return false;

```

Next comes the procedure for placing the kernel in the execution queue. First, let's indicate the number of required threads to launch and the offset. Only after that, we will call the kernel launch function, providing it with information about the number of instances to be launched.

```

//--- put the kernel into the execution queue
int off_set[] = {0};
int NDRange[] = {m_iUnits};
if(!m_cOpenCL.Execute(def_k_AttentionFeedForward, 1, off_set, NDRange))
    return false;
}

```

This concludes the algorithm for launching the kernel of the *Self-Attention* block. However, we still need to add the contents of two buffers and normalize the data in the result buffer. Following the algorithm, first, we find the sum of two vectors (initial data and *Self-Attention* results). This operation is quite

general and can be widely used outside of our neural attention layer class *CNeuronAttention*. Therefore, I decided to add it as a separate method to the data buffer class *CBufferType::SumArray*.

In the parameters to the *SumArray* method, we will pass a pointer to the buffer to be added. Immediately in the body of the method, we check the received pointer and the size of the received buffer. To successfully complete the operation, the size of the current buffer, which will be the first addend, and the resulting buffer (the second addend) must be equal.

```
bool CBufferType::SumArray(CBufferType *src)
{
    //--- check the source data array
    if(!src || src.Total() != Total())
        return false;
```

Like all the methods discussed earlier, the algorithm of this method is split into two threads depending on the execution device. In the block of performing operations using means MQL5 we will first match the matrix formats of both buffers. Then we perform the matrix addition operation. The result of the operation will be saved in the current buffer matrix.

```
if(!m_cOpenCL)
{
    //--- change the matrix size
    MATRIX temp = src.m_mMatrix;
    if(!temp.Reshape(Rows(), Cols()))
        return false;
    //--- add matrices
    m_mMatrix += temp;
}
```

The algorithm for the block of multi-threaded operations is similar to that discussed above. First, we check for the presence of data in the context of OpenCL and, if necessary, load the data from the resulting buffer. Please note that we only check the received buffer. Earlier, when dividing the algorithm depending on the computing device, we already checked the pointer to the current OpenCL context of the buffer. Therefore, we consider the data of the current buffer to have already been transferred to the OpenCL context.

The control block is followed by passing parameters to the kernel and placing it in the execution queue.

```
else
{
    if(src.GetIndex() < 0 && !BufferCreate(m_cOpenCL))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_Sum, def_sum_inputs1, m_myIndex))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_Sum, def_sum_inputs2, src.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_Sum, def_sum_outputs, m_myIndex))
        return false;
    uint off_set[] = {0};
    uint NDRange[] = {(uint)Total()};
    if(!m_cOpenCL.Execute(def_k_Sum, 1, off_set, NDRange))
        return false;
}
//---
```

```

    return true;
}

```

The data normalization process is organized in the *CNeuronAttention::NormlizeBuffer* method. However, while following the general rules for constructing the algorithm, there are two exceptions in this method. First, we eliminated the block for checking the presence of buffers in the OpenCL context. In this case, the risk of using unloaded buffers is minimal. The reason is that before calling this method, the used buffers have already been checked multiple times, and rechecking them would be unnecessary.

```

bool CNeuronAttention::NormlizeBuffer(CBufferType *buffer,
                                     CBufferType *std,
                                     uint std_shift)
{
    if(!m_cOpenCL)
    {
        // MQL5 block
        .....
    }
    else
    {
        if(!m_cOpenCL.SetArgumentBuffer(def_k_LayerNormalize,
                                       def_layernorm_inputs, buffer.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgumentBuffer(def_k_LayerNormalize,
                                       def_layernorm_outputs, buffer.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgumentBuffer(def_k_LayerNormalize,
                                       def_layernorm_std, std.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgument(def_k_LayerNormalize,
                                def_layernorm_vector_size, (int)buffer.Total()))
            return false;
        if(!m_cOpenCL.SetArgument(def_k_LayerNormalize,
                                def_layernorm_std_shift, std_shift))
            return false;
    }
}

```

The second point is related to the use of a local data array and thread synchronization. The reason is that thread synchronization is only available within a work group. We need to explicitly specify its size. The normalization algorithm in the kernel is structured in such a way that the workgroup size cannot be greater than the size of the local array. Let me remind you that the size of the local array is determined by the *LOCAL_SIZE* constant. At the same time, the number of threads cannot be greater than the size of the normalized buffer. Therefore, in the array indicating the dimension of the task space, we will indicate the smaller of the two values. Since we normalize the values of the entire buffer in one batch, the dimensions of the global and local task space will be the same.

Once we have determined the problem dimensions, we enqueue the kernel for execution.

```

int NDRange[] = {(int)MathMin(buffer.Total(), LOCAL_SIZE)};
int off_set[] = {0};
if(!m_cOpenCL.Execute(def_k_LayerNormalize, 1, off_set, NDRange, NDRange))
    return false;
}

```

```
//---
    return true;
}
```

This concludes the block of using OpenCL technology in the feed-forward method of our attention engine class, and we are finished working on this method. Further along, its code remains unchanged. The complete code is given in the description section of [constructing a method](#) using standard MQL5 tools.

We are now moving on to working on one of the backpropagation methods – the method of distributing the error gradient through a hidden layer *CNeuronAttention::CalcHiddenGradient*. The algorithm of our actions remains the same. We will only make an adjustment for the use of two kernels sequentially.

I would like to remind you that when creating backpropagation kernels, we determined the need to use two additional buffers for storing intermediate values of error gradients of the dependency coefficient matrix. So let's take a step back and declare additional buffers: *m_cScoreGrad* and *m_cScoreTemp*.

```
class CNeuronAttention    : public CNeuronBase
{
protected:
    .....
    int                m_cScoreGrad;
    int                m_cScoreTemp;
    .....
};
```

However, in this case, we will not declare instances of buffer objects in main memory. We will not use these buffers to exchange data between the OpenCL context and the main program. They are needed only for temporary storage of data transferred between kernels. This means that their presence in the OpenCL context memory is enough for us. In the main program, we will only declare variables to store pointers to buffers.

Let's get back to working on the *CNeuronAttention::CalcHiddenGradient* method. First, we check the availability and, if necessary, create new data buffers in the *OpenCL* context, used in the first kernel. We intentionally do not create data buffers for the second kernel right away to ensure more efficient memory usage. This will allow us to use larger data buffers when OpenCL context memory resources are limited.

```
bool CNeuronAttention::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    .....
    //--- branching the algorithm across the computing device
    if(!m_cOpenCL)
    {
        // MQL5 block
        .....
    }
    else // OpenCL block
    {
        //--- check data buffers
        if(m_cValues.GetOutputs().GetIndex() < 0)
            return false;
        if(m_cValues.GetGradients().GetIndex() < 0)
            return false;
```

```

if(m_cScores.GetIndex() < 0)
    return false;
if(m_cAttentionOut.GetGradients().GetIndex() < 0)
    return false;
if(m_cScoreGrad < 0)
    return false;
if(m_cScoreTemp < 0)
    return false;

```

After checking all the necessary buffers, we will pass pointers to them to the kernel.

```

//--- pass parameters to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
    def_attscr_outputs_grad, m_cAttentionOut.GetGradients().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
    def_attscr_scores, m_cScores.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
    def_attscr_scores_grad, m_cScoreGrad))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
    def_attscr_scores_temp, m_cScoreTemp))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
    def_attscr_values, m_cValues.GetOutputs().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
    def_attscr_values_grad, m_cValues.GetGradients().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_AttentionScoreGradients,
    def_attscr_window, m_iWindow))
    return false;

```

In addition to the pointers to data buffers, we pass the size of the vector describing one element of the sequence to the kernel.

After passing all the parameters, specify the number of required parallel threads and invoke the function to enqueue the kernel.

```

//--- Place the kernel in the execution queue
int off_set[] = {0};
int NDRange[] = {m_iUnits};
if(!m_cOpenCL.Execute(def_k_AttentionScoreGradients, 1, off_set, NDRange))
    return false;

```

Now we move on to working on the next kernel. Let's check the availability of buffers required for the new kernel.

```

if(m_cQuerys.GetOutputs().GetIndex() < 0)
    return false;
if(m_cQuerys.GetGradients().GetIndex() < 0)
    return false;
if(m_cKeys.GetOutputs().GetIndex() < 0)

```

```

    return false;
    if(m_cKeys.GetGradients().GetIndex() < 0)
        return false;

```

After checking all the necessary data buffers, we will pass pointers to them to the kernel.

```

    if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                                    def_atthgr_keys, m_cKeys.GetOutputs().GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                                    def_atthgr_keys_grad, m_cKeys.GetGradients().GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                                    def_atthgr_querys, m_cQuerys.GetOutputs().GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                                    def_atthgr_querys_grad, m_cQuerys.GetGradients().GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                                    def_atthgr_scores_grad, m_cScoreGrad))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_AttentionHiddenGradients,
                              def_atthgr_key_size, m_iKeysSize))
        return false;

```

In addition to the pointers to data buffers, we pass the size of the key vector for one element of the sequence to the kernel parameters.

After finishing the transfer of all the necessary data to the kernel, we initialize the enqueueing of its execution. The arrays with the specified offset and the number of required kernel instances for execution are already prepared after launching the previous kernel, and we don't need to set them again. Therefore, we simply invoke the function to enqueue the kernel.

```

    if(!m_cOpenCL.Execute(def_k_AttentionHiddenGradients, 1, off_set, NDRange))
        return false;

```

At this point, we conclude our work on building the methods of our attention class and can proceed to test its functionality.

5.1.4 Testing the attention mechanism

Unlike the *LSTM* recurrent block discussed earlier, the attention block works only with current data. Therefore, to create a more representative sample between weight updates during the training of a neural network, we will use random patterns from the general training dataset. We used this approach when testing the fully connected perceptron and the convolutional model. In such a situation, it will be quite logical to take the [convolution_test.mq5](#) script we used for testing the convolutional mode, re-save it with a new name *attention_test.mq5*, and make changes to the description of the created model accordingly.

Note that many changes were required to create the test script. We have removed the description blocks of the convolutional and pooling layers from the script. Instead of them, right after the input data, we will add a description of our attention block. To do this, as with any other neural layer, we will create a new instance of the *CLayerDescription* neural layer description class and immediately check

the result of the operation based on the obtained pointer to the object. Next, we need to provide descriptions for the created neural layer.

In the *type* field, we will pass the *defNeuronAttention* constant, which corresponds to the attention block to be created.

In the *count* field, we must specify the number of elements of the sequence to be analyzed. We request it from the user when running the script and save it to the *BarsToLine* variable. Therefore, in the description of the neural layer, we can pass the value of the variable.

The *window* parameter was used to specify the size of the source data window when describing the convolutional layer. Here we will use it to specify the size of the description vector for one element of the input data sequence. Even though the descriptions are slightly different, the functions are similar. However, unlike the convolutional layer, we will not specify the step of the window, since in this case, it will be equal to the window itself. The number of neurons used to describe one candlestick is also requested from the user in the script parameters. This value is stored in the *NeuronsToBar* variable. As in the previous field case, we simply pass the value from the variable to the specified field.

The *Self-Attention* algorithm does not provide data resizing. At the output of the block, we obtain a tensor of the same size as the original data. It turns out that the *window_out* field in the description of the neural layer will remain unclaimed. But we'll use it to specify the size of the key vector of a single element in the *Key* tensor. In practice, the size of the key is not always different from the size of the vector describing one element. Dimensionality reduction is employed when the size of the description vector for a single element is large to conserve computational resources during the calculation of the dependency coefficient matrix. In our case, when the description vector of one candlestick is only four elements, we will not lower the dimension and pass to the *window_out* field the value of the *NeuronsToBar* variable.

Additionally, we will specify the optimization method and its parameters. In the test case, I used the *Adam* method, as I did in all previous tests.

```
bool CreateLayersDesc(CArrayObj &layers)
{
    CLayerDescription *descr;
    //--- create a source data layer
    .....
    //--- attention layer
    if(!(descr = new CLayerDescription()))
    {
        PrintFormat("Error creating CLayerDescription: %d", GetLastError());
        return false;
    }
    descr.type = defNeuronAttention;
    descr.count = BarsToLine;
    descr.window = NeuronsToBar;
    descr.window_out = NeuronsToBar;
    descr.optimization = Adam;
    descr.activation_params[0] = 1;
    if(!layers.Add(descr))
    {
        PrintFormat("Error adding layer: %d", GetLastError());
        delete descr;
        return false;
    }
}
```



```

    }
    //---hidden layer
    .....
    //--- Results layer
    .....
    //---
    return true;
}

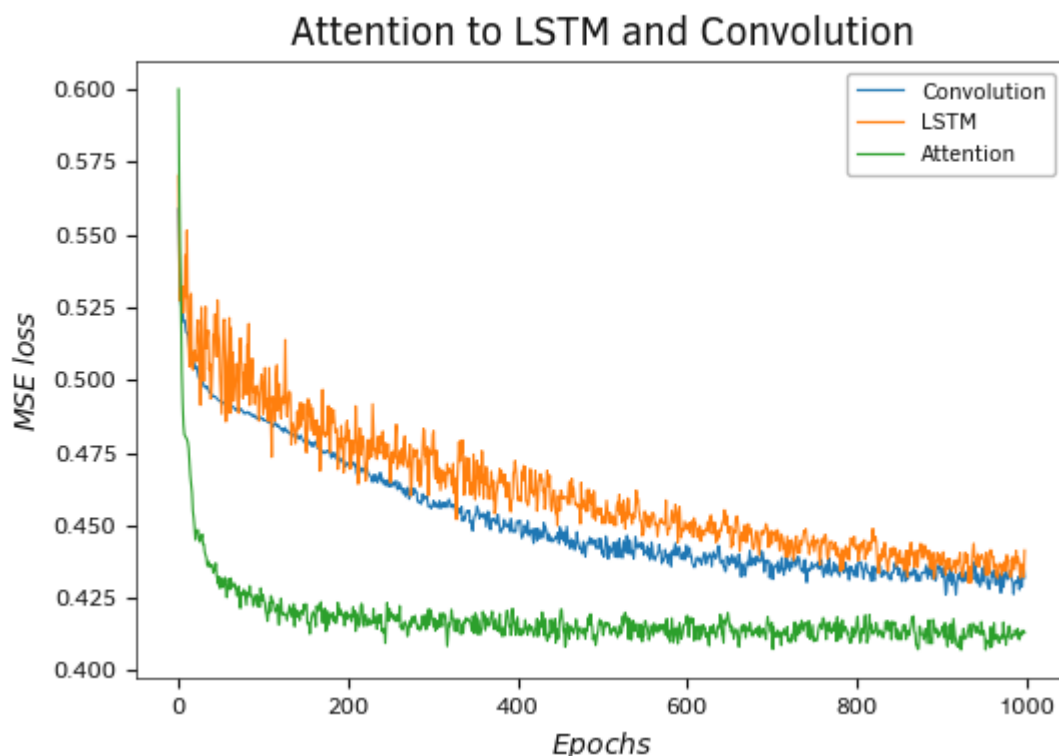
```

After specifying all the parameters, we add the object to the dynamic array of neural layer descriptions. And, of course, we check the results of the operations. The rest of the script code remained unchanged.

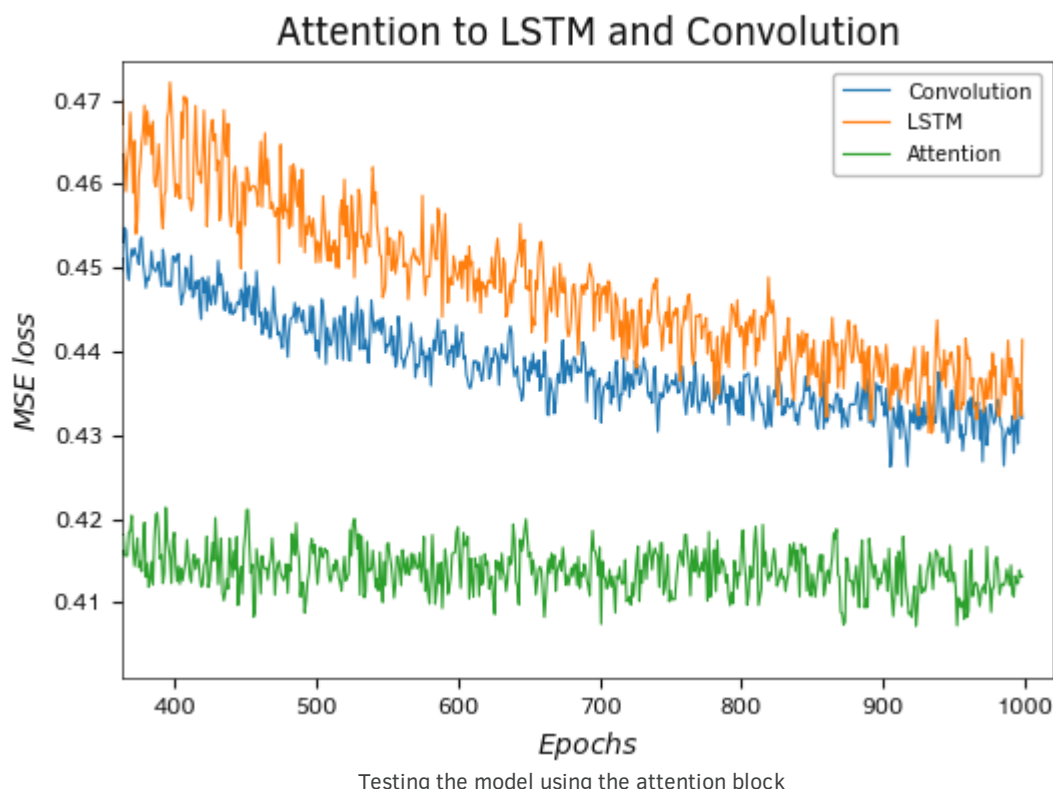
As you can see, when using our library, changing the model configuration is not a very complex procedure. Thus, you will always be able to configure and test various architectural solutions to solve a specific task without making changes to the logic of the main program.

Testing the new model using the attention block was carried out while preserving all the other conditions used to test the previous models. This approach allows the accurate evaluation of how changes in the model architecture affect the training result.

The very first testing showed the superiority of the model with the attention mechanism over the previously considered models. On the training graph, the model using a single attention layer shows faster convergence compared to models using a convolutional layer and a recurrent *LSTM* block.



Testing the model using the attention block



When we scale up the learning curve graph, we can see that the model using the attention method demonstrates lower error throughout the entire training process.

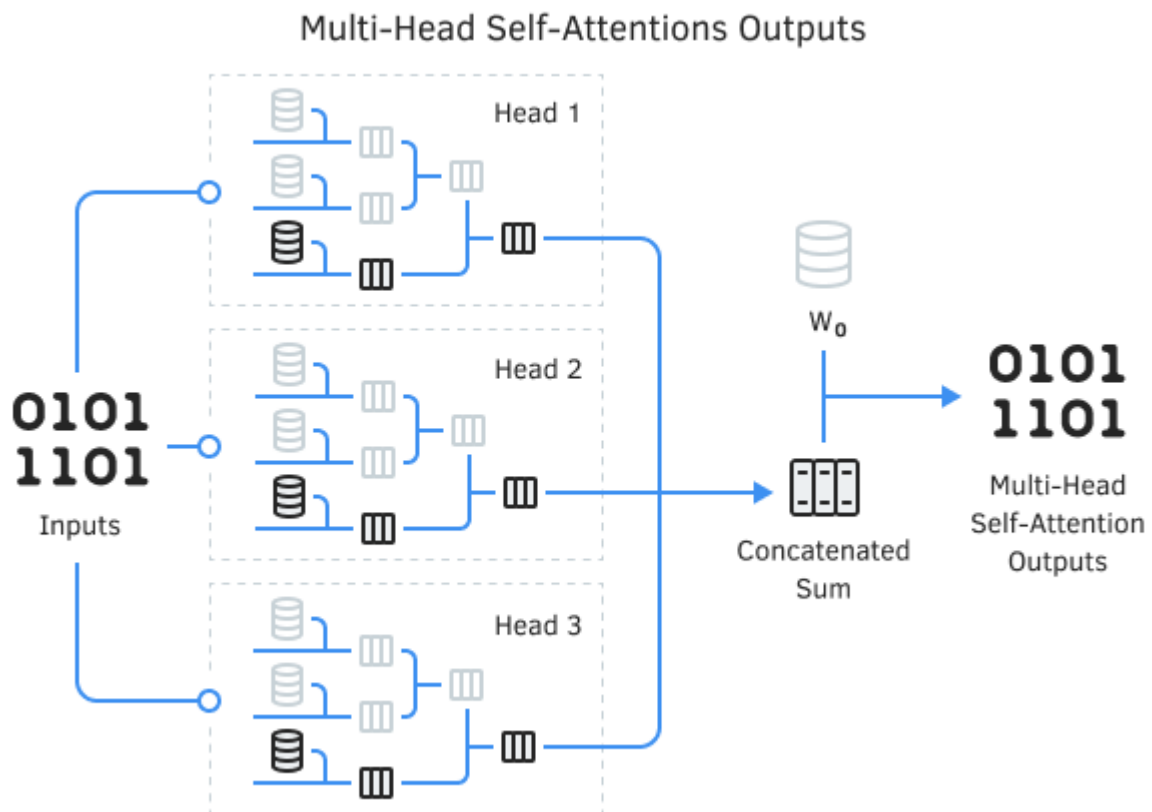
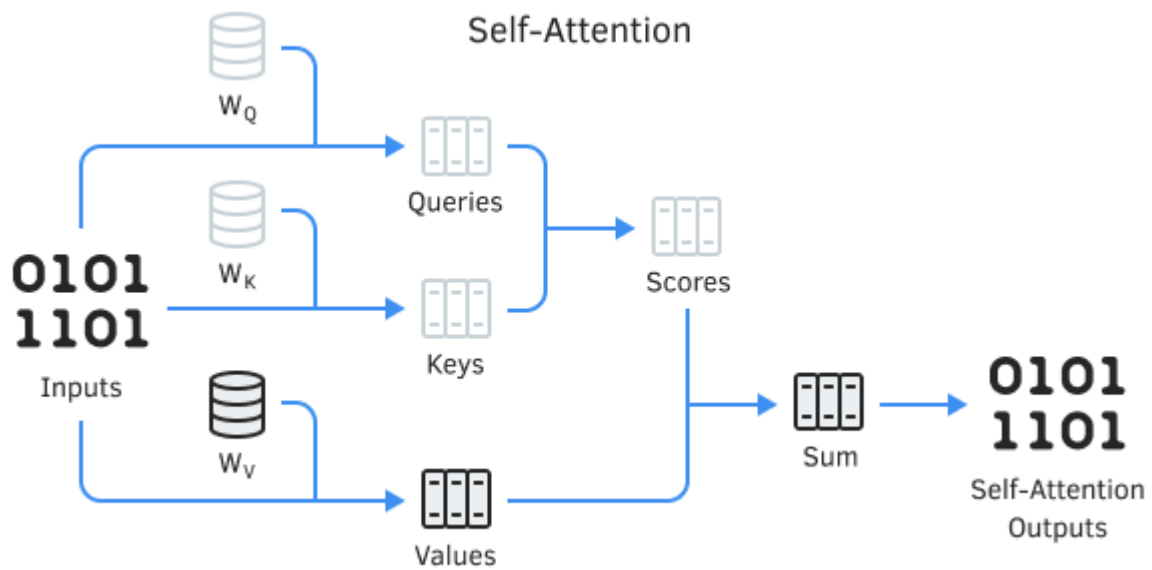
At the same time, it should be noted that using an attention block in this form is rarely encountered in practice. The architecture that has gained the most widespread use is the multi-head attention, which we will explore in the next section.

5.2 Multi-Head attention

In the previous section, we got acquainted with the mechanism of *Self-Attention mechanism*, which was introduced in June 2017 in the article [Attention Is All You Need](#). The key feature of this mechanism is its ability to capture dependencies between individual elements in a sequence. We even implemented it and managed to test it on real data. The model demonstrated its effectiveness.

Recall that the *Self-Attention* algorithm uses three trainable matrices of weights (W_Q , W_K , and W_V). The matrix data is used to obtain 3 three entities: *Query*, *Key*, and *Value*. The first two determine the pairwise relationship between elements of the sequence, while the last one represents the context of the analyzed element.

It's not a secret that situations are not always straightforward. The same situation can often be interpreted from various perspectives. With different points of view, the conclusions can be completely opposite. In such situations, it's important to consider all possible options and only draw a conclusion after a comprehensive analysis. That's why in the same paper, the authors of the method proposed using *Multi-Head Attention* to address such problems. This is the launch of several parallel *Self-Attention* threads, with different weights. Here, each 'head' has its own opinion, and the decision is made by a balanced vote. A solution like this should better identify connections between different elements of the sequence.



Multi-Headed Attention architecture diagram

In the *Multi-Head Attention* architecture, several *Self-Attention* threads with different weights are used in parallel, which simulates a versatile analysis of the situation. The results of the threads are concatenated into a single tensor. The final result of the algorithm is determined by multiplying the tensor by W_O matrix, the parameters of which are selected in the process of training the neural network. This whole architecture replaces the *Self-Attention* block in the encoder and decoder of the transformer architecture.

It is the *Multi-Head Attention* architecture that is most often used to solve practical problems.

5.2.1 Description of the Multi-Head Self-Attention architecture

The *Self-Attention* technology discussed earlier identifies dependencies between sequence objects in a certain context and then ranks them using the *Softmax* function. However, when solving practical problems, it is not always possible to give such an assessment unambiguously. Typically, dependency coefficients between objects change greatly when the point of view or context of the element being analyzed, changes. The final decision on element dependencies is always a compromise. The use of *Multi-Head Self-Attention* is specifically designed to help discover dependencies between elements by comprehensively considering the input data. The additional input trainable matrix of weights will help the model learn to find this compromise.

Perhaps the simplest solution to such a problem would be to expand our *CNeuronAttention* attention by adding an array of *Self-Attention* blocks to it. This approach is possible, but it is irrational. It leads to an increase in the number of objects proportionally to the increase in the number of attention heads. Furthermore, the sequential execution of operations for each attention head does not allow the organization of simultaneous parallel computation of attention for all heads. Additionally, the subsequent operation related to the concatenation of results of attention heads will also require resource and time overhead.

There is a solution, which lies in the realm of matrix operations in mathematics. Having knowledge and understanding of matrix operations in mathematics greatly aids in comprehending the mathematics of neural networks and provides a clear picture of the potential for dividing operations into parallel threads.

Let's go through the *Self-Attention* algorithm and think about transforming operations to implement *Multi-Head Self-Attention*.

1. First, we calculate vectors *Query*, *Key*, and *Value*. These vectors are calculated by multiplying each element of the original sequence by the corresponding matrix W_Q , W_K , and W_V .

$$Q = IW_Q, K = IW_K, V = IW_V$$

To organize *Multi-Head Self-Attention*, we need to repeat this operation based on the number of attention heads. Let's start with a simple example using three attention heads.

$$\begin{aligned} Q_1 &= IW_{Q1}, K_1 = IW_{K1}, V_1 = IW_{V1} \\ Q_2 &= IW_{Q2}, K_2 = IW_{K2}, V_2 = IW_{V2} \\ Q_3 &= IW_{Q3}, K_3 = IW_{K3}, V_3 = IW_{V3} \end{aligned}$$

I think everything is clear here.

Now let's look at the dimensions of tensors. Remember that the architecture of the model provides for the same number of sequence elements at all stages. Each element of the sequence is described by a certain vector of values. Since the *Self-Attention* mechanism is applied in the same way to each element of the sequence, we can analyze operations only with the description vector of one element, as an example. Moreover, the size of this vector is the same for the tensors of the original data and values. However, it may differ from the dimension of the description vector of one element of the sequence in

the query and key tensors. Let's use n_I for the size of the source data vector and n_K for the size of the key vector. Then the tensors will have the following dimensions.

Tensor	I	Q	W_Q	K	W_K	V	W_V
Size	n_I	n_K	$n_I * n_K$	n_K	$n_I * n_K$	n_I	$n_I * n_K$

The specified tensor sizes are applicable to all attention heads. Let's try to combine the corresponding weight matrices into one large one.

$$\text{Concatenate}(W_{Q1}, W_{Q2}, W_{Q3}) = W_{QC}$$

$$\text{Concatenate}(W_{K1}, W_{K2}, W_{K3}) = W_{KC}$$

$$\text{Concatenate}(W_{V1}, W_{V2}, W_{V3}) = W_{VC}$$

Such weight matrices will have size $n_I * 3n_K$ for query matrices W_{QC} and W_{KC} . The matrix W_{VC} will have the size $n_I * 3n_K$, where 3 is the number of attention heads.

Let's substitute the concatenated matrices into the formulas for determining vectors.

$$Q_C = IW_{QC}, K_C = IW_{KC}, V_C = IW_{VC}$$

According to the rules of matrix multiplication, we obtain the following tensor sizes.

Tensor	I	Q_C	W_{QC}	K_C	W_{KC}	V_C	W_{VC}
Size	n_I	$3n_K$	$n_I * 3n_K$	$3n_K$	$n_I * 3n_K$	$3n_I$	$n_I * 3n_K$

Compare the tensor sizes in the two tables: they are very similar. The only difference is that they are multiplied by the number of attention heads. What practical value does it have for us? It's all very straightforward. Instead of creating multiple instances of objects for each attention head, we can create just one object for computing each entity. As when organizing a similar process in the *Self-Attention* mechanism, we can use our convolutional layers, but we will need to increase the window size of the results proportionally to the number of attention heads.

- Next, we define pairwise dependencies between the elements of the sequence. To do this, we will multiply the *Query* vector with the *Key* vectors of all elements of the sequence. This iteration is repeated for the *Query* vector of each element of the sequence. As a result of this iteration, we obtain a *Score* matrix of size $N*N$, where N is the size of the sequence.

$$S = QK^T$$

As a result of this operation, we expect to obtain one coefficient of dependency between a pair of sequence elements for each attention head. However, the operation of multiplying two concatenated vectors will return only one value. As in the case of single-headed *Self-Attention*.

We can change the dimensionality of vectors and convert them into two-dimensional matrices. This makes sense, as we can allocate the data for each attention head into a separate row vector. However, by adding them to the formula above, we will get a square matrix with a side length equal to the number of attention heads, whereas we expected to obtain a vector with a size equal to the number of heads.

There is still a way out. Let's remember the matrix multiplication rule.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

We will substitute here our two-dimensional matrices of multi-head attention. Don't forget that the second matrix is transposed before multiplication.

$$\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \\ Q_{31} & Q_{32} \end{bmatrix} * \begin{bmatrix} K_{11} & K_{21} & K_{31} \\ K_{12} & K_{22} & K_{32} \end{bmatrix} =$$

$$= \begin{bmatrix} Q_{11}K_{11} + Q_{12}K_{12} & Q_{11}K_{21} + Q_{12}K_{22} & Q_{11}K_{31} + Q_{12}K_{32} \\ Q_{21}K_{11} + Q_{22}K_{12} & Q_{21}K_{21} + Q_{22}K_{22} & Q_{21}K_{31} + Q_{22}K_{32} \\ Q_{31}K_{11} + Q_{32}K_{12} & Q_{31}K_{21} + Q_{32}K_{22} & Q_{31}K_{31} + Q_{32}K_{32} \end{bmatrix}$$

As you can see, the vector we expected to obtain forms the diagonal of the matrix of results. And all other operations are just a waste of resources for us. But we can split this procedure into operations. For example, we will not transpose the key matrix and use the Hadamard product of matrices (element-wise matrix multiplication).

$$\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \\ Q_{31} & Q_{32} \end{bmatrix} \circ \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \\ K_{31} & K_{32} \end{bmatrix} = \begin{bmatrix} Q_{11}K_{11} & Q_{12}K_{12} \\ Q_{21}K_{21} & Q_{22}K_{22} \\ Q_{31}K_{31} & Q_{32}K_{32} \end{bmatrix}$$

After this, to obtain the expected result, all we need to do is add the elements of the matrix row by row.

$$\begin{bmatrix} Q_{11}K_{11} & Q_{12}K_{12} \\ Q_{21}K_{21} & Q_{22}K_{22} \\ Q_{31}K_{31} & Q_{32}K_{32} \end{bmatrix} = \begin{bmatrix} Q_{11}K_{11} + Q_{12}K_{12} \\ Q_{21}K_{21} + Q_{22}K_{22} \\ Q_{31}K_{31} + Q_{32}K_{32} \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ S_3 \end{bmatrix}$$

In the end, we got the result in two operations instead of one. However, it's important to note two things:

- We use a transposed matrix in the *Self-Attention* formula, which is also an operation on a matrix, although it is not highlighted separately. And its implementation also requires resources. When splitting into operations, we abandoned this procedure.
 - The vector of coefficients is determined in two operations, regardless of the number of attention heads.
3. The next step is to divide the resulting values by the square root of the dimension of the Key vector and normalize it with the *Softmax* function in the context of each *Query*. In this way, we obtain the coefficients of pairwise interdependence between sequence elements.

At this point, we will not complicate or simplify anything. Matrix division by a constant is always performed element-wise regardless of the matrix size, but we will need to normalize the data on a per-attention-head basis.

4. We multiply each *Value* vector by the corresponding interdependence coefficient and obtain the adjusted value of the element. The goal of this iteration is to focus on relevant elements and reduce the influence of irrelevant values.

To solve this problem, we will use the techniques applied in paragraph 2. First, let's change the dimension of the vector of values and reduce it to a two-dimensional matrix. In it, the rows will correspond to each individual head of attention.

$$\begin{bmatrix} V_{11} & V_{12} & V_{21} & V_{22} & V_{31} & V_{32} \end{bmatrix} = \begin{bmatrix} V_{11} & V_{12} \\ V_{21} & V_{22} \\ V_{31} & V_{32} \end{bmatrix}$$

After this, we can use element-wise multiplication of the dependency coefficient vector by the matrix of values.

$$\begin{bmatrix} S_1 \\ S_2 \\ S_3 \end{bmatrix} \circ \begin{bmatrix} V_{11} & V_{12} \\ V_{21} & V_{22} \\ V_{31} & V_{32} \end{bmatrix} = \begin{bmatrix} S_1 V_{11} & S_1 V_{12} \\ S_2 V_{21} & S_2 V_{22} \\ S_3 V_{31} & S_3 V_{32} \end{bmatrix}$$

5. Then we summarize all the adjusted *Value* vectors for each element. The result of this operation will indeed be the vector of output values of the *Self-Attention* layer.

In the last point, there is nothing more to add. We will summarize the value of vector elements separately in the context of Query queries and attention heads. We can easily parallelize the execution of this task by creating a separate thread for finding each individual vector.

After completing all the points of the *Self-Attention* mechanism in the mode with multiple attention heads, we receive a vector of results for each attention head. Consequently, the overall size of the tensor of results will exceed the size of the original data tensor proportionally to the number of heads. To reduce the dimensionality, the *Multi-Head Self-Attention* algorithm provides for multiplying the concatenated tensor of results by an additional weight matrix W_o . As you can imagine, this procedure is very similar to the operation of a fully connected neural layer without an activation function. We performed similar operations in step 1 to determine *Query*, *Key*, and *Values* vectors. This means that we can use the same solution and use the previously created convolutional layers.

Here, we can also note another point. When describing the operation of the *Self-Attention* block, we paid attention to the moment when the size of the vectors describing one element of the sequence of *Value* tensors and the source data are equal. This requirement was based on the need for subsequent addition of tensors of *Self-Attention* results and initial data. In the case of multi-head attention, we always end up with a concatenated tensor of results that is larger than the tensor of the original data. To align them, multiplication of the result tensor by the matrix W_o is used. Therefore, in order to save resources, we can reduce the dimensionality of the description vector of a single sequence element in the *Value* tensor without risking errors in subsequent data processing.

The rest of the algorithm of the transformer encoder remains unchanged, and we can leverage the developments from the previous section.

Now that we have a complete understanding of the principles behind the algorithm, we can proceed to its implementation.

5.2.2 Building Multi-Head Self-Attention in MQL5

When implementing the *Multi-Head Self-Attention* block, we can note its strong similarity with the previously considered *Self-Attention* block. This is not surprising, because *Multi-Head Self-Attention* is a logical development of *Self-Attention* technology. Therefore, when creating a new class, it would be quite logical to inherit not from the neural layer base class *CNeuronBase* but from the attention block class *CNeuronAttention*.

With this inheritance option, we inherit from the parent class, in addition to the methods and objects of the base class, also objects of the *CNeuronAttention* class, including:

- *m_cQuery*s – convolutional layer for the formation of the query tensor *Query*
- *m_cKeys* – convolutional layer for the formation of the key tensor *Key*
- *m_cValues* – convolutional layer for the formation of the value tensor *Value*
- *m_cScores* – buffer of the matrix of dependency coefficients
- *m_cAttentionOut* – base layer of the source data for recording the results of the *Self-Attention* block operation
- *m_cFF1* and *m_cFF2* – convolutional layers of the *Feed Forward* block

As we defined in the section describing the architectural solution, all objects will be used for their intended purpose. We will only increase their size in proportion to the number of attention heads. Thus, to implement the *Multi-Head Self-Attention* algorithm, we just need to add the internal layer of the W_o matrix and a variable for recording the number of attention heads.

```
class CNeuronMHAttention      : public CNeuronAttention
{
protected:
    CNeuronConv              m_cW0;

    int                      m_iHeads;

public:
    CNeuronMHAttention(void);
    ~CNeuronMHAttention(void);

    //---
    virtual bool             Init(const CLayerDescription *desc) override;
    virtual bool             SetOpenCL(CMyOpenCL *opencl) override;
    virtual bool             FeedForward(CNeuronBase *prevLayer) override;
    virtual bool             CalcHiddenGradient(CNeuronBase *prevLayer) override;
    virtual bool             CalcDeltaWeights(CNeuronBase *prevLayer, bool read) override;
    virtual bool             UpdateWeights(int batch_size, TYPE learningRate,
                                           VECTOR &Beta, VECTOR &Lambda) override;

    //--- file operation methods
    virtual bool             Save(const int file_handle) override;
    virtual bool             Load(const int file_handle) override;
    //--- object identification method
    virtual int              Type(void) override const { return(defNeuronMHAttention); }
};
```

Regarding the class methods, we will override the standard set of methods:

- *Init* – class initialization method
- *SetOpenCL* – method for specifying the handle of the *OpenCL* context to be used
- *FeedForward* – forward pass method
- *CalcHiddenGradient* – method of distributing the gradient error through the hidden layer
- *CalcDeltaWeights* – method of distributing the error gradient to the level of the matrix of weights of the current neural layer
- *UpdateWeights* – method for updating the matrix of weights of the coefficients of the current neural layer
- *Save* – method of saving neural layer data to a file

- *Load* – method of loading neural layer data from a file
- *Type* – method for identifying the type of neural layer

Well, let's start with the class constructor. In it, we create instances of objects necessary for the full functioning of the class and initialize internal variables with default values. Above, we defined only one new object, the convolutional layer *m_cw0*. We will use static objects, just like in the parent class. So, in the class constructor, we just have to specify the initial value for the number of attention heads. The class destructor remains empty.

```
CNeuronMHAttention::CNeuronMHAttention(void) : m_iHeads(8)
{
}
```

In the next step, we will deal with the method of initializing the class. Despite the fact that most of the objects were inherited from the parent class, we cannot use its initialization method, since using them in the *Multi-Head Self-Attention* algorithm will require different tensor sizes. Therefore, we will have to rewrite the initialization method completely. At the same time, to construct the initialization method, we will use an algorithm similar to the corresponding method of the parent class.

Like the similar methods of all previously discussed classes, in the method parameters, we receive a pointer to the object describing the configuration of the neural layer being created. We immediately organize a block for checking the received data. First of all, we check the validity of the received pointer. Only after confirming the validity of its relevance do we check its contents:

- The type of the neural layer to be created in the configuration description must match the type of the class (the *type* parameter).
- The layer you create must have at least one element of the sequence to be analyzed (the *count* parameter).
- The size of the description vector of one source data element must be greater than zero (the *window* parameter).
- The size of the key vector of one element of the sequence must be greater than zero (the *window_out* parameter).
- There must be at least one attention head (the *step* parameter).

```
bool CNeuronMHAttention::Init(const CLayerDescription *desc)
{
    //--- check the initial data
    if(!desc || desc.type != Type() ||
        desc.count <= 0 || desc.window <= 0 || desc.window_out <= 0 ||
        desc.step <= 0)
        return false;
```

It probably looks strange to use the *step* parameter to specify the number of attention heads. But, as you may recall, within the implementation of attention mechanisms, the step size of the input data window is always equal to the size of the window itself. Therefore, this parameter is free. To avoid an unnecessary increase in the size of the neural layer description object, we decided to make the most efficient use of the existing class variables. However, if code readability is a higher priority for you, you can always define the necessary number of variables to describe the architecture of the neural layer being created and name them accordingly.

After successfully passing through the control block, we will save the key parameters of the description of the neural layer being created into local variables.

```
//--- saving the constants
m_iWindow = desc.window;
m_iUnits = desc.count;
m_iKeysSize = desc.window_out;
m_iHeads = desc.step;
```

Like in similar methods of all previously discussed classes, the next step is to call the method of the base neural layer, in which inherited objects will be initialized. We cannot call the method of the parent class because it would create objects of different sizes, and we would need to modify those objects. And we don't want to do the same job twice. Therefore, we "jump over the head" and directly access the method of the base class.

Please note that before calling the method of the base class, we need to make some adjustments to the description of the architecture of the neural layer being created. At the same time, we do not know what plans the user has for the description object of the layer obtained in the parameters. Remember what we talked about objects and pointers to them. In the parameters, we got a pointer to the object. When we make changes to the object, they will be reflected on the side of the main program by the user. If the user applies a single object to describe multiple neural layers, there is a high probability that they will encounter an error when creating subsequent neural layers. Also, layers can be created with incorrect architecture. Therefore, we will create a new object to describe the architecture of the neural layer and populate it with the necessary parameters.

In the parent class, we have worked out a technology with the substitution of pointers to the object, result buffers and error gradients. Therefore, it doesn't matter how these objects are created in the base class method; you can specify any values for the layer size and result window in the parameters. To avoid performing unnecessary operations, we will specify them at least greater than zero.

To eliminate the creation of unnecessary objects, set the size of the source data window to zero and disable the activation function.

We leave the type of neural layer that we received in the description from the user.

Next, we call the method of the base neural layer, passing it the correct description.

```
//--- call the initialization method of the parent class
CLayerDescription* temp = new CLayerDescription();
if(!temp)
    return false;
temp.type = desc.type;
temp.optimization = desc.optimization;
temp.activation = AF_NONE;
temp.count = desc.count;
temp.window_out = 1;
temp.window = 0;
if(!CNeuronBase::Init(temp))
{
    delete temp;
    return false;
}
```

In the above description of the neural layer architecture, we will change the type of the created object and its size. This is enough to create an object of concatenated results of the work of attention heads.

```
//--- initialize AttentionOut
```

```

temp.type = defNeuronBase;
temp.count = (int)(m_iUnits * m_iKeysSize * m_iHeads);
if(!m_cAttentionOut.Init(temp))
{
    delete temp;
    return false;
}
if(!m_cAttentionOut.GetOutputs().m_mMatrix.Reshape(m_iUnits, m_iKeysSize * m_iHead
!m_cAttentionOut.GetGradients().m_mMatrix.Reshape(m_iUnits, m_iKeysSize * m_iHe
return false;

```

After initializing the object, we will slightly change the format of the result buffers and error gradients.

Next, we have to create internal convolutional neural layers. First, we will create internal neural layers to form the *Query*, *Key*, and *Value* tensors. All of them receive a sequence of initial data as input. Therefore, in the *window* and *step* parameters, we will specify the size of the vector describing one element of the source data sequence.

The number of filters of the used convolutional layer, specified in the *window_out* parameter, should correspond to the size of the key vector of one element of the sequence. However, when discussing the architectural solution of this class, we determined the use of concatenated tensors. Therefore, we will increase the number of filters in proportion to the number of attention heads created.

The number of elements in the sequence at all stages remains constant. Therefore, we can write to the *count* parameter the number of elements of the original sequence received from an external program.

The *Multi-Head Self-Attention* architecture does not provide an activation function for the neural layers that are created. Therefore, in the *activation* parameter, we leave the constant *AF_NONE*.

The optimization method for the parameters of all neural layers is the same, and we leave this parameter unchanged.

```

//--- create a description for the inner neural layers
if(!temp)
    return false;
temp.type = defNeuronConv;
temp.window = m_iWindow;
temp.window_out = (int)(m_iKeysSize * m_iHeads);
temp.step = m_iWindow;
temp.count = m_iUnits;

```

First, we initialize the inner layer to create the query tensor *Query*. We check the result of the operation in order to exclude possible critical errors in the further execution of the method code.

```

//--- initializing Querys
if(!m_cQuerys.Init(temp))
{
    delete temp;
    return false;
}
m_cQuerys.SetTransposedOutput(true);

```

After successful initialization of the convolutional neural layer, we set the flag to transpose the result tensor. I'd like to remind you that we introduced this flag to enable the retrieval of a result tensor in which each row contains elements not from a single filter but from all filters for one sequence element.

Similarly, we initialize convolutional neural layer objects to create *Key* and *Value* tensors.

```
//--- initialize Keys
if(!m_cKeys.Init(temp))
{
    delete temp;
    return false;
}
m_cKeys.SetTransposedOutput(true);
```

Please note that during the initialization of the convolutional neural layer object to form the *Value* tensor, we do not align the number of used filters with the size of the input data window, as was done in the single-attention head class *CNeuronAttention*. The use of the W_o matrix allows us to avoid this rule. Reducing the dimensionality of the vector can indeed help save resources and reduce the execution time of operations. In turn, after recreating the complete algorithm of the *Multi-Head Self-Attention* method, you will be able to assess the advantages and disadvantages of such an implementation through practical examples.

```
//--- initialize Values
if(!m_cValues.Init(temp))
{
    delete temp;
    return false;
}
m_cValues.SetTransposedOutput(true);
```

After initializing the first group of internal convolutional layers, following the algorithm of the *Multi-Head Self-Attention* mechanism, we initialize the buffer for the dependency coefficient matrix $m\_cScores$. Fill it with zero values, specifying the required buffer size. Again, let's draw a parallel with the *CNeuronAttention* class. If previously we created a square matrix with a side length equal to the number of elements in the sequence, now we need as many of these matrices as there are attention heads. At the same time, we have agreed to use a concatenated matrix. Therefore, we will increase the buffer size in proportion to the number of attention heads used. Unfortunately, *MQL5* does not support three-dimensional matrices. Within the two-dimensional matrix, we will use rows to distribute the buffer across attention heads.

```
//--- initialize Scores
if(!m_cScores.BufferInit(m_iHeads, m_iUnits * m_iUnits))
{
    delete temp;
    return false;
}
```

Now it's time to initialize the additional convolutional layer that performs the functionality of matrix W_o in the *Multi-Head Self-Attention* algorithm. Let's adjust the description of the architecture of the neural layer being created.

The type of neural layer to be created has already been specified, so we don't need to specify it again.

We determine the size of the input data window as the product of the size of the description vector of one sequence element in the Values tensor and the number of attention heads. In this implementation, we changed the size of the specified vector to the same one in the *Key* tensor. So, the size of the input data window is determined as the product of the size of the key vector of one sequence element and the number of attention heads ($m\_iKeysSize * m\_iHeads$).

We will equate the size of the step of the source data window to the size of the window itself.

According to the *Multi-Head Self-Attention* algorithm, matrix W_o is used to align the sizes of the tensor of results from the multi-head attention block with the tensor of input data. Therefore, we will specify the number of filters in this convolutional layer equal to the size of the description vector of one element of the sequence of initial data fed to the input of the *Multi-Head Self-Attention* block.

The *Multi-Head Self-Attention* algorithm does not provide an activation function for this matrix. Therefore, in the appropriate field, we leave the *AF_NONE* constant.

The optimization method for the weight matrices of all layers in the neural network, including the internal layers of individual blocks, is the same. Therefore, we leave the parameters indicating the optimization method used unchanged.

```
//--- initialize W0
temp.window = (int)(m_iKeysSize * m_iHeads);
temp.step = temp.window;
temp.window_out = m_iWindow;
if(!m_cW0.Init(temp))
{
    delete temp;
    return false;
}
m_cW0.SetTransposedOutput(true);
```

After specifying all the necessary parameters for describing the created neural layer, we call the initialization method of our convolutional neural layer *m_cW0.Init* and check the results of the operations.

At the end of the initialization block of the convolutional layer *m_cW0* we set the flag for transposing the result tensor.

This concludes the work on initializing the objects of the *Multi-Head Self-Attention* block. Next, let's move on to work on the *Feed Forward* block. The functionality and architecture of this block are completely transferred from the *CNeuronAttention* class. However, since we had to completely redefine the initialization method of the class, we will repeat the actions for initializing the internal layers *m_cFF1* and *m_cFF2*.

The algorithm for initializing the neural layer remains the same. We will prepare a description of the neural layer to be created and call the method of its initialization. To describe the convolutional neural layer *m_cFF1*, we will use the description object of the convolutional neural layer which has already been used more than once in this method. Therefore, we will only specify the parameters that are being changed, as the rest are already contained in the neural layer description object.

- The size of the source data window (*window*) is equal to the size of the description vector of one element of the source data tensor sequence fed to the input of our *Multi-Head Self-Attention* block. We receive this parameter from an external program and save it in the *m_iWindow* variable. Consequently, we can pass the value of the specified variable as a parameter.

- We will set the step size of the input data window (*step*) equal to the size of the input data window itself.
- Number of filters used (*window_out*): according to the transformer architecture proposed by the authors, the output size of the first layer of the *Feed Forward* block is four times larger than the size of the original data. Let's use this coefficient. However, during the implementation of your practical tasks, you can always modify this coefficient or even add it to the configuration description of the created neural layer and conduct practical tests to determine the most suitable coefficient for your specific tasks.
- The activation function (*activation*): for this layer, the authors suggest using *ReLU* as an activation function. We replaced it with the close *Swish* function. The graph of this function is very close to the graph of the function proposed by the authors. At the same time, it does not contain kinks and is differentiated throughout the values.
- The optimization parameters of the balance matrix remain unchanged.

```
//--- initialize FF1
temp.window = m_iWindow;
temp.step = temp.window;
temp.window_out = temp.window * 4;
temp.activation = AF_SWISH;
temp.activation_params[0] = 1;
temp.activation_params[1] = 0;
if(!m_cFF1.Init(temp))
{
    delete temp;
    return false;
}
m_cFF1.SetTransposedOutput(true);
```

After we have specified all the parameters in the configuration description of the created convolutional neural layer, we will call its initialization method and check the result of the operations.

Only upon successful initialization of the convolutional neural layer object, we will set the flag for transposing the result tensor.

Now we can proceed to initialize the last object used in the class – the second convolutional layer of the *Feed Forward* block *m_cFF2*. As a result of this neural layer operation, we again return to the dimension of the tensor of the original data. Therefore, in the description object of the structure of the created neural layer, we will need to swap the values of the input data window and the number of used filters. Typically, such an operation requires a local variable to temporarily store one of the values. But in our case, the parameters of the source data window size and its pitch are equal. Hence, we will first write the number of filters of the previous layer to the size parameter of the source data window. Next, in the parameter of the number of filters, specify the value of the window step of the previous convolutional layer. And finally, let's equate the size of the step of the source data window to its size.

The architecture of the transformer does not provide an activation function for this layer. But we will provide an opportunity for the user to experiment. To do this, let's transfer the activation function and its parameters from the architecture description provided by the user to the parameters of this method.

```
//--- initialize FF2
temp.window = temp.window_out;
temp.window_out = temp.step;
```

```

temp.step = temp.window;
temp.activation = desc.activation;
temp.activation_params = desc.activation_params;
if(!m_cFF2.Init(temp))
{
    delete temp;
    return false;
}
m_cFF2.SetTransposedOutput(true);
delete temp;

```

Once all the necessary parameters for describing the structure of the created neural layer are specified, we call its initialization method and set the flag for transposing the result tensor. At the same time, do not forget to check the results of the operations.

Now that all the necessary objects are initialized, we can safely delete the local neural layer description object without any risk of error.

Next, we will apply the technique refined in the *CNeuronAttention* class and substitute pointers to result and error gradient buffers of our multi-head attention class with similar buffers from the internal convolutional neural layer, *m_cFF2*. This will allow us to eliminate unnecessary costs for copying data between buffers. Also, we do not need additional memory to store duplicate data. To do this, we first check the pointers and, if necessary, delete previously created objects that are not needed. Then, we pass pointers to the objects of the convolutional layer *m_cFF2* into the variables.

```

//--- to avoid copying buffers, replace them
if(!SetOutputs(m_cFF2.GetOutputs()))
    return false;
if(m_cGradients)
    delete m_cGradients;
m_cGradients = m_cFF2.GetGradients();
//---
SetOpenCL(m_cOpenCL);
//---
return true;
}

```

In conclusion, to all objects in the method, we will pass a pointer to the used OpenCL context. After that, we exit the method with a positive result.

This concludes our work on the class initialization method. However, we have an open question. At the end of the initialization method, we called the method for passing the OpenCL context pointer. We haven't overridden it yet, and a similar method of the parent class will be called as such. It is functional enough but does not apply to objects declared in the body of this class. Among them, there is only one object: the convolutional layer of *m_cWO*. Therefore, the method will be relatively short.

Like the similar methods of all the previously discussed classes, the *CNeuronMHAttention::SetOpenCL* method in the parameters receives a pointer to the object of working with the OpenCL context. We will have to distribute it to all internal objects. First, it would be necessary to check the validity of the received pointer. Instead, we'll call a similar method of the parent class, which already has all the controls and pointer passing to inherited objects. Thus, after the completion of the parent class method, we just have to pass the pointer to the new objects that were declared in the body of this class. However, in this case, we will pass not the pointer received in the parameters but the pointer

from the local variable of the class inherited from the parent object. The reason is that the method of the parent class checked the received pointer and saved it to a local variable. It also passed it to all the objects that we inherited from the parent class. Therefore, in order for all objects to work in the same context, we pass an already validated pointer to the internal objects.

```
bool CNeuronMHAttention::SetOpenCL(CMyOpenCL *opencl)
{
    //--- call a method of a parent class
    CNeuronAttention::SetOpenCL(opencl);
    //--- call a similar method for the internal layer
    m_cW0.SetOpenCL(m_cOpenCL);
    //---
    return(!m_cOpenCL);
}
```

After passing the pointer to all internal objects, in this case, it's a single convolutional layer, we exit the method and return a result indicating the validity of the used context pointer.

With that, we conclude the process of creating and initializing our multi-head attention class object and move on to the next stage, which is setting up the feed-forward pass.

5.2.2.1 Multi-Head Self-Attention feed-forward method

We have already organized the process of creating and initializing the *CNeuronMHAttention* Multi-Head attention class. And now, when we already have all the internal objects of our class, we can move on to the organization of the forward pass.

The virtual method *FeedForward* is responsible for the implementation of the feed-forward pass in all classes of our library. Adhering to the general system of organization of classes and their methods, as well as the principles of inheritance, in this class we will retain the previously defined structure of methods and override the *FeedForward* method. Like the similar method of the parent class, in the parameters, the feed-forward method receives a pointer to the object of the previous neural layer. According to the framework that has been tested more than once, at the beginning of the method we organize a block of controls. In it, we check the relevance of pointers to all dynamic objects used in the method. In this case, we check the pointer to the neural layer received in the parameters, its result buffer, and the buffer of results from the internal layer of the concatenated output of the attention block.

```
bool CNeuronMHAttention::FeedForward(CNeuronBase *prevLayer)
{
    --- check the relevance of all objects
    if(!prevLayer || !prevLayer.GetOutputs() ||
        !m_cAttentionOut.GetOutputs())
        return false;
```

After successfully passing the control block, we generate concatenated tensors of queries, keys, and values: *Query*, *Key*, and *Value*. To do this, we call the methods of forward pass of the internal convolutional layers *m_cQuerys*, *m_cKeys*, and *m_cValues*. The correspondence of tensors in the *Multi-Head Self-Attention* architecture and the invoke objects is not accidental: it makes the code more readable and allows you to track the algorithm being built.

```
    if(!m_cQuerys.FeedForward(prevLayer))
        return false;
    if(!m_cKeys.FeedForward(prevLayer))
        return false;
    if(!m_cValues.FeedForward(prevLayer))
        return false;
```

Be sure to control the process of performing operations.

Next, according to the *Multi-Head Self-Attention* algorithm, we have to determine the coefficients of dependence between the elements of the sequence and display the concatenated result of the work of all attention heads. This functionality is the link between the internal neural layers. It does not cover the use of other objects and will be built entirely within this method.

As you remember, when building all processes in the methods of our library classes, we create two branches of the algorithm: standard MQL5 tools and multi-threaded calculations on the GPU using OpenCL. As always, in this section, we will consider the implementation of the algorithm using standard MQL5 tools. And we will return to the implementation of multi-threaded computing using OpenCL later.

Now we need to determine how to organize the work. We have three dimensions:

- Attention heads
- Sequence elements
- Vector with the description of one element of the sequence

Matrix operations allow us to operate only with two-dimensional matrices. One of the dimensions used will be a vector describing one element of the sequence. It's not hard to guess that in most cases, the size of the sequence will be tens of times larger than the number of attention heads. Therefore, we will create a loop for iterating through the attention heads, and within the loop, we will analyze the sequences of each attention head.

Before organizing the loop, we need to do a little preparatory work. Let's divide the concatenated results of the previous stage of the implemented algorithm into several attention heads matrices. For this, we will use dynamic arrays of matrices, which will give us a semblance of three-dimensional matrices. The index of an element in the array will indicate the attention head index. Each element in the array will be represented as a tabular matrix, where rows represent individual elements of the sequence. For the convenience of working with arrays, let's give them names that correspond to their content.

```
--- branching of the algorithm by the computing device
MATRIX out;
if(!m_cOpenCL)
{
    if(!out.Init(m_iHeads, m_iUnits * m_iKeysSize))
        return false;
    MATRIX querys[], keys[], values[];
    if(!m_cQuerys.GetOutputs().m_mMatrix.Vsplit(m_iHeads, querys))
        return false;
    if(!m_cKeys.GetOutputs().m_mMatrix.Vsplit(m_iHeads, keys))
        return false;
    if(!m_cValues.GetOutputs().m_mMatrix.Vsplit(m_iHeads, values))
        return false;
}
```

After completing the preparatory work, we can proceed directly to the operations of calculating dependency coefficients. When solving such a problem, we used matrix operations in the forward pass method of the parent class *CNeuronAttention*. Now we will use the same algorithm, but we need to repeat it in a loop with the number of iterations equal to the number of attention heads.

According to the *Multi-Head Self-Attention* algorithm, the dependence coefficients are divided by the square root of the dimension of the *Key* vector, and the obtained values are then normalized with the *Softmax* function in the context of elements of *Query* queries.

$$SoftMax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

Following the algorithm, we multiply the *querys* and transposed *keys* matrices, divide them by the square root of their dimension, and immediately calculate the exponential value. In the resulting matrix, we take line-by-line sums of values and organize a nested loop for data normalization.

```
for(int head = 0; head < m_iHeads; head++)
{
    //--- define Scores
    MATRIX sc = exp(querys[head].MatMul(keys[head].Transpose()) /
                                                            sqrt(m_iKeysSize));

    VECTOR sum = sc.Sum(1);
    for(uint r = 0; r < sc.Rows(); r++)
```

```

        if(!sc.Row(sc.Row(r) / sum[r], r))
            return false;

```

As you can see, the algorithm completely repeats similar operations of the parent class.

Now that we already have a calculated matrix of coefficients of dependencies between elements, we can move on using the *Multi-Head Self-Attention* algorithm and determine the values of the concatenated tensor of the results in terms of the analyzed attention head. To do this, we just need to calculate the products of two matrices containing the coefficients of dependence and the values of *Values*.

```

//--- output of the attention block
MATRIX temp = sc.MatMul(values[head]).Transpose();

```

Special attention should be paid to gathering results into a single concatenated tensor. The entire logic of constructing the algorithm assumes that the tensor of the concatenated result will be a tabular matrix. Each row of the matrix will contain a vector of the concatenated result of a single element of the sequence. I solved this problem as follows.

As a result of the multiplication operation, we obtained a tabular matrix where the number of rows equals the number of elements in the sequence, and the number of columns equals the size of the vector describing one element of the sequence. We transpose the matrix, reshape it into a row matrix, and add this resulting row to the concatenated matrix. At this stage, in the concatenated matrix, each attention head will have its own row.

We do the same with the matrix of dependency coefficients.

```

        if(!temp.Reshape(1, m_iUnits * m_iKeysSize))
            return false;
        if(!sc.Reshape(1, m_iUnits * m_iUnits))
            return false;
        if(!m_cScores.m_mMatrix.Row(sc.Row(0), head))
            return false;
        if(!out.Row(temp.Row(0), head))
            return false;
    }

```

Once the iterations of the loop are completed and the results of all the attention heads are obtained, we will reformat the concatenated matrix. We will make the number of columns equal to the number of elements of the sequence and transpose the matrix. As a result, we will have a number of rows equal to the number of elements in the analyzed sequence. This is the format we need to pass to the next convolutional layer of our multi-head attention block. We will save the matrix to the results buffer of the inner layer *m_cAttentionOut*.

```

        if(!out.Reshape(m_iHeads * m_iKeysSize, m_iUnits))
            return false;
        m_cAttentionOut.GetOutputs().m_mMatrix = out.Transpose();
    }
    else // OpenCL block
    {
        return false;
    }

```

This concludes the section on splitting the algorithm depending on the device for executing operations. Let's go back to using the methods of our internal neural layers. For a block of multi-threaded

operations using OpenCL, we will set a temporary stub in the form of a return of a false value for the execution of method operations. We will return to it in the following sections.

We continue to move according to the *Multi-Head Self-Attention* algorithm. At the next stage, we will need to reduce the dimensionality of the concatenated tensor of results from all attention heads to the size of the original data tensor. For these purposes, the algorithm provides for the use of a trained matrix W_O . This matrix has a dual purpose. First, it serves to change the dimension of the tensor. Second, it performs a weighted summation of all attention heads into a unified entity, thus determining the influence of each attention head on the final result.

To accomplish this task, we will use the object of the convolutional layer. We have already created a convolutional neural layer *m_cWO*, and now we have to call its forward pass method. In the parameters, we pass to the method a pointer to the object of the *m_cAttentionOut* neural layer. Do not forget to check the result of the operation.

```
if(!m_cWO.FeedForward(GetPointer(m_cAttentionOut)))
    return false;
```

After the successful completion of the method operations, the result buffer of our neural layer will be the result of the *Multi-Head Self-Attention* block. According to the Transformer algorithm, we will need to add the obtained result to the original data into a single tensor and normalize the result using the following formulas:

$$\bar{a} = \frac{1}{h} \sum_{i=1}^h a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h} \sum_{i=1}^h (a - \bar{a})^2}}$$

When working on the parent class *CNeuronAttention* we created separate methods for these operations. Now let's make use of the results of the work done earlier.

```
//--- add to the initial data and normalize
if(!m_cWO.GetOutputs().SumArray(prevLayer.GetOutputs()))
    return false;
if(!NormlizeBuffer(m_cWO.GetOutputs(), GetPointer(m_cStd), 0))
    return false;
```

And, of course, don't forget to monitor the process of executing operations at every step.

Monitoring the process of executing operations is very important and should become a good habit, especially when dealing with such a large number of operations.

This concludes the *Multi-Head Self-Attention* block in the transformer encoder algorithm. Next comes its second block – *Feed Forward*. Within this block, we need to propagate the signal through two neural layers. We will do so by sequentially calling the feed-forward methods of each neural layer.

```
//--- FeedForward
if(!m_cFF1.FeedForward(GetPointer(m_cWO)))
    return false;
if(!m_cFF2.FeedForward(GetPointer(m_cFF1)))
```

```
return false;
```

At the end of the forward pass algorithm, we will need to repeat the data normalization procedure. This time we add the result buffers of the *Multi-Head Self-Attention* and *Feed Forward blocks*.

```
//--- add to the output of attention and normalize
if(!m_cOutputs.SumArray(m_cW0.GetOutputs()))
    return false;
if(!NormalizeBuffer(m_cOutputs, GetPointer(m_cStd), 1))
    return false;
//---
return true;
}
```

The normalization procedure completes the feed-forward method. After the specified process completes successfully, we exit the method with a result of *true*. Let's move on to the implementation of the backpropagation method.

5.2.2.2 Multi-Head Self-Attention backpropagation methods

We are confidently moving forward in our learning path. Let's proceed with the implementation of our *Multi-Head Self-Attention* class. In the previous sections, we have already implemented initialization methods and feed-forward methods. However, the neural layer training algorithm is based on the error gradient backpropagation algorithm. We now proceed to implement backpropagation methods.

We have already mentioned that the Multi-Head Self-Attention algorithm is a logical extension of *Self-Attention*. That's why we created our class based on the *CNeuronAttention* class. And yes, the processes are all very similar. However, there are still some minor differences in the implementation of multi-head attention. To implement these differences, we created a new class *CNeuronMHAttention*.

As we progress in creating the methods of the class, let's take a look at the implementation of these differences in the methods of the backpropagation algorithm.

In the parent class, we have overridden three virtual methods to implement the backpropagation algorithm:

- *CNeuronAttention::CalcHiddenGradient* – method for calculating the error gradient through the hidden layer
- *CNeuronAttention::CalcDeltaWeights* – method for calculating the error gradient to the level of the weights matrix
- *CNeuronAttention::UpdateWeights* – method for updating the weights

So, we will also need to override the corresponding methods to organize the multi-head attention backpropagation pass. Let's start with the method of distributing the error gradient through the hidden layer of the *CalcHiddenGradient* neural network.

As in the parent class method, in the parameters of the method, we receive a pointer to the object of the previous neural layer. It is in its error gradient buffer that we are going to record the result of the work being done.

At the beginning of the *CNeuronMHAttention::CalcHiddenGradient* method body, there is the customary and essential attribute of any method: a check of pointers to the objects used in the method. Here, as in the similar method of the parent class, we will perform control checks only for pointers to objects that will be directly accessed from this method without using the methods of internal neural layers. The reason is that all inner neural layer methods have a similar block of controls. By calling them, we again validate the passed pointers to objects. This is an additional cost in resources and time. We can't disable the checks in the methods of the nested neural layers, so we will eliminate explicit duplication of controls in the current method.

We should immediately point out that we only exclude *explicit duplication*, but not *possible*. It's a fine line, but behind it lie great risks.

Explicit is the duplication that will happen anyway. If we see such duplication, we try to keep only one control point before the first use of the object whenever possible.

Note, that there must be at least one control point before the object is accessed for the first time.

I call duplication *possible* when it can occur under certain circumstances. In some cases, it may not happen. We do not eliminate such duplication because the risk of a critical error in the absence of control outweighs the potential benefits of improving program performance.

```
bool CNeuronMHAttention::CalcHiddenGradient(CNeuronBase *prevLayer)
```

```

{
//--- check the relevance of all objects
if(!m_cOutputs || !m_cGradients ||
    m_cOutputs.Total() != m_cGradients.Total())
    return false;

```

After successfully passing the control block, we proceed directly to the error gradient distribution procedure. As you may recall, in the feed-forward pass, the data is normalized at the output of the neural layer. Also, we need to adjust the error gradient by the derivative of the normalization function. In the parent class, we derived this procedure in a separate method entitled *CNeuronAttention::NormalizeBufferGradient*. Now we just need to call it with the appropriate parameters.

```

//--- scale the gradient to normalization
if(!NormalizeBufferGradient(m_cOutputs, m_cGradients, GetPointer(m_cStd), 1))
    return false;

```

Next, we run the error gradient through the inner neural layers of the *Feed Forward* block. These are the two convolutional layers: *m_cFF2* and *m_cFF1*. To propagate the gradient through these neural layers, we sequentially call the analogous methods of the mentioned neural layers. Don't forget to check the results of the operations.

```

//--- propagate the error gradient through the Feed Forward block
if(!m_cFF2.CalcHiddenGradient(GetPointer(m_cFF1)))
    return false;
if(!m_cFF1.CalcHiddenGradient(GetPointer(m_cW0)))
    return false;

```

After passing the error gradient via the *Feed Forward* block, we recall that before normalizing the data at the output of the neural layer, we added up the tensors of the results of the *Multi-Head Self-Attention* and *Feed Forward* blocks. Hence, we must also propagate the error gradient along both directions. For this purpose, after obtaining the error gradient from the *Feed Forward* block in the buffer of the inner neural layer *m_cW0*, we add up the two tensors.

```

if(!m_cW0.GetGradients().SumArray(m_cGradients))
    return false;

```

Let's adjust it for the derivative of the data normalization process.

```

//--- adjust the gradient for normalization
if(!NormalizeBufferGradient(m_cW0.GetOutputs(), m_cW0.GetGradients(),
                           GetPointer(m_cStd), 0))
    return false;

```

We continue utilizing internal neural layer methods. We will call the convolution layer gradient distribution method *m_cW0* and check the result of the operations.

```

//--- distribution of the error gradient by attention heads
if(!m_cW0.CalcHiddenGradient(GetPointer(m_cAttentionOut)))
    return false;

```

Next, we need to propagate the error gradient from the concatenated result of the *Multi-Head Self-Attention* block to the internal neural layers *m_cQuerys*, *m_cKeys*, and *m_cValues*. As you may recall, in the feed-forward pass, the path to *m_cAttentionOut* from the specified inner neural layers was completely recreated inside the method. Similarly, we will have to recreate the progression of the reverse signal.

Since we are creating a new thread of operations, according to our concept, it is necessary to organize two parallel threads of operations: using standard MQL5 tools and in the paradigm of multi-threaded operations using OpenCL.

```
//--- branching of the algorithm by computing device
if(!m_cOpenCL)
{
    MATRIX gradients[];
    MATRIX querys[], querys_grad = MATRIX::Zeros(m_iHeads, m_iUnits * m_iKeysSize);
    MATRIX keys[], keys_grad = MATRIX::Zeros(m_iHeads, m_iUnits * m_iKeysSize);
    MATRIX values[], values_grad = MATRIX::Zeros(m_iHeads, m_iUnits * m_iKeysSize);
    MATRIX attention_grad = m_cAttentionOut.GetGradients().m_mMatrix;
```

As always, in this section, we will consider the implementation using MQL5. We will proceed to the organization of multi-threaded operations later.

So, first, we're going to do some preparatory work. As in the forward pass, in this block, we organize the work separately for individual attention heads. As all the data is stored in concatenated buffers, we will prepare local matrices and split the buffers into individual matrices according to the attention heads.

```
if(!m_cQuerys.GetOutputs().m_mMatrix.Vsplit(m_iHeads, querys) ||
    !m_cKeys.GetOutputs().m_mMatrix.Vsplit(m_iHeads, keys) ||
    !m_cValues.GetOutputs().m_mMatrix.Vsplit(m_iHeads, values) ||
    !attention_grad.Reshape(m_iUnits, m_iHeads * m_iKeysSize) ||
    !attention_grad.Vsplit(m_iHeads, gradients))
    return false;
```

Next, we will create a loop with the number of iterations equal to the number of attention heads used.

```
for(int head = 0; head < m_iHeads; head++)
{
```

During the feed-forward pass, the values of the concatenated buffer of results are assembled by multiplying the values of the *m_cValues* neural layer's tensor results with the corresponding elements of the dependency coefficient matrix, followed by vector addition. Now we need to organize the reverse process: propagating the error gradient along these two directions.

First, we transfer the error gradient to the inner neural layer *m_cValues*. Before that, let's do some preparatory work.

To propagate the gradient to the *m_cValues* neural layer, it is necessary to multiply the error gradient matrix by the dependency coefficient matrix. Hence, we first need to extract such a matrix for the attention head we analyze.

We then multiply the matrices and add the result to a local copy of the concatenated gradient matrix of the *m_cValues* layer.

```
//--- gradient propagation to Values
MATRIX score = MATRIX::Zeros(1, m_iUnits * m_iUnits);
if(!score.Row(m_cScores.m_mMatrix.Row(head), 0) ||
    !score.Reshape(m_iUnits, m_iUnits))
    return false;
MATRIX temp = (score.Transpose().MatMul(gradients[head])).Transpose();
if(!temp.Reshape(1, m_iUnits * m_iKeysSize) ||
```



```
!values_grad.Row(temp.Row(0), head))
return false;
```

After that, we will propagate the gradient along the second path of the algorithm, through the dependency coefficient matrix to the neural layers *m_cQueryys* and *m_cKeys*. In essence, we first need to determine the error gradient at the level of the dependency coefficient matrix and then propagate the error gradient from there to the specified internal neural layers.

Here we should recall that the dependency coefficient matrix is normalized by the *Softmax* function in the *Query* section. To properly adjust the error gradient for the derivative of the *Softmax* function, we need at least the full vector of error gradients for the values involved in a single normalization operation. We can write it into a local matrix.

The task is clear, and we can proceed to implementation. To propagate the error gradient to the dependency coefficient matrix, it is sufficient to multiply the obtained gradient by the matrix of the results from the last feed-forward pass of the *m_cValues* neural layer.

After obtaining the error gradient vector at the dependency coefficient matrix level, we should adjust it using the derivative of the *Softmax* function.

$$Softmax(x_i)' = \begin{cases} i = j, & y_i(1 - y_i) \\ i \neq j, & -y_i y_j \end{cases}$$

We will organize a loop in which we adjust the error gradient using the derivative of the Softmax normalization function.

```
//--- gradient distribution up to Score
gradients[head] = gradients[head].MatMul(values[head].Transpose());
//--- gradient correction by Softmax derivative
for(int r = 0; r < m_iUnits; r++)
{
    MATRIX ident = MATRIX::Identity(m_iUnits, m_iUnits);
    MATRIX ones = MATRIX::Ones(m_iUnits, 1);
    MATRIX result = MATRIX::Zeros(1, m_iUnits);
    if(!result.Row(score.Row(r), 0))
        return false;
    result = ones.MatMul(result);
    result = result.Transpose() * (ident - result);
    if(!gradients[head].Row(result.MatMul(gradients[head].Row(r)) /
                                sqrt(m_iKeysSize), r))
        return false;
}
```

In the next step, we distribute the error gradient to the result values of the *m_cQueryys* and *m_cKeys* neural layers. However, we will not immediately write the values into the data buffers of the specified neural layers. We will only accumulate the sums of the error gradients into the pre-prepared matrices *queryys_grad* and *keys_grad*.

Technically, we multiply the adjusted error gradient by the opposite matrix. Multiplying it by the *Keys* matrix, we get the error gradient for *Queryys*, and vice versa. We reformat the obtained matrices and add them to the corresponding local matrices.

```
//--- gradient propagation to Queryys and Keys
temp = (gradients[head].MatMul(keys[head])).Transpose();
```

```

        if(! temp.Reshape(1, m_iUnits * m_iKeysSize) ||
           !querys_grad.Row(temp.Row(0), head))
            return false;
        temp = (gradients[head].Transpose().MatMul(querys[head])).Transpose();
        if(! temp.Reshape(1, m_iUnits * m_iKeysSize) ||
           !keys_grad.Row(temp.Row(0), head))
            return false;
    }

```

After completing the iterations of the loop, we obtain concatenated matrices of error gradients for all internal layers. Finally, we need to format the matrices as required and copy the values into the respective data buffers.

```

        if(!querys_grad.Reshape(m_iHeads * m_iKeysSize, m_iUnits) ||
           !keys_grad.Reshape(m_iHeads * m_iKeysSize, m_iUnits) ||
           !values_grad.Reshape(m_iHeads * m_iKeysSize, m_iUnits))
            return false;
        m_cQuerys.GetGradients().m_mMatrix = querys_grad.Transpose();
        m_cKeys.GetGradients().m_mMatrix = keys_grad.Transpose();
        m_cValues.GetGradients().m_mMatrix = values_grad.Transpose();
    }
    else // OpenCL block
    {
        return false;
    }

```

As a result, we have propagated the error gradient to the level of internal neural layers. We have successfully addressed the previously set task and are concluding the section on algorithm partitioning based on the computational device. In the multi-threaded operations branch, we will temporarily set the method exit with a false result. We will complete this part later.

We haven't propagated the error gradient to the previous layer yet. We will further propagate the error gradient using internal neural layer methods.

We've already filled the error gradient buffers of all the inner layers. We only need to call the method for error gradient propagation through the layer to obtain the error gradient at the level of the original data. However, one question remains open: all three internal neural layers (*m_cQuerys*, *m_cKeys*, *m_cValues*) use the same tensor from the previous layer as their input data. This means that all three layers must pass the error gradient to the previous layer's buffer. In addition, the result of the *Multi-Head Self-Attention* block was added to the tensor of the original data before normalization. Hence, this is the fourth thread of the error gradient that we need to pass to the previous layer level.

However, our gradient propagation methods are constructed in a way that when the error gradient is saved in the buffer of the previous layer, it overwrites the previous values, erasing the prior information. This is done intentionally to avoid unnecessary buffer-clearing operations before starting each iteration of the backpropagation pass. To address this issue, after running the *CalcHiddenGradient* method for each internal neural layer, we will copy the error gradient data to a separate buffer, where we will accumulate it with the previously stored values. At this point we should recall that the error gradient at the output of the *Multi-Head Self-Attention* block is already contained in the error gradient buffer of the neural layer *m_cW0*. It might seem that this buffer would be suitable for accumulating the error gradient for the previous layer. But that's a misconception. If we were to accumulate the error gradient in the mentioned buffer right now, it would distort the data during the subsequent error gradient propagation to the weight matrix of that layer. At the same time, we can implement the error gradient

propagation to the matrix of the *m_cw0* layer right now. There's all the data you need to do that. We call the *CalcDeltaWeights* method of the specified neural layer and then use its buffer to accumulate the total error gradient.

```
//--- propagate the error gradient to the previous layer
if(!m_cw0.CalcDeltaWeights(GetPointer(m_cAttentionOut), false))
    return false;
CBufferType* attention_grad = m_cw0.GetGradients();
if(!m_cValues.CalcHiddenGradient(prevLayer))
    return false;
if(!attention_grad.SumArray(prevLayer.GetGradients()))
    return false;
if(!m_cQuerys.CalcHiddenGradient(prevLayer))
    return false;
if(!attention_grad.SumArray(prevLayer.GetGradients()))
    return false;
if(!m_cKeys.CalcHiddenGradient(prevLayer))
    return false;
if(!prevLayer.GetGradients().SumArray(attention_grad))
    return false;
//---
return true;
}
```

Attention should be paid to the last group of commands. During the previous operations, we copied data from the gradient buffer of the previous layer, but at the end of the method, we reversed the process by taking the cumulative error gradient from the internal neural layer's buffer and adding it to the values of the buffer of the previous layer. It is in the buffer of the previous layer where we need to obtain the result. From it, the methods of the previous layer will take the error gradient and distribute it further through the neural network.

This completes the task set for this method. We complete the method with a positive result.

Next, we will work on two more methods that will continue the execution of the error backpropagation algorithm in this class.

After propagating the error through all the neural layers of our network, we need to propagate the error gradient to the level of each weight. Our *CNeuronMHAttention* class does not contain a separate buffer for the weight matrix. All trained parameters are encapsulated in internal neural layers. Therefore, the only thing we need to do in the method for propagating the error gradient to the *CalcDeltaWeights* weight matrix is to consistently call the same method for all inner layers. At the same time, we should check the results of the operations.

Recall that in the previous method, we have already passed the error gradient to the weight matrix of the *m_cw0* inner layer. It is necessary to exclude it from this iteration.

```
bool CNeuronMHAttention::CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
{
//--- call the same method for all inner layers
if(!m_cFF2.CalcDeltaWeights(GetPointer(m_cFF1), false))
    return false;
if(!m_cFF1.CalcDeltaWeights(GetPointer(m_cw0), false))
    return false;
}
```

```

        if(!m_cQueryys.CalcDeltaWeights(prevLayer, false))
            return false;
        if(!m_cKeys.CalcDeltaWeights(prevLayer, false))
            return false;
        if(!m_cValues.CalcDeltaWeights(prevLayer, read))
            return false;
    //---
    return true;
}

```

After propagating the error gradients to the weight matrices, the only remaining step is to update the weights of our internal neural layers. This functionality is assigned to the *UpdateWeights* method. Despite the complexity of the class itself, the method for updating the weight matrices turns out to be very concise and straightforward. It was object inheritance that helped us with this.

We created our *CNeuronMHAttention* class as a descendant of the *CNeuronAttention* class. We added only one object of the inner *m_cW0* neural layer. During the operations of the *UpdateWeights* method of the convolutional neural layers used, all operations are performed only on elements within the object, without accessing data from other objects. That's why we can call a similar method from the parent class, where this process is already implemented for inherited objects. After successfully executing the method of the parent class, we only need to update the coefficient matrix of the *m_cW0* internal neural layer.

```

bool CNeuronMHAttention::UpdateWeights(int batch_size, TYPE learningRate,
                                       VECTOR &Beta, VECTOR &Lambda)
{
    //--- call the method of the parent class
    if(!CNeuronAttention::UpdateWeights(batch_size, learningRate, Beta, Lambda))
        return false;
    //--- call the same method for all inner layers
    if(!m_cW0.UpdateWeights(batch_size, learningRate, Beta, Lambda))
        return false;
    //---
    return true;
}

```

Of course, we verify the result of all operations and return a boolean value indicating their execution to the caller.

Thus, we are nearing the completion of the *Multi-Head Self-Attention* technology implementation class. We have already implemented the whole algorithm using standard MQL5 tools. You can even create a script and test how it works. However, we still need to supplement our class with file handling methods.

5.2.2.3 File operations

We already had good progress with our work on the implementation of the *Multi-Head Self-Attention* algorithm. In the previous sections, we implemented the feed-forward and backpropagation operations of our *CNeuronMHAttention* class using standard MQL5 tools. Now, in order to fully utilize it in our models, we need to complement it with file methods. The proper functioning of these methods is just as important for industrial use as the correct functioning of the feed-forward and backpropagation methods.

True, we can create a model and test its performance without saving the training results. However, to conduct a repeated test, we will have to retrain our model from scratch. In real-life operations, we wouldn't want to repeat the training process each time. On the contrary, quite often significant efforts are invested in developing and training a model on large datasets, which enables the creation of a truly functional model. At the same time, it is expected that during practical application, it will be sufficient to start the model, and it will be fully ready to operate on real data. Therefore, when approaching the development of file handling methods, we must design their functionality in such a way that we can fully restore the model's state with minimal effort. Well, we have done this work several times already, so let's use the established algorithm once again.

First, let's look at the structure of our multi-head attention class *CNeuronMHAttention*.

```
class CNeuronMHAttention      : public CNeuronAttention
{
protected:
    CNeuronConv              m_cw0;
    int                      m_iHeads;

public:
                                CNeuronMHAttention(void);
                                ~CNeuronMHAttention(void);

    //---
    virtual bool               Init(const CLayerDescription *desc) override;
    virtual bool               SetOpenCL(CMyOpenCL *opencl) override;
    virtual bool               FeedForward(CNeuronBase *prevLayer) override;
    virtual bool               CalcHiddenGradient(CNeuronBase *prevLayer) override;
    virtual bool               CalcDeltaWeights(CNeuronBase *prevLayer, bool read) override;
    virtual bool               UpdateWeights(int batch_size, TYPE learningRate,
                                            VECTOR &Beta, VECTOR &Lambda) override;

    //--- methods of working with files
    virtual bool               Save(const int file_handle) override;
    virtual bool               Load(const int file_handle) override;
    //--- object identification method
    virtual int                Type(void) override const { return(defNeuronMHAttention); }
};
```

Seemingly, there's nothing complicated here. In the class body, we declare only one convolution layer *m_cw0* and one variable *m_iHeads* indicating the number of attention heads used. Most of the objects are inherited from the parent class *CNeuronAttention*. We already created a similar method when working on the parent class, and now we can use it. I suggest looking again at the *CNeuronAttention::Save* parent class method and making sure it has a save of all the data we need. After that, we can start working on the method for saving the current class data. This time, everything here is indeed very simple.

In the parameters, the *CNeuronMHAttention::Save* method gets the handle of the file to which it will write the data. In the body of the method, we immediately pass the obtained handle to a similar method of the parent class, where all the control logic is already implemented. In addition to controls, the parent class method also implements the saving of inherited objects and their data. Therefore, by checking the result of the parent class method, we immediately get a consolidated result of passing through the control block and saving inherited objects. We only need to save the number of attention heads used and the *m_cw0* convolutional layer data.

```
bool CNeuronMHAttention::Save(const int file_handle)
{
    //--- call the method of the parent class
    if(!CNeuronAttention::Save(file_handle))
        return false;
    //--- save constants
    if(FileWriteInteger(file_handle, m_iHeads) <= 0)
        return false;
    //--- call the same method for all inner layers
    if(!m_cw0.Save(file_handle))
        return false;
    //---
    return true;
}
```

The *CNeuronMHAttention::Load* method loads data from a file in accordance with the sequence of their recording. Therefore, in the body of the method, we immediately pass the received file handle as a parameter to the corresponding method of the parent class and check the result.

```
bool CNeuronMHAttention::Load(const int file_handle)
{
    //--- call the method of the parent class
    if(!CNeuronAttention::Load(file_handle))
        return false;
}
```

After executing the operations of the parent class method, we read the number of attention heads used and the data of the *m_cw0* internal convolution layer from the file. Loading a constant is very simple: we just read the value from the file and save it to our *m_iHeads* variable. But before calling the load method, we must check the type of the object to be loaded. Only if the object types match, we call the data loading method and check the result.

```
m_iHeads = FileReadInteger(file_handle);
if(CheckPointer(m_cw0) == POINTER_INVALID)
{
    m_cw0 = new CNeuronConv();
    if(CheckPointer(m_cw0) == POINTER_INVALID)
        return false;
}
if(FileReadInteger(file_handle) != defNeuronConv ||
    !m_cw0.Load(file_handle))
    return false;
```

It is expected that after the successful execution of the parent class operations, we will have fully restored inherited objects. However, we inherited the objects but initialized them in the corresponding method of this class with parameters different from the parent class. In this class, we adjusted almost

all objects for the number of attention heads used. In the data loading method of the parent class, we not only load data from the file but also initialize unsaved objects. These are objects whose data are only used within a single iteration of feed-forward and backpropagation passes.

So, let's return to the parent class method and critically evaluate all the operations once again. Pay attention to the following lines of code.

```
bool CNeuronAttention::Load(const int file_handle)
{
    .....
    m_iUnits = FileReadInteger(file_handle);
    .....
    if(!m_cScores.BufferInit(m_iUnits, m_iUnits, 0))
        return false;
    .....
    //---
    return true;
}
```

They initialize the *m_cScores* dependency coefficient matrix buffer. As you can see, the initialization is done with zero values with the size sufficient for only one attention head. However, this does not satisfy the requirements of our *Multi-Head Self-Attention* algorithm. It would make sense to add a reinitialization of the buffer in our class loading method, giving it the necessary size.

```
//--- initialize Scores
if(!m_cScores.BufferInit(m_iHeads, m_iUnits * m_iUnits))
    return false;
//---
return true;
}
```

After completing all the operations, we exit the method with a positive result.

This completes the implementation of the *CNeuronMHAttention* class using standard MQL5 tools. We have implemented the *Multi-Head Self-Attention* algorithm. In the next section, we will add the ability to perform multi-threaded operations using OpenCL.

5.2.3 Organizing parallel computing for Multi-Head Self-Attention

We continue our steady progress on the path of knowledge and building a library for creating machine learning models within the MQL5 environment. In this section, we are planning to complete work on *CNeuronMHAttention* which is another class of neural layers. This class implements the *Multi-Head Self-Attention* algorithm. In the previous sections, we have already fully implemented the algorithm using standard MQL5 tools. Now let's supplement its functionality with the ability to use OpenCL technology to organize the computation process in multi-threaded mode using *GPU* resources.

We have already implemented similar work for each of the previously discussed neural layers. Let me remind you of the general algorithm for constructing this process. First, we create an OpenCL program. Next, we enhance the main program code with the functionality for calling this program and passing the necessary data in both directions. We will need to send the input data to the program before its execution and calculate the results after its execution.

As usual, we start by creating an OpenCL program. Do we really need to create a new program? Why do we need to create new kernels? The answer is obvious: we need to implement functionality. But let's recall that we inherited our class from a similar class implementing the *Self-Attention* algorithm. We have repeatedly talked about the continuity of these algorithms. Can we use the kernels created earlier to implement processes in this class?

Considering the similarity of processes, it would be more advantageous for us to use the same kernels for both implementations. Firstly, this reduces the number of OpenCL objects, and the system can only handle a limited number of them. Secondly, it's always more convenient to maintain and optimize one object rather than duplicating shared blocks across multiple similar objects, be it kernels or classes.

So, how can we implement this? The previously created kernels work within the same head of attention. Of course, on the main program side, we can copy data into separate buffers and sequentially invoke kernels for each attention head. This approach is possible, but it is irrational. Excessive copying of data in itself is not the best solution. Moreover, calling kernels sequentially for each attention head doesn't allow for the simultaneous calculation of all attention heads in parallel threads.

In fact, we can use the previously created kernels without unnecessary copying, by implementing some minor modifications.

One thing we have already done from the very beginning when creating the class is fully utilizing concatenated data buffers. That is, all our data buffers contain data from all attention heads at once. By transferring data to context memory, we transfer data from all attention heads. This means that on the OpenCL side, we can work with all attention heads in parallel. We just need to correctly determine the offset in the data buffer to the required values. These are the changes that we must make to the kernel.

To determine this bias, we need to understand the total number of attention heads used and the ordinal number of the working attention head. We can pass the total quantity in parameters but can't do the same for the serial number of the current one. To implement the transfer of such data, we would need to create a loop with a sequential kernel call for each attention head, which we try to avoid.

Let's remember how the function of queuing the kernel is organized. The *CLExecute* function has a *work_dim* parameter, which is responsible for the dimension of the task space. The function also receives in parameters a dynamic array *global_work_size[]*, which indicates the total number of tasks being performed in each dimension.

```
bool CLExecute(
// handle to the OpenCL program kernel
    int          kernel,
// dimension of the problem space
    uint         work_dim,
// initial offset in task space
    const uint&  global_work_offset[],
// total number of tasks
    const uint&  global_work_size[]
);
```

Earlier we used only one dimension, and now we can use two. We will continue to use one dimension for iterating over the elements of the sequence and the other dimension for iterating over the attention heads.

Well, a solution has been found and we can begin implementation. But there is one more question: to create a new kernel or not. Everything points towards modifying the previously created one. But in this case, after finishing the work, we will have to take a step back and adjust the methods of the *CNeuronAttention* class. Otherwise, we will get a critical error when trying to launch the kernel.

For my part, I decided to make changes to the previously created kernel and the methods of the main program. You can choose your preferred options.

Now, let's look at the changes made to the feed-forward kernel.

In the kernel body, we request the identifiers of the launched thread and the total number of threads in two dimensions. The first dimension will specify the index of the processed request and the length of the sequence. The second dimension will indicate the number of the active attention head.

We also determine the offset to the beginning of the vector being analyzed in the query tensor and the attention coefficient matrix.

```
const int q = get_global_id(0);
const int units = get_global_size(0);
const int h = get_global_id(1);
const int heads = get_global_size(1);
int shift_query = key_size * (q * heads + h);
int shift_scores = units * (q * heads + h);
```

As you can see, compared to the previous version, the kernel differs by having a second dimension that accounts for attention heads. Accordingly, the calculation of offsets is also adjusted to account for multi-head attention.

Next, we create a system of two nested loops to calculate one vector of the matrix of dependence coefficients. This is due to the fact that to calculate one element of the sequence at the output of the attention block, we need a whole vector of the matrix of dependence coefficients. Such calculation applies to one attention head.

Also, before starting the loop system, we will prepare a local variable *summ* to sum all the values of the vector. We will need this sum later to normalize the vector values.

The outer loop has a number of iterations equal to the number of sequence elements. It will immediately indicate the analyzed element in the key tensor *Key* and the column number in the attention coefficient matrix. In the body of the loop, we will determine the offset in the key tensor to the beginning of the vector of the analyzed element in the sequence and prepare a variable for calculating the result of multiplying two vectors.

In the nested loop with a number of iterations equal to the size of the key vector, we will perform the operation of multiplying the query vector by the key vector.

After completing the iterations in the nested loop, we will take the exponential of the obtained result from the vector multiplication, record the resulting value in the tensor of attention coefficient matrices, and add it to our sum of vector values.

```
TYPE summ = 0;
for(int s = 0; s < units; s++)
{
    TYPE score = 0;
    int shift_key = key_size * (s * heads + h);
    for(int k = 0; k < key_size; k++)
```

```

        score += queries[shift_query + k] * keys[shift_key + k];
    score = exp(score / sqrt((TYPE)key_size));
    summ += score;
    scores[shift_scores + s] = score;
}

```

After completing all iterations of the loop system, we will have a vector with computed but unnormalized attention coefficients for one query vector with respect to all key vectors. To complete the vector normalization process, we need to divide the contents of the vector by the sum of all its values, which we have collected in the *summ* variable.

To perform this operation, we will create another loop with the number of iterations equal to the number of elements in the sequence.

```

for(int s = 0; s < units; s++)
    scores[shift_scores + s] /= summ;

```

As you can see, this block differs from the previous implementation only in terms of calculating the offsets of elements in tensors. Now that we have the normalized attention vector for one query with respect to all elements in the key tensor sequence, we can calculate the weighted vector for one element of the sequence at the output of one attention head. To do this, we will create a system of two nested loops.

First, we will determine the offset in the result tensor to the beginning of the vector for the analyzed element.

Then we will create an outer loop based on the number of elements in the result vector. In the body of the loop, we will first prepare a variable for accumulating the value of one element of the vector. We will create a nested loop with the number of iterations equal to the number of sequence elements, in which we will iterate through all the elements of the tensor of values. In each element of the description vector of an element, we will take one value corresponding to the counter of the outer loop iteration and multiply it by the element of the normalized attention coefficient vector according to the counter of the nested loop iteration. After completing the full cycle of iterations in the nested loop, the *query* variable will contain one value of the description vector for the analyzed element of the attention block sequence. We will write it to the corresponding element of the kernel work results buffer.

```

shift_query = window * (q * heads + h);
for(int i = 0; i < window; i++)
{
    TYPE query = 0;
    for(int v = 0; v < units; v++)
        query += values[window * (v * heads + h) + i] * scores[shift_scores + v];
    outputs[shift_query + i] = query;
}
}

```

After completing the iterations of the outer loop, we will obtain a complete description vector for one element of the sequence in the result tensor buffer.

As you can see, the operations of one kernel result in one description vector for an element of the sequence in the result tensor of one attention head. To calculate the complete tensor, we need to launch a task pool with a size equal to the product of the number of elements in the sequence and the number of attention heads. This is what we do when running the kernel in a two-dimensional task space.

To transform the kernel from the single-head attention plane to multi-head attention, we simply needed to organize the kernel launch in a two-dimensional space and adjust the offset calculations in the data buffers.

Let's do a similar job with backpropagation kernels. As you may recall, in the *Self-Attention* block, in contrast to the implementation of other neural layers, we implemented the propagation of the error gradient through the internal space of the hidden neural layer using two consecutive kernels. So, we need to transfer both kernels into the area of multi-head attention. However, let's consider things in order.

First, we will look at the *AttentionCalcScoreGradient* kernel. The kernel parameters remain unchanged. Here we have the same data buffers and one constant size of the description vector of one element.

```
__kernel void AttentionCalcScoreGradient(__global TYPE *scores,
                                         __global TYPE *scores_grad,
                                         __global TYPE *values,
                                         __global TYPE *values_grad,
                                         __global TYPE *outputs_grad,
                                         __global TYPE *scores_temp,
                                         int window)
{
```

In the kernel body, similar to the feed-forward kernel, we add the retrieval of thread identification in the second dimension and adjust the calculation of offsets in data buffers accordingly.

```
    const int q = get_global_id(0);
    const int units = get_global_size(0);
    const int h = get_global_id(1);
    const int heads = get_global_size(1);
    int shift_value = window * (q * heads + h);
    int shift_score = units * (q * heads + h);
```

We do not change the kernel algorithm. As with the implementation of the *Self-Attention* algorithm, the kernel can be logically divided into two blocks.

In the first algorithm, we distribute the error gradient to the tensor of values *Values*. Here we create a system of two nested loops. The outer loop will have a number of iterations equal to the size of the description vector of one sequence element in the value tensor. In the loop body, we create a local variable to collect the error gradient of the analyzed element.

It should be understood that during the feed-forward pass, each element in the sequence of the value tensor has a significant influence on the value of each element in the sequence of the result tensor. The strength of this influence is determined by the corresponding column of the attention coefficient matrix, where each row corresponds to one element in the sequence tensor of results. Therefore, to obtain the error gradient vector for one element in the sequence tensor of values, we need to multiply the corresponding column of the attention coefficient matrix by the error gradient tensor at the level of the attention block results.

$$G_{Value} = Score^T * G_{Self-Attention}$$

To perform this operation, we organize a nested loop with the number of iterations equal to the number of elements in the sequence. In the body of this loop, we will multiply two vectors and write the result to the corresponding element of the error gradient buffer of the value tensor.

```
//--- Distributing the gradient on Values
for(int i = 0; i < window; i++)
{
    TYPE grad = 0;
    for(int g = 0; g < units; g++)
        grad += scores[units * (g * heads + h) + q] *
                outputs_grad>window * (g * heads + h) + i];
    values_grad[shift_value + i] = grad;
}
```

Here, we made changes only in terms of determining the offsets to the analyzed elements in the data buffers.

The second block of this kernel is responsible for propagating the gradient to the level of the dependency coefficient matrix. First, we create a system of two nested loops and calculate the error gradient for one row of the dependency coefficient matrix. There is a very important moment here. We calculate the error gradient specifically for a matrix row, not a column. The normalization of the matrix with the *Softmax* function was performed row-wise, so we should also adjust it row-wise with respect to the *Softmax* derivative. To determine the error gradient for one row of the matrix, we need to take the corresponding vector from the error gradient tensor at the level of attention block results and multiply it by the key tensor of the corresponding attention head.

$$G_{Score} = G_{Self-Attention} * V^T$$

To perform the multiplication operation, we organize a nested loop.

```
//--- Gradient distribution on Score
for(int k = 0; k < units; k++)
{
    TYPE grad = 0;
    for(int i = 0; i < window; i++)
        grad += outputs_grad[shift_value + i] *
                values>window * (k * heads + h) + i];
    scores_temp[shift_score + k] = grad;
}
```

After running a full cycle of iterations of our tensor loop system, we will obtain a single row of error gradients for the dependency coefficient matrix. Before passing the error gradient further, it is necessary to correct it by the derivative of the *Softmax* function.

```
//--- Adjust for the Softmax derivative
for(int k = 0; k < units; k++)
{
    TYPE grad = 0;
    TYPE score = scores[shift_score + k];
    for(int i = 0; i < units; i++)
        grad += scores[shift_score + i] *
                ((int)(i == k) - score) * scores_temp[shift_score + i];
    scores_grad[shift_score + k] = grad;
}
```

The operation results are written into the corresponding elements of the error gradient tensor.

This completes the work with the first kernel of the backpropagation algorithm. As you may have noticed, the changes affected only the definition of the offset in the data buffers and the additional dimension of the task space.

Let's move on to the second kernel of the *AttentionCalcHiddenGradient* error backpropagation algorithm. In this kernel, we need to propagate the error gradient from the dependency coefficient matrix to the buffers of the *m_cQuery*s and *m_cKeys* internal neural layers.

This operation is not difficult from a mathematical point of view. We have already determined the error gradient at the level of the dependency coefficient matrix in the previous kernel. Now we need to multiply the dependency coefficient matrix by the opposite tensor.

As in the previous kernel, the kernel header and parameters have not changed at all. Here we see the same set of buffers and parameters.

```
__kernel void AttentionCalcHiddenGradient(__global TYPE *querys,
                                         __global TYPE *querys_grad,
                                         __global TYPE *keys,
                                         __global TYPE *keys_grad,
                                         __global TYPE *scores_grad,
                                         int key_size)
{
```

In the kernel body, we identify the thread in two dimensions of tasks. The second dimension has been added for the identification of the active attention head. We adjust the offsets in the gradient buffers accordingly, ensuring they are aligned with the elements of the sequence being analyzed.

```
    const int q = get_global_id(0);
    const int units = get_global_size(0);
    const int h = get_global_id(1);
    const int heads = get_global_size(1);
    int shift_query = key_size * (q * heads + h);
    int shift_score = units * (q * heads + h);
```

As mentioned earlier, in the kernel body, we need to distribute the error gradient to two internal neural layers from a single source. The same algorithm is used for gradient error distribution in both directions. And both recipient vectors have the same size. All of this allows us to calculate the error gradient for both tensors in parallel within the body of a single loop system. The number of iterations in the outer loop is equal to the size of the vector for which we are calculating the error gradient. In its body, we prepare variables for accumulating the error gradients and create a nested loop with a number of iterations equal to the number of elements in the sequence. In the body of the nested loop, we simultaneously calculate values from the product of two pairs of vectors.

```
    //--- Propagate the gradient on Querys and Keys
    const TYPE k = 1 / sqrt((TYPE)key_size);
    //---
    for(int i = 0; i < key_size; i++)
    {
        TYPE grad_q = 0;
        TYPE grad_k = 0;
        for(int s = 0; s < units; s++)
        {
            grad_q += keys[key_size * (s * heads + h) + i] *
                    scores_grad[shift_score + s];
```

```

        grad_k += queries[key_size * (s * heads + h) + i] *
                scores_grad[units * (s * heads + h) + q];
    }
    queries_grad[shift_query + i] = grad_q * k;
    keys_grad[shift_query + i] = grad_k * k;
}
}

```

After exiting the nested loop, each variable has one value for the error gradient vectors of the required tensors. We write them into the corresponding elements of the tensors. After completing the full number of iterations of the loop system, we obtain the two desired vectors of error gradients.

We finish working with OpenCL program kernels. Here, we have only made slight changes to the kernels of the *Self-Attention* algorithm to transfer them to the area of multi-headed attention.

Now we have to supplement the main program with the functionality of calling kernel data from methods of both the *CNeuronAttention* class and the *CNeuronMHAttention* class. We usually start this work by creating constants for working with kernels. But in this case, the constants have already been created.

Next, we created kernels in the OpenCL context. But this time we did not create new kernels. The ones that we slightly adjusted are already declared in the body of the main program. Therefore, we skip this step too.

Let's move on to making changes directly to class methods. For new kernels to work in the *CNeuronAttention* class, we add a second element to the offset and task space arrays. For offset, we specify 0 in both dimensions. For the task space, we leave the first value unchanged, and in the second element of the array, we introduce 1 (indicating the use of a single attention head). Additionally, when enqueueing the kernel for execution, we specify the two-dimensionality of the task space.

```

int off_set[] = {0, 0};
int NDRange[] = {m_iUnits, 1};
if(!m_cOpenCL.Execute(def_k_AttentionFeedForward, 2, off_set, NDRange))
    return false;

```

After this, we can fully use the updated feed-forward kernel.

We do such simple manipulations to call all three kernels in the methods of the *CNeuronAttention* class.

So, we have restored the functionality of the methods of the *CNeuronAttention* class, which implements the *Self-Attention* algorithm. There are also some changes on the main program side.

Let's move on to working on our *CNeuronMHAttention* class with the implementation of the *Multi-Head Self-Attention* algorithm. As usual, we'll start with the feed-forward method. Before we queue the kernel for operations, we need to do some preparatory work. First of all, we check the presence of the necessary buffers in the OpenCL context memory.

```

bool CNeuronMHAttention::FeedForward(CNeuronBase *prevLayer)
{
    .....
    //--- branching of the algorithm across the computing device
    MATRIX out;
    if(!m_cOpenCL)
    {
        .....
    }
}

```

```

    }
    else // OpenCL block
    {
        //--- check data buffers
        if(m_cQuerys.GetOutputs().GetIndex() < 0)
            return false;
        if(m_cKeys.GetOutputs().GetIndex() < 0)
            return false;
        if(m_cValues.GetOutputs().GetIndex() < 0)
            return false;
        if(m_cScores.GetIndex() < 0)
            return false;
        if(m_cAttentionOut.GetOutputs().GetIndex() < 0)
            return false;
    }

```

After checking all the necessary buffers, we pass pointers to the buffers to the kernel parameters. There we also pass the constants necessary for the operation of the kernel.

Please note that when passing parameters to the kernel, we specified the *m_iKeysSize* variable, which contains the size of the key vector for one element of the sequence, twice. We specified it for both the key vector size parameter and the value vector size parameter. Two parameters in the kernel are a necessary measure. When using a single attention head, for the size of the value vector, we would need to specify the size of the input data vector. This is a requirement of the *Self-Attention* algorithm. However, when using multi-head attention, the *WO* matrix allows us to use different sizes for the value vector.

```

//--- pass parameters to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward,
                                def_attff_keys, m_cKeys.GetOutputs().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward,
                                def_attff_outputs, m_cAttentionOut.GetOutputs().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward,
                                def_attff_querys, m_cQuerys.GetOutputs().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward,
                                def_attff_scores, m_cScores.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward,
                                def_attff_values, m_cValues.GetOutputs().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_AttentionFeedForward,
                           def_attff_key_size, m_iKeysSize))
    return false;
if(!m_cOpenCL.SetArgument(def_k_AttentionFeedForward,
                           def_attff_window, m_iKeysSize))
    return false;

```

This concludes the preparatory work, and we can move on to organizing the kernel launch procedure. To do this, we indicate the size of the problem space in two dimensions. In the first dimension, we indicate the size of the sequence; in the second one, we indicate the number of attention heads. Let's call the method for adding the kernel to the execution queue.

```

//--- putting the kernel into the execution queue
int off_set[] = {0, 0};
int NDRange[] = {m_iUnits, m_iHeads};
if(!m_cOpenCL.Execute(def_k_AttentionFeedForward, 2, off_set, NDRange))
    return false;
}

```

Here, we conclude our work on the feed-forward method and transition to the *CalcHiddenGradient* method that propagates the error gradient through the hidden layer. To implement the process of this method, we have prepared two kernels, which we need to launch sequentially. First, we will run the error gradient propagation kernel up to the *AttentionCalcScoreGradient* dependency coefficient matrix.

The algorithm for carrying out the preparatory work and launching the kernel is similar to what we used above when launching the forward pass kernel.

```

bool CNeuronMHAttention::CalcHiddenGradient(CNeuronBase *prevLayer)
{
//--- branching of the algorithm across the computing device
if(!m_cOpenCL)
{
.....
// MQL5 block
}
else // OpenCL block
{
//--- check data buffers
if(m_cValues.GetOutputs().GetIndex() < 0)
    return false;
if(m_cValues.GetGradients().GetIndex() < 0)
    return false;
if(m_cScores.GetIndex() < 0)
    return false;
if(m_cAttentionOut.GetGradients().GetIndex() < 0)
    return false;
if(m_cScoreGrad < 0)
    return false;
if(m_cScoreTemp < 0)
    return false;
}
}

```

After checking the buffers, we pass pointers to them and the necessary constants as parameters to the kernel.

```

//--- passing parameters to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
    def_attscr_outputs_grad, m_cAttentionOut.GetGradients().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
    def_attscr_scores, m_cScores.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
    def_attscr_scores_grad, m_cScoreGrad))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,

```



```

def_attscr_scores_temp, m_cScoreTemp))

return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                                def_attscr_values, m_cValues.GetOutputs().GetIndex()))

return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                                def_attscr_values_grad, m_cValues.GetGradients().GetIndex()))

return false;
if(!m_cOpenCL.SetArgument(def_k_AttentionScoreGradients,
                           def_attscr_window, m_iKeysSize))

return false;

```

We place the kernel in the queue for performing operations. As with the feed-forward pass, we create a two-dimensional task space. In the first dimension, we specify the number of elements being analyzed in the sequence, and in the second dimension, we specify the number of attention heads.

```

//--- place the kernel into the execution queue
int off_set[] = {0, 0};
int NDRange[] = {m_iUnits, m_iHeads};
if(!m_cOpenCL.Execute(def_k_AttentionScoreGradients, 2, off_set, NDRange))

return false;

```

We immediately begin the preparatory work before launching the second kernel. Checking the data buffers in the OpenCL context memory. Only those buffers that we did not check when launching the first kernel are subject to verification.

```

//--- check data buffers
if(m_cQuerys.GetOutputs().GetIndex() < 0)

return false;
if(m_cQuerys.GetGradients().GetIndex() < 0)

return false;
if(m_cKeys.GetOutputs().GetIndex() < 0)

return false;
if(m_cKeys.GetGradients().GetIndex() < 0)

return false;

```

We pass pointers to data buffers to the parameters of the second kernel. We also add the necessary constants there.

```

//--- pass arguments to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                                def_atthgr_keys, m_cKeys.GetOutputs().GetIndex()))

return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                                def_atthgr_keys_grad, m_cKeys.GetGradients().GetIndex()))

return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                                def_atthgr_querys, m_cQuerys.GetOutputs().GetIndex()))

return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                                def_atthgr_querys_grad, m_cQuerys.GetGradients().GetIndex()))

return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,

```

```

def_atthgr_scores_grad, m_cScoreGrad))
    return false;
if(!m_cOpenCL.SetArgument(def_k_AttentionHiddenGradients,
                           def_atthgr_key_size, m_iKeysSize))
    return false;

```

After completing the preparatory work, we call the method for placing the kernel in the tasks execution queue. Please note that this time we are not creating new arrays specifying the task space because it has not changed, and we can use the existing arrays from the previous kernel launch.

```

//--- place the kernel into the execution queue
if(!m_cOpenCL.Execute(def_k_AttentionHiddenGradients, 2, off_set, NDRange))
    return false;
}

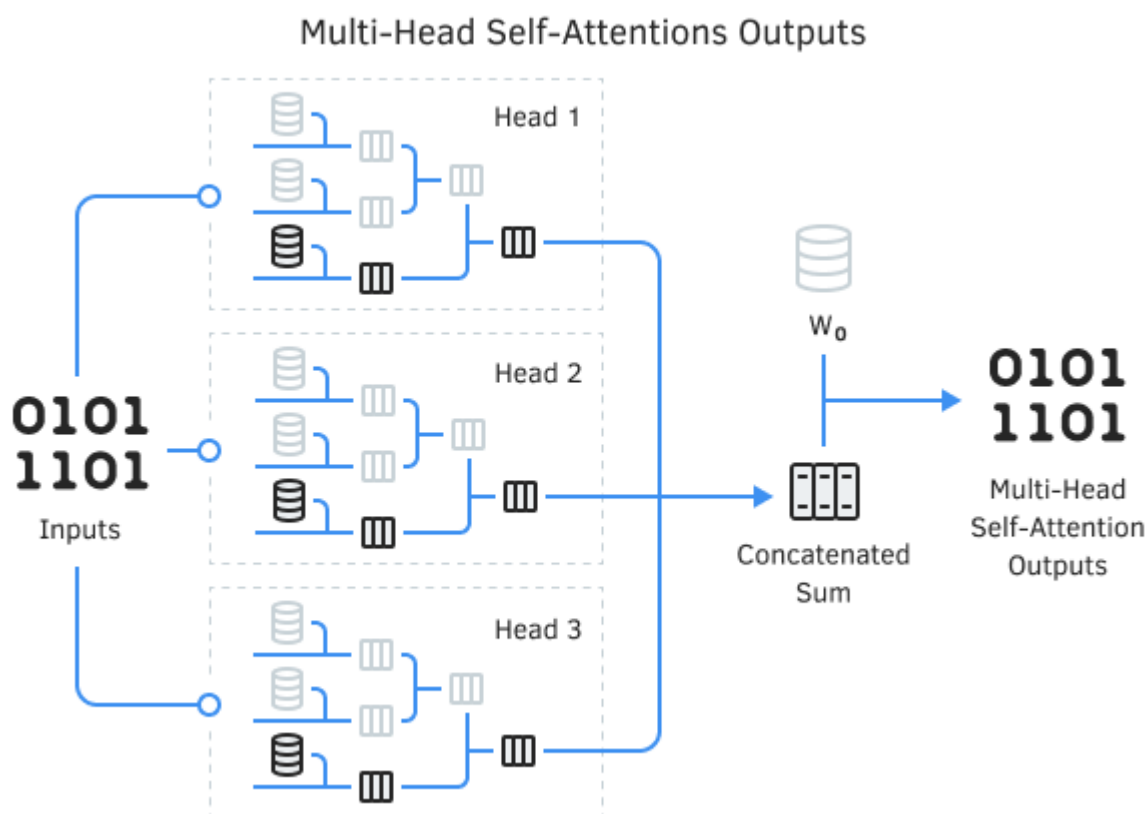
```

This concludes our work on the implementation of the *Multi-Head Self-Attention* algorithm, including general and multi-threaded calculations. We have implemented all the functionality of the *CNeuronMHAttention* class. Now we can proceed with comprehensive testing of its performance using training and testing datasets.

5.2.4 Building Multi-Head Self-Attention in Python

We have already implemented the *Multi-Head Self-Attention* algorithm using MQL5 and have even added the ability to perform multi-threaded calculations using OpenCL. Now let's look at an option for implementing such an algorithm in Python using the *Keras* library for *TensorFlow*. We had to deal with this library when creating previous models. Indeed, up to this point, we have been using only pre-built neural layers offered by the library, and with their help, we constructed linear models.

The *Multi-Head Self-Attention* model cannot be called linear. The parallel work of several heads of attention in itself is a rejection of the linearity of the model. In the *Self-Attention* algorithm itself, the source data simultaneously goes in four directions.



Therefore, to build a *Multi-Head Self-Attention* model, we will consider another functionality offered by this library, which is creating custom neural layers.

A layer is a callable object that takes one or more tensors as input and outputs one or more tensors. It includes computation and status.

All neural layers in the *Keras* library represent classes inherited from the *tf.keras.layers.Layer* base class. Therefore, when creating a new custom neural layer, we will also inherit from the specified base class.

The base class provides the following parameters:

- *trainable* – flag that indicates the need to train the parameters of the neural layer
- *name* – layer name
- *dtype* – type of layer results and weighting factors
- *dynamic* – flag that indicates that the layer cannot be used to create a graph of static calculations

```
tf.keras.layers.Layer(
    trainable=True, name=None, dtype=None, dynamic=False, **kwargs
)
```

Also, the library architecture defines a minimum set of methods for each layer:

- *__init__* – layer initialization method
- *call* – calculation method (feed-forward pass)

In the initialization method, we define the custom attributes of the layer and create weight matrices, the structure of which does not depend on the format and structure of the input data. However, when

solving practical problems, we often do not know the structure of the input data, and as a result, we cannot create weight matrices without understanding the dimensionality of the input data. In such cases, the initialization of weight matrices and other objects is transferred to the *build(self, input_shape)* method. This method is called once, during the first call of the *call* method.

The *call* method describes the forward-pass operations that must be performed with the initial data. The results of the operations are returned as one or more tensors. For layers used in linear models, there is a restriction on the result in the form of a single tensor.

Each neural layer has the following attributes (a list of the most commonly used attributes is provided):

- *name* – layer name
- *dtype* – type of weighting factors
- *trainable_weights* – list of variables to be trained
- *non_trainable_weights* – list of non-trainable variables
- *weights* – combines lists of trainable and non-trainable variables
- *trainable* – logical flag that indicates the need to train layer parameters
- *activity_regularizer* – additional regularization function for the output of the neural layer.

The advantages of this implementation are obvious: we are not creating backpropagation methods. All functionality is implemented by the library. We just need to correctly describe the logic of the feed-forward pass in the *call* method.

This approach makes it possible to create rather complex architectural solutions. Moreover, the created layer may contain other nested neural layers. At the same time, the parameters of the internal neural layers are included in the list of parameters of the external neural layer.

5.2.4.1 Creating a new neural layer class

Let's get to the practical part and look at the implementation of our multi-head attention neural layer. To implement it, we create a new *MHAttention* class that inherits from the base class of all neural layers *tf.keras.layers.Layer*.

```
# Multi-Head Self-Attention Model
class MHAttention(tf.keras.layers.Layer):
```

First, we'll override the layer initialization method `__init__`. In the parameters of the initialization method, we will specify two constants:

- *key_size* – size of the vector describing one element of the sequence in the tensor of Keys
- *heads* – number of attention heads

In the body of the method, we will save the parameters in local variables for future use and immediately calculate the size of the concatenated output of attention heads into the variable *m_iDimension*.

For your convenience, I made an effort to repeat the names of variables from the MQL5 implementation as much as possible.

Next, we declare the internal objects of our neural layer. However, note that in this case, we do not specify the vector size of one element of the source data sequence. This is made possible by the use of multidimensional tensors.

The *TensorFlow* library works with multidimensional arrays or tensors represented as objects. This approach makes understanding the model more convenient and visual. To be able to implement the task in OpenCL, we were forced to use one-dimensional data buffers. To gain access to the required element, we calculated the offset in the one-dimensional buffer. Now, when using multidimensional arrays, to access the matrix element, we just need to specify the row and column of the element. It is convenient and clear.

Another advantage of this approach is that we do not need to specify the dimension of the source data. We can get it from the tensor itself. We will take advantage of this. We won't ask the user for the size of the description vector for one element of the input data sequence. Instead, we will receive the input data tensor as a matrix. Each line of such a matrix is a vector description of one element of the sequence. We can operate with the size of this vector. That is, the first dimension indicates the number of elements of the sequence, and the second means the length of the description vector of one element of the sequence.

However, there is also the other side of the coin. At the time of class initialization, we have not yet received the initial data. So, we do not know its size, as the user did not specify them in the parameters. Therefore, we cannot create all objects in the initialization method. But it doesn't matter. We will do what we can.

In the initialization method, we will declare objects that can be created without understanding the dimension of the source data:

- *m_cQueries* – neural layer for the formation of the concatenated tensor of queries *Query*
- *m_cKeys* – neural layer for the formation of the concatenated tensor of keys *Key*
- *m_cValues* – neural layer for the formation of the concatenated tensor of values *Values*
- *m_cNormAttention* – data normalization layer for the *Multi-Head Self-Attention* block
- *m_cNormOutput* – normalization layer for the results of the neural layer

```

def __init__(self, key_size, heads, **kwargs):
    super(MHAttention, self).__init__(**kwargs)

    self.m_iHeads = heads
    self.m_iKeysSize = key_size
    self.m_iDimension = self.m_iHeads * self.m_iKeysSize;

    self.m_cQuerys = tf.keras.layers.Dense(self.m_iDimension)
    self.m_cKeys = tf.keras.layers.Dense(self.m_iDimension)
    self.m_cValues = tf.keras.layers.Dense(self.m_iDimension)
    self.m_cNormAttention = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.m_cNormOutput = tf.keras.layers.LayerNormalization(epsilon=1e-6)

```

After creating the initialization method, we proceed to the *build* method. This method will allow us to initialize the missing objects. This method is run only once before the first call of the *call* method. It receives the source data size in the parameters. Knowing this size, we can initialize objects, structures, and/or parameters that depend on the size of the source data.

In the method body, we save the last dimension of the source data tensor as the size of the description vector of one element of the source data sequence to the *m_iWindow* local variable. After that, we will create three more internal neural layers:

- *m_cW0* – fully connected layer of the reduction matrix W_o
- *m_cFF1* – the first fully connected layer of the *Feed Forward* block
- *m_cFF2* – the second fully connected layer of the *Feed Forward* block

```

def build(self, input_shape):
    self.m_iWindow = input_shape[-1]
    self.m_cW0 = tf.keras.layers.Dense(self.m_iWindow)
    self.m_cFF1 = tf.keras.layers.Dense(4 * self.m_iWindow,
                                         activation=tf.nn.swish)
    self.m_cFF2 = tf.keras.layers.Dense(self.m_iWindow)

```

So, we have defined all the internal objects necessary to implement the *Multi-Head Self-Attention* algorithm inside our new layer. Before proceeding with the implementation, let's once again look at how we can write the algorithm of multi-head attention using matrix mathematics since when working with multidimensional tensors, we must operate with matrix operations.

The first step is to define the *Query*, *Key*, and *Value* tensors. To obtain query data, we need to multiply the tensor of the source data by the corresponding matrix of weights. This operation is performed in three internal neural layers.

```

def call(self, data):
    batch_size = tf.shape(data)[0]
    query = self.m_cQuerys(data)
    key = self.m_cKeys(data)
    value = self.m_cValues(data)

```

The second step is to determine the matrix of dependency coefficients. According to the *Self-Attention* algorithm, we first need to multiply the query tensor by the transposed key tensor.

$$Score = Query * Key^T$$

Everything is simple for just one attention head. But we have concatenated tensors, which in the last dimension contain the data of all attention heads. Multiplying them in this form will give us a result comparable to one-headed attention. As an option, we can transform the two-dimensional tensor into a three-dimensional one, separating the attention head into a distinct dimension.

$$[Units, Concatenate] \Rightarrow [Units, Heads, Vector]$$

Multiplying the last two dimensions in this form is also not quite what we would like to get. However, if we swap the first and second dimensions, then we can multiply the last two dimensions to get the result we are looking for.

$$[Units, Heads, Vector] \Rightarrow [Heads, Units, Vector]$$

The described procedure will be placed in a separate function *split_heads*.

```
def split_heads(self, x, batch_size):
    x = tf.reshape(x, (batch_size, -1,
                      self.m_iHeads,
                      self.m_iKeysSize))
    return tf.transpose(x, perm=[0, 2, 1, 3])
```

Inside the *call* method, we transform tensors and multiply them according to the *Self-Attention* algorithm.

```
query = self.split_heads(query, batch_size)
key = self.split_heads(key, batch_size)
value = self.split_heads(value, batch_size)
score = tf.matmul(query, key, transpose_b=True)
```

Next, we need to divide the obtained dependence coefficients by the square root of the dimension of the key vector and normalize it with the *Softmax* function according to the last dimension of the tensor.

```
score = score / tf.math.sqrt(tf.cast(self.m_iKeysSize, tf.float32))
score = tf.nn.softmax(score, axis=-1)
```

Now we only need to multiply the normalized dependency coefficients by the *Value* tensor.

```
attention = tf.matmul(score, value)
```

As a result of this operation, we will get the attention block result for each attention head. To continue the algorithm, we need a concatenated tensor of all attention heads. Therefore, we need to carry out the reverse procedure of the tensor transformation. Once again, we rearrange the first and second dimensions and change the dimension of the tensor from three-dimensional to two-dimensional.

```
attention = tf.transpose(attention, perm=[0, 2, 1, 3])
attention = tf.reshape(attention, (batch_size, -1, self.m_iDimension))
```

After that, using the W_0 matrix, we convert the concatenated tensor of the results to the size of the tensor of the initial data. Add the two tensors and normalize the result.

```
attention = self.m_cW0(attention)
attention = self.m_cNormAttention(data + attention)
```

This concludes the first block of the *Multi-Head Self-Attention* algorithm, followed by two consecutive fully connected layers of the *Feed Forward* block. The first neural layer will be with the *Swish* activation function, and the second one will have no activation function.

```
output=self.m_cFF1(attention)
output=self.m_cFF2(output)
```

At the end of the method, we add the result tensors of the *Multi-Head Self-Attention* and *Feed Forward* blocks and normalize the layer. The result of the operations is returned in the form of a tensor.

```
output=self.m_cNormOutput(attention+output)
return output
```

We have implemented a minimal set of methods of the class, sufficient to test its functionality. However, we will not be able to save the model with this class in this form. This is not good because our goal is to build and train a model with the subsequent possibility of practical use. Therefore, the ability to save the model and then restore it is one of the key requirements.

First, to enable the saving of the new object, which is our neural layer, it is necessary to add it to the list of custom objects and provide serialization capabilities for the object. This allows us to make a directive `register_keras_serializable`, which we will add before declaring the class of our neural layer.

```
# Multi-Head Self-Attention model
@tf.keras.utils.register_keras_serializable(package="Custom", name='MHAttention')
class MHAttention(tf.keras.layers.Layer):
```

But that's not all. We still need to add the `get_config` method, which will return the contents of variables to save to a file. Note that among the variables there are both those specified by the user when initializing the class object and those saved from the size of the initial data. Our weights are tuned to these dimensions.

```
def get_config(self):
    config={'key_size': self.m_iKeysSize,
           'heads': self.m_iHeads,
           'dimension': self.m_iDimension,
           'window': self.m_iWindow
          }
    base_config = super(MHAttention, self).get_config()
    return dict(list(base_config.items()) + list(config.items()))
```

The `from_config` method is responsible for restoring data from the configuration list. However, please note the following. In the usual logic, the parameters from the class initialization method are specified in the configuration dictionary. But we also saved data that depends on the size of the initial data. And, as you remember, they are not included in the parameters of the initialization method. In its pure form, we will get an error about the presence of unknown parameters. Therefore, at the beginning of the method, we remove them from the configuration directory, but at the same time save the values to local variables. And only after that, we restore the layer.

```
@classmethod
def from_config(cls, config):
    dimension=config.pop('dimension')
    window=config.pop('window')
    layer = cls(**config)
    layer._build_from_signature(dimension, window)
    return layer
```

After initializing our neural layer from the configuration dictionary, we need to pass the values we previously extracted about the configuration of the input data into the respective variables. To perform this functionality, we will call the `_build_from_signature` method, which we will also have to override.


```
def _build_from_signature(self, dimension, window):  
    self.m_iDimension=dimension  
    self.m_iWindow=window
```

With that, we conclude our work on the class of our neural layer and can move on to creating a model to test the newly created *Multi-Head Self-Attention* neural layer.

5.2.4.2 Creating a script to test Multi-Head Self-Attention

To test the operation of our new class of the *Multi-Head Self-Attention* neural layer, we will create a script with the implementation of the neural network model, in which we will use the new type of neural layer. We will create our script based on the [lstm.py](#) script, which we used earlier to test recurrent models. Before we start, let's create a copy of the specified script with the file name *attention.py*. In the new copy of the script, we will delete the previously created models, leaving only the convolution model and the best recurrent model. They will serve as a basis for comparing new models.

```
# A model with a 2-dimensional convolutional layer
model3 = keras.Sequential([keras.Input(shape=inputs),
    # Reformat the tensor into a 4-dimensional one.
    # Specify 3 dimensions, as the 4th dimension is determined by the size of the
    keras.layers.Reshape((-1,4,1)),
    # A convolution layer with 8 filters
    keras.layers.Conv2D(8,(3,1),1,activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    # Subsample layer
    keras.layers.MaxPooling2D((2,1),strides=1),
    # Reformat the tensor to 2-dimensional for fully connected layers
    keras.layers.Flatten(),
    keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(targetts, activation=tf.nn.tanh)
])

# LSTM block model with no fully connected layers
model4 = keras.Sequential([keras.Input(shape=inputs),
    # Reformat the tensor into 3-dimensional.
    # Specify 2 dimensions, as the 3rd dimension is determined by the size of the
    keras.layers.Reshape((-1,4)),
    # 2 consecutive LSTM blocks
    # 1 contains 40 elements
    keras.layers.LSTM(40,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5),
    return_sequences=False),
    # 2nd gives the result instead of a fully connected layer
    keras.layers.Reshape((-1,2)),
    keras.layers.LSTM(targetts)
])
```

To build the initial model, we created a fairly simple architecture consisting of one attention layer, three fully connected hidden layers, and one fully connected output layer. We used almost the same model architecture above to build the convolutional model. The use of similar models enables the accurate evaluation of the impact of new solutions on the overall performance of the model.

```
heads=8
key_dimension=4
```

```

model5 = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
    # Reformat the tensor into 3-dimensional. Specify 2 dimensions,
    # as the 3rd dimension is determined by the size of the batch
    # first dimension is for sequence elements
    # second dimension is for the vector describing of one element
    keras.layers.Reshape((-1,4)),
    MHAttention(key_dimension,heads),

```

Since our attention layer returns a tensor of the same size as its input, we need to reshape the data into a two-dimensional space before using the block of fully connected layers.

```

    # Reformat the tensor to 2-dimensional for fully connected layers
    keras.layers.Flatten(),
    keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
    keras.layers.Dense(targets, activation=tf.nn.tanh)
])

```

It should be noted that despite the external similarity of the models, the model utilizing the attention mechanism layer uses five times fewer parameters.

However, using a single attention layer is a simplified model and is employed solely for comparative experimentation purposes. In practice, it's more common to use multiple consecutive attention layers. I suggest evaluating the impact of multiple attention layers used in the model on real-world data. To conduct such an experiment, we will sequentially add three more attention layers with the same parameters to our previous model.

Layer (type)	Output Shape	Param #
=====		
reshape_3 (Reshape)	(None, 40, 4)	0

mh_attention (MHAttention)	(None, 40, 4)	776

flatten_1 (Flatten)	(None, 160)	0

dense_7 (Dense)	(None, 40)	6440

dense_8 (Dense)	(None, 40)	1640

dense_9 (Dense)	(None, 40)	1640

dense_10 (Dense)	(None, 2)	82
=====		
Total params:		10,578

Model using the Multi-Heads Self-Attention layer

```

model6 = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
    # Reformat the tensor into 3-dimensional. Specify 2 dimensions,

```

5. Attention mechanisms

```
# as the 3rd dimension is determined by the size of the package
# first dimension is for sequence elements
# second dimension is for the vector describing one element
keras.layers.Reshape((-1,4)),
MHAttention(key_dimension,heads),
MHAttention(key_dimension,heads),
MHAttention(key_dimension,heads),
MHAttention(key_dimension,heads),
# Reformat the tensor to 2-dimensional for fully connected layers
keras.layers.Flatten(),
keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
keras.layers.Dense(targerts, activation=tf.nn.tanh)
])
```

We will compile all neural models with the same parameters: the *Adam* optimization method, standard deviation as the network error, and an additional *accuracy* metric.

```
model3.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
```

We compiled neural network models with the same parameters as before.

Recurrent models are sensitive to the sequence of input signals provided. Therefore, when training a recurrent neural network, unlike the other models, you cannot shuffle the input data. Exactly for this purpose, when launching a recurrent model, we set the *shuffle* parameter to *False*. The convolution model and models using the attention layer have this parameter set to *True*. The remaining training parameters for the models remain unchanged, including the early stopping criterion when reaching a minimum error on the training dataset.

```
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=20)

history3 = model3.fit(train_data, train_target,
                     epochs=500, batch_size=1000,
                     callbacks=[callback],
                     verbose=2,
                     validation_split=0.01,
                     shuffle=True)
```

After slightly training the models, we visualize the results. We plot two graphs. On one of them, we will display the dynamics of error changes during the training and validation process.

```
# Plot the results of model training
plt.figure()
plt.plot(history3.history['loss'], label='Conv2D train')
plt.plot(history3.history['val_loss'], label='Conv2D validation')
plt.plot(history4.history['loss'], label='LSTM only train')
plt.plot(history4.history['val_loss'], label='LSTM only validation')
plt.plot(history5.history['loss'], label='MH Attention train')
```

```
plt.plot(history5.history['val_loss'], label='MH Attention validation')
plt.plot(history6.history['loss'], label='MH Attention 4 layers train')
plt.plot(history6.history['val_loss'], label='MH Attention 4 layers validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics')
plt.legend(loc='upper right', ncol=2)
```

In the second graph, we plot similar results for *Accuracy*.

```
plt.figure()
plt.plot(history3.history['accuracy'], label='Conv2D train')
plt.plot(history3.history['val_accuracy'], label='Conv2D validation')
plt.plot(history4.history['accuracy'], label='LSTM only train')
plt.plot(history4.history['val_accuracy'], label='LSTM only validation')
plt.plot(history5.history['accuracy'], label='MH Attention train')
plt.plot(history5.history['val_accuracy'], label='MH Attention validation')
plt.plot(history6.history['accuracy'], label='MH Attention 4 layers train')
plt.plot(history6.history['val_accuracy'], label='MH Attention 4 layers validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics')
plt.legend(loc='lower right', ncol=2)
```

Then we will load the test dataset and evaluate the performance of the pretrained models on it.

```
# Load testing dataset
test_filename = os.path.join(path, 'test_data.csv')
test = np.asarray( pd.read_table(test_filename,
                                sep=',',
                                header=None,
                                skipinitialspace=True,
                                encoding='utf-8',
                                float_precision='high',
                                dtype=np.float64,
                                low_memory=False))

# Split the test sample into input data and targets
test_data=test[:,0:inputs]
test_target=test[:,inputs:]

# Validation of model results on a test sample
test_loss3, test_acc3 = model3.evaluate(test_data, test_target, verbose=2)
test_loss4, test_acc4 = model4.evaluate(test_data, test_target, verbose=2)
test_loss5, test_acc5 = model5.evaluate(test_data, test_target, verbose=2)
test_loss6, test_acc6 = model6.evaluate(test_data, test_target, verbose=2)
```

The results of the model's performance on the test sample will be numerically logged and visualized on the graph.

```
# Output test results to the log
print('Conv2D model')
print('Test accuracy:', test_acc3)
print('Test loss:', test_loss3)
```

```

print('LSTM only model')
print('Test accuracy:', test_acc4)
print('Test loss:', test_loss4)

print('MH Attention model')
print('Test accuracy:', test_acc5)
print('Test loss:', test_loss5)

print('MH Attention 4l Model')
print('Test accuracy:', test_acc5)
print('Test loss:', test_loss5)

plt.figure()
plt.bar(['Conv2D', 'LSTM', 'MH Attention', 'MH Attention\n4 layers'],
        [test_loss3, test_loss4, test_loss5, test_loss6])
plt.ylabel('$MSE$ $loss$')
plt.title('Test results')

plt.figure()
plt.bar(['Conv2D', 'LSTM', 'MH Attention', 'MH Attention\n4 layers'],
        [test_acc3, test_acc4, test_acc5, test_acc6])
plt.ylabel('$Accuracy$')
plt.title('Test results')
plt.show()

```

We finalize our work on the *Multi-Head Self-Attention* mechanism. We have recreated this mechanism by means of MQL5 and in Python. In this section, we have prepared a Python script that creates a total of four neural network models:

- Convolution model
- Recurrent neural network
- Two models using *Multi-Head Self-Attention* technology

While running the script, we will conduct a brief training session for all four models using the same dataset. We will compare the performance of the trained models on the test dataset and analyze the results. This will allow us to compare the performance of various architectural solutions on real-world data. The test results will be provided in the next chapter.

5.2.5 Comparative testing of Attention models

We have done a lot of work while studying and implementing the *Multi-Head Self-Attention* algorithm. We even managed to implement it on several platforms. Earlier we created new classes only for our library in MQL5. This time we got acquainted with the possibility of creating custom neural layers in Python using the *TensorFlow library*. Now it's time to look at the results of our labor and evaluate the opportunities offered to us by the new technology.

As usual, we start testing with models created using standard MQL5 tools. We have already started this work when testing the operation of the *Self-Attention* algorithm. To run the new test, we will take [attention_test.mq5](#) from the previous test and create a copy of it named *attention_test2.mq5*.

When creating a new class for multi-head attention, we largely inherited processes from the *Self-Attention* algorithm. In some cases, methods were inherited entirely, while in others they used the *Self-Attention* methods as a foundation and created new functionality through minor adjustments. So here,

the testing script will not require major changes, and all changes will affect only the block for declaring a new layer.

Our first change is, of course, the type of neural layer we are creating. In the *type* parameter, we will specify the *defNeuronMHAttention* constant, which corresponds to the multi-head attention class.

We also need to indicate the number of attention heads used. We will specify this value in the *step* parameter. I agree that the name of the parameter is not at all consonant. However, I decided not to create an additional parameter but to use the available free fields instead.

After that, we will once again go through the script code and carefully examine the key checkpoints for executing operations.

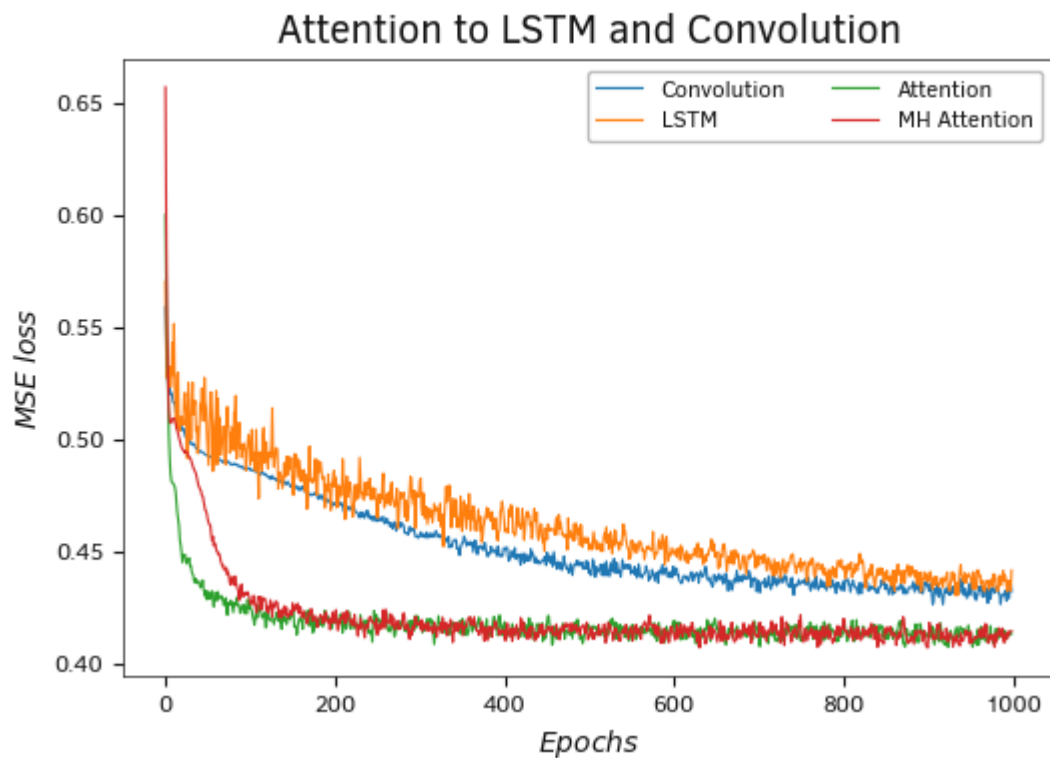
That's it. Such changes are sufficient for the first test to evaluate the net impact of the solution architecture on the model results.

```
//--- Attention layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
descr.type = defNeuronMHAttention;
descr.count = BarsToLine;
descr.window = NeuronsToBar;
descr.window_out = 8;
descr.step = 8;           // Number of attention heads
descr.optimization = Adam;
descr.activation_params[0] = 1;
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

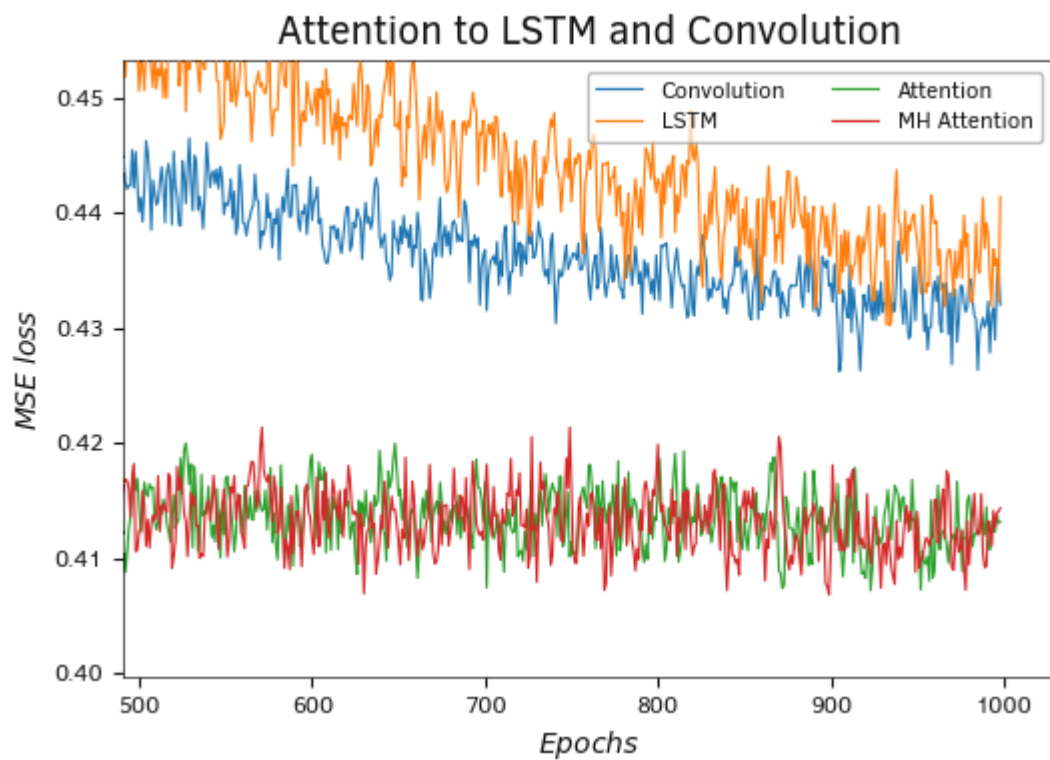
We conducted the testing directly on the same training dataset, keeping all other model parameters unchanged. Their results are shown in the graph below.

We have seen that even the use of *Self-Attention* gives us superiority over the previously considered architectural solutions of convolutional and recurrent models. Increasing the attention heads also yields a positive result.

The presented graphs depicting the neural network error dynamics on the training dataset clearly show that models using the attention mechanism train much faster than other models. Increasing the number of parameters when adding attention heads requires slightly more training time. However, this increase is not critical. At the same time, additional attention heads can reduce the error in the model operation.



If we zoom in we can clearly see that the error of models using the attention mechanism remains lower throughout the entire training. At the same time, the use of additional attention heads further improves the performance.



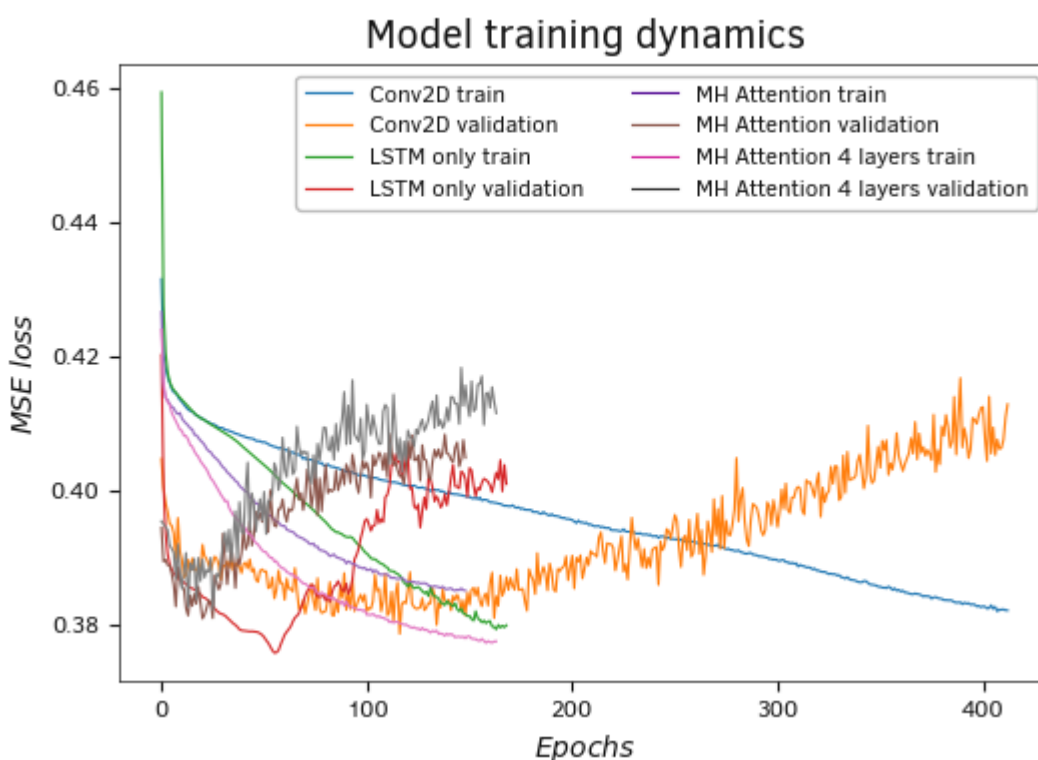
Note that the model using the convolutional layer has the highest number of trainable parameters. This provides an additional reason to reconsider the rationality of resource usage and start exploring new technologies that emerge every day.

When talking about the rational use of resources, I also want to caution against an inadequate increase in the number of attention heads being used. Each attention head means the consumption of additional resources. Find a balance between the amount of resources consumed, and the benefits that they give to the overall result. There is no universal answer. Such a decision should be made on a case-by-case basis.

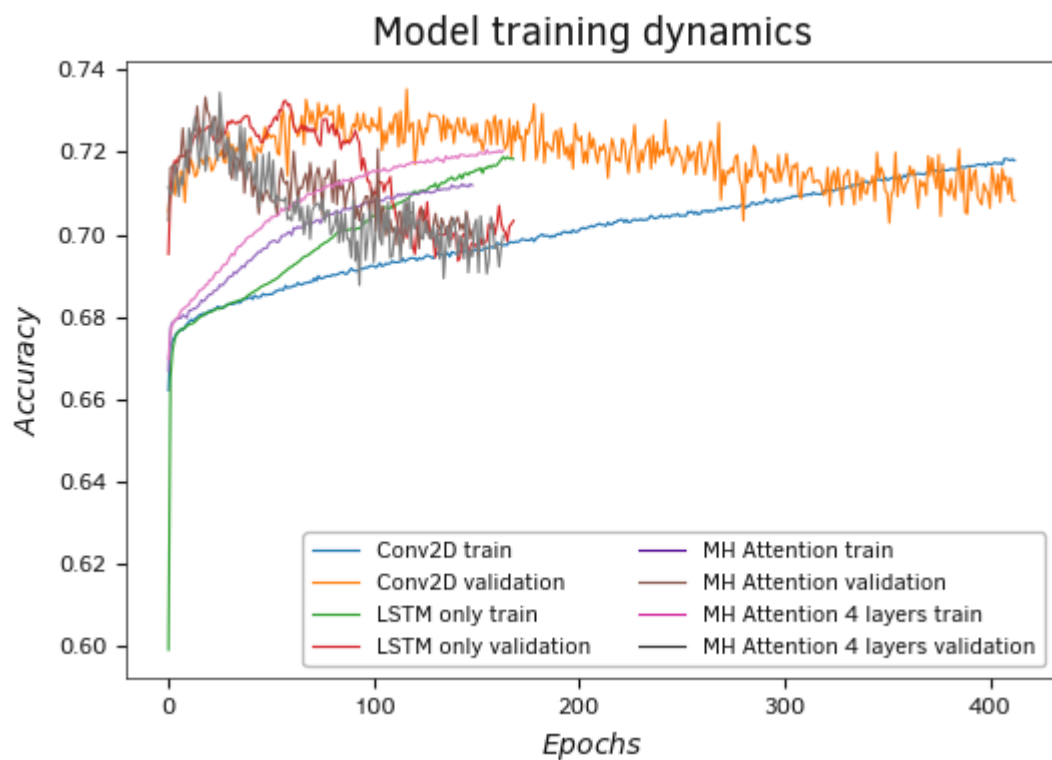
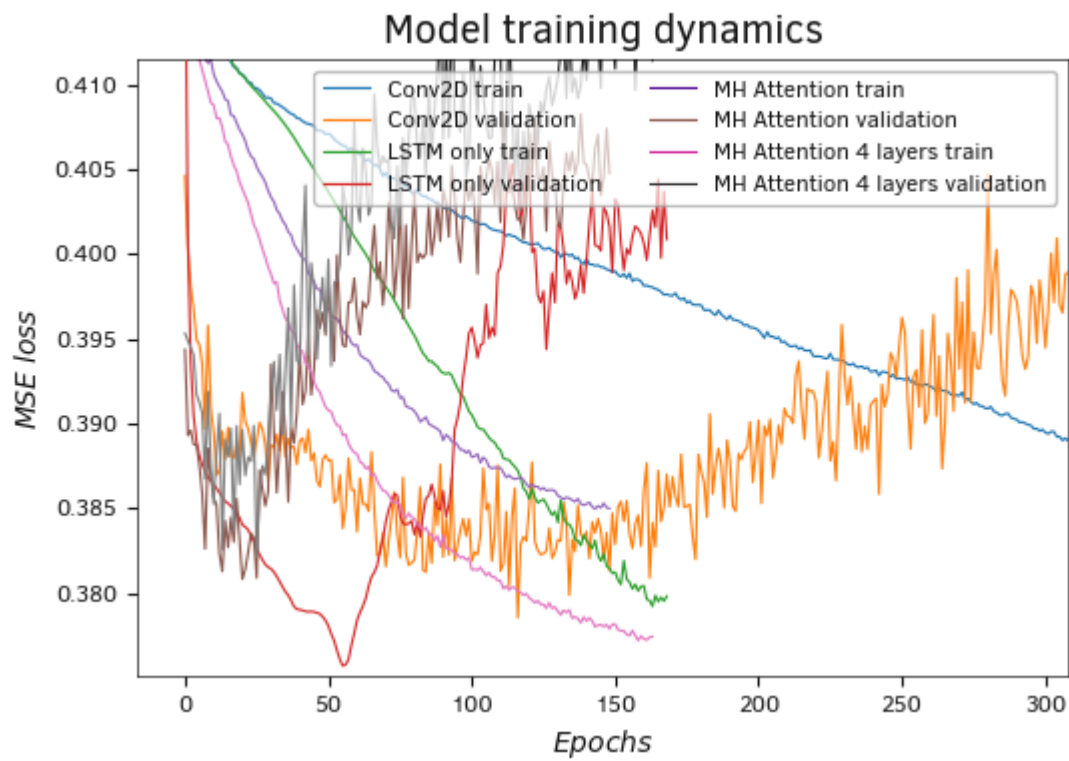
The results of test training of models written in Python also confirm the above conclusions. Models that employ attention mechanisms train faster and are also less susceptible to model overfitting. This is confirmed by a smaller gap between the error graphs for training and validation. Increasing the number of used attention layers allows the reduction of the overall model error under otherwise equal conditions.

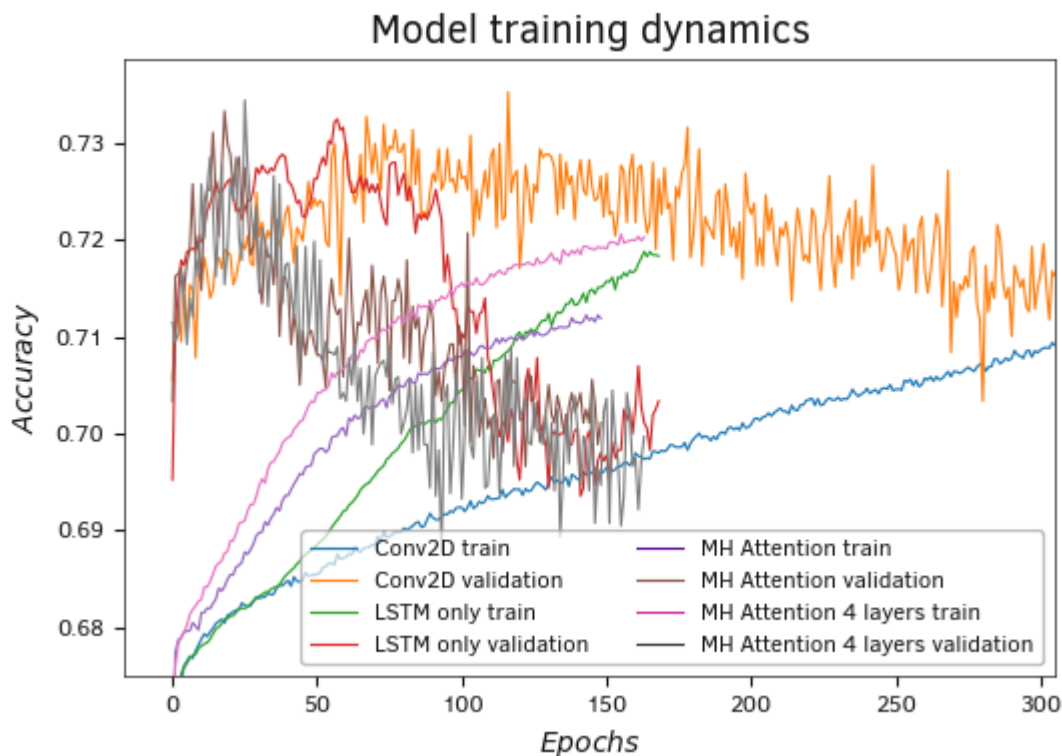
As you zoom in, you'll notice that models using attention mechanisms have straighter lines and fewer breaks. This indicates a clearer identification of dependencies and a progressive movement towards minimizing error. Partly, this can be explained by the normalization of results within the *Self-Attention* block which allows you to have a result with the same statistical indicators at the output.

The graph of the test results for the *Accuracy* metric also confirms our conclusions.



Results of test training of Python attention models





Results of test training of Python attention models

5.3 GPT architecture

In June 2018, OpenAI introduced *GPT*, the neural network model, which immediately showed the best results in a number of language tests. In February 2019, they released *GPT-2*, and in May 2020, everyone learned about *GPT-3*. These models demonstrated the possibility of a neural network to generate texts. Experiments were also conducted on the generation of music and images. The main disadvantage of the models is the requirements for computing resources. It took a month to train the first *GPT* on a machine with 8 GPUs. This disadvantage is partially compensated by the ability to use pre-trained models to solve new problems. However, the size of the model requires resources for its functioning.

Conceptually, *GPT* models are built on the basis of the transformer we have already looked at. The main idea is to pre-train a model without a teacher on a large volume of data and then fine-tune it on a relatively small amount of labeled data.

The reason for the two-step training is the size of the model. Modern deep machine learning models, such as *GPT*, have a large number of parameters, numbering in the hundreds of millions or more. Therefore, training of such neural networks requires a huge training dataset. When using supervised learning, creating a labeled training dataset can require significant effort. At the same time, there are numerous digitized texts available on the internet which are not unlabeled, making them suitable for unsupervised learning models. However, the results of unsupervised learning are statistically inferior to supervised learning. Therefore, after unsupervised learning, the model undergoes fine-tuning on a relatively small labeled dataset.

Unsupervised learning allows *GPT* to learn a language model, while fine-tuning using labeled data tailors the model for specific tasks. In this way, a single pre-trained model can be replicated and configured to

perform different language tasks. The limitation lies in the language of the source dataset for unsupervised learning.

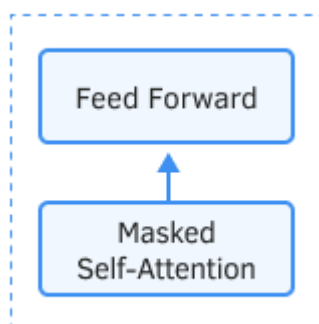
As practical experience has shown, such an approach yields good results across a wide range of language tasks. For example, the *GPT-3* model is able to generate related texts on a given topic. But it's worth noting that the mentioned model contains 175 billion parameters and was pre-trained on a dataset of 570 GB.

Despite the fact that *GPT* models were designed for natural language processing, they also showed impressive results in music and image generation tasks.

Theoretically, it is possible to use *GPT* models with any sequences of digitized data. The question lies in having sufficient data and resources for unsupervised pre-training.

5.3.1 Description of the architecture and implementation principles

Let's consider the differences between *GPT* models and the previously considered *Transformer*. First, *GPT* models do not use the encoder while only using the decoder. This has led to the disappearance of the inner layer of *Encoder-Decoder Self-Attention*. The figure below shows the transformer block in *GPT*.



GPT Block

As in the classic *Transformer*, these blocks in *GPT* models are lined up on top of each other. Each block has its own weight matrices for the attention engine and fully connected *Feed Forward* layers. The number of such blocks determines the size of the model. As it turns out, the stack of blocks can be quite large. There are 12 of them in *GPT-1* and the smallest of *GPT-2* (*GPT-2 Small*), 48 in *GPT-2 Extra Large*, and 96 in *GPT-3*.

Like traditional language models, *GPT* allows you to find relationships only with the previous elements of the sequence, not allowing you to look into the future. However, unlike the *Transformer*, it doesn't use masking of elements but rather introduces changes to the computation process. In *GPT*, the attention coefficients in the *Score* matrix for subsequent elements are zeroed.

At the same time, *GPT* can be attributed to autoregressive models. Generating one token of the sequence at a time, the generated token is added to the input sequence and fed into the model for the next iteration.

As in the classic transformer, three vectors are generated inside the *Self-Attention* mechanism for each token: *Query*, *Key*, and *Value*. In an autoregressive model, when the input sequence changes by only 1 token on each new iteration, there is no need to recalculate the vectors for each token from scratch.

That's why in GPT, each layer calculates vectors only for the new elements in the sequence and stores them for each element in the sequence. Each transformer block saves its vectors for later use.

This approach allows the model to generate text word by word until it reaches the end token.

And of course, *GPT* models use the *Multi-Head Self-Attention* mechanism.

5.3.2 Building a GPT model using MQL5

Before you start working on a *GPT* model, don't expect to get some kind of a beast at the end of the section that can solve any problems. We only build the model algorithms. The operation of these algorithms will be comparable to the computational resources involved. Of course, we will get and evaluate the results of these algorithms. But first things first.

Let's briefly recap the algorithm:

1. The Multi-Head Self-Attention block received, as input, a tensor of initial data where each element of the sequence is represented by a token (a vector of values).

One sequence for all heads (threads). The actions in steps 2-5 are identical for each attention head.

2. For each token, three vectors (*Query*, *Key*, *Value*) are calculated by multiplying the token vector by the corresponding trainable matrix of weights W .
3. By multiplying the *Query* and *Key* vectors, we determine the pairwise dependencies between the elements of the sequence. At this step, the *Query* vector of each element of the sequence is multiplied by the *Key* vectors of the current and all previous elements of the sequence.
4. The matrix of the obtained dependence coefficients is normalized using the *Softmax* function in the context of each query (*Query*). In this case, a zero attention coefficient is set for subsequent elements of the sequence.
5. As a result of steps 3 and 4, we get a square *Score* matrix with a dimension equal to the number of elements in the sequence, where the sum of all elements in the context of each *Query* is equal to one.
6. Then we multiply the normalized attention coefficients by the *Value* vectors of the corresponding elements of the sequence, add the resulting vectors, and get the attention-adjusted value for each element of the sequence.
7. Next, we determine the weighted attention result. To do this, we multiply the concatenated tensor of the results of all attention heads by the trained matrix W_0 .
8. The resulting tensor is added to the input sequence and normalized.
9. The *Multi-Heads Self-Attention* mechanism is followed by two fully connected layers of the *Feed Forward* block. The first (hidden) layer contains four times as many neurons as the input sequence with the *ReLU* activation function (we used the *Swish* function instead). The dimension of the second layer is equal to the dimension of the input sequence, and neurons do not use the activation function.
10. The result of the fully connected layers is summed up with the tensor input to the *Feed Forward* block and the resulting tensor is normalized.

Now that we have refreshed the basic steps of the process, let's proceed with the implementation. To implement the new type of neural layer, let's create a new class *CNeuronGPT*, inheriting from the *CNeuronBase* neural layer base class of our model. Despite using the *Self-Attention* algorithm in the

model, I chose not to inherit from our existing classes of neural layers using attention mechanisms. This is due to some peculiarities in the model implementation, which we will become familiar with during the process.

Perhaps one of the main differences is the ability to build multiple homogeneous layers within one class. Previously we used separate layers to implement parts of the model functionality, while now we are talking about the full-fledged creation of several copies of the layer being created, each with its own weights. To achieve this, in the body of the method, we declare not individual neural layers but entire collections of layers. Among them, you will see familiar variable names from working with previous classes, but they will now contain pointers to collections of neural layers. At the same time, we have preserved the functionality hidden behind the object names. Additionally, we have added two new variables:

- *m_iLayers* – number of neural layers in the block
- *m_iCurrentPosition* – number of the current element in the sequence

```
class CNeuronGPT      : public CNeuronBase
{
protected:
    CArrayLayers      m_cQuerys;
    CArrayLayers      m_cKeys;
    CArrayLayers      m_cValues;
    CArrayLayers      m_cScores;
    CArrayLayers      m_cAttentionOut;
    CArrayLayers      m_cW0;
    CArrayLayers      m_cFF1;
    CArrayLayers      m_cFF2;
    //---
    int               m_iLayers;
    int               m_iWindow;
    int               m_iUnits;
    int               m_iKeysSize;
    int               m_iHeads;
    CBufferType       m_dStd[];
    int               m_iCurrentPosition;
    int               m_iScoreTemp;

    virtual bool      NormlizeBuffer(CBufferType *buffer, CBufferType *std,
                                    uint std_shift);

    virtual bool      NormlizeBufferGradient(CBufferType *output,
                                    CBufferType *gradient, CBufferType *std, uint std_shift);
public:
    CNeuronGPT(void);
    ~CNeuronGPT(void);

    //---
    virtual bool      Init(const CLayerDescription *desc) override;
    virtual bool      SetOpenCL(CMyOpenCL *openc1) override;
    virtual bool      FeedForward(CNeuronBase *prevLayer) override;
    virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
    virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer, bool read) override;
    virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                    VECTOR &Beta, VECTOR &Lambda) override;
```

```

//---
virtual int      GetUnits(void) const { return m_iUnits; }
virtual int      GetLayers(void) const { return m_iLayers; }
//--- methods for operations with files
virtual bool     Save(const int file_handle) override;
virtual bool     Load(const int file_handle) override;
//--- object identification methods
virtual int      Type(void) override const { return(defNeuronGPT); }
};

```

The addition of the *m_iCurrentPosition* variable is the second architectural feature of this model. We have already said that *GPT* refers to autoregressive models. At each step, it returns one element of the sequence and feeds it as input at a new iteration. We mentioned something similar about recurrent models. However, in recurrent models, the hidden state was added to the current state of the environment, while in the case of *GPT*, generating the language model involves creating a new state. Of course, concerning financial markets, we deviate slightly from this feedback and input the actual new state, but we will preserve the signal processing principles.

The logic is as follows: if only one element of the sequence is updated at each new iteration, there is no need to recalculate the same values every time. It's not efficient. Let's recalculate only the new element of the sequence, and for the previous elements of the sequence, let's use the values from previous iterations. This is why we introduce the *m_iCurrentPosition* variable to store the index of the current element in the sequence. We will get acquainted with its usage principles as we proceed with the implementation.

Let's take things step by step. As usual, we will start working on the methods of the class with the class constructor. In it, we initialize variables with initial values. Similar to the attention mechanism classes discussed earlier, we use static objects that do not instantiate in the class constructor. The class destructor remains empty.

```

CNeuronGPT::CNeuronGPT(void) :    m_iHeads(8),
                                   m_iWindow(0),
                                   m_iKeysSize(0),
                                   m_iUnits(0),
                                   m_iLayers(0),
                                   m_iCurrentPosition(0)
{
}

```

Following our previously used pattern of working with classes, next, we will construct the initialization method of the class. This method is inherited from the parent class *CNeuronBase* and is overridden in each new class.

In the parameters, the method receives a pointer to an object describing the created neural layer, and we immediately perform a validity check on the received pointer, as well as verify the presence of the specified minimum necessary parameters for the correct initialization of the class instance.

```

bool CNeuronGPT::Init(const CLayerDescription *desc)
{
//--- checking the initial data
if(!desc || desc.type != Type() || desc.count <= 0 || desc.window <= 0 ||
    desc.window_out <= 0 || desc.step <= 0 || desc.layers <= 0)
    return false;

```

After successfully passing the control block, we save the received parameters to the appropriate variables of our class.

```
//--- save the constants
m_iWindow    = desc.window;
m_iUnits     = desc.count;
m_iKeysSize  = desc.window_out;
m_iHeads     = desc.step;
m_iLayers    = desc.layers;
if(!ArrayResize(m_dStd, m_iLayers))
    return false;
for(int l = 0; l < m_iLayers; l++)
    if(!m_dStd[l].BufferInit(1, 2, 1))
        return false;
```

Then, similar to the previously created classes using the attention mechanism, we will slightly adjust the description of the created neural layer and call the initialization method of the parent class. I would like to remind you that in the description of the created neural layer, we set the window size parameter of the input data to zero before calling the method of the parent class. This allows us to remove unused buffer objects from the parent class.

```
//--- call the initialization method of the parent class
CLayerDescription *temp = new CLayerDescription();
if(!temp || !temp.Copy(desc))
    return false;
temp.window_out = 1;
temp.window     = 0;
temp.activation = AF_NONE;
if(!CNeuronBase::Init(desc))
    return false;
delete temp;
```

After that, we create a loop with the number of iterations equal to the number of homogeneous neural layers created. All other objects will be created in the body of this loop.

```
//--- run a loop to create objects of internal layers
for(int layer = 0; layer < m_iLayers; layer++)
{
```

The operations in the loop body are very similar to the operations performed in the class initialization methods using the *Self-Attention* mechanism, but there are still differences.

Firstly, within the loop body, we create an instance of the *CLayerDescription* object to describe the neural layers being created and fill it with the necessary data. Since we have decided to input only the state update to the neural network, rather than the entire pattern information, I chose to forgo using convolutional neural layers and opted for a basic fully connected neural layer. Therefore, in the *type* field of the layer description object, we set the constant *defNeuronBase*. In this case, the window size of the input data will be equal to the size of the vector describing one element of the sequence. In this case, the entire volume of input data is perceived as the description of one element of the sequence.

Next, we recall that the model uses the *Multi-Head Self-Attention* mechanism, so we need to create three vectors (*Query*, *Key*, *Value*) for each attention head from one vector of the initial data. I would like to remind you of another detail: when implementing the *Multi-Head Self-Attention* mechanism, we used concatenated vectors. Now we are going further: we will not only create a single tensor for all attention

heads but we also combine all three entities mentioned above at once (*Query*, *Key*, *Value*). However, since it will contain only one element of the sequence, its size will not be so large. In the *count* field specify a size equal to the three vectors of one element of the key tensor sequence for each attention head. The newly created layer will not have an activation function, just like before. We will use the parameter optimization method specified by the user in the neural layer description from the method parameters.

```
temp = new CLayerDescription();
if(!temp)
    return false;
temp.type = defNeuronBase;
temp.window = m_iWindow;
temp.count = (int)(3 * m_iKeysSize * m_iHeads);
temp.activation = AF_NONE;
temp.optimization = desc.optimization;
```

After creating the neural layer description object and specifying all the necessary parameters, we create the first internal neural layer *Query*s. We initialize it using a pre-created neural layer description object. It is essential to monitor the process of performing operations. After successfully completing the first two operations, we add the layer to the corresponding collection.

```
//--- initialize Querys
CNeuronBase *Querys = new CNeuronBase();
if(!Querys)
{
    delete temp;
    return false;
}
if(!Querys.Init(temp))
{
    delete Querys;
    delete temp;
    return false;
}
if(!m_cQuerys.Add(Querys))
{
    delete Querys;
    delete temp;
    return false;
}
```

Despite creating a concatenated tensor, we have kept the name *Query*s for the neural layer, maintaining continuity with the previously created attention mechanism classes. However, we will also create internal neural layers for *Keys* and *Values*, although with different parameters.

We will use the internal neural layers *Keys* and *Values* to accumulate historical data on the received current states. It is, so to speak, the memory of our neural layer, and it should be sufficient to store the entire pattern being analyzed. However, since we have already calculated the state of these vectors in the fully connected neural layer *Query*s, we do not need matrices of weights in them. Therefore, before initializing the mentioned internal neural layers, we will make a change to the description object of the neural layer: we will set the size of the input data window to zero and ensure that the neural layer has enough elements to store the entire pattern description tensor.

```

//--- initialize Keys
CNeuronBase *Keys = new CNeuronBase();
if(!Keys)
{
    delete temp;
    return false;
}
temp.window = 0;
temp.count = (int)(m_iUnits * m_iKeysSize * m_iHeads);
if(!Keys.Init(temp))
{
    delete Keys;
    delete temp;
    return false;
}
if(!Keys.GetOutputs().Reshape(m_iUnits, m_iKeysSize * m_iHeads))
    return false;
if(!m_cKeys.Add(Keys))
{
    delete Keys;
    delete temp;
    return false;
}

```

The rest of the algorithm for creating an internal neural layer is similar to creating the *Query*s layer:

- Create a new instance of the neural layer object.
- Initialize the neural layer.
- Add the neural layer to the corresponding collection.

```

//--- initialize Values
CNeuronBase *Values = new CNeuronBase();
if(!Values)
{
    delete temp;
    return false;
}
if(!Values.Init(temp))
{
    delete Values;
    delete temp;
    return false;
}
if(!Values.GetOutputs().Reshape(m_iUnits, m_iKeysSize * m_iHeads))
    return false;
if(!m_cValues.Add(Values))
{
    delete Values;
    delete temp;
    return false;
}

```

After creating the neural layers *Query*, *Keys*, and *Values*, we proceed to create the dependency coefficient matrix *Score*. There are implementation nuances here as well. This matrix in the *Self-Attention* implementation algorithm has a square size with each side of the square equal to the number of elements of the sequence. Each element of the matrix represents the coefficient of the pairwise relationship between the elements of the sequence, where the rows of the matrix correspond to the vectors of the tensor of the *Query* queries and the columns of the matrix correspond to the vectors of the *Key* tensor.

Now, let's think about how we can implement such a matrix if we have one *Query* vector that describes only the last state. Therefore, the *Score* matrix in this case degenerates into a vector. Of course, for each attention head. Certainly, the neural layer of the *Score* dependency coefficient vector does not contain a matrix of weights. Therefore, we adjust the number of elements in the neural layer and create a new internal neural layer using the algorithm mentioned above. Let's take advantage of the opportunity and make the matrix rectangular. The rows of the matrix will correspond to the attention heads.

```
//--- initialize Scores
CNeuronBase *Scores = new CNeuronBase();
if(!Scores)
{
    delete temp;
    return false;
}
temp.count = (int)(m_iUnits * m_iHeads);
if(!Scores.Init(temp))
{
    delete Scores;
    delete temp;
    return false;
}
if(!Scores.GetOutputs().Reshape(m_iHeads, m_iUnits))
    return false;
if(!m_cScores.Add(Scores))
{
    delete Scores;
    delete temp;
    return false;
}
```

The next object we will create is a neural layer for the concatenated output of the *AttentionOut* attention heads. Here, the situation is similar to the dependency coefficient matrix. We have already discussed the reasons for the degeneration of the matrix of dependence coefficients into a vector, and to obtain the result of the work of the attention head according to the *Self-Attention* algorithm, we need to multiply the matrix of dependence coefficients by the *Value* tensor.

$$Output = Softmax\left(\frac{Query * Key^T}{\sqrt{Dimension_{Key}}}\right)Value$$

But in our case, with one *Query* vector at the output, we also get one vector for each attention head. Therefore, we will specify the correct layer size and execute the algorithm for its initialization.

```

//--- initialize AttentionOut
CNeuronBase *AttentionOut = new CNeuronBase();
if(!AttentionOut)
{
    delete temp;
    return false;
}
temp.count = (int)(m_iKeysSize * m_iHeads);
if(!AttentionOut.Init(temp))
{
    delete AttentionOut;
    delete temp;
    return false;
}
if(!AttentionOut.GetOutputs().Reshape(m_iHeads, m_iKeysSize))
    return false;
if(!m_cAttentionOut.Add(AttentionOut))
{
    delete AttentionOut;
    delete temp;
    return false;
}

```

Following the multi-head attention algorithm, our next step will be to organize the results of all attention heads into a unified vector and adjust its size to match the size of the input data vector. In the algorithm of the *Multi-Head Self-Attention* mechanism, this operation is performed using the W_0 matrix. However, we will perform this operation using a basic fully connected neural layer without an activation function.

Again, we will create a new instance of the neural layer object. Do not forget to check the result of the operation.

```

//--- initialize W0
CNeuronBase *W0 = new CNeuronBase();
if(!W0)
{
    delete temp;
    return false;
}

```

In the neural layer description object, we enter the necessary parameters:

- The size of the input data window is equal to the size of the previously created layer for the concatenated results of attention heads.
- The number of elements at the output of the neural layer is equal to the size of the source data vector.
- The activation function is not used.

We initialize the neural layer using the neural layer description object.

```

temp.window = temp.count;
temp.count = m_iWindow;
temp.activation = AF_NONE;

```

```

if(!W0.Init(temp))
{
    delete W0;
    delete temp;
    return false;
}
if(!m_cW0.Add(W0))
{
    delete W0;
    delete temp;
    return false;
}

```

After the successful initialization of the neural layer object, we add it to the appropriate collection.

This concludes the work on initializing the objects of the *Multi-Head Self-Attention* mechanism, and we just have to create two neural layers of the *Feed Forward* block. The first neural layer has four times as many neurons in its output as the tensor received as input, and it is activated using the *Swish* function.

```

//--- initialize FF1
CNeuronBase *FF1 = new CNeuronBase();
if(!FF1)
{
    delete temp;
    return false;
}
temp.window = m_iWindow;
temp.count = temp.window * 4;
temp.activation = AF_SWISH;
temp.activation_params[0] = 1;
temp.activation_params[1] = 0;
if(!FF1.Init(temp))
{
    delete FF1;
    delete temp;
    return false;
}
if(!m_cFF1.Add(FF1))
{
    delete FF1;
    delete temp;
    return false;
}

```

The second neural layer of the *Feed Forward* block does not have the activation function. It returns the size of the tensor to the size of the initial data. Here we also use a basic fully connected neural layer. We will make the necessary adjustments to the description object of the neural layer and initialize the neural layer.

```

//--- initialize FF2
CNeuronBase *FF2 = new CNeuronBase();
if(!FF2)
{

```

```

        delete temp;
        return false;
    }
    temp.window = temp.count;
    temp.count = m_iWindow;
    temp.activation = AF_NONE;
    if(!FF2.Init(temp))
    {
        delete FF2;
        delete temp;
        return false;
    }
    if(!m_cFF2.Add(FF2))
    {
        delete FF2;
        delete temp;
        return false;
    }
    delete temp;
}

```

We check the results of the operations at each step and add the created neural layer to the appropriate collection.

At this stage, we have created all the objects necessary for the operation of a single neural layer. We remove the description object of the neural layer and proceed to the next iteration of our loop, where we will create objects for the operation of the next layer.

Thus, upon completing all iterations of the loop, we will obtain objects for the operation of as many neural layers as the user specified when calling the initialization method of this neural layer.

Furthermore, to avoid copying data between the buffers of internal neural layers and the current layer, we will replace the pointers to the result and gradient buffers of the current layer.

```

//--- to avoid copying buffers, we will replace them
if(m_cFF2.Total() < m_iLayers)
    return false;
if(!m_cOutputs)
    delete m_cOutputs;
CNeuronBase *neuron = m_cFF2.At(m_iLayers - 1);
if(!neuron)
    return false;
m_cOutputs = neuron.GetOutputs();
if(!m_cGradients)
    delete m_cGradients;
m_cGradients = neuron.GetGradients();

```

In conclusion, we call the method for distributing pointers to the OpenCL context among the class object and exit the initialization method.

```

    SetOpenCL(m_cOpenCL);
//---
    return true;
}

```

To fully address the issue of class initialization, I suggest considering a method for distributing the OpenCL context object pointer among the internal layer objects.

Despite the change in the type of internal objects, from a neural layer to a collection of neural layers, the structure and algorithm of the pointer propagation method to the OpenCL context have not changed much. This became possible thanks to the similar method we previously wrote in the neural layer collection class.

In the parameters, our *SetOpenCL* method gets a pointer to the OpenCL context object. In the body of the method, we first call the relevant method of the parent class, where all the necessary controls are already implemented, and the pointer is saved in the corresponding class variable. After that, we alternately check the pointers of all the internal objects of the neural layer and call a similar method for them.

```
bool CNeuronGPT::SetOpenCL(CMyOpenCL *openc1)
{
    CNeuronBase::SetOpenCL(openc1);
    m_cQuerys.SetOpenc1(m_cOpenCL);
    m_cKeys.SetOpenc1(m_cOpenCL);
    m_cValues.SetOpenc1(m_cOpenCL);
    m_cScores.SetOpenc1(m_cOpenCL);
    m_cAttentionOut.SetOpenc1(m_cOpenCL);
    m_cW0.SetOpenc1(m_cOpenCL);
    m_cFF1.SetOpenc1(m_cOpenCL);
    m_cFF2.SetOpenc1(m_cOpenCL);
    if(m_cOpenCL)
    {
        uint size = sizeof(TYPE) * m_iUnits * m_iHeads;
        m_iScoreTemp = m_cOpenCL.AddBuffer(size, CL_MEM_READ_WRITE);
        for(int l = 0; l < m_iLayers; l++)
            m_dStd[l].BufferCreate(m_cOpenCL);
    }
    else
    {
        for(int l = 0; l < m_iLayers; l++)
            m_dStd[l].BufferFree();
    }
    //---
    return(!m_cOpenCL);
}
```

Thus, we conclude the class initialization and proceed directly to implementing the neural layer operational algorithm. As always, we will start with the implementation of the feed-forward method.

5.3.2.1 GPT feed-forward method

We continue our work on implementing the *GPT* algorithm proposed by the *OpenAI* team. We have already created the basic skeleton of the class with objects to implement the algorithm. Now we are proceeding directly to its implementation. Yes, the class will utilize the familiar *Self-Attention* algorithm, but with some implementation specifics.

As in all the previously discussed classes, all the feed-forward functionality is implemented in the *CNeuronGPT::FeedForward* method. As you know, this method is virtual, is inherited from the base neural network class, and is overridden in each class to implement a specific algorithm. In the method parameters, it receives a pointer to the object of the previous neural layer, which contains the initial data in its buffer for executing the algorithm.

As in all previous implementations, we start the method with the control block. In this block, we check the validity of pointers to the objects involved in the method. This operation allows us to prevent many critical errors when accessing invalid objects.

```
bool CNeuronGPT::FeedForward(CNeuronBase *prevLayer)
{
    //--- check the relevance of all objects
    if(!prevLayer || !prevLayer.GetOutputs())
        return false;
```

Next, we increment the *m_iCurrentPosition* index of the current object in the *Key* and *Value* buffers. We need this pointer for organizing the stack in these buffers. In fact, the *Self-Attention* algorithm performs a weighted summation of different contexts into a single vector. According to the mathematical rules, rearranging the places of the summands does not change the sum. That is, it is absolutely irrelevant at which position of the data buffer the element is located. What's important is its presence. This is the disadvantage of this algorithm for handling timeseries, but also a plus for our implementation. When organizing the data stack in the *Key* and *Value* buffers, we will not perform a costly full data shift. Instead, we will move the pointer along the stack and overwrite the data in the corresponding data buffer elements.

```
//--- increment the pointer to the current object in the data stack
m_iCurrentPosition++;
if(m_iCurrentPosition >= m_iUnits)
    m_iCurrentPosition = 0;
```

The next straightforward step is taken to organize the correct functioning of our internal multi-layered architecture. The pointer to the previous neuron layer obtained in the parameters is needed only for the first internal layer. Further internal neural layers will use the output from the preceding internal neural layer as their input data. Therefore, for internal use, we introduce a local variable to store a pointer to the previous neural layer. Now we will assign it the pointer obtained from the method parameters, but after the iterations of each internal neural layer, we will write a new pointer into it. So, we can organize the loop operation through all internal neural layers. In this case, we will work with one object pointer variable in the loop body. In reality, each neural layer will access a buffer of its own input data.

```
CNeuronBase *prevL = prevLayer;
```

As mentioned before, the main functionality of our feed-forward method will be implemented within the body of the loop iterating through the internal neural layers. Therefore, the next step is to create such a loop. Right within the loop, we extract from the collection the pointer to the *Querys* object

corresponding to the current internal neural layer. We check the validity of the extracted pointer and then execute the feed-forward method of the corresponding object.

```
//--- run the loop through all internal layers
for(int layer = 0; layer < m_iLayers; layer++)
{
    CNeuronBase *Queryys = m_cQueryys.At(layer);
    if(!Queryys || !Queryys.FeedForward(prevL))
        return false;
```

Further functionality is not covered by the methods of internal objects. Therefore, as in previous *Self-Attention* implementations, we will implement it within the body of the method. Here, it is important to remember that in all implementations of our library, we provided the user with the option to choose the device and the technology for performing mathematical operations. In this class, we will not deviate from our principles and will also implement algorithm separation based on the chosen computational device. But first, let's perform some preparatory work and extract pointers to the objects of the analyzed internal layer from the collections. Do not forget to validate the obtained pointers.

```
CNeuronBase *Queryys = m_cQueryys.At(layer);
if(!Queryys || !Queryys.FeedForward(prevL))
    return false;
CNeuronBase *Keys = m_cKeys.At(layer);
if(!Keys)
    return false;
CNeuronBase *Values = m_cValues.At(layer);
if(!Values)
    return false;
//--- initializing Scores
CNeuronBase *Scores = m_cScores.At(layer);
if(!Scores)
    return false;
//--- initializing AttentionOut
CNeuronBase *AttentionOut = m_cAttentionOut.At(layer);
if(!AttentionOut)
    return false;
```

Next, we split the algorithm based on the chosen computational device. In this chapter, we will discuss the organization of the process using standard MQL5 tools, and we will revisit the implementation of multi-threaded computations using the OpenCL technology in the following sections.

```
//--- branching of the algorithm by the computing device
if(!m_cOpenCL)
{
    MATRIX array[];
    if(!Queryys.GetOutputs().m_mMatrix.Vsplit(3, array))
        return false;
    if(!Keys.GetOutputs().Row(array[1].Row(0), m_iCurrentPosition))
        return false;
    if(!Values.GetOutputs().Row(array[2].Row(0), m_iCurrentPosition))
        return false;
```

As you may recall, during the execution of the feed-forward pass of the specified object, we simultaneously construct all the vectors for the *Query*, *Key*, and *Value* tensors for all attention heads. In

the next step, we move the vectors of the last two tensors to the corresponding stacks. For this purpose, we will divide the result buffer of the *Query*s layer into 3 equal parts: query, key, and value. First, we copy the data into the appropriate data buffers. When copying data, we will use the *m_iCurrentPosition* variable to determine the offset in the buffers.

Then we will do a bit of preparatory work. To facilitate access to the elements of the objects, we will create local pointers to the result buffers of the internal neural layers for *Query* and *Key*. We will also prepare dynamic arrays to perform the computational part.

```
MATRIX out;
if(!out.Init(m_iHeads, m_iKeysSize))
    return false;
MATRIX array_keys[], array_values[];
MATRIX array_querys[];
MATRIX keys = Keys.GetOutputs().m_mMatrix;
MATRIX values = Values.GetOutputs().m_mMatrix;
```

Similarly to the construction of the feed-forward algorithm in the previously discussed implementation of multi-head attention, we will split the data matrices according to the attention heads.

```
if(!array[0].Vsplitt(m_iHeads, array_querys))
    return false;
if(!keys.Reshape(m_iUnits, m_iHeads * m_iKeysSize))
    return false;
if(!keys.Vsplitt(m_iHeads, array_keys))
    return false;
if(!values.Reshape(m_iUnits, m_iHeads * m_iKeysSize))
    return false;
if(!values.Vsplitt(m_iHeads, array_values))
    return false;
```

After that, we create a nested loop for computations. In it, we iterate through the attention heads used. Right here in the body, we extract the *Query* vector and the *Keys* matrix of the analyzed attention head. We multiply them and divide the resulting vector by the square root of the dimension of the description vector for one element in the *Keys* matrix. We normalize it using the *Softmax* function.

```
//--- define Scores
for(int head = 0; head < m_iHeads; head++)
{
    MATRIX score=array_querys[head].MatMul(array_keys[head].Transpose())/
                                                sqrt(m_iKeysSize);

    //--- normalize Scores
    if(!score.Activation(score,AF_SOFTMAX))
        return false;
    if(!Scores.GetOutputs().Row(score.Row(0), head))
        return false;
```

Thus, after normalizing the data, the sum of all dependency coefficients will be equal to one. This gives us reason to expect a vector with appropriate characteristics at the output of the *Self-Attention* block. We save the normalized data in a buffer for later use during the backpropagation pass.

After calculating and normalizing the dependency coefficient vector, we have all the necessary data to calculate the output values of the *Self-Attention* block. We multiply the normalized *Score* vector by the *Value* tensor. Then we copy the resulting vector into the local result matrix.

```
//--- attention block output
MATRIX o = score.MatMul(array_values[head]);
if(!out.Row(o.Row(0), head))
    return false;
}
```

As a result of performing all iterations of the loop system in our *out* matrix, the concatenated output of the *Multi-Heads Self-Attention* block will be collected. We transfer them to the result buffer of the *AttentionOut* neural layer to use in our algorithm later.

```
if(!out.Reshape(1, m_iHeads * m_iKeysSize))
    return false;
AttentionOut.GetOutputs().m_mMatrix = out;
}
else // OpenCL block
{
    return false;
}
```

This completes the operation separation block depending on the computing device. Next, we will use the methods of our internal objects.

According to the *Multi-Heads Self-Attention* algorithm, the next step is to create a single ordered weighted vector of results for the entire multi-head attention block from the concatenated output of all attention heads. For this purpose, the matrix W_o is provided in the method algorithm. In contrast, we have assigned this functionality to the internal fully connected neural layer WO . We extract the pointer to the object of the corresponding neural layer and call its feed-forward method. To prevent critical errors, we must validate the pointer to the object before calling its method.

```
//--- weighted output of all heads of attention
CNeuronBase *W0 = m_cW0.At(layer);
if(!W0 || !W0.FeedForward(AttentionOut))
    return false;
```

We are nearing the completion of the implementation of the *Multi-Heads Self-Attention* block algorithm. According to the *GPT* model algorithm, we need to add the obtained result to the original data and normalize the result using the formulas.

$$\bar{a} = \frac{1}{h} \sum_{i=1}^h a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h} \sum_{i=1}^h (a - \bar{a})^2}}$$

First, we call the *CBufferType::SumArray* method of summarizing two buffers. Then we normalize the data using the *CNeuronGPT::NormalizeBuffer* method. Its algorithm completely repeats the relevant method of the *CNeuronAttention* class.

```
//--- add to the input data and normalize
if(!W0.GetOutputs().SumArray(prevL.GetOutputs()))
    return false;
```

```

    if(!NormalizeBuffer(W0.GetOutputs(), GetPointer(m_dStd[layer]), 0))
        return false;

```

After successfully normalizing all the data, we will pass the signal through two internal neural layers of the *Feed Forward* block. This operation is straightforward: we sequentially extract pointers to the respective neural layer objects, validate the pointers, and call the feed-forward method for each internal layer.

```

//--- forward pass of Feed Forward block
CNeuronBase *FF1 = m_cFF1.At(layer);
if(!FF1 || !FF1.FeedForward(W0))
    return false;
CNeuronBase *FF2 = m_cFF2.At(layer);
if(!FF2 || !FF2.FeedForward(FF1))
    return false;

```

Finally, we add the result of the *Feed Forward* block to the result of the *Multi-Heads Self-Attention* block. Then we normalize the obtained values.

```

//--- perform summation with the attention output and normalizing
CBufferType *prev = FF2.GetOutputs();
if(!prev.SumArray(W0.GetOutputs()))
    return false;
if(!NormalizeBuffer(prev, GetPointer(m_dStd[layer]), 1))
    return false;

```

This completes the feed-forward pass for one internal layer. We can proceed to the next iteration of the loop and the next internal neural layer. But first, we need to change the pointer to the neural layer of the initial data, as we discussed at the beginning of the method. The results of the forward pass are contained in the buffer of the internal neural layer *FF2*. We write the pointer to it into the local variable *prevL*, with which we work at the next iteration of the loop.

```

    prevL = FF2;
}
//---
return true;
}

```

So, upon completing all iterations of the nested neural layer enumeration loop, we obtain a complete recalculation of the feed-forward pass for our block. To change the number of such neural layers, we only need to modify one parameter when calling the initialization method of the *CNeuronGPT* class in the *GPT* model.

With this, we conclude the work on the feed-forward pass method and move on to organizing the backpropagation process.

5.3.2.2 GPT backpropagation methods

In the previous sections, we looked at the architecture of the *GPT* model and even implemented methods to initialize our new class and implement the feed-forward pass through the model algorithm. Now let's look at a possible implementation of the backpropagation pass for this algorithm.

To implement the backpropagation pass in each new class we override three methods:

- *CalcHiddenGradient* – method for calculating the error gradient through the hidden layer
- *CalcDeltaWeights* – method for calculating the error gradient to the level of the weight matrix
- *UpdateWeights* – method for updating weights

This class will not be an exception, and we will redefine all three methods. Let's start with the first backpropagation and, probably, the most complex method: error gradient propagation through the hidden layer. It is in this method that we have to repeat the entire feed-forward algorithm in reverse order.

In the parameters, the method receives a pointer to the object of the previous layer, to which we have to pass the error gradient. Again, in the body of the method, we implement a block of checks. In it, according to the already established good tradition, we check the validity of pointers to all objects used in the method. This approach helps eliminate many critical errors during the execution of the method code.

```
bool CNeuronGPT::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    //--- check the relevance of all objects
    if(!m_cOutputs || !m_cGradients ||
        m_cOutputs.Total() != m_cGradients.Total())
        return false;
```

Next, by analogy with the forward pass method, we organize a loop for searching through the internal neural layers. But this time, in accordance with the principles of backward pass, we also organize the cycle with a countdown of iterations. All further iterations will be performed in the body of the loop and repeated for all nested layers of our model.

```
//--- run a loop through all internal layers in reverse order
for(int layer = m_iLayers - 1; layer >= 0; layer--)
{
    CNeuronBase *FF2 = m_cFF2.At(layer);
    if(!FF2)
        return false;
    CBufferType *Gradients = FF2.GetGradients();
    //--- scale the gradient for normalization
    if(!NormlizeBufferGradient(FF2.GetOutputs(), Gradients,
                                GetPointer(m_dStd[layer]), 1))
        return false;
```

In the body of the loop, we first retrieve a pointer to the corresponding neural layer of the output of the *Feed Forward* FF2 block and adjust its error gradient buffer to the derivative of the normalization function. We discussed the reasons for this operation in detail when constructing a similar method for the *Self-Attention* algorithm.

After this, we sequentially call the error gradient distribution methods for the internal layers of the *Feed Forward* block. We also call the methods in the reverse order: first for the second layer, and then for the first one.

```
//--- propagate a gradient through the Feed Forward block
CNeuronBase *FF1 = m_cFF1.At(layer);
if(!FF2.CalcHiddenGradient(FF1))
    return false;
CNeuronBase *W0 = m_cW0.At(layer);
if(!FF1.CalcHiddenGradient(W0))
    return false;
```

During the feed-forward pass, we added up the results of the *Multi-Heads Self-Attention* and *Feed Forward* blocks. Also, now we need to draw an error gradient in two directions. We add the error gradients at the output level of the specified blocks. Then we adjust the total tensor by the derivative of the layer normalization function.

```
CBufferType *attention_grad = W0.GetGradients();
if(!attention_grad.SumArray(Gradients))
    return false;
//--- scale the gradient for normalization
if(!NormlizeBufferGradient(W0.GetOutputs(), attention_grad,
                           GetPointer(m_dStd[layer]), 0))

    return false;
```

Next, we distribute the error gradient across the attention heads by calling the error gradient distribution method of the internal neural layer W_O .

```
//--- initialize Scores
CNeuronBase *Scores = m_cScores.At(layer);
if(!Scores)
    return false;
//--- distribute the error gradient across the heads of attention
CNeuronBase *AttentionOut = m_cAttentionOut.At(layer);
if(!W0.CalcHiddenGradient(AttentionOut))
    return false;
```

Until now, everything was simple and transparent. We simply called the corresponding methods of our internal neural layers in reverse order. But then comes the algorithm block that is not covered by the methods of internal neural layers. It was implemented inside the feed-forward method. Therefore, we also have to completely recreate the error gradient backpropagation functionality.

First, let's do the preparatory work and create local pointers to *Querys*, *Keys*, and *Values* objects. At this point, don't forget to check the validity of the received object pointers.

```
//--- get pointers to Querys, Keys, Values objects
CNeuronBase *Querys = m_cQuerys.At(layer);
if(!Querys)
    return false;
CNeuronBase *Keys = m_cKeys.At(layer);
if(!Keys)
    return false;
CNeuronBase *Values = m_cValues.At(layer);
if(!Values)
```

```
return false;
```

Next, we need to create two options for implementing the algorithm: using standard MQL5 tools and in multi-threaded operations mode using OpenCL technology. We create a branching of the algorithm depending on the selected device for performing mathematical operations. As usual, in this section, we will look at the implementation of the algorithm using standard MQL5 tools and will return to the implementation of the multi-threaded operations block in other sections.

To organize calculations using standard MQL5 tools, we prepare dynamic arrays. In one array, we load error gradient data from the buffer. Some arrays are filled with the results of the feed-forward pass and others are initialized with zero values for subsequent gradient error accumulation operations.

```
//--- branching of the algorithm across the computing device
attention_grad = AttentionOut.GetGradients();
if(!m_cOpenCL)
{
    MATRIX gradients[];
    if(!attention_grad.m_mMatrix.Vsplit(m_iHeads, gradients))
        return false;
    if(!Querys.GetGradients().m_mMatrix.Reshape(3, m_iHeads * m_iKeysSize))
        return false;
    MATRIX values[];
    if(!Values.GetOutputs().m_mMatrix.Vsplit(m_iHeads, values))
        return false;
    MATRIX keys[];
    if(!Keys.GetOutputs().m_mMatrix.Vsplit(m_iHeads, keys))
        return false;
    MATRIX querys[];
    MATRIX query = Querys.GetOutputs().m_mMatrix;
    if(!query.Reshape(3, m_iHeads * m_iKeysSize) ||
        !query.Resize(1, query.Cols()))
        return false;
    if(!query.Vsplit(m_iHeads, querys))
        return false;
    MATRIX querys_grad = MATRIX::Zeros(m_iHeads, m_iKeysSize);
    MATRIX keys_grad = querys_grad;
    MATRIX values_grad = querys_grad;
```

First, we will distribute the error gradient to the *Value* tensor. It's important to note that we'll be distributing the error gradient not across the entire tensor but only for the current element. This is reasonable when we consider the purpose of error gradient distribution. We aim to optimize the model parameters throughout the training process, and distributing the error gradient helps us obtain guidelines for this optimization.

When distributing the error gradient to the *Value* tensor, we need to pass it in two directions: to the previous layer and to the weight matrix responsible for forming the current layer's tensor.

We can only transfer the error gradient for the current state to the previous layer. The buffer of the previous layer is unable to accept more because, during the feed-forward pass, it only provides the current state for which it expects the error gradient.

Also, only the current state error gradient can be propagated to the weight matrix. To distribute the error from previous states, we would need the input data from those previous states. However, the previous layer does not provide this information, and we did not save it in the buffers of our layer.

Therefore, distributing the gradient to the elements of the value tensor, except for the current state, is a dead-end task and does not make sense.

The general approach is as follows: during the feed-forward pass, we calculate only the current state and additionally retrieve from memory those already calculated in previous iterations. A similar situation applies during the backpropagation pass: it is assumed that the error gradient from previous states has already been considered in the backpropagation methods in previous iterations. This significantly reduces the number of operations for each iteration of the feed-forward and backpropagation passes.

I hope the logic is clear. Let's return to our backpropagation method. We paused at passing the error gradient to the *Value* tensor. To execute this iteration, we will first create a local pointer to the attention coefficient vector and then organize a loop.

Our loop will iterate through the active attention heads. Here, we immediately save the attention coefficient vector corresponding to the analyzed attention head in a local matrix. We multiply the gradient vector obtained from previous iterations by the attention coefficient for the current element of the sequence. The resulting values are saved in the error gradient matrix in the *Values* buffer.

```
for(int head = 0; head < m_iHeads; head++)
{
    MATRIX score = MATRIX::Zeros(1, m_iUnits);
    if(!score.Row(Scores.GetOutputs().m_mMatrix.Row(head), 0))
        return false;
    //--- distribution of the gradient on Values
    if(!values_grad.Row((gradients[head] *
                        score[0, m_iCurrentPosition]).Row(0), head))
        return false;
```

Next, we need to distribute the gradient in the second direction: through the matrix of dependency coefficients on the *Query* and *Key* tensors. But first, we need to propagate the gradient through the vector of dependence coefficients. We multiply the error gradient matrices at the output of the attention block and the *Values* matrix and obtain a gradient at the level of the vector of dependence coefficients.

So, we have a vector of error gradients for one attention head. But I would like to remind you that during the feed-forward pass, we normalized the vector of dependence coefficients with the *Softmax* function. Therefore, the obtained error gradients are valid for normalized data. To further distribute the error gradients, we need to adjust the error gradients to the derivative of the specified function.

A special feature of the *Softmax* function is the requirement for a complete set of tensor values to compute the value of each element. Similarly, to compute the derivative of one element, we need a complete set of values for the function results. In our case, the results of the function are the normalized vector of dependency coefficients, which we obtained during the forward pass. We have also already obtained the vector of error gradients. Thus, we have all the necessary initial data to perform the operations of finding the derivative of a function and adjusting the error gradient. The formula for the derivative of the *Softmax* function is as follows:

$$(\text{Softmax}(x_i))' = \begin{cases} i = j & y_i(1 - y_i) \\ i \neq j & -y_i y_j \end{cases}$$

The practical part of the error gradient adjustment operations is implemented using MQL5 matrix operations. After adjusting the error gradients, we divide the resulting vector by the square root of the dimension of the *Key* vector of one element of the sequence. We performed the same operation during the feed-forward pass to prevent uncontrolled growth of non-normalized dependency coefficients.

```
//--- gradient distribution to Querys and Keys
MATRIX score_grad = gradients[head].MatMul(values[head].Transpose());
//---
MATRIX ident = MATRIX::Identity(m_iUnits, m_iUnits);
MATRIX ones = MATRIX::Ones(m_iUnits, 1);
score = ones.MatMul(score);
score = score.Transpose() * (ident - score);
score_grad = score_grad.MatMul(score.Transpose()) /
                                                    sqrt(m_iKeysSize);

MATRIX temp = score_grad.MatMul(keys[head]);
if(!querys_grad.Row(temp.Row(0), head))
    return false;
temp = querys[head] * score_grad[0, m_iCurrentPosition];
if(!keys_grad.Row(temp.Row(0), head))
    return false;
}
```

As a result of these operations, we obtain the adjusted error gradient for one element of the dependency coefficient vector. But we will not save it to the next data buffer. Instead, we will immediately distribute it to the corresponding elements of the *Query* and *Key* tensors. To do this, we need to multiply this value by the vector of the opposite tensor. To determine the error gradient on the *Query* vector, we have a complete set of sequence elements in the *Key* tensor. However, in the *Query* tensor, we only have one sequence element. Therefore, the error gradient on the *Key* tensor will be propagated only for the current element of the sequence. We save the obtained error gradient values into the matrices we prepared earlier.

By obtaining error gradients at the levels of *Query* and *Keys* tensors, we complete the operations of the loop through attention heads.

As soon as the full loop of iterations is completed, our *querys_grad*, *keys_grad*, and *values_grad* matrices will contain the accumulated error gradients for the current sequence element across all attention heads. All we have to do is transfer its values to the error gradient buffer of our internal *Querys* layer.

```
if(!querys_grad.Reshape(1, m_iHeads * m_iKeysSize) ||
    !keys_grad.Reshape(1, m_iHeads * m_iKeysSize) ||
    !values_grad.Reshape(1, m_iHeads * m_iKeysSize))
    return false;
if(!Querys.GetGradients().Row(querys_grad.Row(0), 0) ||
    !Querys.GetGradients().Row(keys_grad.Row(0), 1) ||
    !Querys.GetGradients().Row(values_grad.Row(0), 2))
    return false;
if(!Querys.GetGradients().Reshape(1, Querys.GetGradients().Total()))
    return false;
```

```

    }
    else // OpenCL block
    {
        return false;
    }

```

This concludes the block for separating the operations of the algorithm depending on the device for performing the operations. Next, we will continue executing the algorithm using the methods of our internal neural layers.

Previously, we obtained a concatenated tensor of error gradients that includes data from all attention heads and from all three entities (*Query*, *Key*, *Value*). Now, using the method that propagates the gradient through the hidden layer of our internal neural layer *Query*s.CalcHiddenGradient, we can transfer the error gradient to the previous layer buffer. Before performing this operation, we need to decide in which object's buffer we will write the error gradients. We created this class as a multi-layer block, and all operations of the method are performed in a loop iterating through the active layer of our block. Therefore, to the object of the previous neural layer, whose pointer we received in the parameters of this method, we transfer data only from the first neural layer of our block. It will have index 0 in the collection of nested neural layers of our *GPT* block. All other nested neural layers must pass the error gradient to the internal neural layer buffer *FF2* of the previous nested neural layer. Let me remind you that *FF2* is the internal neural layer with the results of the *Feed Forward* block.

Therefore, we will create a local pointer to the object of the previous neural layer and assign it a pointer to the required object depending on the index of the active nested neural layer in our *GPT* block. Only after obtaining the correct pointer to the object of the correct previous layer, we transfer the error gradient to its buffer.

```

    //--- transfer the error gradient to the previous layer
    CNeuronBase *prevL = (layer == 0 ? prevLayer : m_cFF2.At(layer - 1));
    if(!Queryys.CalcHiddenGradient(prevL))
        return false;
    if(!prevL.GetGradients().SumArray(W0.GetGradients()))
        return false;
}
//---
return true;
}

```

Please note that when constructing similar methods in the implementation classes of attention mechanisms, at this point, we created a complete procedure for summing error gradients from four directions. Now, thanks to the use of the concatenated error gradient buffer, we obtain the total error gradient from three directions by executing the method of only one neural layer. We still have to add gradients, but only once. To the obtained error gradient, we will add the error gradient at the level of the outputs of the multi-head attention block. You remember that during the feed-forward pass, we also added the original data with the tensor of the multi-head attention block's outputs. Therefore, the error gradient must go through all the steps that the signal goes through during the feed-forward pass, but in reverse order.

This concludes the operations in the body of the loop iterating through the nested neural layers of our *GPT* block, as well as the overall operations of our method. We close the loop and exit the method.

And once again, I want to emphasize: do not forget to monitor every step of the operation execution. This helps minimize the risk of critical errors and makes the program operation more controlled and reliable.

We have discussed the organization of the error gradient propagation method to the previous layer. But this is only one of the three backpropagation methods that we must override for this class. Therefore, after propagating the error gradient to the previous neural layer, we need to propagate the error gradient to the internal weight matrices contained within the depths of a considerable number of internal objects of the neural layers. In accordance with the structure of our class methods, this functionality is performed in the *CalcDeltaWeights* method.

To propagate the error gradient to the weight matrix of any of the previously discussed neural layers, two things are necessary:

- The error gradient at the output level of a given neural layer to the activation function.
- The initial data provided by the previous neural layer.

To organize this process, we already have all the necessary data. In the previous method, we distributed the error gradient to each neural layer. We will get a pointer to the previous neural layer in the parameters of the *CNeuronGPT::CalcDeltaWeights* method.

As usual, in the body of the method, we organize a control block to check the pointers of all used internal objects. The control block should be minimal and sufficient. Eliminate redundant and explicitly repetitive controls, as they do not add value to the program operation and can slow it down. Moreover, each operation, including control, requires resources and time. Let's think about the objects for which we should update weight matrices. These include:

- The *Query* neural layer, which returns a concatenated tensor of three entities (Query, Key, Value).
- The W_o matrix neural layer.
- Two neural layers of the *Feed Forward* block.

All the mentioned objects are declared static. Therefore, there is no need to check their pointers since their presence is controlled by the system. This allows us to exclude the control block from this method.

Everything else is straightforward and simple. Let's organize a loop through all the nested neural layers of our *GPT* block. In the body of the block, we extract all the objects of the above collections, one by one. First, we check the pointer to the object, and then we call its method to propagate the error gradient to the level of the weight matrix.

```
bool CNeuronGPT::CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
{
    //--- in a loop, we call the method for each internal object
    for(int layer = 0; layer < m_iLayers; layer++)
    {
        if(!m_cFF2.At(layer))
            return false;
        CNeuronBase *temp = m_cFF2.At(layer);
        if(!temp.CalcDeltaWeights(m_cFF1.At(layer), false))
            return false;
        temp = m_cFF1.At(layer);
        if(!temp.CalcDeltaWeights(m_cW0.At(layer), false))
            return false;
    }
}
```

```

    temp = m_cW0.At(layer);
    if(!temp.CalcDeltaWeights(m_cAttentionOut.At(layer), false))
        return false;
    temp = m_cQuerys.At(layer);
    if(!temp)
        return false;
    CNeuronBase *prevL = (layer == 0 ? prevLayer : m_cFF2.At(layer - 1));
    if(!temp.CalcDeltaWeights(prevL, (read && layer == m_iLayers - 1)))
        return false;
}
//---
return true;
}

```

It is worth mentioning a few words about the order in which methods of internal objects are called. From the perspective of mathematical operations, the order of method calls does not affect the final result. However, the order of method calls used in the loop body is not random. Note that in the loop body, we explicitly check the pointers for only two objects that do not serve as the input data for other internal layers. The reason is that the called methods of neural layers also have a control block that checks the incoming data, including the received pointers. To eliminate repeated checks of object pointers, we first pass a pointer to the object as input to another object, check the result of the operations of the called method, which, among other things, confirms the validity of the passed pointer, and then confidently access the object because its pointer was checked during the execution of the previous object method. In this way, we organize a comprehensive check of all object pointers without explicit control within the method body and eliminate redundant pointer checks that could slow down the program execution.

Next, we will consider the method for updating model parameters. This function does not require external object data. There is not a single object pointer in the method parameters, as there are only parameter values for executing the specified parameter optimization algorithm.

In the method body, we also organize a loop to iterate through the nested neural layers of our *GPT* block. In the loop body, we extract one object from each collection, check the validity of the pointer, and call the method to update the weight matrix of each object.

```

bool CNeuronGPT::UpdateWeights(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda)
{
    //--- in a loop we call the method for each internal object
    for(int layer = 0; layer < m_iLayers; layer++)
    {
        CNeuronBase *temp = m_cFF2.At(layer);
        if(!temp || !temp.UpdateWeights(batch_size, learningRate, Beta, Lambda))
            return false;
        temp = m_cFF1.At(layer);
        if(!temp || !temp.UpdateWeights(batch_size, learningRate, Beta, Lambda))
            return false;
        temp = m_cW0.At(layer);
        if(!temp || !temp.UpdateWeights(batch_size, learningRate, Beta, Lambda))
            return false;
        temp = m_cQuerys.At(layer);
        if(!temp || !temp.UpdateWeights(batch_size, learningRate, Beta, Lambda))
            return false;
    }
}

```

```
    }  
    //---  
    return true;  
}
```

Since the called methods do not access external objects, our control optimization approach will not work here due to the absence of explicitly repetitive controls. Therefore, we need to explicitly check each object pointer before calling its method.

We have discussed the implementation of three backpropagation methods and with that, we conclude our work on implementing the *GPT* model algorithm in our *CNeuronGPT* class. For the complete implementation of functionality using standard MQL5 tools, we need to override the methods for working with files. We've already discussed the importance of these methods for the operation of neural network models.

5.3.2.3 File operations

We continue working on our *GPT* model implementation class. We have already implemented the functionality of this model in the methods of our *CNeuronGPT* class. In the previous sections, we discussed object initialization methods and created the processes of feed-forward and backpropagation passes. The specified functionality is sufficient for creating a test model, and it is even possible to conduct a series of tests to assess the model functionality.

However, we have already discussed the importance of file handling methods for the practical operation of any neural network model. The main significance of this process is attributed to the cost of the model training process because it requires both time and resources. Often such costs are quite high. Therefore, there is a strong desire to train the model once and then use it with maximum workload in the shortest possible time.

The unpredictable and highly volatile nature of financial markets does not leave us with hope for an indefinitely prolonged usage of a model trained once. However, even in this case, with the volatility of the environment, retraining the model in new conditions will require fewer resources and time compared to training the model from scratch with random weights.

Therefore, let's continue our work and implement methods for working with files. As always, let's start with the *CNeuronGPT::Save* method that saves data to a file.

When starting to work on the data saving method, as usual, we take a critical look at the structure of our class and evaluate the necessity of saving the data for each object.

```
class CNeuronGPT      : public CNeuronBase
{
protected:
    CArrayLayers      m_cQuerys;
    CArrayLayers      m_cKeys;
    CArrayLayers      m_cValues;
    CArrayLayers      m_cScores;
    CArrayLayers      m_cAttentionOut;
    CArrayLayers      m_cW0;
    CArrayLayers      m_cFF1;
    CArrayLayers      m_cFF2;
    //---
    int               m_iLayers;
    int               m_iWindow;
    int               m_iUnits;
    int               m_iKeysSize;
    int               m_iHeads;
    CBufferType       m_dStd[];
    int               m_iCurrentPosition;
    int               m_iScoreTemp;

    virtual bool      NormlizeBuffer(CBufferType *buffer, CBufferType *std,
                                     uint std_shift);

    virtual bool      NormlizeBufferGradient(CBufferType *output,
                                             CBufferType *gradient, CBufferType *std, uint std_shift);

public:
    CNeuronGPT(void);
    ~CNeuronGPT(void);
```

```

//---
virtual bool      Init(const CLayerDescription *desc) override;
virtual bool      SetOpenCL(CMyOpenCL *openc1) override;
virtual bool      FeedForward(CNeuronBase *prevLayer) override;
virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer, bool read) override;
virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda) override;

//---
virtual int        GetUnits(void) const { return m_iUnits; }
virtual int        GetLayers(void) const { return m_iLayers; }
//--- methods for working with files
virtual bool      Save(const int file_handle) override;
virtual bool      Load(const int file_handle) override;
//--- object identification method
virtual int        Type(void) override const { return(defNeuronGPT); }
};

```

At this point, we realize that besides constants, our class contains only collections of objects. The resources required to recreate the collection objects with a complete description of their structure will be much higher than the potential savings in disk space resources. Therefore, we organize the saving of all collections in a data file for model recovery.

In the parameters, this method receives a file handle to save the data. To avoid duplicate controls and reduce the total amount of program code, we do not check the received handle. Instead, we call the similar method of the parent class, to which we pass the received handle. The advantages of this approach are obvious. With a single command, we check the received handle and save the data of objects inherited from the parent class. By checking the result of the parent class method, we control the entire specified process.

```

bool CNeuronGPT::Save(const int file_handle)
{
//--- calling a method of a parent class
if(!CNeuronBase::Save(file_handle))
    return false;

```

After the successful execution of the method of the parent class, we save the following constants of our method to the file:

- *m_iLayers* – the number of nested neural layers of the *GPT* block
- *m_iWindow* – the size of the source data window (the size of the description vector of one element of the source data sequence)
- *m_iKeysSize* – the size of the description vector of one element of the *Keys* key tensor
- *m_iHeads* – the number of attention heads used
- *m_iUnits* – the number of elements in the sequence
- *m_iCurrentPosition* – the position of the currently analyzed element

```

//--- save the constants
if(FileWriteInteger(file_handle, m_iLayers) <= 0)
    return false;
if(FileWriteInteger(file_handle, m_iWindow) <= 0)
    return false;

```

```

    if(FileWriteInteger(file_handle, m_iKeysSize) <= 0)
        return false;
    if(FileWriteInteger(file_handle, m_iHeads) <= 0)
        return false;
    if(FileWriteInteger(file_handle, m_iUnits) <= 0)
        return false;
    if(FileWriteInteger(file_handle, m_iCurrentPosition) <= 0)
        return false;

```

Saving the position of the current analyzed element is necessary for the proper functioning of the *Key* and *Value* stacks. However, in real usage conditions, I would recommend that before using the model, you sequentially input data into it in a volume sufficient to fully fill the stacks. This approach will allow you to control the process of data loading into the model and eliminate the risk of possible omissions, which could potentially impact the accuracy of the model performance in the initial stages after data loading. Of course, the model will level out after the stack is completely filled. But the risk of losses up to this point increases.

Next, we sequentially check the pointers to objects in all our collections and call their data-saving methods.

```

//--- call the method for all collections of inner layers
    if(!m_cQuerys.Save(file_handle))
        return false;
    if(!m_cKeys.Save(file_handle))
        return false;
    if(!m_cValues.Save(file_handle))
        return false;
    if(!m_cScores.Save(file_handle))
        return false;
    if(!m_cAttentionOut.Save(file_handle))
        return false;
    if(!m_cW0.Save(file_handle))
        return false;
    if(!m_cFF1.Save(file_handle))
        return false;
    if(!m_cFF2.Save(file_handle))
        return false;
//---
    return true;
}

```

Then we exit the data saving method.

We have created a method for saving an object of our class. Now we can move on to work on the method of recovering an object from the data written to the file. As a reminder, the primary requirement for methods restoring the functionality of objects from saved data is to read the data in strict accordance with the sequence of their recording.

Similar to the file-writing method, our data-loading method *CNeuronGPT::Load* receives in parameters the handle of the file containing the data to be read. Just like when writing data, we first call the analogous method of the parent class. First, we read the data in strict accordance with the writing sequence. Second, we use the idea voiced when studying the method of writing data, that is, we use

the controls implemented in the method of the parent class and exclude their duplication. Of course, before proceeding further, we check the result of the parent method operations.

```
bool CNeuronGPT::Load(const int file_handle)
{
    //--- call the method of a parent class
    if(!CNeuronBase::Load(file_handle))
        return false;
```

After the successful execution of the parent class method, we read the constants of our block operating parameters. Their values are read in the order in which they are written. After reading the constant values, we should adjust the size of the dynamic array for writing standard deviations used in normalizing the results of our block operation. The size of the array must be sufficient to store data from all nested neural layers. Otherwise, we run the risk of encountering a critical error due to exceeding array dimensions during program execution.

```
//--- read constants from a file
m_iLayers = FileReadInteger(file_handle);
m_iWindow = FileReadInteger(file_handle);
m_iKeysSize = FileReadInteger(file_handle);
m_iHeads = FileReadInteger(file_handle);
m_iUnits = FileReadInteger(file_handle);
m_iCurrentPosition = FileReadInteger(file_handle);
if(ArrayResize(m_dStd, m_iLayers) <= 0)
    return false;
for(int i = 0; i < m_iLayers; i++)
    if(!m_dStd[i].BufferInit(1, 2, 1))
        return false;;
```

Then all we have to do is load the data from our object collections. However, before calling the method to load collection object data, we need to ensure the relevance of the collection object pointer and, if necessary, create a new instance of the collection object. Only then we can call the data loading method. Of course, do not forget that the order of loading objects is in strict accordance with the order of their writing. We also control the data loading process at each iteration.

```
//--- call the method for all collections of inner layers
if(!m_cQuerys.Load(file_handle))
    return false;
if(!m_cKeys.Load(file_handle))
    return false;
if(!m_cValues.Load(file_handle))
    return false;
if(!m_cScores.Load(file_handle))
    return false;
if(!m_cAttentionOut.Load(file_handle))
    return false;
if(!m_cW0.Load(file_handle))
    return false;
if(!m_cFF1.Load(file_handle))
    return false;
if(!m_cFF2.Load(file_handle))
    return false;
```

After loading all objects, we create another loop and reformat the result buffers of all created objects. In this case, we do not perform a validity check on the object pointers as in the previous iterations, all these objects loaded data from the file, which means they were created and verified.

```
//--- reformat the result matrices
for(int i = 0; i < m_iLayers; i++)
{
    CNeuronBase* temp = m_cKeys.At(i);
    if(!temp.GetOutputs().Reshape(m_iUnits, m_iKeysSize * m_iHeads))
        return false;
    temp = m_cValues.At(i);
    if(!temp.GetOutputs().Reshape(m_iUnits, m_iKeysSize * m_iHeads))
        return false;
    temp = m_cScores.At(i);
    if(!temp.GetOutputs().Reshape(m_iHeads, m_iUnits))
        return false;
    temp = m_cAttentionOut.At(i);
    if(!temp.GetOutputs().Reshape(m_iHeads, m_iKeysSize))
        return false;
}
```

At the end of the method, we replace the buffers and terminate its work.

```
//--- replace data buffers to avoid excessive copying
CNeuronBase *last = m_cFF2.At(m_cFF2.Total() - 1);
if(!m_cOutputs)
    delete m_cOutputs;
m_cOutputs = last.GetOutputs();
if(!m_cGradients)
    delete m_cGradients;
m_cGradients = last.GetGradients();
//---
return true;
}
```

Now that the file handling methods have been created, we can proceed further. Next, our plan involves creating the capability to perform parallel mathematical operations using OpenCL.

5.3.3 Organizing parallel computing in the GPT model

We continue to work on our model class, *GPT CNeuronGPT*. In the previous sections, we have already recreated the model algorithm using standard MQL5 tools. Now it's time to supplement the model with the ability to perform mathematical operations in multi-threaded mode using the computing power of the GPU. This is the opportunity provided by OpenCL.

To organize this process, we have to perform two subtasks:

- Create an OpenCL program.
- Organize the call of the OpenCL program from the main program.

Let's start by creating an executable program on the OpenCL side. In this program, we need to implement that part of the algorithm that is not covered by the use of internal object methods. We

have two such blocks: one in the feed-forward part, and the second, mirrored to the first, included in the error gradient propagation method when performing the backpropagation pass.

To execute the feed-forward algorithm, we will create the *GPTFeedForward* kernel. In part, the kernel algorithm will resemble a similar kernel for classes using attention mechanisms. This is not surprising since they all use the *Self-Attention* mechanism. However, each implementation has its nuances. Last time, instead of creating a new kernel for organizing multi-head attention, we were able to quickly modify the existing kernel of the *Self-Attention* algorithm. Now, creating a new kernel seems less costly compared to trying to create a universal kernel for all tasks.

Unlike the implementation of the *Multi-Heads Self-Attention* mechanism in which we translated the kernel into a two-dimensional task space, in this implementation we return to a one-dimensional space. This is due to the lack of the possibility of splitting the task into parallel threads in the context of the elements of the *Query* tensor sequence since the *GPT* model implementation only allows processing one query per iteration. In this case, we are left with a division by threads only in the context of attention heads.

In the parameters of the *GPTFeedForward* kernel, we will continue to pass pointers to five data buffers. However, the number of variables increases: earlier we obtained the size of the sequence from the dimension of the task space, but now we have to explicitly specify it in the kernel parameters. Here, an additional parameter is used to specify the current element in the sequences of keys and values.

```
__kernel void GPTFeedForward(__global TYPE *queries,
                             __global TYPE *keys,
                             __global TYPE *scores,
                             __global TYPE *values,
                             __global TYPE *outputs,
                             int key_size,
                             int units,
                             int current)
{
```

As mentioned earlier, the created kernel will operate in a one-dimensional space, focusing on the attention heads. Therefore, the first thing we do in the body of the kernel is determine the active attention head based on the identifier of the executing thread and the total number of attention heads, considering the total number of running threads.

```
    const int h = get_global_id(0);
    const int heads = get_global_size(0);
    int shift_query = key_size * h;
    int shift_scores = units * h;
```

We immediately determine the offset in the *Query* and *Score* (dependency coefficient matrix) tensors.

Next, according to the *Self-Attention* algorithm that is being built, we determine the dependency coefficients between the elements of the sequence. To do this, we multiply the *Query* vector by the *Key* tensor. To implement these operations, we will create a system of two nested loops. The outer loop will iterate over the elements of the *Key* tensor sequence and, accordingly, the elements of the *Score* vector with dependency coefficients. In the body of the loop, we will define the offset in the *Key* tensor and prepare a local variable to count the intermediate values.

After that, we organize a nested loop with the number of iterations equal to the size of the description vector of one element of the sequence. In the body of this cycle, we perform the operation of multiplying a pair of vectors. The resulting value is divided by the square root of the dimension of the

vector, and we take the exponent from it. We write the result of the operation into the corresponding element of the dependency coefficient vector and add it to the cumulative sum of all elements in the dependency coefficient vectors for subsequent data normalization.

We should consider the issue of the concatenated buffer of the results of the *Query* layer. The values of the key and query vectors of the current element in the sequence have not yet been transferred to the corresponding buffers. Therefore, we check the element that we are accessing in the key tensor. Before accessing the current element, we first copy the data to the buffer. Of course, for current operations, we could take data from the *queries* buffer. But we will need this data in subsequent iterations. Therefore, transferring them to the buffer is inevitable.

```
TYPE summ = 0;
for(int s = 0; s < units; s++)
{
    TYPE score = 0;
    int shift_key = key_size * (s * heads + h);
    for(int k = 0; k < key_size; k++)
    {
        if(s == current)
            keys[shift_key + k] = queries[shift_query + k + heads * key_size];
        score += queries[shift_query + k] * keys[shift_key + k];
    }
    score = exp(score / sqrt((TYPE)key_size));
    summ += score;
    scores[shift_scores + s] = score;
}
```

As a result of performing a full cycle of iterations of the system created above from two loops, we get a vector of dependency coefficients. According to the *Self-Attention* algorithm, before further use of the obtained coefficients, they will have to be normalized by the *Softmax* function. When obtaining the exponent from the products of vectors, we have already executed part of the algorithm of the specified function. To complete the normalization operation, we just need to divide the values stored in the vector by their total sum, which we prudently collected in the local variable *summ*. Therefore, we organize another loop with the number of iterations equal to the size of the vector of dependency coefficients. In the body of this loop, we will divide all the values of the vector by the value of the local variable *summ*.

```
for(int s = 0; s < units; s++)
    scores[shift_scores + s] /= summ;
```

Thus, after completing the iterations of the loop in the *Score* vector, we get the normalized values of the dependency coefficients with the total sum of all elements in the block. In fact, the obtained coefficients give us an idea of the proportion of influence of each element of the sequence from the *Value* tensor on the final value of the analyzed element of the sequence in the tensor of the results of the current attention head.

This means that in order to obtain the final values, we need to multiply the *Score* vector with normalized dependency coefficients by the *Value* tensor. To perform this operation, we need another system of two nested loops. But before running it, we will determine the offset in the tensor of the results before the beginning of the vector of the analyzed element of the sequence.

The outer loop, with the number of iterations equal to the size of the vector describing one element of the sequence, indicates the ordinal number of the collected element in the result vector. The nested

loop, with the number of iterations equal to the number of elements in the sequence, helps correlate the vectors of the *Value* tensor with the dependency coefficients from the *Score* vector. In the body of the nested loop, we multiply the vector from the *Value* tensor by the corresponding element dependency coefficient. The resulting values of the products will be accumulated in a local variable. After completing the iterations of the inner loop, we save the obtained value in the buffer of the *Self-Attention* block results.

```

shift_query = key_size * h;
for(int i = 0; i < key_size; i++)
{
    TYPE query = 0;
    for(int v = 0; v < units; v++)
    {
        if(v == current)
            values[key_size * (v * heads + h) + i] =
                queries[(2 * heads + h) * key_size + i];
        query += values[key_size * (v * heads + h) + i] *
                scores[shift_scores + v];
    }
    outputs[shift_query + i] = query;
}
}

```

As a result of completing the full cycle of iterations within the loop system, we obtain the vector describing one element of the sequence in the tensor of results for one attention head. The task assigned to this kernel has been completed, and we can exit it.

This concludes the work with the feed-forward kernel, and we move further along. Now we need to organize the backpropagation process. The implementation of this task will be split into two kernels. In the *GPTCalcScoreGradient* kernel, we will propagate the error gradient to the vector of the dependency coefficients. In the *GPTCalcHiddenGradient* kernel, we will continue the propagation of the error gradient up to the level of the *Query* and *Key* tensors.

Let's take it step by step. The *GPTCalcScoreGradient* kernel in the parameters receives pointers to six data buffers and three parameters:

- *scores* – buffer for the vector of dependency coefficients
- *scores_grad* – buffer for the error gradient vector at the level of dependency coefficients
- *values* – buffer for the *Value* tensor
- *values_grad* – buffer for the error gradient tensor at the *Value* level
- *outputs_grad* – error gradient tensor buffer at the result level of the *Self-Attention* block
- *scores_temp* – buffer for writing intermediate values
- *window* – size of the vector describing one element of the sequence in the *Value* tensor
- *units* – number of elements in the sequence
- *current* – ordinal number of the current item in the *Value* stack

```

__kernel void GPTCalcScoreGradient(__global TYPE *scores,
                                   __global TYPE *scores_grad,
                                   __global TYPE *values,
                                   __global TYPE *values_grad,
                                   __global TYPE *outputs_grad,

```

```

        __global__ TYPE *scores_temp,
        int window,
        int units,
        int current)
{

```

As with the feed-forward pass, we run the kernel in one-dimensional space by the number of attention heads used. In the body of the kernel, we immediately determine the active attention head based on the thread identifier and the total number of attention heads considering the total number of launched threads.

```

    const int h = get_global_id(0);
    const int heads = get_global_size(0);
    int shift_value = window * (2 * heads + h);
    int shift_score = units * h;

```

We also determine the offset in the tensors of the error gradients at the level of *Value* and in the vector of the dependency coefficients. Note that the offset in the tensor of error gradients for *Value* and in the *Value* tensor itself will be different in this case.

In this implementation of the *GPT* model, we used one internal neural layer to generate a concatenated tensor containing the values of *Query*, *Key*, and *Value* for all attention heads. Accordingly, we assemble the error gradient into a similar concatenated tensor of the error gradients for the specified neural layer. However, this tensor contains only the current element of the sequence. At the same time, the *Value* stack tensor contains complete information about the entire sequence but only for the *Value* tensor.

After the preparatory work, we distribute the error gradient to the *Value* tensor. As mentioned above, we distribute the error gradient only for the current element of the sequence. To do this, we organize a loop with the number of iterations equal to the size of the description vector of one element of the sequence in the *Value* tensor. In the body of the loop, will multiply the error gradient vector at the result level of the *Self-Attention* block by the corresponding dependency coefficient from the *Score* vector. The obtained values are saved in the buffer of the concatenated tensor of error gradients.

```

//--- Gradient distribution to Values
for(int i = 0; i < window; i++)
    values_grad[shift_value + i] = scores[units * h + current] *
                                     outputs_grad[window * h + i];

```

After calculating the error gradient on the *Value* tensor, we will determine the value of the error gradient at the level of the dependency coefficient vector. To perform this operation, we will need a system of two loops: an outer loop with the number of iterations equal to the number of elements in the sequence and a nested loop with the number of iterations equal to the size of the description vector for one element in the *Value* tensor. In the body of the nested loop, we will multiply 2 vectors (*Value* and error gradient). The resulting value is stored in the temporary data buffer.

```

//--- Gradient distribution to Score
for(int k = 0; k < units; k++)
{
    TYPE grad = 0;
    for(int i = 0; i < window; i++)
        grad += outputs_grad[shift_value + i] *
                values[window * (k * heads + h) + i];
    scores_temp[shift_score + k] = grad;
}

```

```
}
```

After completing the full cycle of iterations within the loop system, in the temporary buffer, we will obtain a fully populated gradient vector of error for the dependency coefficient vector. But to distribute the error gradient further, we first need to correct it to the derivative of the *Softmax* function.

Let's organize another system of two nested cycles. Both loops will contain the number of iterations equal to the number of elements in the sequence. In the body of the nested loop, we will calculate the derivative of the function using the formula.

$$(\text{Softmax}(x_i))' = \begin{cases} i = j & y_i(1 - y_i) \\ i \neq j & -y_i y_j \end{cases}$$

```
//--- Adjust to the Softmax derivative
for(int k = 0; k < units; k++)
{
    TYPE grad = 0;
    TYPE score = scores[shift_score + k];
    for(int i = 0; i < units; i++)
        grad += scores[shift_score + i] * ((int)(i == k) - score) *
            scores_temp[shift_score + i];
    scores_grad[shift_score + k] = grad;
}
}
```

We will save the obtained values in the error gradient buffer at the level of the dependency coefficient vector. At this stage, we complete the work of the first kernel and move on to the second one.

In the second kernel of the *GPTCalcHiddenGradient* backpropagation process, we have to propagate the error gradient further and bring it to the level of *Query* and *Key* tensors.

In parameters, the *GPTCalcHiddenGradient* kernel receives pointers to 4 data buffers and 3 parameters.

```
__kernel void GPTCalcHiddenGradient(__global TYPE *querys,
    __global TYPE *querys_grad,
    __global TYPE *keys,
    __global TYPE *scores_grad,
    int key_size,
    int units,
    int current)
{
```

Note that we talked about distributing the gradient into the *Query* and *Key* tensors. In the kernel parameters, there is a pointer only to the *Query* error gradient buffer. This situation is made possible by the use of a concatenated buffer, in which we have already saved the error gradient at the level of the *Value* tensor. Now we add error gradients at the level of the *Query* and *Key* tensors to the same buffer.

In the kernel body, we determine the ordinal number of the analyzed attention head based on the thread identifier and the number of used attention heads considering the total number of tasks launched.

```
const int h = get_global_id(0);
const int heads = get_global_size(0);
```

```

int shift_query = key_size * h;
int shift_key = key_size * (heads + h);
int shift_score = units * h;

```

Here we also define the offsets in the data buffers before the beginning of the analyzed vectors.

Next, we organize a system of two nested loops, in the body of which we will determine the error gradients at the level of the tensors we are looking for. To do this, multiply the error gradient at the level of the dependency coefficient vector by the opposite tensor.

```

//--- Gradient distribution on Querys and Keys
const TYPE k = 1 / sqrt((TYPE)key_size);
//---
for(int i = 0; i < key_size; i++)
{
    TYPE grad_q = 0;
    TYPE grad_k = 0;
    for(int s = 0; s < units; s++)
    {
        grad_q += keys[key_size * (s * heads + h) + i] *
                scores_grad[shift_score + s];

        if(s == current)
            grad_k += querys[key_size * h + i] *
                scores_grad[units * h + current];
    }
    querys_grad[shift_query + i] = grad_q * k;
    querys_grad[shift_key + i] = grad_k * k;
}
}

```

Note that we calculate the error gradient only for the current element of the sequence and save the obtained values in the corresponding elements of the error gradient buffer.

As a result of all iterations of our loop system, we get a fully filled concatenated tensor of error gradients of all three entities (*Query*, *Key*, and *Value*). We complete the work on building the OpenCL program and move on to building the functionality on the side of the main program.

To make it more convenient to manage the constructed kernels in the main program, let's create named constants for calling kernels and accessing their elements. To do this, we open our constants file [defines.mqh](#) and create kernel access constants in it.

```

#define def_k_GPTFeedForward          34
#define def_k_GPTScoreGradients       35
#define def_k_GPTHiddenGradients      36

```

Then we add access constants to kernel parameters.

```

//--- GPT feed-forward pass
#define def_gptff_querys              0
#define def_gptff_keys                1
#define def_gptff_scores              2
#define def_gptff_values              3
#define def_gptff_outputs             4
#define def_gptff_key_size            5

```



```

#define def_gptff_units          6
#define def_gptff_current        7

//--- determine the gradient at the matrix of GPT dependency coefficients
#define def_gptscr_scores        0
#define def_gptscr_scores_grad   1
#define def_gptscr_values        2
#define def_gptscr_values_grad   3
#define def_gptscr_outputs_grad  4
#define def_gptscr_scores_temp    5
#define def_gptscr_window        6
#define def_gptscr_units         7
#define def_gptscr_current       8

//--- gradient distribution via GPT
#define def_gpthgr_queries        0
#define def_gpthgr_queries_grad   1
#define def_gpthgr_keys          2
#define def_gpthgr_scores_grad    3
#define def_gpthgr_key_size       4
#define def_gpthgr_units         5
#define def_gpthgr_current       6

```

After that, we go to the dispatch service class of the *CNet* neural network model and, in the OpenCL initialization method *InitOpenCL*, we change the total number of kernels in our program. Next, we initialize the creation of new kernels in the OpenCL context.

```

bool CNet::InitOpenCL(void)
{
    .....
    if(!m_cOpenCL.SetKernelsCount(37))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        return false;
    }
    .....
    if(!m_cOpenCL.KernelCreate(def_k_GPTFeedForward, "GPTFeedForward"))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        return false;
    }
    if(!m_cOpenCL.KernelCreate(def_k_GPTScoreGradients, "GPTCalcScoreGradient"))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        return false;
    }
    if(!m_cOpenCL.KernelCreate(def_k_GPTHiddenGradients, "GPTCalcHiddenGradient"))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
    }
}

```

```

        return false;
    }
    //---
    return true;
}

```

This concludes the preparatory work and goes directly to the methods of our *CNeuronGPT* class. In them, we have to perform three stages of work to call each kernel:

- Preparing the input data and transferring it to the memory of the OpenCL context.
- Placing the kernel in the execution queue.
- Loading the results of program execution into the memory of the main program.

First, we modify the *CNeuronGPT::FeedForward* method. In the block for organizing multi-threaded computing using OpenCL, we first check for the presence of an already created buffer in the memory of the OpenCL context.

```

bool CNeuronGPT::FeedForward(CNeuronBase *prevLayer)
{
    .....
    for(int layer = 0; layer < m_iLayers; layer++)
    {
        .....
        //--- branching of the algorithm by the computing device
        if(!m_cOpenCL)
        {
            // Program block using standard MQL5 tools
            .....
        }
        else // OpenCL block
        {
            //--- checking data buffers
            if(Querys.GetOutputs().GetIndex() < 0)
                return false;
            if(Keys.GetOutputs().GetIndex() < 0)
                return false;
            if(Values.GetOutputs().GetIndex() < 0)
                return false;
            if(Scores.GetOutputs().GetIndex() < 0)
                return false;
            if(AttentionOut.GetOutputs().GetIndex() < 0)
                return false;

```

When all buffers have been created, and those that are necessary for kernel operation have been passed to the OpenCL context memory, we pass pointers to the used data buffers and the necessary constants to the kernel parameters.

```

        //--- pass parameters to the kernel
        if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTFeedForward,
                                         def_gptff_keys, Keys.GetOutputs().GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTFeedForward,
                                         def_gptff_outputs, AttentionOut.GetOutputs().GetIndex()))

```

```

        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTFeedForward,
                                    def_gptff_queries, Querys.GetOutputs().GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTFeedForward,
                                    def_gptff_scores, Scores.GetOutputs().GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTFeedForward,
                                    def_gptff_values, Values.GetOutputs().GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_GPTFeedForward,
                              def_gptff_key_size, m_iKeysSize))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_GPTFeedForward,
                              def_gptff_units, m_iUnits))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_GPTFeedForward,
                              def_gptff_current, m_iCurrentPosition))
        return false;

```

At this stage, the preparatory work is completed. Let's move on to the stage of placing the kernel in the execution queue. Here, we first create two dynamic arrays in which we specify the offset and the number of running threads in each task subspace. Then call the *m_cOpenCL.Execute* method that places the kernel in the queue.

```

    //--- Place a kernel in the queue
    int off_set[] = {0};
    int NDRange[] = {m_iHeads};
    if(!m_cOpenCL.Execute(def_k_GPTFeedForward, 1, off_set, NDRange))
        return false;
}

```

This concludes the *CNeuronGPT::FeedForward* method. But we still have to do similar work in the *CNeuronGPT::CalcHiddenGradient* method of the backpropagation algorithm.

Let me remind you that in order to implement the backpropagation method, we have created two kernels that will be called sequentially one after the other. Therefore, the kernel maintenance work must be repeated for each of them.

First, let's create data buffers for the first kernel.

```

bool CNeuronGPT::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    .....
    for(int layer = m_iLayers - 1; layer >= 0; layer--)
    {
        .....
        //--- branching of the algorithm by the computing device
        attention_grad = AttentionOut.GetGradients();
        if(!m_cOpenCL)
        {
            // Program block using standard MQL5 tools
            .....

```

```

    }
else // OpenCL block
{
    //--- checking data buffers
    if(Values.GetOutputs().GetIndex() < 0)
        return false;
    if(Querys.GetGradients().GetIndex() < 0)
        return false;
    if(Scores.GetOutputs().GetIndex() < 0)
        return false;
    if(attention_grad.GetIndex() < 0)
        return false;
    if(Scores.GetGradients().GetIndex() < 0)
        return false;
    if(m_iScoreTemp < 0)
        return false;
}

```

Following our algorithm for working with the OpenCL context, after creating data buffers and passing all the necessary information to the context memory, we pass pointers to the used data buffers and constants for executing the program algorithm to the parameters of the kernel being launched.

```

//--- pass parameters to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
                                def_gptscr_outputs_grad, attention_grad.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
                                def_gptscr_scores, Scores.GetOutputs().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
                                def_gptscr_scores_grad, Scores.GetGradients().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
                                def_gptscr_scores_temp, m_iScoreTemp))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
                                def_gptscr_values, Values.GetOutputs().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
                                def_gptscr_values_grad, Querys.GetGradients().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_GPTScoreGradients,
                           def_gptscr_window, m_iKeysSize))
    return false;
if(!m_cOpenCL.SetArgument(def_k_GPTScoreGradients,
                           def_gptscr_units, m_iUnits))
    return false;
if(!m_cOpenCL.SetArgument(def_k_GPTScoreGradients,
                           def_gptscr_current, m_iCurrentPosition))
    return false;

```

Note that instead of the *Value* tensor error gradient buffer, we pass a pointer to the gradient buffer of the inner neural layer *Querys*. This is because we used a concatenated error gradient buffer for all three

tensors. To eliminate the subsequent data copy operation, we will immediately write the data to the concatenated buffer.

After that, we perform the operation of placing the kernel in the queue. Let me remind you that we are launching a kernel to perform tasks in one-dimensional space in the context of attention heads.

Let's specify the offset in the task space and the number of threads to be started in the corresponding dynamic arrays. After that, we call the method of queuing our kernel.

```
//--- Place the kernel in queue
int off_set[] = {0};
int NDRange[] = {m_iHeads};
if(!m_cOpenCL.Execute(def_k_GPTScoreGradients, 1, off_set, NDRange))
    return false;
```

This concludes the work on the first kernel, and we move on to building a similar algorithm for the second kernel of the backpropagation pass.

Now we will check the additional buffers in the memory of the OpenCL context.

```
if(Querys.GetOutputs().GetIndex() < 0)
    return false;
if(Keys.GetOutputs().GetIndex() < 0)
    return false;
```

We pass the parameters to the kernel.

```
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTHHiddenGradients,
                                def_gpthgr_keys, Keys.GetOutputs().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTHHiddenGradients,
                                def_gpthgr_querys, Querys.GetOutputs().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTHHiddenGradients,
                                def_gpthgr_querys_grad, Querys.GetGradients().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTHHiddenGradients,
                                def_gpthgr_scores_grad, Scores.GetGradients().GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_GPTHHiddenGradients,
                           def_gpthgr_key_size, m_iKeysSize))
    return false;
if(!m_cOpenCL.SetArgument(def_k_GPTHHiddenGradients,
                           def_gpthgr_units, m_iUnits))
    return false;
if(!m_cOpenCL.SetArgument(def_k_GPTHHiddenGradients,
                           def_gpthgr_current, m_iCurrentPosition))
    return false;
```

After that, we will put the kernel in the execution queue. Note that this time we do not create arrays of offset and dimensionality of the task space. We simply use the arrays created during the execution of the previous kernel without modification.

```
if(!m_cOpenCL.Execute(def_k_GPTHHiddenGradients, 1, off_set, NDRange))
    return false;
```

```
}
```

This completes the work on building a *GPT* model class, and we can proceed to evaluate the results of the work done.

5.3.4 Comparative testing of implementations

We have completed the *CNeuronGPT* neural layer class using the attention mechanisms. In this class, we attempted to recreate the *GPT* (*Generative Pre-trained Transformer*) model proposed by the *OpenAI* team in 2018. This model was developed for language tasks but later showed quite good results for other tasks as well. The third generation of this model (*GPT-3*) is the most advanced language model at the time of writing this book.

The distinguishing feature of this model from other variations of the Transformer model is its autoregressive algorithm. In this case, the model is not fed with the entire volume of data describing the current state but only the changes in the state. In language problem solving examples, we can input into the model not the whole text at once, but one word at a time. Furthermore, the output generated by the model represents a continuation of the sentence. We input this word again into the model without repeating the previous phrase. The model maps it to the stored previous states and generates a new word. In practice, such an autoregressive model allows the generating of coherent texts. By avoiding the reprocessing of previous states, the model's computational workload is significantly reduced without sacrificing its performance quality.

We will not set the task of generating a new chart candlestick. For a comparative analysis of the model performance with previously discussed architectural solutions, we will keep the same task and the previously used training dataset. However, we will make the task more challenging for this model. Instead of providing the entire pattern as before, we will only input a small part of it consisting of the last five candles. To do this, let's modify our test script a bit.

We will write the script for this test to the file *gpt_test.mq5*. As a template, we take one of the previous attention model testing scripts: [attention_test.mq5](#). At the beginning of the script, we define a constant for specifying the size of the pattern in the training dataset file and external parameters for configuring the script.

```
#define GPT_InputBars      5
#define HistoryBars       40
//+-----+
//| External parameters for script operation |
//+-----+
// Name of the file with the training sample
input string   StudyFileName = "study_data.csv";
// File name for recording the error dynamics
input string   OutputFileName = "loss_study_gpt.csv";
// Number of historical bars in one pattern
input int      BarsToLine    = 40;
// Number of input layer neurons per 1 bar
input int      NeuronsToBar   = 4;
// Use OpenCL
input bool     UseOpenCL     = false;
// Batch size for updating the weights matrix
input int      BatchSize     = 10000;
// Learning rate
```

```

input double   LearningRate   = 0.00003;
// Number of hidden layers
input int      HiddenLayers   = 3;
// Number of neurons in one hidden layer
input int      HiddenLayer    = 40;
// Number of loops of updating the weights matrix
input int      Epochs         = 1000;

```

As you can see, all the external parameters of the script have been inherited from the test script of the previous model. The constant for the size of the pattern in the training dataset is necessary for organizing the correct loading of data because, in this implementation, the size of the data passed to the model will be significantly different from the size of the pattern in the training dataset. I didn't make this constant an external parameter because we are using a single training dataset, so there's no need to change this parameter during testing. At the same time, the introduction of an additional external parameter can potentially add confusion for the user.

After declaring the external parameters of the test script we are creating, we include our library for creating neural network models.

```

//+-----+
//| Connecting the neural network library |
//+-----+
#include "..\..\..\Include\NeuroNetworksBook\realization\neuronnet.mqh"

```

Here we finish creating global variables and can proceed with the script.

In the body of the script, we need to make changes to two functions. The first changes will be made to the *CreateLayersDesc* function that describes the architecture of the model. As mentioned above, we will only feed information about the last five candlesticks to the model input. So, we reduce the size of the raw data layer to 20 neurons. But we will make the script architecture flexible and specify the size of the source data layer as the product of the external parameter of the number of neurons per description of one candlestick in *NeuronsToBar* and the constant of the number of candlesticks to load in *GPT_InputBars*.

```

bool CreateLayersDesc(CArrayObj &layers)
{
    CLayerDescription *descr;
    //--- create source data layer
    if(!(descr = new CLayerDescription()))
    {
        PrintFormat("Error creating CLayerDescription: %d", GetLastError());
        return false;
    }
    descr.type          = defNeuronBase;
    int prev_count = descr.count = NeuronsToBar * GPT_InputBars;
    descr.window        = 0;
    descr.activation     = AF_NONE;
    descr.optimization  = None;
    if(!layers.Add(descr))
    {
        PrintFormat("Error adding layer: %d", GetLastError());
        delete descr;
        return false;
    }
}

```

```
}
```

Note that in case of an error occurring while adding an object to the dynamic array, we output a message to the log for the user and **make sure** to delete the objects we created before the script finishes. It should become a good practice for you to always clean up memory before the program terminates, whether it's normal termination or due to an error.

After adding a neural layer to the dynamic array of descriptions, we proceed to the next neural layer. We create a new instance of an object to describe the neural layer. We cannot use the previously created instance because the variable only holds a pointer to the object. This same pointer was passed to the dynamic array of pointers to objects describing neural layers. Therefore, when making changes to the object through the pointer in the local variable, all new data will be reflected when accessing the object through the pointer in the dynamic array. Thus, by using one pointer, we will only have copies of the same pointer in the dynamic array, and the program will create a model consisting of identical neural layers instead of the desired architecture.

```
//--- GPT block
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    delete descr;
    return false;
}
```

As the second layer, we will create a *GPT* block. The model will know about this from the *defNeuronGPT* constant in the *type* field of the created neural layer.

In the *count* field, we will specify the stack size to store the pattern information. Its value will determine the size of the buffers for the *Key* and *Value* tensors, and will also affect the size of the vector of dependency coefficients *Score*.

We will set the size of the input window equal to the number of elements in the previous layer, which we have saved in a local variable.

The size of the description vector of one element in the *Key* tensor will be equal to the number of description elements of one candlestick. This is the value we used when performing previous tests with attention models. This approach will help us to put more emphasis on the impact of the solution architecture itself, rather than the parameters used.

We also transfer the rest of the parameters unchanged from the scripts of previous tests with attention models. Among them are the number of attention heads used and the parameter optimization function. I'll remind you that the activation functions for all internal neural layers are defined by the Transformer architecture, so there's no need for an additional activation function for the neural layer here.

```
descr.type = defNeuronGPT;
descr.count = BarsToLine;
descr.window = prev_count;
descr.window_out = NeuronsToBar; // Size of Key vector
descr.step = 8;                  // Attention heads
descr.layers = 4;
descr.activation = AF_NONE;
descr.optimization = Adam;
```


Besides, when testing the *Multi-Head Self-Attention* architecture, we created four identical neural layers. Now, to create such an architecture, we only need to create one description of a neural layer and specify the number of identical neural layers in the *layers* parameter.

We add the created description of the neural layer to our collection of descriptions of the architecture of the created model.

```
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

Next comes a block of hidden fully connected neural layers, transferred in an unchanged form from the scripts of the previous tests, as well as the results layer. At the output of our model, there will be a results layer represented by a fully connected neural layer with two elements and a linear activation function.

The next block we will modify is the *LoadTrainingData* function for loading the training sample.

First, we create two dynamic data buffer objects. One will be used for loading pattern descriptions, and the other for target values.

```
bool LoadTrainingData(string path, CArrayObj &data, CArrayObj &result)
{
    CBufferType *pattern;
    CBufferType *target;
```

After that, we open the training dataset file for reading. When opening the file, we use the *FILE_SHARE_READ* flag, which allows other programs to read this file without blocking it.

```
//--- open the file with the training dataset
int handle = FileOpen(path, FILE_READ | FILE_CSV | FILE_ANSI |
                      FILE_SHARE_READ, ",", CP_UTF8);
if(handle == INVALID_HANDLE)
{
    PrintFormat("Error opening study data file: %d", GetLastError());
    return false;
}
```

Now we check the resulting file handle.

After successfully opening the training dataset file, we create a loop to read the data up to the end of the file. To enable the script to be forcibly stopped, we will add the *IsStopped* function to check the interruption of the program closure.

```
//--- display the progress of training data loading in the chart comment
uint next_comment_time = 0;
uint OutputTimeout = 250; // not more than once every 250 milliseconds
//--- organize a loop to load the training sample
while(!FileIsEnding(handle) && !IsStopped())
{
    if(!(pattern = new CBufferType()))
    {
```

```

        PrintFormat("Error creating Pattern data array: %d", GetLastError());
        return false;
    }
    if(!pattern.BufferInit(1, NeuronsToBar * GPT_InputBars))
        return false;
    if(!(target = new CBufferType()))
    {
        PrintFormat("Error creating Pattern Target array: %d", GetLastError());
        return false;
    }
    if(!target.BufferInit(1, 2))
        return false;

```

In the body of the loop, we create new instances of data buffers for writing individual patterns and their target values, for which we have already declared local variable pointers earlier. As always, we control the process of object creation. Otherwise, there is an increased risk of encountering a critical error when subsequently accessing the created object.

It's worth pointing out that we will create new objects at each iteration of the loop. This is due to the principles of working with pointers to object instances which were described a bit above when creating the model description.

After successful creation of objects, we proceed directly to reading the data. When creating a training dataset, we first recorded descriptions of 40 candlestick patterns followed by 2 target value elements. We will read the data in the same sequence. First, we organize a loop to read the pattern description vector. We will read from the file one value at a time into a local variable, while simultaneously checking the position of the loaded element. We will only save those elements in the data buffer that fall within the size of our analysis window.

```

int skip = (HistoryBars - GPT_InputBars) * NeuronsToBar;
for(int i = 0; i < NeuronsToBar * HistoryBars; i++)
{
    TYPE temp = (TYPE)FileReadNumber(handle);
    if(i < skip)
        continue;
    pattern.m_mMatrix[0, i - skip] = temp;
}

```

We read the target values in the same way, only here we leave both values.

```

for(int i = 0; i < 2; i++)
    target.m_mMatrix[0, i] = (TYPE)FileReadNumber(handle);

```

After successfully reading information about one pattern from the file, we save the loaded information into dynamic arrays of our database. We save the pattern information in the dynamic *data* array and the target values in the *result* array.

```

if(!data.Add(pattern))
{
    PrintFormat("Error adding study data to array: %d", GetLastError());
    return false;
}

if(!result.Add(target))

```

```

{
    PrintFormat("Error adding study data to array: %d", GetLastError());
    return false;
}

```

Meanwhile, we monitor the process of operations.

At this point, we have fully loaded and saved information about one pattern. Before moving on to loading information about the next pattern, let's display on the chart of the instrument the number of loaded patterns for visual control by the user.

Let's move on to the next iteration of the loop.

```

//--- output download progress in chart comment
//--- (not more than once every 250 milliseconds)
if(next_comment_time < GetTickCount())
{
    Comment(StringFormat("Patterns loaded: %d", data.Total()));
    next_comment_time = GetTickCount() + OutputTimeout;
}
}
FileClose(handle);
return(true);
}

```

When all iterations of the loop are complete, the two dynamic arrays (*data* and *result*) will contain all the information about the training dataset. We can close the file and at the same time terminate the data loading block. This completes the function.

GPT is a regression model. This means it is sensitive to the sequence of input elements. To meet such a requirement of the model for the training loop, let's apply the developments of a recurrent algorithm. We randomly select only the first element of the training batch and, in the interval between model parameter updates, we input consecutive patterns.

```

bool NetworkFit(CNet &net, const CArrayObj &data, const CArrayObj &target,
                VECTOR &loss_history)
{
    //--- training
    int patterns = data.Total();
    //--- loop through epochs
    for(int epoch = 0; epoch < Epochs; epoch++)
    {
        ulong ticks = GetTickCount64();
        //--- training by batches
        //--- select a random pattern
        int k = (int)((double)(MathRand() * MathRand()) / MathPow(32767.0, 2) *
                    (patterns - BarsToLine-1));

        k = fmax(k, 0);
        for(int i = 0; (i < (BatchSize + BarsToLine) && (k + i) < patterns); i++)
        {
            //--- check to see if the training has stopped
            if(IsStopped())
            {
                Print("Network fitting stopped by user");
            }
        }
    }
}

```

```

        return true;
    }
    if(!net.FeedForward(data.At(k + i)))
    {
        PrintFormat("Error in FeedForward: %d", GetLastError());
        return false;
    }
    if(i < BarsToLine)
        continue;
    if(!net.Backpropagation(target.At(k + i)))
    {
        PrintFormat("Error in Backpropagation: %d", GetLastError());
        return false;
    }
}
}
//--- reconfigur the network weights
net.UpdateWeights(BatchSize);
printf("Use OpenCL %s, epoch %d, time %.5f sec",
       (string)UseOpenCL, epoch, (GetTickCount64() - ticks) / 1000.0);
//--- report on a bygone epoch
TYPE loss = net.GetRecentAverageLoss();
Comment(StringFormat("Epoch %d, error %.5f", epoch, loss));
//--- remember the epoch error to save to file
loss_history[epoch] = loss;
}
return true;
}

```

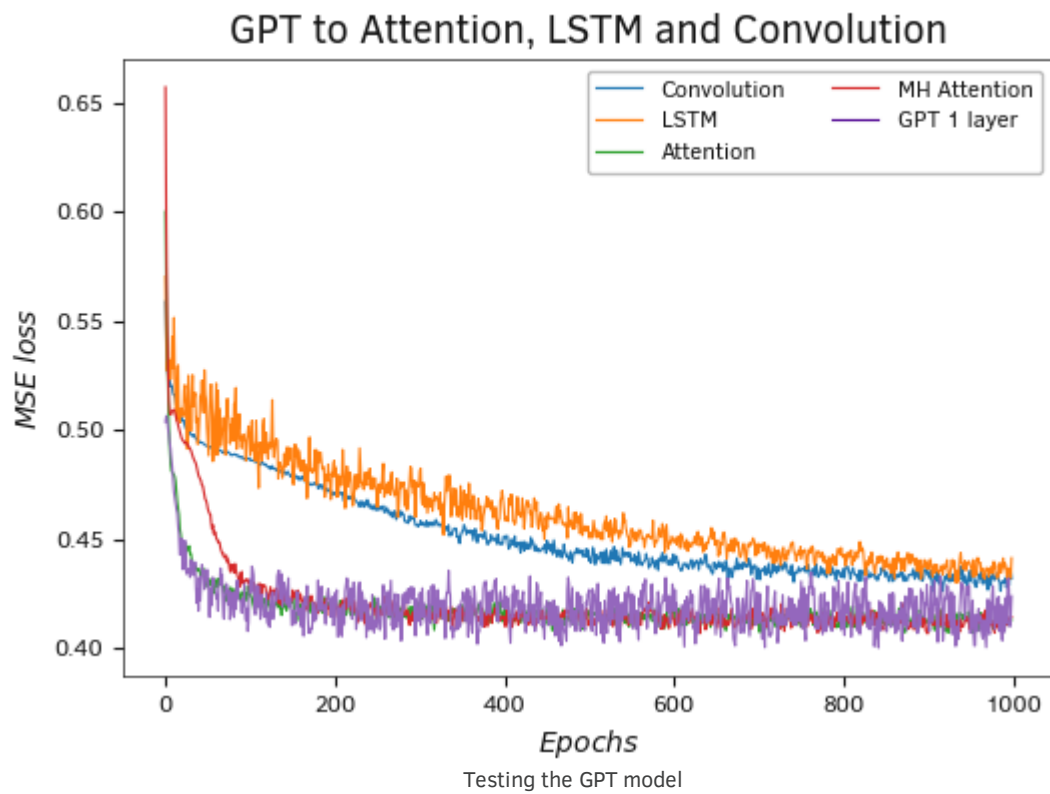
Meanwhile, we monitor the process of operations.

Further script code is transferred in an unchanged form.

Now we will run the script and compare the results with those obtained earlier when testing the previous models.

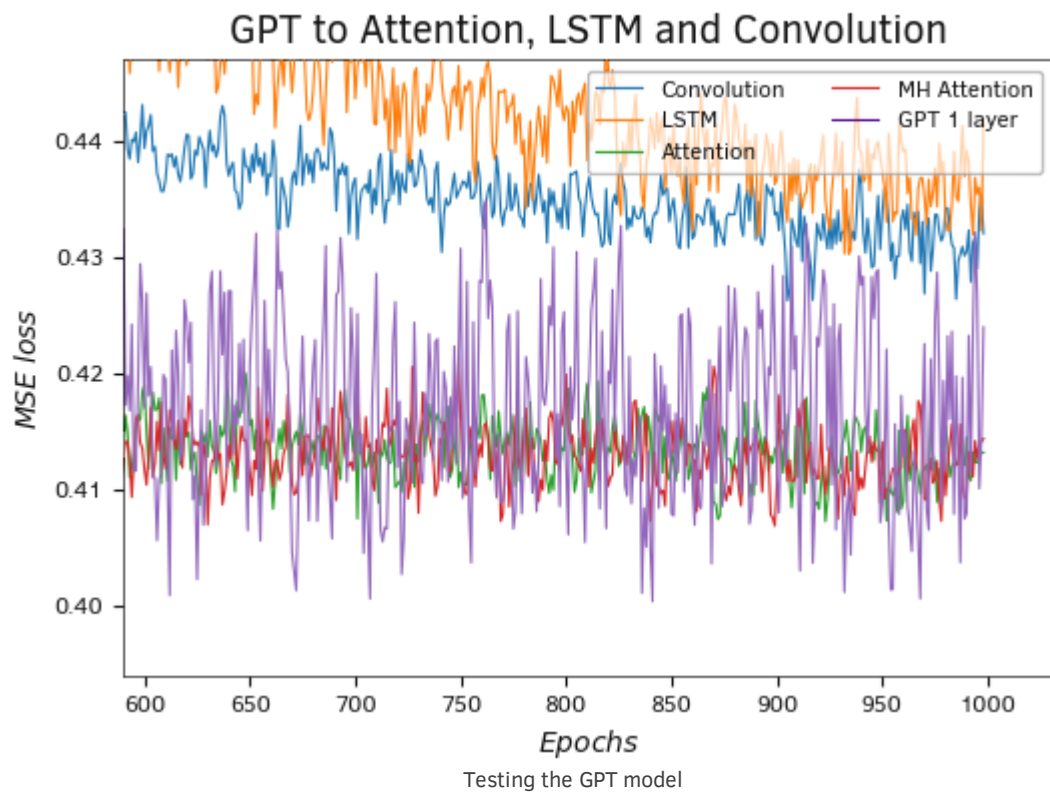
We performed the first test with all training parameters intact and a single layer in the *GPT* block. The graph of the model error in the learning dynamics has relatively large fluctuations. This can be caused by the unevenness of the data distribution between weight matrix updates due to a lack of data shuffling and a reduction in the amount of data fed into the model, which results in a decreased gradient error propagation to the weight matrix at each feed-forward iteration. I would like to remind you that during the implementation of the model, we discussed the issue of gradient error propagation only within the scope of the current state.

At the same time, despite the significant noise, the proposed architecture raises the quality bar for the model performance. It demonstrates the highest performance among all the models considered.



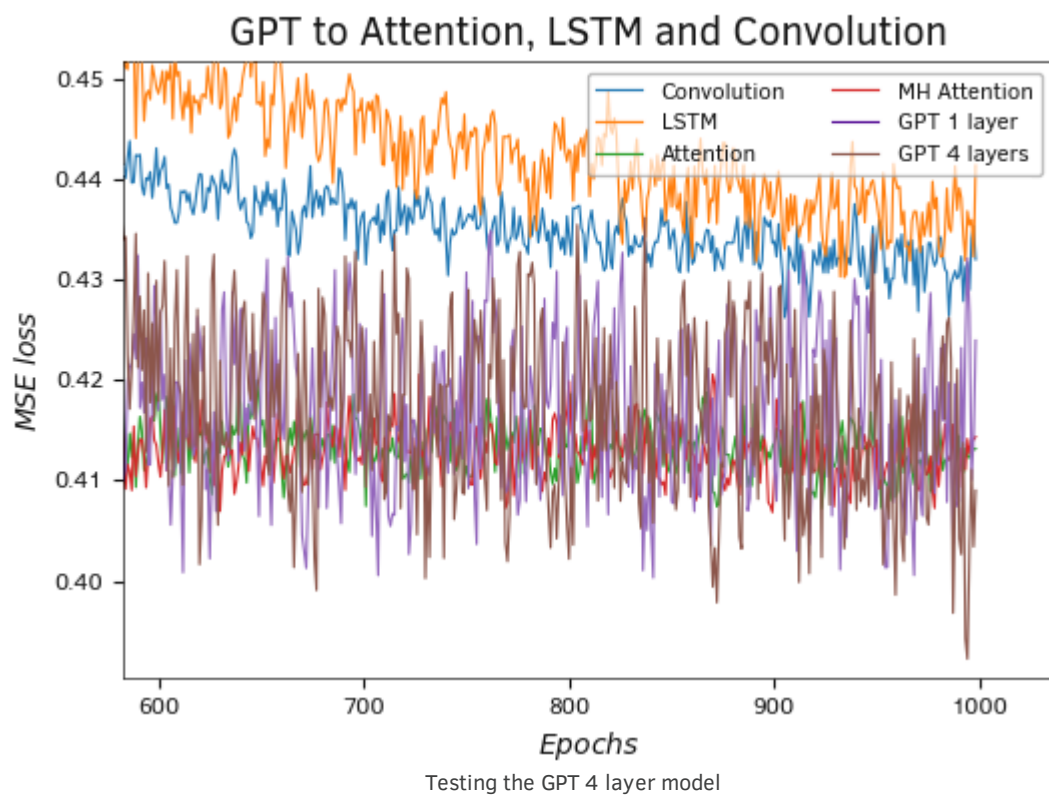
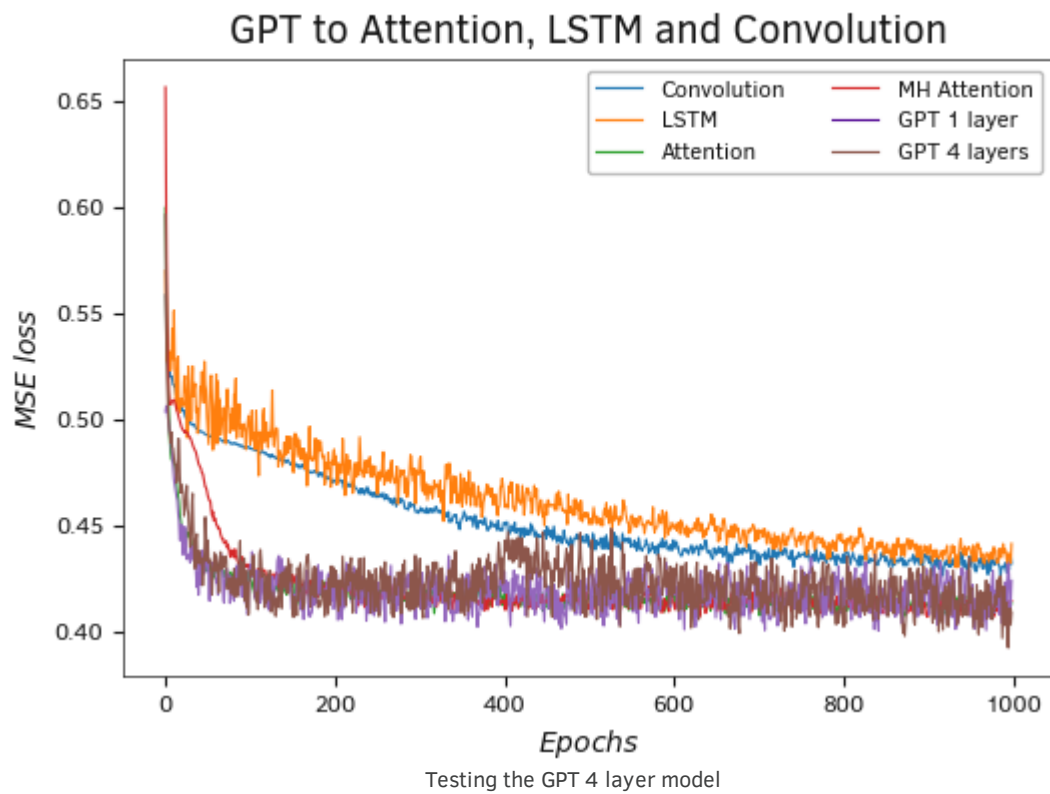
Zooming up the graph demonstrates how well the model lowers the minimum error threshold.

Here it should be added that during testing, we trained our model "from scratch". The authors of the architecture suggest unsupervised pre-training of the *GPT* block on a large dataset and then fine-tuning the pre-trained model for specific tasks during supervised learning.



Let's continue testing our implementation. All known implementations of the *GPT* architecture use multiple blocks of this architecture. For the next test, we increased the number of layers in the *GPT* block to four. The rest of the script parameters are left unchanged.

The testing results were as expected. Increasing the number of neural layers invariably leads to an increase in the total number of model parameters. A larger number of parameters requires a greater number of update iterations to achieve optimal results. In doing so, the model learns better to separate patterns and is more prone to overlearning. This is what the results of the model training demonstrated. We see the same noise in the error plot. In addition, we observe an even greater reduction in the minimum error metrics of the model.

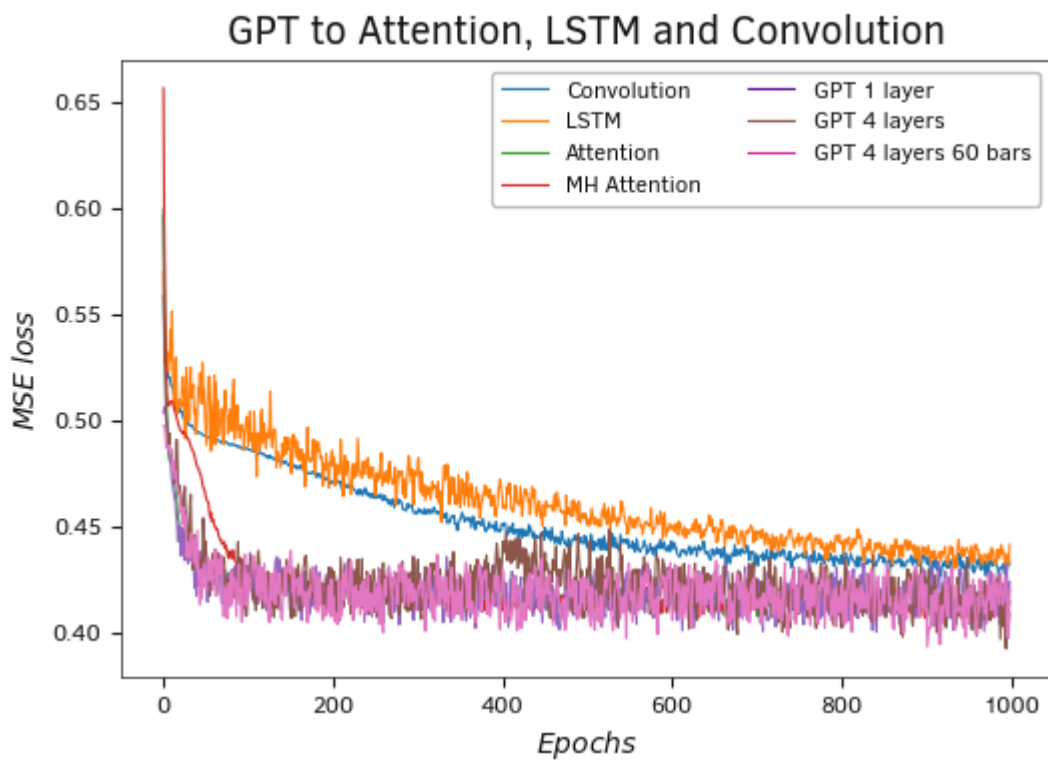


We would like to add that, from the practice of using attention models, their benefits are most clearly demonstrated when long sequences are used. *GPT* is no exception. It's more like the other way around. Since the model recalculates only the current state and uses archived copies of previous states, this

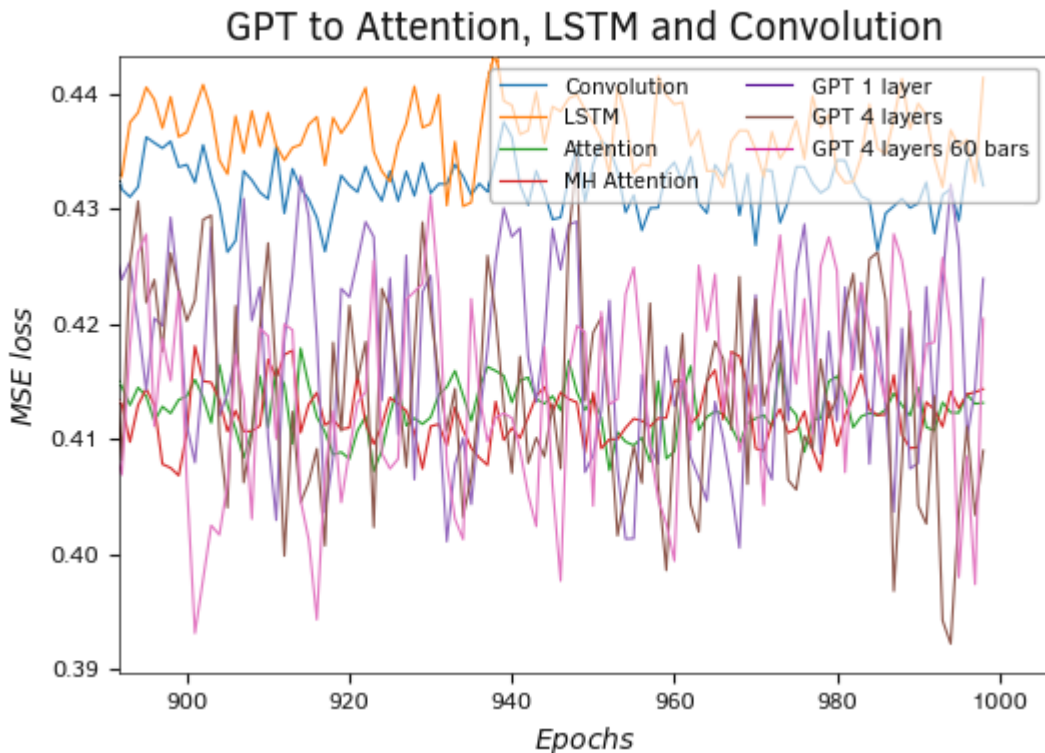
significantly reduces the number of operations when analyzing large sequences. As the number of iterations decreases, the speed of the whole model increases.

For the next test, we increase the stack size to 60 candles. Thanks to the architectural design of *GPT*, we can increase the length of the analyzed sequence by simply increasing one external parameter without changing the program code. Among other things, we do not need to change the amount of data fed to the model input. It should be noted that changing the stack size does not change the number of model parameters. Yes, increasing the stack of *Key* and *Value* tensors leads to an increase in the *Score* vector of dependency coefficients. But there is absolutely no change in any of the weight matrices of the internal neural layers.

The test results demonstrated a reduction in the model's performance error. Moreover, the overall trend suggests that there is a high likelihood of seeing improved results from the model as we continue with further training.



Testing the GPT model with an enlarged stack



We have constructed yet another architectural model of a neural layer. The testing results of the model using this new architectural solution demonstrate significant potential in its utilization. At the same time, we employed small models with rather short training periods. This is sufficient for demonstrating the functionality of architectural solutions but not adequate for real-world data usage. As practice shows, achieving optimal results requires various experiments and unconventional approaches. In most cases, the best results are achieved through a blend of different architectural solutions.

6. Architectural solutions for improving model convergence

We have discussed several different architectural solutions for neural layers. We have created classes to implement the discussed architectural solutions and small models using them. However, in the process of studying neural networks, we cannot bypass the issue of improving neural network convergence. We have considered [theoretical aspects](#) of such practices but have not yet implemented them in any model.

In this chapter, we dive deeper into the construction and application of architectural solutions improving the convergence of neural networks, such as Batch Normalization and Dropout techniques. Regarding [batch normalization](#), we start by examining its basic principles, then proceed to a detailed discussion of creating a [batch normalization class using the MQL5 programming language](#), including forward and backward pass methods, as well as file handling methods. The section also covers [multi-threaded computing](#) in the context of batch normalization and provides the implementation of this approach in Python, including the creation of a script for testing. An important part of the discussion is the [comparative testing of models using batch normalization](#), showing the practical effectiveness of the approaches considered.

Moving on to the topic of [Dropout](#), we will examine its implementation in MQL5, including feed-forward, backpropagation and file handling methods. We will see multi-threaded operations for the Dropout

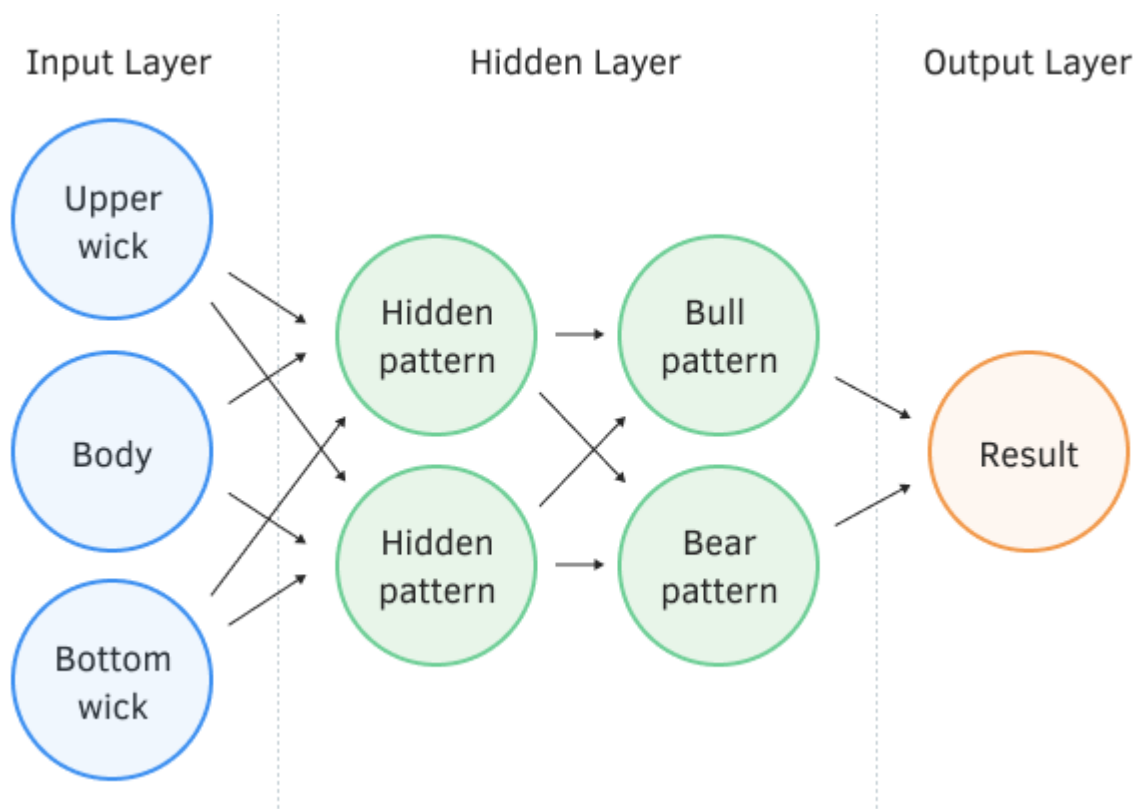
mechanism and its [implementation in Python](#). As we close this chapter, we will conduct [comparative testing of models using Dropout](#) and see the impact of this technique on the convergence and efficiency of neural networks. Thus, we will not only study the theoretical aspects but will also practically apply them to improve the performance and convergence of models.

6.1 Batch normalization

One such practice is [batch normalization](#). It's worth noting that data normalization is quite common in neural network models in various forms. Remember when we created our first fully connected perceptron model, one of the tests involved comparing the model performance on the training dataset with normalized and non-normalized data. Testing showed the advantage of using normalized data.

We also encountered data normalization when studying attention models. The *Self-Attention* mechanism uses data normalization at the output of the Attention block and at the output of the *Feed Forward* block. The difference from the previous normalization is in the area of data normalization. In the first case, we took each individual parameter and normalized its values with respect to historical data, while in the second case, we didn't look at the history of values for a single indicator; on the contrary, we took all the indicators at the current moment and normalized their values within the context of the current state. We can say that the data was normalized along the time interval and across it. The first option refers to batch data normalization, and the second is called *Layer Normalization*.

However, there are other possible uses for data normalization. Let me remind you of the main problem solved by data normalization. Consider a fully connected perceptron with two hidden layers. With a forward pass, each layer generates a set of data that serves as a training sample for the next layer. The output layer result is compared with reference data, and during the backpropagation pass, the error gradient is propagated from the output layer through the hidden layers to the input data. Having obtained the error gradient on each neuron, we update the weights, adjusting our neural network to the training samples from the last forward pass. Here lies a conflict: we are adapting the second hidden layer to the data output of the first hidden layer, while by changing the parameters of the first hidden layer, we have already altered the data array. That is, we adjust the second hidden layer to the dataset that no longer exists. A similar situation arises with the output layer, which adapts to the already altered output of the second hidden layer. If you also consider the distortion between the first and second hidden layers, the error scales increase. Furthermore, the deeper the neural network, the stronger the manifestation of this effect. This phenomenon is called the internal covariance shift.



In classical neural networks, the mentioned problem was partially addressed by reducing the learning rate. Small changes in the weights do not significantly change the distribution of the dataset at the output of the neural layer. However, this approach does not solve the problem of scaling with an increase in the number of layers in the neural network as it reduces the learning rate. Another problem with a low learning rate is the risk of getting stuck in local minima.

In February 2015, Sergey Ioffe and Christian Szegedy proposed a *Batch Normalization* method to solve the problem of internal covariance shift. The idea of the method was to normalize each individual neuron on a certain time interval with the median of the sample shifting to zero and scaling the dataset variance to one.

Experiments conducted by the method authors demonstrate that the use of the *Batch Normalization* method also acts as a regularizer. With this, there is no need to use other regularization methods, in particular *Dropout*. Moreover, there are more recent studies that show that the combined use of *Dropout* and *Batch Normalization* adversely affects the training results of a neural network.

In modern neural network architectures, variations of the proposed normalization algorithm can be found in various forms. The authors suggest using *Batch Normalization* immediately before non-linearity (activation formula).

6.1.1 Principles of batch normalization implementation

The authors of the method proposed the following normalization algorithm. First, we calculate the average value from the data sample.

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i$$

where:

- μ_B = arithmetic mean of the dataset
- m = dataset size (batch)

Then we calculate the variance of the initial sample.

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

Next, we will normalize the dataset by making it have a zero mean and a unit variance.

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

Please note that a small positive number ε is added to the denominator of the dataset variance to avoid division by zero.

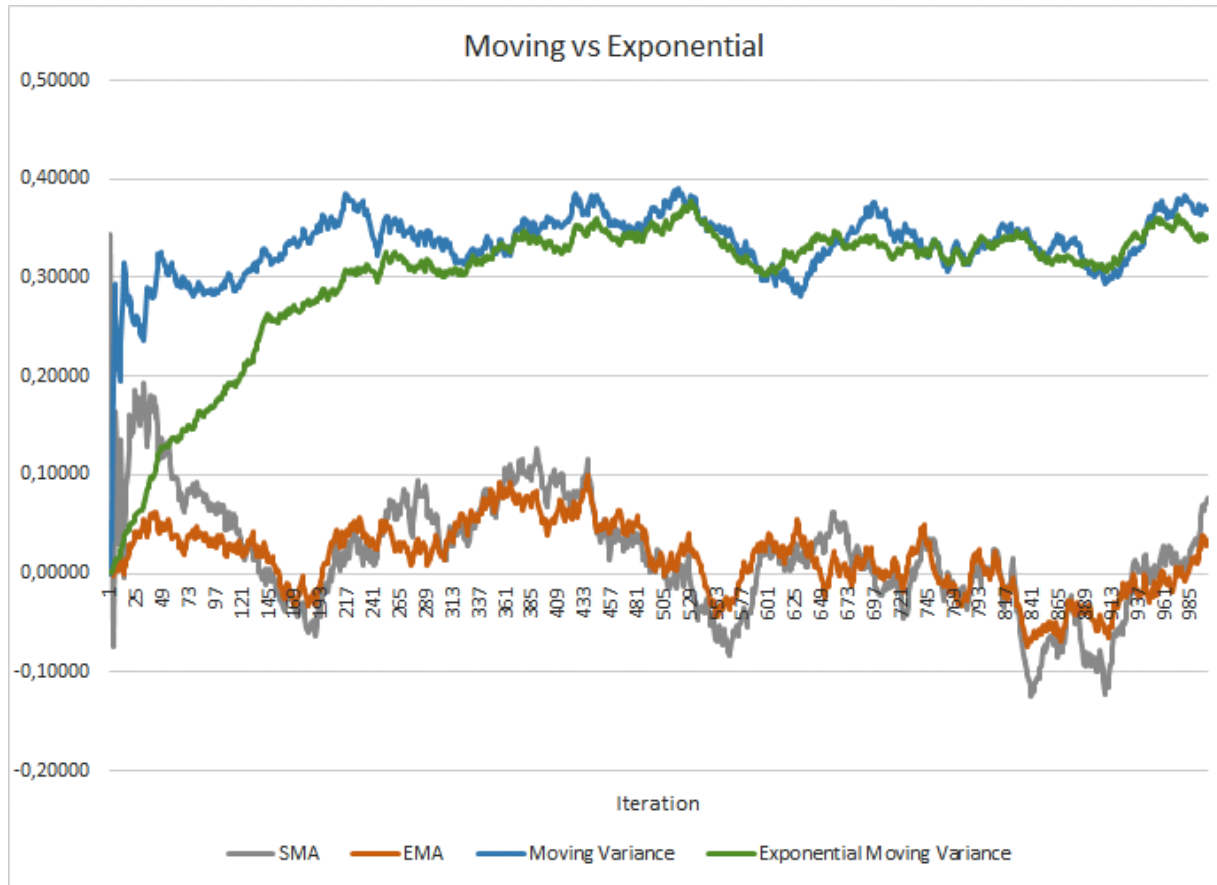
However, it has been found that such normalization can distort the impact of the initial data. Therefore, the authors of the method added another step that includes scaling and shifting. They introduced variables γ and β , which are trained along with the neural network using gradient descent.

$$y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma\beta}(x_i)$$

The application of this method allows obtaining a dataset with the same distribution at each training step, which, in practice, makes neural network training more stable and enables an increase in the learning rate. Overall, this can improve the training quality while reducing the time and computational resources required for neural network training.

However, at the same time, the costs of storing additional coefficients increase. Additionally, the calculation of moving averages and variances requires additional memory allocation to store historical data for each neuron across the entire batch size. Here you can look at the exponential average. To calculate *EMA* (Exponential Moving Averages), we only need the previous value of the function and the current element of the sequence.

The figure below provides a visual comparison of moving averages and moving variances for 100 elements against exponential moving averages and exponential moving variances for the same 100 elements. The graph was plotted for 1,000 random items in the range from -1.0 to 1.0.



Comparison of moving and exponential average graphs

As seen in the graph, the moving average and exponential moving average converge after around 120-130 iterations, and beyond that point, there is minimal deviation that can be disregarded. The exponential moving average chart also looks smoother. To calculate *EMA*, you only need the previous value of the function and the current element of the sequence. Let me remind you of the exponential moving average formula.

$$\mu_i = \frac{m-1}{m} \mu_{i-1} + \frac{1}{m} x_i = \frac{\mu_{i-1}(m-1) + x_i}{m}$$

where:

- μ_i = exponential average of the sample at the i th step
- m = dataset size (batch)
- x_i = current value of the indicator

To align the plots of moving variance and exponential moving variance, it took slightly more iterations (around 310-320), but overall the picture is similar. In the case of variance, the use of exponential moving averages not only saves memory but also significantly reduces the number of computations since for moving variance, we would need to recalculate the deviation from the mean for the entire batch of historical data, which can be computationally expensive.

In my opinion, the use of such a solution significantly reduces memory usage and the computational overhead during each iteration of the forward pass.

6.1.2 Building a batch normalization class in MQL5

After considering the theoretical aspects of the normalization method, we will move on to its practical implementation within our library. To do this, we will create a new *CNeuronBatchNorm* class derived from the *CNeuronBase* base class of the fully connected neural layer.

To ensure the full functionality of our class, we need to add a few things. We will add just one buffer for recording normalization parameters for each element of the sequence and a variable to store the batch size for normalization. For the rest, we will use base class buffers with minor amendments. We will talk about them during the implementation of the methods.

```
class CNeuronBatchNorm      : public CNeuronBase
{
protected:
    CBufferType      m_cBatchOptions;
    uint             m_iBatchSize;      // batch size

public:
    CNeuronBatchNorm(void);
    ~CNeuronBatchNorm(void);

    //---
    virtual bool      Init(const CLayerDescription* description) override;
    virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
    virtual bool      FeedForward(CNeuronBase* prevLayer) override;
    virtual bool      CalcHiddenGradient(CNeuronBase* prevLayer) override;
    virtual bool      CalcDeltaWeights(CNeuronBase* prevLayer, bool read) override;
    //--- methods for working with files
    virtual bool      Save(const int file_handle) override;
    virtual bool      Load(const int file_handle) override;
    //--- object identification method
    virtual int        Type(void) override const { return(defNeuronBatchNorm); }
};
```

We'll be redefining the same set of basic methods:

- *Init* – method for initializing a class instance
- *FeedForward* – feed-forward method
- *CalcHiddenGradient* – method of distributing error gradients through a hidden layer
- *CalcDeltaWeights* – method for distributing error gradients to the weight matrix
- *Save* – method for saving neural layer parameters
- *Load* – method for restoring the neural layer performance from the saved data

Let's start working on the class with its constructor. In this method, we only set an initial value for the normalization batch size. The class destructor remains empty.

```
CNeuronBatchNorm::CNeuronBatchNorm(void) : m_iBatchSize(1)
{
}
```

After that, we move on to working on the class initialization method. But before we start implementing this method, let's pay attention to the nuances of our implementation.

First of all, the normalization method does not involve changing the number of elements. The output of the neural layer will have the same number of neurons as the input. Therefore, the size of the source data window should be equal to the number of neurons in the layer being created. Of course, we can ignore the source data window size parameter and only use the number of neurons in the layer. However, in this case, we would lose additional control during the neural layer initialization stage and would have to constantly check whether the number of neurons matches during each feed-forward and backpropagation pass.

The second point is related to the lack of a matrix of weights in our usual form. Let's look at mathematical formulas again.

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma\beta}(x_i)$$

To calculate the normalized value, we use only the mean and standard deviation, which are calculated for the dataset and do not have adjustable parameters. We have only two configurable parameters when we shift and scale the values of γ and β . Both parameters are selected individually for each value from the source data tensor.

Now let's remember the mathematical formula for a displaced neuron.

$$Output = \sum_{i=1}^N w_i x_i + \beta$$

Don't you think that when $N = 1$, the formulas will look identical? We will use this similarity.

$$Output = wx + \beta \leftrightarrow \gamma \hat{x} + \beta = BN_{\gamma\beta}(x)$$

Now let's get back to our method of initializing an object instance. This method is virtual and inherits from the parent class. According to the rules of inheritance, this method stores the return type and the list of method parameters. The parameters of our method contain only one pointer to the object describing the neural layer being created.

In the body of the method, we immediately check the received pointer to the description object of the created neural layer, while also simultaneously verifying the correspondence between the size of the input data window and the number of neurons in the created layer. We discussed this point a little earlier.

After successfully checking the obtained object, we change the size of the initial data window by one in accordance with the similarity shown above. Now we call the parent class initialization method, remembering to check the results of the operations.

```
bool CNeuronBatchNorm::Init(const CLayerDescription *description)
{
    if(!description ||
        description.window != description.count)
        return false;
    CLayerDescription *temp = new CLayerDescription();
```

```

    if(!temp || !temp.Copy(description))
        return false;
    temp.window = 1;
    if(!CNeuronBase::Init(temp))
        return false;
    delete temp;

```

It should be noted here that during the initialization of the parent class, the weight matrix is initialized with random values. However, for batch normalization, the recommended initial values are 1 for the scaling coefficient γ and 0 for the offset β . As an experiment, we can leave it as it is, or we can fill the weight matrix buffer now.

```

//--- initialize the training parameter buffer
if(!m_cWeights.m_mMatrix.Fill(0))
    return false;
if(!m_cWeights.m_mMatrix.Col(VECTOR::Ones(description.count), 0))
    return false;

```

After successfully initializing the objects of the parent class, we proceed to create objects and specify initial values for the variables and constants of the new class.

First, we initialize the normalization parameter buffer. In this buffer, we need three elements for each element in the sequence. There we will save:

0. μ – average value from previous iterations of the forward pass.
1. σ^2 – dataset variance over previous iterations of the forward pass.
2. $\hat{x}$ – normalized value before scaling and shifting.

I deliberately numbered the values starting from 0. This is exactly the indexing that values in our data buffer will get. At the initial stage, we initialize the entire buffer with zero values and check the results of the operations.

```

//--- initialize the normalization parameter buffer
if(!m_cBatchOptions.BufferInit(description.count, 3, 0))
    return false;
if(!m_cBatchOptions.Col(VECTOR::Ones(description.count), 1))
    return false;

```

At the end of the initialization method of our class, we save the batch normalization size into a specially created variable. We then exit the method with a positive result.

```

    m_iBatchSize = description.batch;
//---
    return true;
}

```

At this point, we conclude our work with the auxiliary initialization methods and move on to building the algorithms for the class. As always, we will begin this work by constructing a method for the feed-forward pass.

6.1.2.1 Batch normalization feed-forward methods

We continue moving forward along the path of building the batch normalization class, and simultaneously, along the path of understanding the structure and methods of organizing neural networks. Earlier, we discussed various architectures for constructing neural layers to solve practical tasks. However, the operation of the batch normalization layer is equally important in organizing the functioning of a neural network, although its task may not be immediately apparent. Rather, it is hidden within the organization of the processes of the neural network itself and serves more for the stability of our model.

We have already built the class initialization methods. Now it's time to build the algorithm of method operation directly. We begin this process with the *FeedForward* method. This method is declared virtual in the *CNeuronBase* neural layer base class of our library and is overridden in each new class.

I would like to remind you that this approach allows us to eliminate the use of dispatch methods and functions for reallocating information flows and calling various methods depending on the class of the object being used. In practice, we can simply pass a pointer to any derived object into a local variable of the base class of the neural layer and call the method declared in the base class. At the same time, the system will perform all dispatching functions without our participation. It will call the method related to the actual type of the object.

This property is exactly what we exploit when expecting to receive a pointer to an object of the base class of the neural layer in the method parameters. At the same time, a pointer to any of the neural layer objects in our library can be passed in the parameters. We can work with it through the use of overridden virtual functions.

The operation of the feed-forward method itself starts with a control block for checking pointers to the objects used by the method. Here we check both the pointer to the object of the previous layer obtained in the parameters and pointers to internal objects.

```
bool CNeuronBatchNorm::FeedForward(CNeuronBase *prevLayer)
{
    //--- control block
    if(!prevLayer || !prevLayer.GetOutputs() || !m_cOutputs ||
        !m_cWeights || !m_cActivation)
        return false;
```

Please note that along with other objects, we also check the pointer to the activation function object. Although the batch normalization algorithm does not use an activation function, we will not limit the user's capabilities and will provide them with the option to use an activation function as they see fit. Moreover, there are practical cases where applying an activation function after data normalization is beneficial. For example, the method authors recommend normalizing data immediately before applying the activation function. At first glance, applying such an approach would require modifications to every previously discussed class. However, we can implement the same functionality without modifying the existing classes. We simply need to declare the required neural layer without an activation function, followed by a normalization layer with the desired activation function. Therefore, I believe the use of the activation feature in our class is justified.

Next, we will branch the algorithm for a case when the normalization batch size is equal to 1 or less. It should be understood that when the batch is equal to 1, no normalization is performed, and we simply pass the tensor of the original data to the output of the neural layer. After completing the data copy from the buffer, we call the activation method and exit the method after verifying the results of the operations.

6. Architectural solutions for improving model convergence

```
//--- check the size of the normalization batch
if(m_iBatchSize <= 1)
{
    m_cOutputs.m_mMatrix = prevLayer.GetOutputs().m_mMatrix;
    if(m_cOpenCL && !m_cOutputs.BufferWrite())
        return false;
    if(!m_cActivation.Activation(m_cOutputs))
        return false;
    return true;
}
```

Next, we need to construct the algorithm of the method. Following the concept we have adopted, we will create two variants of the algorithm implementation: by standard MQL5 tools and in the multi-threaded calculations mode using OpenCL. Therefore, next, we create another branching of the algorithm depending on the user's choice of the computational device. In this section, we will consider the construction of the algorithm using MQL5. In further sections, we will return to the construction of the algorithm using OpenCL.

```
//--- branching of the algorithm over the computing device
if(!m_cOpenCL)
{
```

We start the block of operations using MQL5 with a small preparatory work. To simplify the process of accessing the data, we save a sequence of raw data into a local matrix.

```
MATRIX inputs = prevLayer.GetOutputs().m_mMatrix;
if(!inputs.Reshape(1, prevLayer.Total()))
    return false;
```

According to the data normalization algorithm, we find the mean value. In considering the architecture of our solution, we have decided to use an exponential moving average, which is determined by the formula.

$$\mu_i = \frac{\mu_{i-1}(m-1) + x_i}{m}$$

```
VECTOR mean = (m_cBatchOptions.Col(0) * ((TYPE)m_iBatchSize - 1.0) +
    inputs.Row(0)) / (TYPE)m_iBatchSize;
```

After determining the moving average, we find the average variance.

```
VECTOR delt = inputs.Row(0) - mean;
VECTOR variance = (m_cBatchOptions.Col(1) * ((TYPE)m_iBatchSize - 1.0) +
    MathPow(delt, 2)) / (TYPE)m_iBatchSize;
```

Once the mean and variance values are found, we can easily compute the normalized value of the current element in the sequence.

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

```
VECTOR std = sqrt(variance) + 1e-32;
VECTOR nx = delt / std;
```

Note that we add a small constant to the variance to eliminate the potential zero division error.

The next step of the batch normalization algorithm is shift and scaling.

```
VECTOR res = m_cWeights.Col(0) * nx + m_cWeights.Col(1);
```

After that, we only need to save the obtained values into the respective elements of the buffers. Please note that we save not only the results of the algorithm operations in the result buffer but also our intermediate values in the normalization parameters buffer. We will need them in subsequent iterations of the algorithm. Do not forget to check the results of the operations.

```
if(!m_cOutputs.Row(res, 0) ||
    !m_cBatchOptions.Col(mean, 0) ||
    !m_cBatchOptions.Col(variance, 1) ||
    !m_cBatchOptions.Col(nx, 2))
    return false;
}
else // OpenCL block
{
    return false;
}
```

This completes the algorithm splitting depending on the computing device used. As always, we will set a temporary stub for the OpenCL block in the form of a false value return. We will return to this part later.

Now, before exiting the method, we activate the values in the result buffer of our class. To do this, we call the *Activation* method of our special object to work with the *m_cActivation* activation function. After checking the result of the operation, we terminate the method.

```
if(!m_cActivation.Activation(m_cOutputs))
    return false;
//---
return true;
}
```

With that, we conclude our work on the feed-forward method of the *CNeuronBatchNorm* batch normalization class. I hope that understanding the logic behind its construction wasn't difficult for you. Now, let's move on to building the backpropagation methods.

6.1.2.2 Batch normalization backpropagation methods

In the previous sections, we began studying the algorithm of the batch normalization method. To implement it in our library, we have created a separate neural layer in the form of the *CNeuronBatchNorm* class and have even built methods for initializing the class of the feed-forward algorithm. Now it's time to move on to building the backpropagation algorithm for our class. Let me remind you that the backpropagation algorithm in all neural layers of our library is represented by four virtual methods:

- The ***CalcOutputGradient*** method for calculating the error gradient at the output of the neural network,
- The ***CalcHiddenGradient*** method for propagating the gradient through the hidden layer,
- The ***CalcDeltaWeights*** method for calculating weight adjustment values, and
- The ***UpdateWeights*** method for updating the weight matrix.

All of them were declared in our *CNeuronBase* neural layer base class. They are overridden in each new class as needed.

In this class, we will override only two methods: error gradient propagation through the hidden layer and the calculation of weight adjusting values.

We will not override the error gradient method at the output of the neural network because I do not know of a scenario where it would be necessary to use batch normalization as the last layer of a neural network. Moreover, experiments show that the use of batch normalization immediately before the neural network result layer can adversely affect the results of the model.

As for the method for updating the weight matrix, we intentionally designed the operation of the buffer for the matrix of trainable parameters in such a way that it became possible to use the method from the parent class to update its parameters.

Now let's move on to the practical part and look at the implementation of the specified *CalcHiddenGradient* backpropagation methods. This is a virtual method that was defined in the *CNeuronBase* neural layer base class. The method is overridden in each new class of the neural layer to implement a specific algorithm. In the parameters, the method receives a pointer to the object of the previous neural layer and returns the logical result of the operations.

In the method body, we add a control block in which we check the validity of pointers both to the previous layer object received in the parameters and to the internal objects used in the method operation. We have talked about the importance of such a process on multiple occasions because accessing an object through an invalid pointer leads to a critical error and a complete termination of the program.

```
bool CNeuronBatchNorm::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    //--- control block
    if(!prevLayer || !prevLayer.GetOutputs() || !prevLayer.GetGradients() ||
        !m_cActivation || !m_cWeights)
        return false;
```

Next, we need to adjust the error gradient obtained from the next layer to the derivative of the activation function of our layer. In the base class, we have encapsulated all the work with the activation function into a separate object of the *CActivation* class. Therefore, now, to adjust the error gradient, we

should simply call the appropriate method of this class and provide a pointer to the error gradient buffer of our class as a parameter. As always, do not forget to check the result of the operation.

```
//--- adjust the error gradient to the derivative of the activation function
if(!m_cActivation.Derivative(m_cGradients))
    return false;
```

After that, we check the size of the specified normalization batch. If it is not more than one, simply copy the gradient buffer data of the current layer to the buffer of the previous layer. Then we exit the method with the result of copying the data.

```
//--- check the size of the normalization batch
if(m_iBatchSize <= 1)
{
    prevLayer.GetGradients().m_mMatrix = m_cGradients.m_mMatrix;
    if(m_cOpenCL && !prevLayer.GetGradients().BufferWrite())
        return false;
    return true;
}
```

Next, we sequentially calculate the gradients for all functions of the algorithm.

$$\Delta \hat{x} \rightarrow \Delta \sigma^2 \rightarrow \Delta \mu \rightarrow \Delta x_i$$

I suggest going through the process and looking at the mathematical formulas for the propagation of the error gradient. At the initial stage, we have the error gradient for the results of our normalization layer, which corresponds to the scaling and shifting function values. Let me remind you of the formula:

$$BN_{\gamma\beta}(x_i) = \gamma \hat{x}_i + \beta$$

To adjust the error gradient, we need to multiply it by the derivative of the function. According to the rules for calculating the derivative for $\hat{x}_i$, the shift β acts as a constant and its derivative is zero. The derivative of the product is equal to the second factor. Thus, our derivative will be equal to the scaling factor γ .

$$\begin{aligned} (BN_{\gamma\beta}(x_i))' &= \gamma \\ \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} &= \gamma G_i \end{aligned}$$

where G_i is the gradient of the i th element at the output of the scaling and shift function.

In the method code, this operation will be expressed in the following lines.

```
//--- branching of the algorithm by computing device
if(!m_cOpenCL)
{
    MATRIX mat_inputs = prevLayer.GetOutputs().m_mMatrix;
    if(!mat_inputs.Reshape(1, prevLayer.Total()))
        return false;
    VECTOR inputs = mat_inputs.Row(0);
    CBufferType *inputs_grad = prevLayer.GetGradients();
    ulong total = m_cOutputs.Total();
```

6. Architectural solutions for improving model convergence

```
VECTOR gnx = m_cGradients.Row(0) * m_cWeights.Col(0);
```

Let's move on. We determine the normalized value using the formula.

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

From here, we need to distribute the error gradient to each of the components. I will not show the entire process of deriving partial differential formulas. I will only provide a ready-made formula for calculating the error gradient presented by the authors of the method in the article [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#).

$$\begin{aligned} \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} &= \sum_{i=1}^m \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * (x_i - \mu) * \frac{-1}{2} (\sigma^2 + \varepsilon)^{-\frac{3}{2}} \\ \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \mu} &= \left(\sum_{i=1}^m \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * \frac{-1}{\sqrt{\sigma^2 + \varepsilon}} \right) + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} * \frac{\sum_{i=1}^m -2(x_i - \mu)}{m} \\ \frac{\partial BN_{\gamma\beta}(x_i)}{\partial x_i} &= \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * \frac{1}{\sqrt{\sigma^2 + \varepsilon}} + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} * \frac{2(x_i - \mu)}{m} + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \mu} * \frac{1}{m} \\ \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \gamma} &= \sum_{i=1}^m \partial G_i * \hat{x}_i \\ \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \beta} &= \sum_{i=1}^m \partial G_i \end{aligned}$$

The last two formulas will be needed for the next method, where we will propagate the error gradient to the level of the trainable parameter matrix. Therefore, in the code of this method, we implement only the formulas given above.

```
VECTOR temp = MathPow(MathSqrt(m_cBatchOptions.Col(1) + 1e-32), -1);
VECTOR gvar = (inputs - m_cBatchOptions.Col(0)) /
    (-2 * pow(m_cBatchOptions.Col(1) + 1.0e-32, 3.0 / 2.0)) * gnx;
VECTOR gmu = temp * (-1) * gnx - gvar * 2 *
    (inputs - m_cBatchOptions.Col(0)) / (TYPE)m_iBatchSize;
VECTOR gx = temp * gnx + gmu / (TYPE)m_iBatchSize + gvar * 2 *
    (inputs - m_cBatchOptions.Col(0)) / (TYPE)m_iBatchSize;
```

Note that the formulas are the sums of the values across the entire normalization dataset. We perform calculations only for the current value. Nevertheless, we do not deviate from the above formulas. The reason is that our dataset is stretched over time, and we return the error gradient at each step. During the period between updates of the trainable parameters of our class, we accumulate the error gradient on them, thereby summing it over the entire duration of our normalization dataset stretched along the time scale.

Now we only need to save the obtained error gradient into the corresponding element of the buffer and check the result of the operation.

```
if(!inputs_grad.Row(gx, 0))
```

```

        return false;
    if(!inputs_grad.Reshape(prevLayer.Rows(), prevLayer.Cols()))
        return false;
    }
    else // OpenCL block
    {
        return false;
    }
    //---
    return true;
}

```

As a result of performing these operations, we obtained a filled buffer for the gradient tensor of the previous layer. So, the task set for this method has been completed, and we can conclude the branching of the algorithm depending on the used device. We will set a temporary stub for the block of organizing multi-threaded computing using OpenCL, as in similar cases when working with other methods. Thus, we finish working on our *CNeuronBatchNorm::CalcHiddenGradient* method at this point.

We will continue to organize the process of the backpropagation pass. Let's move on to the next method *CNeuronBatchNorm::CalcDeltaWeights*. Usually, this method is responsible for distributing the error gradient to the level of the weight matrix. But in our case, we have slightly different trainable parameters, on which we will distribute the error gradient.

The *CalcDeltaWeights* method like the previous one, receives a pointer to the object of the previous layer in the parameters. However, in this case, it is more of a fulfillment of the requirement of method inheritance than a functional necessity. The formulas for propagating the error gradient to trainable variables have already been provided above, but I will list them again for reference.

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \gamma} = \sum_{i=1}^m \partial G_i * \hat{x}_i$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \beta} = \sum_{i=1}^m \partial G_i$$

As can be seen from the above formulas, the error gradient of the parameters does not depend on the values of the previous layer. The gradient of the scaling coefficient depends on the normalized value, while the gradient of the bias is equal to the error gradient at the output of the batch normalization layer. Of course, the normalized value itself depends on the values of the previous layer. However, to avoid its recalculation, we simply saved the normalized values in a buffer with a feed-forward pass. Therefore, in the body of this method, we will not refer to the elements of the previous layer. Hence, there is no point in wasting time checking the resulting pointer to the previous layer. At the same time, we will not completely exclude the control block as we check not only external pointers but also pointers to internal objects.

```

bool CNeuronBatchNorm::CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
{
    //--- control block
    if(!m_cGradients || !m_cDeltaWeights)
        return false;
}

```

After successfully passing the control block, we check the value of the normalization batch. It should be at least greater than one. Otherwise, we exit the method.

```
//--- check the size of the normalization batch
if(m_iBatchSize <= 1)
    return true;
```

After successfully passing all the controls, we proceed to the direct implementation of the method algorithm. We always implement the algorithm in two versions: using standard MQL5 tools and using multi-threaded computing technology using OpenCL. Therefore, before continuing the operations, we will create a branching of the algorithm depending on the device used for computing operations.

```
//--- branching of the algorithm by the computing device
if(!m_cOpenCL)
{
```

In the branch of algorithm implementation using standard MQL5 tools we will use matrix operations. According to the formulas provided above, we determine the error gradient for the scaling coefficient and the bias. We add the obtained values to the previously accumulated error gradients of the corresponding elements and update the values in the error gradient accumulation buffer.

```
    VECTOR grad = m_cGradients.Row(0);
    VECTOR delta = m_cBatchOptions.Col(2) * grad + m_cDeltaWeights.Col(0);
    if(!m_cDeltaWeights.Col(delta, 0))
        return false;
    if(!m_cDeltaWeights.Col(grad + m_cDeltaWeights.Col(1), 1))
        return false;
```

After completing all the operations, we will have a fully updated error gradient buffer at the level of the batch normalization layer's trainable parameters. In other words, the task for this method is solved, and we close the branch of the algorithm depending on the computing device, along with the entire method. However, first, we add a stub in the block of the multi-threaded computing algorithm using OpenCL.

```
    }
    else // OpenCL block
    {
        return false;
    }
//---
    return true;
}
```

Above, we have redefined two methods from the backpropagation algorithm. The method of updating the weights, and in this case the trained parameters, was inherited from the parent class. Thus, the work on the backpropagation methods in terms of the organization of the process using standard MQL5 tools can be considered complete. Let's move on to the file handling methods.

6.1.2.3 File operations

We are confidently approaching the completion of work on the methods of the *CNeuronBatchNorm* batch normalization class. Previously, we have already built methods for initializing the class, as well as built an algorithm for the operation of feed-forward and backpropagation passes using standard MQL5 capabilities. Let's move on to working on file handling methods. We have discussed the importance of having and correctly functioning these methods several times as the performance of these methods determines how quickly we can deploy a trained model into operational use.

We have already done similar work more than once for other classes in our library. Now we will follow the same established algorithm. First, we evaluate the need to write each element of the class to the data file. In the structure of our class, we have created only one new data buffer and one variable. Both of these elements are important for organizing correct object operations in our class. Therefore, we save both elements to a data file.

```
class CNeuronBatchNorm      : public CNeuronBase
{
protected:
    CBufferType      m_cBatchOptions;
    uint             m_iBatchSize;      // batch size

public:
                                CNeuronBatchNorm(void);
                                ~CNeuronBatchNorm(void);

    //---
    virtual bool      Init(const CLayerDescription* description) override;
    virtual bool      SetOpenCL(CMyOpenCL *openc1) override;
    virtual bool      FeedForward(CNeuronBase* prevLayer) override;
    virtual bool      CalcHiddenGradient(CNeuronBase* prevLayer) override;
    virtual bool      CalcDeltaWeights(CNeuronBase* prevLayer, bool read) override;
    //--- methods for working with files
    virtual bool      Save(const int file_handle) override;
    virtual bool      Load(const int file_handle) override;
    //--- object identification method
    virtual int        Type(void) override const {return defNeuronBatchNorm;}
};
```

Having determined the scope of our work, we now proceed directly to creating the file handling methods for our class. As always, the first step is to create the *CNeuronBatchNorm::Save* method for writing data to the file. Like all the methods we have discussed so far, this one is also created as a virtual method in the base neural layer class and is overridden in each new neural layer class to fully save all the necessary information for subsequent restoration of the correct operation of the saved objects. In parameters, the method receives a file handle of to write the data.

```
bool CNeuronBatchNorm::Save(const int file_handle)
{
    //--- call the method of the parent class
    if(!CNeuronBase::Save(file_handle))
        return false;
}
```

The obtained file handle for writing data is not checked, as this control is already implemented in the same-named method of the parent class, which is called in the body of this method. Thus, we check the result of the operations of the method of the parent class.

```

    if(!CNeuronBase::Save(file_handle))
        return false;

```

It is very convenient to use the method of the parent class. This usage serves a dual purpose. The first purpose is a control function because the parent class already implements a set of controls that do not need to be duplicated in the new method. We only need to call the parent class method and check its execution result. The second purpose is functional. The method of the parent class already stores all inherited objects and variables. Here, it's the same situation: we call the parent class method once, thereby saving all inherited objects and variables. Convenient, isn't it? Moreover, we do not need to call the method for each individual functionality. With one call, we accomplish two tasks: control and saving of inherited objects. Checking the result of the function execution confirms the correct execution of both functions of the method.

After successfully executing the parent class method, we understand that the handle to the file provided as a parameter is valid. Now we can proceed with further file operations without the risk of getting a critical error. First, we save the normalization batch size, which is stored in the *m_iBatchSize* variable. Also, we make sure to check the result of the operation.

```

//--- save the size of the normalization batch
    if(FileWriteInteger(file_handle, m_iBatchSize) <= 0)
        return false;

```

At the end of the method, we save the buffer of *m_cBatchOptions* normalization parameters. To do this, we just call the corresponding method of the specified object and check its operation result.

```

//--- save normalization settings
    if(!m_cBatchOptions.Save(file_handle))
        return false;
//---
    return true;
}

```

As you can see, by using parent class methods and internal objects, we have described the method for saving all the necessary information easily and quite concisely. The main data saving controls and operations are hidden in these methods.

Similarly, let's create a method for loading data from the *CNeuronBatchNorm::Load* file. It should be noted that this method is responsible not only for reading data from a file but also for fully restoring the functionality of the object to the state at the time of data saving. Therefore, this method should include operations for creating instances of objects required for the correct functioning of our batch normalization class. In addition, we must initialize all unsaved objects and variables with initial values.

In the parameters, the *CNeuronBatchNorm::Load* method, like the previous data saving method, receives the handle of the file with the saved data. We have to organize the reading of data from the file in strict accordance with the sequence of their writing to the file. This time, in the body of the method, we immediately call the method of the parent class. The calculation here is the same: by calling the parent class method once, we immediately execute the entire functionality with inherited objects and variables. At the same time, we only need to check the result of the parent class method once to ensure the correctness of all its operations.

```

bool CNeuronBatchNorm::Load(const int file_handle)
{
    //--- call the method of the parent class
    if(!CNeuronBase::Load(file_handle))

```

```
return false;
```

After the successful execution of the parent class method, we move on to loading the data of the objects of the batch normalization class. According to the sequence in which the data is written to the file, we first read the size of the normalization batch.

```
m_iBatchSize = FileReadInteger(file_handle);
```

Finally, it remains to load the buffer data of the *m_cBatchOptions* normalization parameters.

```
//--- initialize a dynamic array of optimization parameters
if(!m_cBatchOptions.Load(file_handle))
    return false;
//---
return true;
}
```

After successfully loading all the data, we will conclude the method execution with a positive result.

We have finished creating a batch data normalization layer using standard MQL5 tools. To complete the work on the *CNeuronBatchNorm* class, we need to supplement its functionality with the ability to perform multi-threaded mathematical operations using OpenCL. We'll do that in the next section. But now we have the opportunity to conduct the first tests.

6.1.3 Organizing multi-threaded computations in the batch normalization class

We continue working on our batch normalization class *CNeuronBatchNorm*. In the previous sections, we have already fully implemented the functionality of the class using standard MQL5 tools. In order to complete the work on the class, according to our concept, it remains to supplement its functionality with the ability to perform multi-threaded mathematical operations using OpenCL. Recall, the implementation of this functionality can be roughly divided into two sub-processes:

- Create an OpenCL program.
- Modify the methods of the main program to organize data exchange with the context and call the OpenCL program.

Let's start by creating the OpenCL program. First, we implement the *BatchNormFeedForward* forwards pass kernel. In the parameters, we pass pointers to four buffers and two constants to the kernel:

- *inputs* – buffer of raw data (previous layer results)
- *options* – normalization parameter buffer
- *weights* – trainable parameter matrix buffer (named after the class buffer)
- *output* – result buffer
- *batch* – size of the normalization batch
- *total* – size of the result buffer

```
__kernel void BatchNormFeedForward(__global TYPE *inputs,
                                   __global TYPE *options,
                                   __global TYPE *weights,
                                   __global TYPE *output,
                                   int batch,
```

```

        int total)
    {

```

The last parameter is necessary because we use vector variables of *TYPE4* type to optimize the computation process. This approach allows parallel computing not at the software level, but at the microprocessor level. The use of a vector of four elements of type *double* allows you to fully fill a 256-bit microprocessor register and perform calculations on the entire vector in a single clock cycle. Thus, in one clock cycle of the microprocessor, we perform operations on four elements of our data array. OpenCL supports vector variables of 2, 3, 4, 8, and 16 elements. Before choosing a vector dimension, please check the specifications of your hardware.

In the kernel body, we immediately identify the current thread ID. We will need it to determine the offset in the buffers of the tensors before the analyzed variables.

We also check the size of the normalization batch. If it is not greater than one, we simply copy the corresponding elements from the gradient buffer of the current layer to the gradient buffer of the previous layer and terminate further execution of the kernel.

```

    int n = get_global_id(0);
    if(batch <= 1)
    {
        D4ToArray(output, ToVect4(inputs, n * 4, 1, total, 0), n * 4, 1, total, 0);
        return;
    }

```

Please note that when calling the function to convert tensor values into vector representation and vice versa for the bias parameter in the tensor, we increase the thread identifier by a factor of four. This is because when using vector operations with *TYPE4*, each thread simultaneously processes four elements of the tensor. Therefore, the number of launched threads will be four times smaller than the size of the processed tensor.

If the normalization batch size is greater than one, and we continue program execution, we need to determine the offset in the normalization parameter tensor buffers, taking into account the identifier of the current thread and the vector operation size (*TYPE4*)

```

    int shift = n * 4;
    int shift_options = n * 3 * 4;
    int shift_weights = n * 2 * 4;

```

We now move on directly to the execution of our algorithm. First, we create a vector with the analyzed input data and calculate the exponential average. Let's use the previous mean and variance to determine the first iteration. We divide the pre-obtained averaging value by the dataset package size only on the second and subsequent iterations. This is because the average of the first element is the element itself.

After determining the mean, we find the deviation of the current value from the mean and calculate the dataset variance.

```

    TYPE4 inp = ToVect4(inputs, shift, 1, total, 0);
    TYPE4 mean = ToVect4(options, shift, 3, total * 3, 0) * ((TYPE)batch - 1) + inp ;
    if(options[shift_options] != 0 && options[shift_options + 1] > 0)
        mean /= (TYPE4)batch;
    TYPE4 delt = inp - mean;
    TYPE4 variance = ToVect4(options, shift, 3, total * 3, 1) * ((TYPE)batch - 1) + pc
    if(options[shift_options + 1] > 0)

```

```
variance /= (TYPE4)batch;
```

Having mean and sample variance, we can easily calculate the normalized value of the parameter.

```
TYPE4 nx = delt / sqrt(variance + 1e-37f);
```

Next, according to the batch normalization algorithm, it is necessary to perform the shift and scaling of the normalized value. But before that, I want to remind you that at the initial stage, we initialized the buffer of trainable parameter matrices with zero values. As such, we get 0 for all values regardless of the previously normalized value.

$$y_i = 0\hat{x}_i + 0 = 0$$

Therefore, we check for a zero value for the scaling factor and replace it with one if necessary.

```
if(weights[shift_weights] == 0)
    D4ToArray(weights, (TYPE4)1, shift, 2, total * 2, 0);
```

Note that we are checking for equality to zero only for the first element of the analyzed value vector. We will replace the entire vector with one. This approach is acceptable because I expect to get null values only on the first pass. At this point, we will have all buffer elements equal to zero and need to replace them. The coefficients will then be determined and optimized and, therefore, it will be different from zero during model training.

After such a simple operation, we can safely scale up and shift.

```
TYPE4 res = ToVect4(weights, shift, 2, total * 2, 0) * nx +
    ToVect4(weights, shift, 2, total * 2, 1);
```

Now we only need to save the received data to the appropriate buffer elements. In doing so, we maintain not only the last result but also the intermediate values we need.

```
D4ToArray(options, mean, shift, 3, total * 3, 0);
D4ToArray(options, variance, shift, 3, total * 3, 1);
D4ToArray(options, nx, shift, 3, total * 3, 2);
D4ToArray(output, res, shift, 1, total, 0);
}
```

This completes the work on the *BatchNormFeedForward* feed-forward kernel and we can move on to the work on the backpropagation kernels.

To implement the backpropagation algorithm, we create two kernels, one for the propagation of the error gradient to the level of the previous layer and the other one for the propagation of the error gradient to the level of the matrix of trainable parameters.

We start by creating an error gradient propagation kernel through a hidden layer of the *BatchNormCalcHiddenGradient* neural network. In the parameters of this method, this time we will pass five data buffers and two constants:

- *inputs* – buffer of input data (previous layer results)
- *options* – normalization parameter buffer
- *weights* – trainable parameter matrix buffer (named after the class buffer)
- *gradient* – error gradient buffer at the result level of the current layer
- *gradient_inputs* – error gradient buffer at the level of the previous layer results (in this case the kernel result)

- *batch* – size of the normalization batch
- *total* – size of the result buffer

```
__kernel void BatchNormCalcHiddenGradient(__global TYPE *options,
                                          __global TYPE *gradient,
                                          __global TYPE *inputs,
                                          __global TYPE *gradient_inputs,
                                          __global TYPE *weights,
                                          int batch,
                                          int total)
{
```

At the beginning of the kernel, as in the feed-forward kernel, we determine the current flow identifier and check the normalization batch size. If the normalization batch size is not greater than one, we simply copy the error gradients from the current layer buffer to the previous layer buffer and stop executing the kernel.

```
int n = get_global_id(0);
int shift = n * 4;
if(batch <= 1)
{
    D4ToArray(gradient_inputs, ToVect4(gradient, shift, 1, total, 0),
              shift, 1, total, 0);
    return;
}
```

If, however, the size of the normalization batch is greater than one, and we continue with the kernel operations, then we have to propagate the error gradient throughout the chain from the results level of the current layer to the results level of the previous layer. Below are the mathematical formulas we have to implement.

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} = \gamma G_i$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} = \sum_{i=1}^m \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * (x_i - \mu) * \frac{-1}{2} (\sigma^2 + \varepsilon)^{-\frac{3}{2}}$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \mu} = \left(\sum_{i=1}^m \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * \frac{-1}{\sqrt{\sigma^2 + \varepsilon}} \right) + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} * \frac{\sum_{i=1}^m -2(x_i - \mu)}{m}$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial x_i} = \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * \frac{1}{\sqrt{\sigma^2 + \varepsilon}} + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} * \frac{2(x_i - \mu)}{m} + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \mu} * \frac{1}{m}$$

```
TYPE4 inp = ToVect4(inputs, shift, 1, total, 0);
TYPE4 gnx = ToVect4(gradient, shift, 1, total, 0) *
            ToVect4(weights, shift, 2, total * 2, 0);
TYPE4 temp = 1 / sqrt(ToVect4(options, shift, 3, total * 3, 1) + 1e-37f);
TYPE4 delt = inp - ToVect4(options, shift, 3, total * 3, 0);
TYPE4 gvar = delt / (-2 * pow(ToVect4(options, shift, 3, total * 3, 1) +
                             1.0e-37f, 3.0f / 2.0f)) * gnx;
```

6. Architectural solutions for improving model convergence

```
TYPE4 gmu = (-temp) * gnx - gvar * 2 * delt / (TYPE4)batch;  
TYPE4 gx = temp * gnx + gmu/(TYPE4)batch + gvar * 2 * delt/(TYPE4)batch;
```

After the calculation, we save the result of the operations and complete the kernel.

```
D4ToArray(gradient_inputs, gx, shift, 1, total, 0);  
}
```

This completes the first kernel in implementing the backpropagation algorithm of our batch normalization class, and we move on to the final phase of the OpenCL program, which is the creation of the second backpropagation kernel in which the error gradient is propagated to the level of the *BatchNormCalcDeltaWeights* trainable parameter matrix.

We pass three data buffers to this kernel in parameters. These are:

- *options* – normalization parameter buffer
- *delta_weights* – error gradient buffer at the trainable parameter matrix level (in this case the result of the kernel)
- *gradient* – error gradient buffer at the result level of the current layer.

```
__kernel void BatchNormCalcDeltaWeights(__global TYPE *options,  
                                         __global TYPE *delta_weights,  
                                         __global TYPE *gradients)  
{
```

We need to implement only two mathematical formulas in this kernel:

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \gamma} = \sum_{i=1}^m \partial G_i * \hat{x}_i$$
$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \beta} = \sum_{i=1}^m \partial G_i$$

As you can see, the operations are quite simple and will not require too long code to implement the algorithm. This time we don't even use vector operations.

At the beginning of the kernel, we define the ID of the current thread and the offset in the buffers with the normalization parameter tensors. The offset in the error gradient buffers will match the thread ID.

```
const int n = get_global_id(0);  
int shift_options = n * 3;  
int shift_weights = n * 2;
```

To reduce global memory access, we will first save the gradient error value in a local variable, and then calculate and immediately write the corresponding values for the current step to the gradient error accumulation buffer elements using the formulas mentioned above.

```
TYPE grad = gradients[n];  
delta_weights[shift_weights] += grad * options[shift_options + 2];  
delta_weights[shift_weights + 1] += grad;  
}
```

As you can see, the error gradients are written to the corresponding buffer elements. So, the task assigned to this kernel is complete and we can finish its operation.

We have created all three kernels to implement feed-forward and backpropagation passes in our batch normalization class. We can now proceed to make changes to the main program to organize data exchange with the OpenCL context and call the corresponding program kernel.

Let's start this work as usual by creating constants for OpenCL kernels. Go to [defines.mqh](#) and add program kernel identifier constants at the beginning.

```
#define def_k_BatchNormFeedForward      37
#define def_k_BatchNormCalcHiddenGradient 38
#define def_k_BatchNormCalcDeltaWeights 39
```

Then add the kernel parameter identifiers.

```
//--- feed-forward pass of batch normalization
#define def_bnff_inputs      0
#define def_bnff_options     1
#define def_bnff_weights     2
#define def_bnff_outputs     3
#define def_bnff_batch       4
#define def_bnff_total       5

//--- gradient distribution through the batch normalization layer
#define def_bnhgr_options    0
#define def_bnhgr_gradient   1
#define def_bnhgr_inputs     2
#define def_bnhgr_gradient_inputs 3
#define def_bnhgr_weights    4
#define def_bnhgr_batch      5
#define def_bnhgr_total      6

//--- gradient distribution to optimized batch normalization parameters
#define def_bndelt_options    0
#define def_bndelt_delta_weights 1
#define def_bndelt_gradient   2
```

The next step is to initialize the new kernels in the program. To do this, we switch to the method of initializing the OpenCL program of the main dispatch class of our model [CNet::InitOpenCL](#). First, we change the total number of kernels used.

```
if(!m_cOpenCL.SetKernelsCount(40))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}
if(!m_cOpenCL.KernelCreate(def_k_BatchNormFeedForward,
                           "BatchNormFeedForward"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}
if(!m_cOpenCL.KernelCreate(def_k_BatchNormCalcHiddenGradient,
                           "BatchNormCalcHiddenGradient"))
```



```

{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}
if(!m_cOpenCL.KernelCreate(def_k_BatchNormCalcDeltaWeights,
                           "BatchNormCalcDeltaWeights"))
{
    m_cOpenCL.Shutdown();
    delete m_cOpenCL;
    return false;
}

```

Now that the kernels have been created, and we can access them, we move on to working with the methods of our batch normalization class.

As is already the tradition, we will start with the feed-forward method. We only make changes to the implementation of the multithread algorithm using OpenCL. All other method code remains unchanged.

According to the algorithm of preparing the kernel to run, we must first pass all the necessary data to the OpenCL context memory. So, we check for created buffers in the context memory.

```

bool CNeuronBatchNorm::FeedForward(CNeuronBase *prevLayer)
{
    .....
    //--- branching of the algorithm by the computing device
    if(!m_cOpenCL)
    {
        //--- Implementation using MQL5 tools
        .....
    }
    else // OpenCL block
    {
        //--- checking data buffers
        CBufferType *inputs = prevLayer.GetOutputs();
        if(inputs.GetIndex() < 0)
            return false;
        if(m_cBatchOptions.GetIndex() < 0)
            return false;
        if(m_cWeights.GetIndex() < 0)
            return false;
        if(m_cOutputs.GetIndex() < 0)
            return false;
    }
}

```

In the next step, we pass pointers to data buffers and the values of required constants to the kernel parameters.

```

//--- pass parameters to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormFeedForward,
                                def_bnff_inputs, inputs.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormFeedForward,
                                def_bnff_weights, m_cWeights.GetIndex()))

```

```

        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormFeedForward,
                                    def_bnff_options, m_cBatchOptions.GetIndex()))

        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormFeedForward,
                                    def_bnff_outputs, m_cOutputs.GetIndex()))

        return false;
    if(!m_cOpenCL.SetArgument(def_k_BatchNormFeedForward,
                              def_bnff_total, (int)m_cOutputs.Total()))

        return false;
    if(!m_cOpenCL.SetArgument(def_k_BatchNormFeedForward,
                              def_bnff_batch, m_iBatchSize))

        return false;

```

After completing the preparatory work, we proceed to enqueue the kernel for execution. First, we need to fill in two dynamic arrays. In one of them, we fill in the dimension of the task space, and in the other one, we fill in the offset in each dimension of the task space. We will run the kernel in a one-dimensional zero-offset task space. The number of threads to run will be four times smaller than the tensor size of the current layer results. However, since the dimension of the tensor will not always be a multiple of four, and we need to run the computations for all elements of the result tensor, we will provide an additional thread that will compute the "tail" part of the tensor that is not a multiple of four.

After calculating the number of threads and filling the buffers, we call the kernel queuing method.

```

//--- queuing for execution
    uint off_set[] = {0};
    uint NDRange[] = { (int)(m_cOutputs.Total() + 3) / 4 };
    if(!m_cOpenCL.Execute(def_k_BatchNormFeedForward, 1, off_set, NDRange))
        return false;
}
//---
    if(!m_cActivation.Activation(m_cOutputs))
        return false;
//---
    return true;
}

```

This completes the work with the feed-forward method, in which we have already implemented full functionality, including the ability to organize parallel computations on GPUs using OpenCL technology.

Let's proceed to backpropagation methods, in which we need to perform similar work. When implementing the backpropagation method in pure MQL5, we have overridden two methods. Consequently, we need to supplement both methods with multi-threaded computing functionality. First, let's add functionality to the method that propagates the error gradient up to the previous neural layer *CNeuronBatchNorm::CalcHiddenGradient*. As with the feed-forward method, we first create the necessary data buffers in the OpenCL context.

```

bool CNeuronBatchNorm::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    .....
//--- branching of the algorithm by the computing device
    if(!m_cOpenCL)
    {

```

```
//--- Implementation using MQL5 tools
.....
}
else // OpenCL block
{
//--- checking data buffers
CBufferType* inputs = prevLayer.GetOutputs();
CBufferType* inputs_grad = prevLayer.GetGradients();
if(inputs.GetIndex() < 0)
    return false;
if(m_cBatchOptions.GetIndex() < 0)
    return false;
if(m_cWeights.GetIndex() < 0)
    return false;
if(m_cOutputs.GetIndex() < 0)
    return false;
if(m_cGradients.GetIndex() < 0)
    return false;
if(inputs_grad.GetIndex() < 0)
    return false;
```

Then, according to our algorithm for implementing multi-threaded computations using OpenCL, we pass the parameters of the induced kernel.

```
//--- pass parameters to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcHiddenGradient,
                                def_bnhgr_inputs, inputs.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcHiddenGradient,
                                def_bnhgr_weights, m_cWeights.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcHiddenGradient,
                                def_bnhgr_options, m_cBatchOptions.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcHiddenGradient,
                                def_bnhgr_gradient, m_cGradients.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcHiddenGradient,
                                def_bnhgr_gradient_inputs, inputs_grad.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_BatchNormCalcHiddenGradient,
                           def_bnhgr_total, (int)m_cOutputs.Total()))
    return false;
if(!m_cOpenCL.SetArgument(def_k_BatchNormCalcHiddenGradient,
                           def_bnhgr_batch, m_iBatchSize))
    return false;
```

After passing all parameters, we prepare the kernel for the run queue. Recall that when creating the kernel we defined the use of vector operations with type *TYPE4*. Accordingly, we reduce by four times the number of threads running. Now we call the queuing method of the kernel.

```
//--- queuing
int off_set[] = {0};
```

6. Architectural solutions for improving model convergence

```
int NDRange[] = { (int)(m_cOutputs.Total() + 3) / 4 };
if(!m_cOpenCL.Execute(def_k_BatchNormCalcHiddenGradient, 1, off_set, NDRange))
    return false;
}
//---
return true;
}
```

This concludes the method that propagates the error gradient through the hidden layer *CNeuronBatchNorm::CalcHiddenGradient*. We need to repeat the operations for the second backpropagation method *CNeuronBatchNorm::CalcDeltaWeights*.

Again, we repeat the algorithm for queuing the kernel. This time the kernel uses three data buffers.

```
bool CNeuronBatchNorm::CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
{
    .....
    //--- branching of the algorithm by the computing device
    if(!m_cOpenCL)
    {
        //--- Implementation using MQL5 tools
        .....
    }
    else
    {
        //--- check data buffers
        if(m_cBatchOptions.GetIndex() < 0)
            return false;
        if(m_cGradients.GetIndex() < 0)
            return false;
        if(m_cDeltaWeights.GetIndex() < 0)
            return false;
    }
}
```

Then we pass pointers to the created buffers as parameters to the launched kernel.

```
//--- pass parameters to the kernel
if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcDeltaWeights,
                                def_bndelt_delta_weights, m_cDeltaWeights.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcDeltaWeights,
                                def_bndelt_options, m_cBatchOptions.GetIndex()))
    return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcDeltaWeights,
                                def_bndelt_gradient, m_cGradients.GetIndex()))
    return false;
```

Then we place the kernel in the execution queue. This time the number of threads will be equal to the number of elements in the results tensor of our batch normalization layer.

```
//--- queuing
int off_set[] = {0};
int NDRange[] = {(int)m_cOutputs.Total()};
if(!m_cOpenCL.Execute(def_k_BatchNormCalcDeltaWeights, 1, off_set, NDRange))
    return false;
```

```

        if(read && !m_cDeltaWeights.BufferRead())
            return false;
    }
    //---
    return true;
}

```

This concludes our work with *CNeuronBatchNorm* batch normalization class. It is ready for use as it now fully implements the data normalization algorithm. We have implemented the algorithm in two versions: using standard MQL5 tools and using OpenCL multi-threaded computing technology. This gives the user the opportunity to choose the technology used according to their requirements.

I now propose to look at the implementation of the batch normalization method in Python.

6.1.4 Implementing batch normalization in Python

We have already discussed the batch data normalization algorithm and even implemented a batch normalization layer for our library in MQL5. Additionally, we have added the capability to utilize multithreading technology with OpenCL. Now let's see the version of the method implementation offered by the familiar *Keras* library for *TensorFlow*.

This library provides the *tf.keras.layers.BatchNormalization* layer.

```

tf.keras.layers.BatchNormalization(
    axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True,
    beta_initializer='zeros', gamma_initializer='ones',
    moving_mean_initializer='zeros',
    moving_variance_initializer='ones', beta_regularizer=None,
    gamma_regularizer=None, beta_constraint=None, gamma_constraint=None, **kwargs
)

```

Batch normalization applies a transformation to maintain the mean of the results around zero and the standard deviation around one.

It's important to note that the batch normalization layer operates differently during training and during logical output.

During the training period, the layer normalizes its output data using the mean and standard deviation of the current batch of input data. For each normalized channel, the layer returns:

$$BN_{Out}^{Train} = \frac{\gamma * (X - \bar{x})}{\sqrt{\sigma_X^2 + \varepsilon}} + \beta$$

where:

- ε = small constant (configured as part of the constructor arguments) to avoid division by zero error
- γ = trainable scale factor (initialized as 1), which can be disabled by setting *scale=False* in the object constructor
- β = trainable offset factor (initialized as 0), which can be disabled by setting *center=False* in the object constructor
- X = tensor of the input data batch
- $\bar{x}$ = average value of the data batch

6. Architectural solutions for improving model convergence

- σ_X^2 = variance of the data batch

During operation, the layer normalizes its output data using the moving average and standard deviation of the batches it encountered during training.

$$BN_{Out} = \frac{\gamma * (x - \bar{x}_{Train})}{\sqrt{\sigma_{Train}^2 + \varepsilon}} + \beta$$

Thus, the layer will normalize the input data during inference only after being trained on data with similar statistical characteristics.

You can pass the following arguments to the layer constructor:

- *axis* – integer, the axis to be normalized (usually a feature axis)
- *momentum* – momentum for the moving average
- *epsilon* – small constant to avoid division by zero error
- *center* – if True, adds a *beta* offset to normalized tensor, if False, *beta* is ignored
- *scale* – if True it is multiplied by *gamma*, if False *gamma* is not used; when the next layer is a linear layer, it can be turned off because the scaling will be performed by the subsequent layer
- *beta_initializer* – beta weight initializer type
- *gamma_initializer* – type of gamma weight initializer
- *moving_mean_initializer* – type of initializer for moving average
- *moving_variance_initializer* – type of initializer for moving average variance
- *beta_regularizer* – additional regularizer for beta weight
- *gamma_regularizer* – additional regularizer for gamma weight
- *beta_constraint* – optional constraint for beta weight
- *gamma_constraint* – optional constraint for gamma weight

The following parameters can be used when accessing a layer:

- *inputs* – tensor of initial data, it is allowed to use tensor of any rank
- *training* – logical flag indicating the mode of operation of the layer: training or operation (the difference between the modes of operation is specified above)
- *input_shape* – used to describe the dimensionality of input data in case the layer is specified first in the model

At the output, the layer produces a tensor of results while preserving the dimensionality of the original data.

In addition, the layer design allows the use of the *layer.trainable* setting that blocks parameters from being changed during training. This is optional and usually means that the layer operates in output mode. This mode is usually enabled by the "training" parameter, which can be passed when calling the layer. However, please note that "Parameter Freeze" and "Output Mode" are two different concepts.

However, in the case of a *BatchNormalization* layer, setting *trainable = False* means that the layer will subsequently run in logical output mode. This means that it will use the moving average and moving variance to normalize the current batch instead of the mean and variance of the current dataset.

This behavior was added in *TensorFlow 2.0* to ensure that when *layer.trainable = False*, you get the most commonly expected behavior in the case of fine-tuning.

6. Architectural solutions for improving model convergence

Note that setting *trainable* for a model containing other layers will recursively set the *trainable* value for all inner layers.

If the *trainable* value of the attribute changes after a model is compiled, the new value does not take effect for that model until the model is recompiled again.

6.1.4.1 Creating a script to test batch normalization

To analyze the effect of batch normalization on the result, let's take the simplest models with a fully connected perceptron. One of our very first tests was to check the influence of preprocessing normalization of input data on the model's performance. In that test, we concluded that it was important to normalize the initial data and used normalized initial data in all subsequent models. However, the preliminary normalization of the initial data always has costs and is not very convenient for working in financial markets, when the initial data goes in a continuous stream. In this case, the normalization of the source data must be written in the program code. When changing the dataset, whether it's due to time-dependent factors or alterations in the analyzed instrument, it may require modifications to the code or external parameters that need to be defined outside the model. This is an additional cost. After that, you would need to retrain the model. Therefore, it would be logical to find a way to incorporate the data normalization process into the model and update its parameters during the model training. Don't you think that the batch normalization model we are looking at is suitable for solving this problem? This will be our first test.

To conduct such an experiment, we will use the script for testing perceptron models [perceptron.py](#) and create a copy of it named *batch_norm.py*. Let's make small changes to it.

At the beginning of the script, we import the necessary libraries as usual.

```
# Import libraries
import os
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import MetaTrader5 as mt5
```

Before training, we need to load training datasets which are available in the sandbox of the MetaTrader 5 terminal. To determine the path to the sandbox, we connect to the terminal and find the path to the terminal data folder. We add *MQL5\Files* to the resulting path and thus get the path to the terminal sandbox. If you saved the training dataset to a subdirectory, you also need to add it to this sandbox path. Now you can disconnect from the terminal. We will create two local variables with the full path to the files of the training dataset, one with normalized data and the second one with non-normalized data.

```
# Load the training dataset
if not mt5.initialize():
    print("initialize() failed, error code =",mt5.last_error())
    quit()

path=os.path.join(mt5.terminal_info().data_path,r'MQL5\Files')
mt5.shutdown()
filename = os.path.join(path,'study_data.csv')
filename_not_norm = os.path.join(path,'study_data_not_norm.csv')
```

First, we load data from the normalized set.

```
data = np.asarray( pd.read_table(filename,
                             sep=',',
                             header=None,
```


6. Architectural solutions for improving model convergence

```
skipinitialspace=True,  
encoding='utf-8',  
float_precision='high',  
dtype=np.float64,  
low_memory=False))
```

Then we divide the uploaded data into patterns and goals. Let me remind you that when creating the training dataset, we wrote all the information about the pattern to the file in one line. At the same time, each line contains information about only one pattern. The last two elements in the row contain the target values of the pattern. Let's use this property and determine the number of elements in the second dimension of our data array. Subtracting the number of elements from the obtained value by the target values, we get the number of elements of one pattern description. Using this information, we divide the data into two arrays.

```
# Divide the training sample into initial data and goals  
targets=2  
inputs=data.shape[1]-targets  
train_data=data[:,0:inputs]  
train_target=data[:,inputs:]
```

After that, we load and divide the data of the non-normalized training dataset in the same way.

```
#load unnormalized training dataset  
data = np.asarray( pd.read_table(filename_not_norm,  
                                sep=',',  
                                header=None,  
                                skipinitialspace=True,  
                                encoding='utf-8',  
                                float_precision='high',  
                                dtype=np.float64,  
                                low_memory=False))  
  
# Split the non-normalized training sample into initial data and goals  
train_nn_data=data[:,0:inputs]  
train_nn_target=data[:,inputs:]  
  
del data
```

After dividing the training dataset into two tensors, we delete the source data object in order to use our resources more efficiently.

The next step after loading the data is to create neural network models for testing.

First, we will create a small fully connected perceptron with one hidden layer of 40 elements and a result layer of 2 elements.

```
# Creating the first model with one hidden layer  
model1 = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),  
                           keras.layers.Dense(40, activation=tf.nn.swish),  
                           keras.layers.Dense(targets, activation=tf.nn.tanh)  
                           ])
```

After this, we create a callback object for early termination if the model's error on the training dataset doesn't decrease for more than five epochs. When compiling the model, we specify the *Adam* parameter optimization method and the standard deviation as a function of the model's training error.

6. Architectural solutions for improving model convergence

In addition to the error function to track the quality of training, we add the *Accuracy* metric, which shows the proportion of correct responses to the model.

```
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=5)
model1.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
model1.summary()
```

Next, we create a second model, in which we simply add a batch normalization layer between the source data layer and the hidden model layer.

```
# Add batch normalization to the source data
# to a model with one hidden layer
model1bn = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                             keras.layers.BatchNormalization(),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dense(targets, activation=tf.nn.tanh)
                             ])
```

And we compile the model with the same parameters.

```
model1bn.compile(optimizer='Adam',
                 loss='mean_squared_error',
                 metrics=['accuracy'])
model1bn.summary()
```

The models for our first experiment are ready.

In the second experiment, I would like to evaluate the impact of using batch normalization within the network between hidden layers of the model. To conduct this experiment, we will also create fully connected perceptrons, but with three similar hidden layers. In the first model, we'll create a model without using batch normalization. Let's just take the first model from this script and add two hidden layers to it, similar to the first hidden layer. The source data and results layers remain unchanged.

```
# Create a model with three hidden layers
model2 = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dense(targets, activation=tf.nn.tanh)
                             ])
```

For the sake of experiment purity, we will compile the model with the same parameters.

```
model2.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
model2.summary()
```

Now let's add a batch normalization layer before each hidden layer. Note that we **do not add** a batch normalization layer before the result layer, because the authors of the method do not recommend it. In their experiments, this worsened the results of the models.

```
# Add batch normalization for the source data and hidden layers of the second model
```

6. Architectural solutions for improving model convergence

```
model2bn = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                             keras.layers.BatchNormalization(),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.BatchNormalization(),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.BatchNormalization(),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dense(targets, activation=tf.nn.tanh)
                             ])
```

As before, the model is compiled without changing the parameters.

```
model2bn.compile(optimizer='Adam',
                 loss='mean_squared_error',
                 metrics=['accuracy'])
model2bn.summary()
```

Now that all the models are built, we can start training them. All models will be trained with the same parameters. To train the model, we will use batches of 1000 patterns between weight matrix updates. Training will last for 500 epochs unless early stopping occurs. The last 10% of the training dataset will be used for validation. At the same time, the patterns will be mixed during the learning process.

First, let's train a model with one hidden layer using normalized data.

```
# Train the first model on non-normalized data
history1 = model1.fit(train_data, train_target,
                     epochs=500, batch_size=1000,
                     callbacks=[callback],
                     verbose=2,
                     validation_split=0.1,
                     shuffle=True)
model1.save(os.path.join(path, 'perceptron1.h5'))
```

Next, we train the same model using non-normalized data.

```
# Train the first model on non-normalized data
history1nn = model1.fit(train_nn_data, train_nn_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.1,
                      shuffle=True)
```

Now we train a similar model using a batch normalization layer between the source data and the hidden layer. Training will be carried out on a non-normalized training dataset.

```
history1bn = model1bn.fit(train_nn_data, train_nn_target,
                        epochs=500, batch_size=1000,
                        callbacks=[callback],
                        verbose=2,
                        validation_split=0.1,
                        shuffle=True)
model1bn.save(os.path.join(path, 'perceptron1bn.h5'))
```

6. Architectural solutions for improving model convergence

The results of the first two trainings will serve as benchmarks for evaluating the model performance with the batch normalization layer.

At this stage, we gather enough information to draw a conclusion from the first experiment: data normalization during data preprocessing can be replaced with a batch normalization layer between the raw data and the trainable model.

Let's move on to working on the second experiment and determine the impact of the addition of a batch normalization layer before the hidden layer of the model on the training process and the overall performance of the trained model. To do this, we need to train two more models.

First, we train a model with three hidden layers using pre-normalized data. We use the same training parameters to train the model.

```
history2 = model2.fit(train_data, train_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.1,
                      shuffle=True)
model2.save(os.path.join(path, 'perceptron2.h5'))
```

Next, we train the model using a non-normalized training dataset, but with a batch normalization layer before of each hidden layer. In particular, the batch normalization layer is also used before the first hidden layer after the source data layer.

```
history2bn = model2bn.fit(train_nn_data, train_nn_target,
                          epochs=500, batch_size=1000,
                          callbacks=[callback],
                          verbose=2,
                          validation_split=0.1,
                          shuffle=True)
model2bn.save(os.path.join(path, 'perceptron2bn.h5'))
```

After training these two models, we have enough information to draw conclusions from the results of the second experiment. For clarity, let's create graphs showing the change in training and validation errors as a function of the number of training epochs.

First, let's plot the change in the standard deviation of the data of our models from the target data for the first experiment.

```
# Drawing model training results with one hidden layer
plt.plot(history1.history['loss'], label='Normalized inputs train')
plt.plot(history1.history['val_loss'], label='Normalized inputs validation')
plt.plot(history1nn.history['loss'], label='Unnormalized inputs train')
plt.plot(history1nn.history['val_loss'], label='Unnormalized inputs validation')
plt.plot(history1bn.history['loss'],
          label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history1bn.history['val_loss'],
          label='Unnormalized inputs\nvs BatchNormalization validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n1 hidden layer')
plt.legend(loc='upper right', ncol=2)
```

In addition to the first graph, let's plot the dynamics of changes in the *Accuracy* metric.

```
plt.figure()
plt.plot(history1.history['accuracy'], label='Normalized inputs train')
plt.plot(history1.history['val_accuracy'], label='Normalized inputs validation')
plt.plot(history1nn.history['accuracy'], label='Unnormalized inputs train')
plt.plot(history1nn.history['val_accuracy'], label='Unnormalized inputs validation')
plt.plot(history1bn.history['accuracy'],
          label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history1bn.history['val_accuracy'],
          label='Unnormalized inputs\nvs BatchNormalization validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n1 hidden layer')
plt.legend(loc='lower right', ncol=2)
```

We build similar graphs to display the results of the second experiment.

```
# Drawing the results of training models with three hidden layers
plt.figure()
plt.plot(history2.history['loss'], label='Normalized inputs train')
plt.plot(history2.history['val_loss'], label='Normalized inputs validation')
plt.plot(history2bn.history['loss'],
          label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history2bn.history['val_loss'],
          label='Unnormalized inputs\nvs BatchNormalization validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n3 hidden layers')
plt.legend(loc='upper right', ncol=2)

plt.figure()
plt.plot(history2.history['accuracy'], label='Normalized inputs train')
plt.plot(history2.history['val_accuracy'], label='Normalized inputs validation')
plt.plot(history2bn.history['accuracy'],
          label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history2bn.history['val_accuracy'],
          label='Unnormalized inputs\nvs BatchNormalization validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n3 hidden layers')
plt.legend(loc='lower right', ncol=2)
```

So, at this stage, we have trained all the models using data from the training set. For us, the training dataset represents historical data. Of course, the fact that the model can approximate historical data is a good thing. But we would like the model to work well in real-time. To check how the model behaves on unknown data, let's check the operation of the models on a test sample.

We load the test dataset in the same way as we loaded the training datasets. First, let's load the normalized test dataset.

```
# Uploading a test dataset
test_filename = os.path.join(path, 'test_data.csv')
test = np.asarray( pd.read_table(test_filename,
```

6. Architectural solutions for improving model convergence

```
sep=',',
header=None,
skipinitialspace=True,
encoding='utf-8',
float_precision='high',
dtype=np.float64,
low_memory=False))
```

Now we divide the loaded data into patterns and target values.

```
# Separation of the test sample into initial data and goals
test_data=test[:,0:inputs]
test_target=test[:,inputs:]
```

Then we repeat the algorithm to load the non-normalized test dataset.

```
test_filename = os.path.join(path, 'test_data_not_norm.csv')
test = np.asarray( pd.read_table(test_filename,
                                sep=',',
                                header=None,
                                skipinitialspace=True,
                                encoding='utf-8',
                                float_precision='high',
                                dtype=np.float64,
                                low_memory=False))
```

```
# Split the test dataset into initial data and goals
test_nn_data=test[:,0:inputs]
test_nn_target=test[:,inputs:]
```

```
del test
```

After copying the data, we delete the array of initial data, which will allow us to manage our resources more efficiently.

Next, we will test the operation of all models on test samples. We check the operation of models without batch normalization layers on normalized data. We will test models using batch normalization layers on non-normalized test sample data.

```
# Checking the results of models on a test sample
test_loss1, test_acc1 = model1.evaluate(test_data, test_target, verbose=2)
test_loss1bn, test_acc1bn = model1bn.evaluate(test_nn_data, test_nn_target,
                                              verbose=2)
test_loss2, test_acc2 = model2.evaluate(test_data, test_target, verbose=2)
test_loss2bn, test_acc2bn = model2bn.evaluate(test_nn_data, test_nn_target,
                                              verbose=2)
```

Testing results are output to the log.

```
# Output test results to the journal
print('Model 1 hidden layer')
print('Test accuracy:', test_acc1)
print('Test loss:', test_loss1)

print('Model 1 hidden layer with BatchNormalization')
```

```

print('Test accuracy:', test_acc1bn)
print('Test loss:', test_loss1bn)

print('Model 3 hidden layers')
print('Test accuracy:', test_acc2)
print('Test loss:', test_loss2)

print('Model 3 hidden layer with BatchNormalization')
print('Test accuracy:', test_acc2bn)
print('Test loss:', test_loss2bn)

```

For clarity, we make a graphical representation of the results separately for the standard deviation and for the *Accuracy* metric.

```

plt.figure()
plt.bar(['1 hidden layer', '1 hidden layer\nvs BatchNormalization',
        '3 hidden layers', '3 hidden layers\nvs BatchNormalization'],
        [test_loss1, test_loss1bn, test_loss2, test_loss2bn])
plt.ylabel('$MSE$ $Loss$')
plt.title('Test results')

plt.figure()
plt.bar(['1 hidden layer', '1 hidden layer\nvs BatchNormalization',
        '3 hidden layers', '3 hidden layers\nvs BatchNormalization'],
        [test_acc1, test_acc1bn, test_acc2, test_acc2bn])
plt.ylabel('$Accuracy$')
plt.title('Test results')

plt.show()

```

After creating the graphs, we call the command to render them on the user's screen.

With this, we conclude our work on the script that allows testing of how the use of batch normalization layer affects training results and model performance. We will get familiar with the results in the next section, dedicated to testing models.

6.1.5 Comparative testing of models using batch normalization

We have done a lot of work together and created a new class for batch normalization implementation. Its main purpose is to solve the internal covariance shift problem. As a result, the model should learn faster and the results should become more stable. Let's do some experiments and see if that's the case.

First, we will test the model using our class written in MQL5. For experiments, we will use very simple models that consist of only fully connected layers.

In the first experiment, we will try to use a batch normalization layer instead of pre-normalization in the data preparation stage. This approach will reduce the cost of data preparation both for model training and during commercial operation. In addition, the inclusion of normalization in the model allows it to be used with real-time data streams. This is how stock quotes are delivered, and processing them in real-time gives you an advantage.

To test the approach, we will create a script that uses one fully connected hidden layer and a fully connected layer as the neural layer of the results. Between the hidden layer and the initial data layer, we will set up a batch normalization layer.

The task is clear, so let's move on to practical implementation. To create the script, we will use the script from the first test of a fully connected perceptron [perceptron_test.mq5](#) as a base. Let's create a copy of the file with the name *perceptron_test_norm.mq5*.

At the beginning of the script are the external parameters. We will transfer them to the new script without any changes.

```
//+-----+
//| External parameters for script operation |
//+-----+
// Name of the file with the training sample
input string StudyFileName = "study_data_not_norm.csv";
// File name for recording the error dynamics
input string OutputFileName = "loss_study_vs_norm.csv";
// Number of historical bars in one pattern
input int BarsToLine = 40;
// Number of input layer neurons per 1 bar
input int NeuronsToBar = 4;
// Use OpenCL
input bool UseOpenCL = false;
// Packet size for updating the weights matrix
input int BatchSize = 10000;
// Learning rate
input double LearningRate = 3e-5;
// Number of hidden layers
input int HiddenLayers = 1;
// Number of neurons in one hidden layer
input int HiddenLayer = 40;
// Number of iterations of updating the weights matrix
input int Epochs = 1000;
```

In the script, we will only make changes to the *CreateLayersDesc* function that serves to specify the model architecture. In the parameters, this function receives a pointer to a dynamic array object, which we are to fill with descriptions of the neural layers to be created. To exclude possible misunderstandings, let's clear the obtained dynamic array immediately.

```
bool CreateLayersDesc(CArrayObj &layers)
{
    layers.Clear();
    CLayerDescription *descr;
    ///--- creating initial data layer
    if(!(descr = new CLayerDescription()))
    {
        PrintFormat("Error creating CLayerDescription: %d", GetLastError());
        return false;
    }
}
```

First, we create a layer to receive the initial data. Before passing the description of the layer to be created, we must create an instance of the neural layer description object *CLayerDescription*. We

create an instance of the object and immediately check the result of the operation. Please note that in case of an error, we display a message to the user, then we delete the dynamically created array object that was created earlier and only then terminate the program execution.

When the object is successfully created, we begin populating it with the desired content. For the initial data neural layer, we specify the basic type of a fully connected neural layer with zero initial data window without activation function and parameter optimization. We specify the number of neurons to be sufficient to receive the entire sequence of the pattern description. In this case, the number is equal to the product of the number of neurons in the pattern description by the number of elements in one candlestick description.

```
descr.type          = defNeuronBase;
int prev =
    descr.count      = NeuronsToBar * BarsToLine;
descr.window        = 0;
descr.activation     = AF_NONE;
descr.optimization   = None;
```

Once all the parameters of the created neural layer are specified, we add our neural layer description object to the dynamic array of the model architecture description and immediately check the result of the operation.

```
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

After adding the description of a neural layer to the dynamic array, we proceed to the description of the next neural layer. Again, we create a new instance of the neural layer description object and check the result of the operation.

```
//--- batch normalization layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
```

The second layer of the model will be the batch normalization layer. We will tell the model about this by specifying an appropriate constant in the *type* field. We specify the number of neurons and the size of the initial data window according to the size of the previous neuron layer. A new *batch* size parameter is added for the batch normalization layer. For this parameter, we will specify a value equal to the batch size between updates of the batch parameters. We do not use the activation function, but we specify a method to optimize the *Adam* parameters.

```
descr.type = defNeuronBatchNorm;
descr.count = prev;
descr.window = descr.count;
descr.batch = BatchSize;
descr.activation = AF_NONE;
descr.optimization = Adam;
```

After specifying all the necessary parameters of the new neural layer, we add it to the dynamic array of the model architecture description. As always, we check the result of the operation.

```
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

Next in our model, there is a block of hidden layers. The number of hidden layers is specified in the external script parameters by the user. All hidden layers are basic fully connected layers with the same number of neurons, which is specified in the external parameters of the script. Therefore, to create all the hidden neural layers, you only need one description object for the neural layer, which can be added multiple times to the dynamic array describing the model architecture.

Hence, the next step is to create a new instance of the neural layer description object and check the result of the operation.

```
//--- block of hidden layers
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
```

After creating the object, we fill it with the necessary values. Also, we specify the base type of the neuron layer *defNeuronBase*. The number of elements in the neural layer is transferred from the external parameter of the *HiddenLayer* script. We will use *Swish* as the activation function and *Adam* as the parameter optimization method.

```
descr.type          = defNeuronBase;
descr.count         = HiddenLayer;
descr.activation    = AF_SWISH;
descr.optimization  = Adam;
descr.activation_params[0] = 1;
```

Once sufficient information is provided for creating a neural layer, we organize a loop with the number of iterations equal to the number of hidden layers. Within the loop, we will add the created description of the hidden neural layer to the dynamic array describing the model architecture. At the same time, don't forget to control the process of adding the description object to the array at each iteration.

```
for(int i = 0; i < HiddenLayers; i++)
{
    if(!layers.Add(descr))
    {
        PrintFormat("Error adding layer: %d", GetLastError());
        delete descr;
        return false;
    }
}
```

In conclusion of the model description, you need to add the description of the output neural layer. For this, we create another instance of the neural layer description object and immediately check the result of the operation.

```
//--- layer of results
```

6. Architectural solutions for improving model convergence

```
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
```

After creating the object, we fill it with the description of the neural layer to be created. For the output layer, you can use the basic type of fully connected neural layer with two elements (corresponding to the number of target values for the pattern). We use the linear activation function and the *Adam* optimization method as we did for the other model layers.

```
descr.type          = defNeuronBase;
descr.count         = 2;
descr.activation     = AF_LINEAR;
descr.optimization   = Adam;
descr.activation_params[0] = 1;
```

Add the prepared description of the neural layer to the dynamic array describing the architecture of the model.

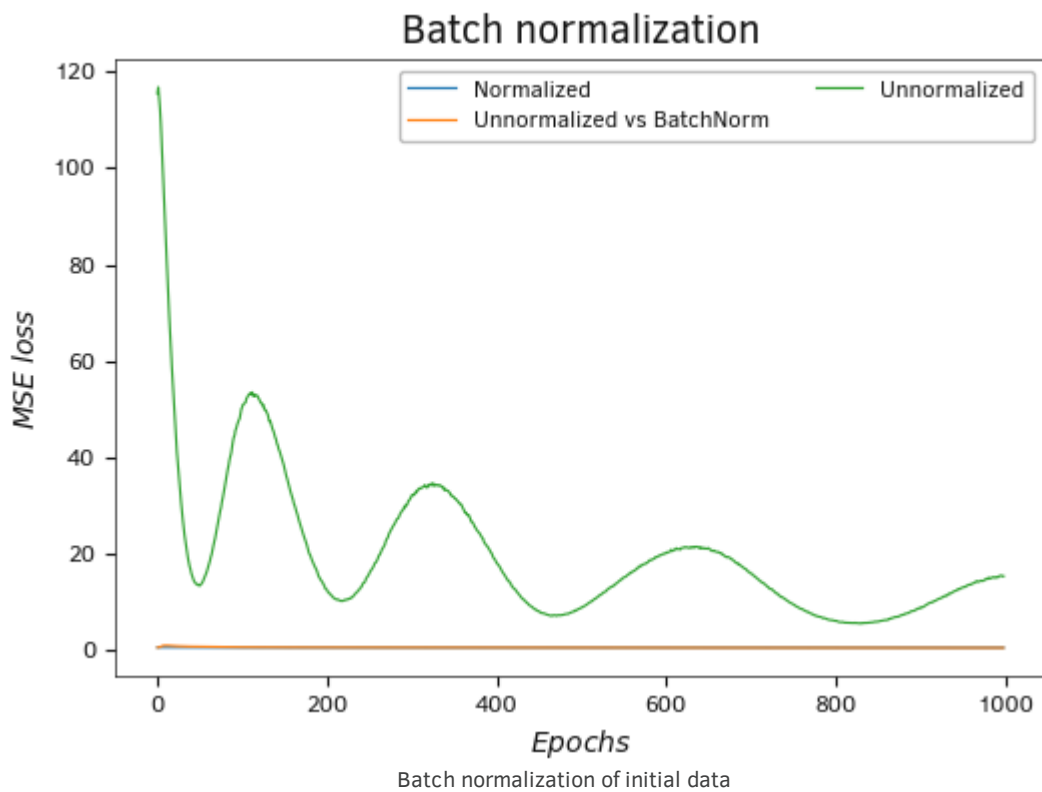
```
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
return true;
}
```

And of course, don't forget to check the result of the operation

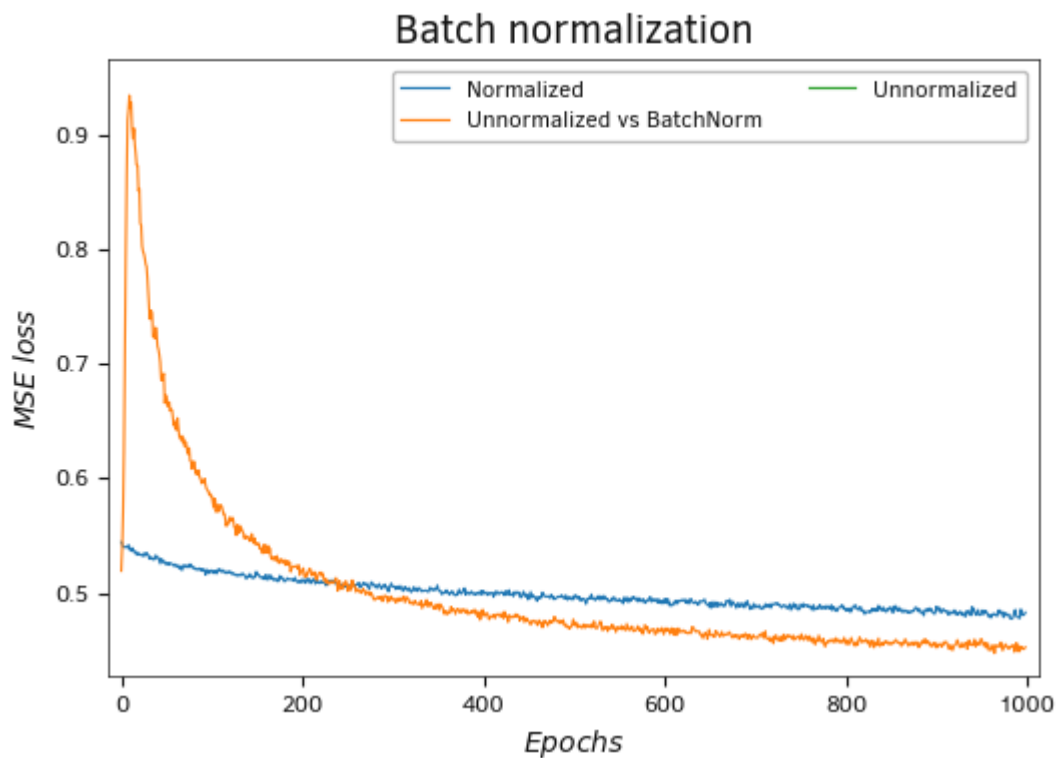
This concludes our work on building the script to run the first test, while the rest of the script code is transferred to this one in an unchanged form. We will run the script on the previously prepared training dataset. I will remind you that for the purity of experiments, all models within this book are trained on a single training dataset. This applies to models created in the MQL5 environment and those written in Python.

From the test results presented in the figures below, it can be confidently stated that the use of batch normalization layers can effectively replace the preprocessing normalization procedure at the data preparation stage.

The direct impact of batch normalization layers can be assessed by comparing the error dynamics graph of a model without a batch normalization layer when trained on a non-normalized training dataset. The gap between the charts is enormous.

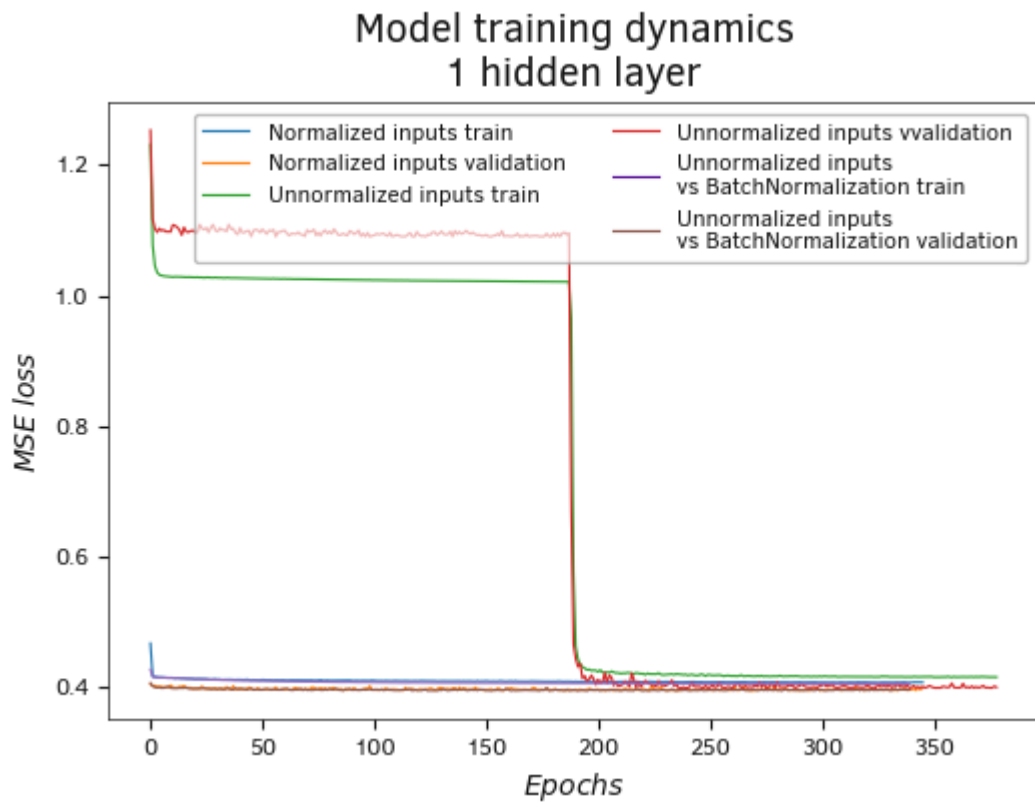


At the same time, differences in the error dynamics graphs of the model during training on normalized data and on non-normalized data with the use of batch normalization layers may only become apparent when you zoom in on the graph. Only at the beginning of training, there is a gap between the performance of the models. As training iterations increase, the accuracy gap of the models shrinks dramatically. After 200 iterations, the model using the normalization layer shows even better performance. This further confirms the possibility of including batch normalization layers in a model for real-time data normalization, providing additional evidence of its effectiveness.

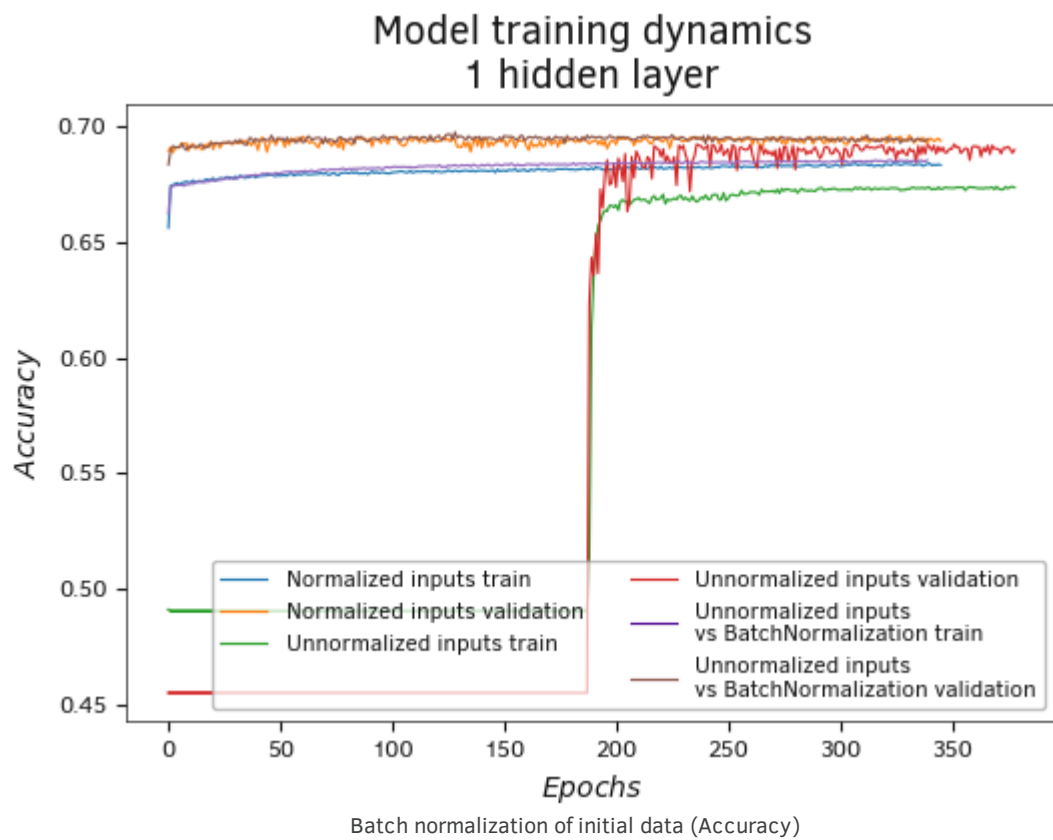
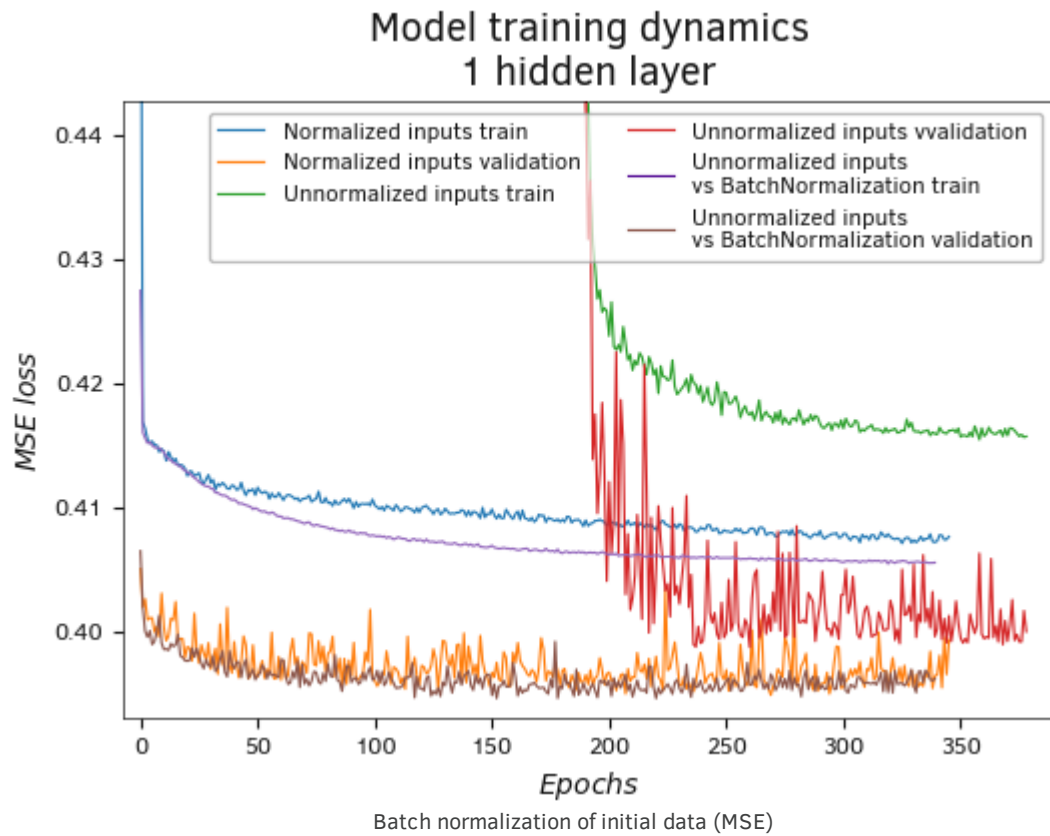


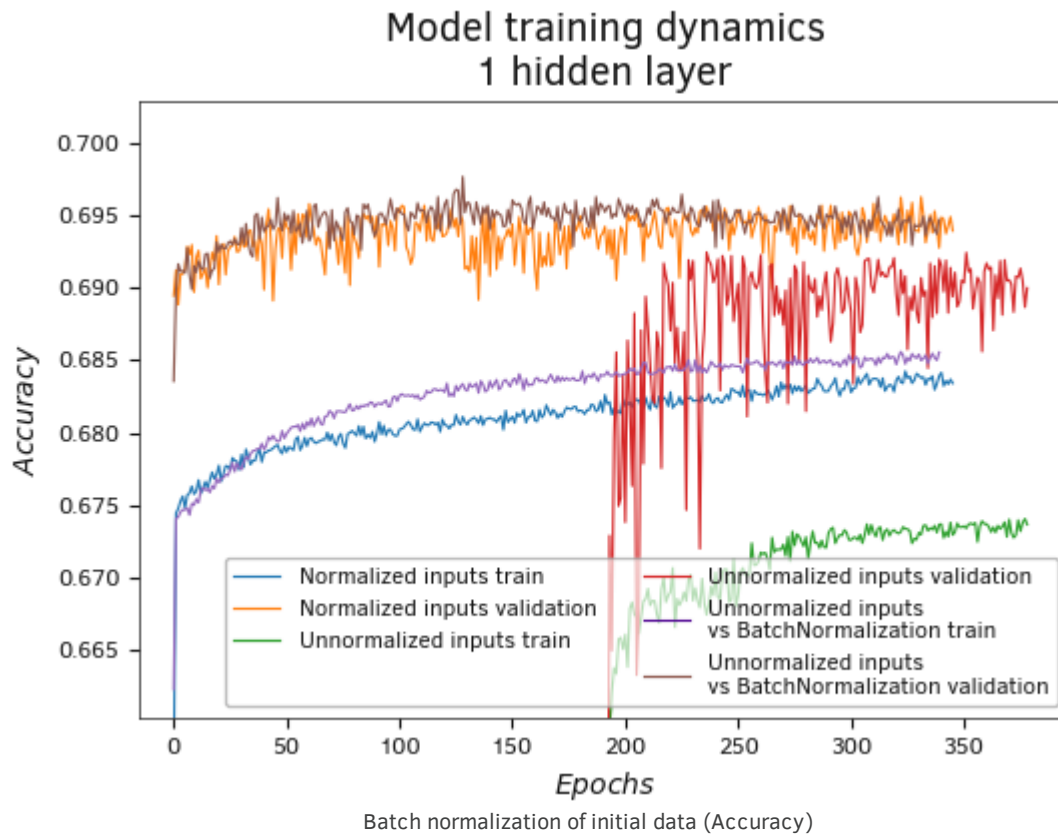
Batch normalization of initial data

We performed a similar experiment with models created in Python. This experiment confirmed the earlier findings.



Batch normalization of initial data (MSE)





Furthermore, within the scope of this experiment, the model using batch normalization layers demonstrated slightly better results both on the training dataset and during validation.

The analysis of the graph for the Accuracy metric suggests similar conclusions.

The second and probably the main option for using a batch normalization layer is to put the batch normalization layer before the hidden layers. The authors proposed using this method in exactly this way to address the problem of internal covariate shift. To test the effectiveness of this approach, let's create a copy of our script with the name *perceptron_test_norm2.mq5*. We will make small changes in the block of creating hidden layers. This is because, in the new script, we need to alternate between fully connected hidden layers and batch normalization layers, so we will include the creation of batch normalization layers within the loop.

```
//--- Batch Normalization Layer
CLayerDescription *norm = new CLayerDescription();
if(!norm)
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
norm.type = defNeuronBatchNorm;
norm.count = prev;
norm.window = descr.count;
norm.batch = BatchSize;
norm.activation = AF_NONE;
norm.optimization = Adam;
//--- Hidden layer
if(!(descr = new CLayerDescription()))
```

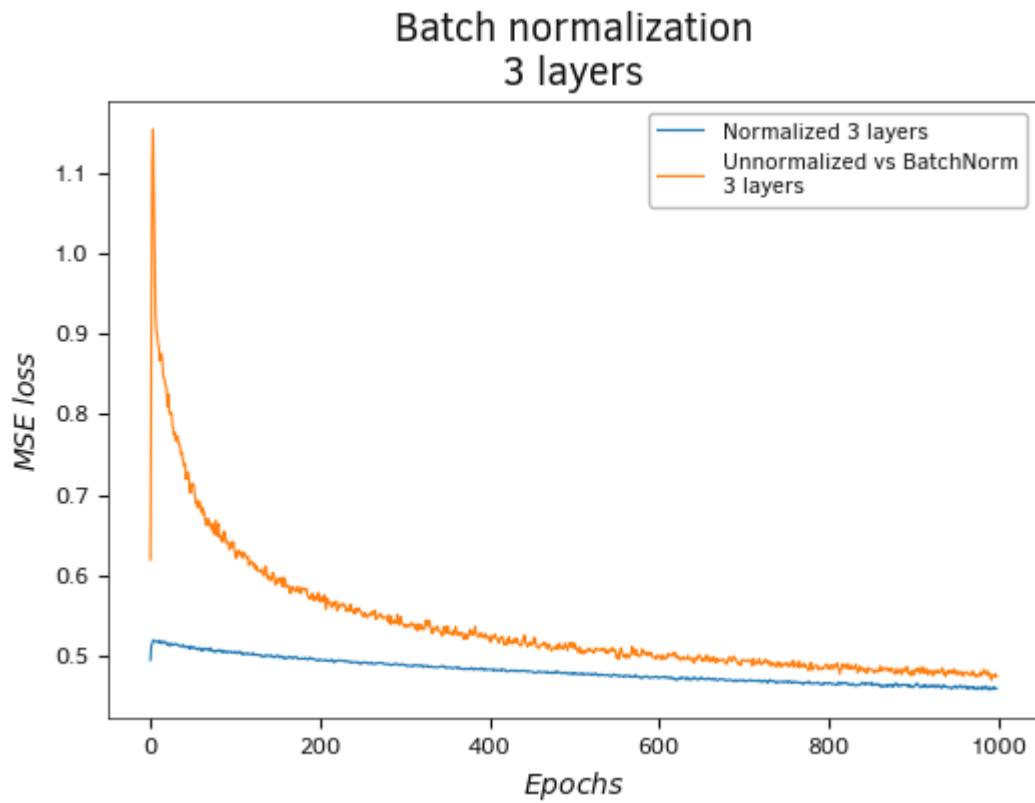
```

{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    delete norm;
    return false;
}
descr.type          = defNeuronBase;
descr.count         = HiddenLayer;
descr.activation     = AF_SWISH;
descr.optimization  = Adam;
descr.activation_params[0] = 1;
for(int i = 0; i < HiddenLayers; i++)
{
    if(!layers.Add(norm))
    {
        PrintFormat("Error adding layer: %d", GetLastError());
        delete descr;
        delete norm;
        return false;
    }
    CLayerDescription *temp = new CLayerDescription();
    if(!temp)
    {
        PrintFormat("Error creating CLayerDescription: %d", GetLastError());
        delete descr;
        return false;
    }
    temp.Copy(norm);
    norm = temp;
    norm.count = descr.count;
    if(!layers.Add(descr))
    {
        PrintFormat("Error adding layer: %d", GetLastError());
        delete descr;
        delete norm;
        return false;
    }
}
delete norm;

```

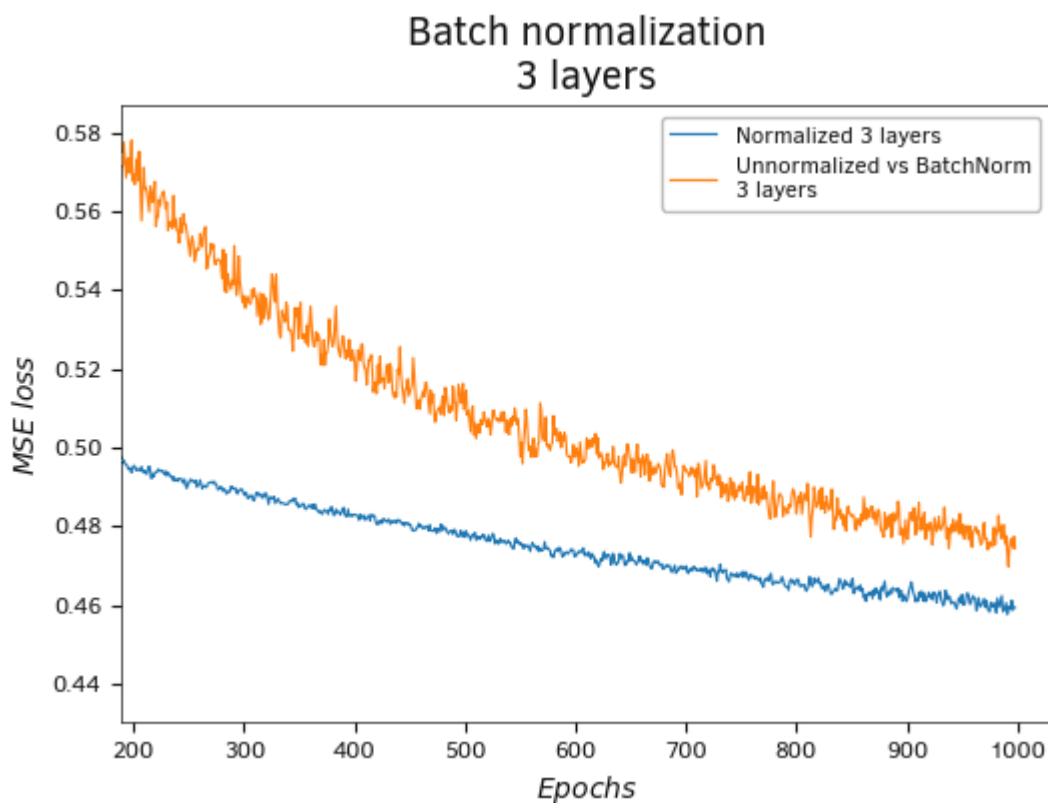
Otherwise, the script remains unchanged.

Testing of the script operation fully confirmed the earlier conclusions. Initially, when trained on non-normalized data, the model with batch normalization takes a little time to adapt. However, the gap in the accuracy of the models is shrinking dramatically.

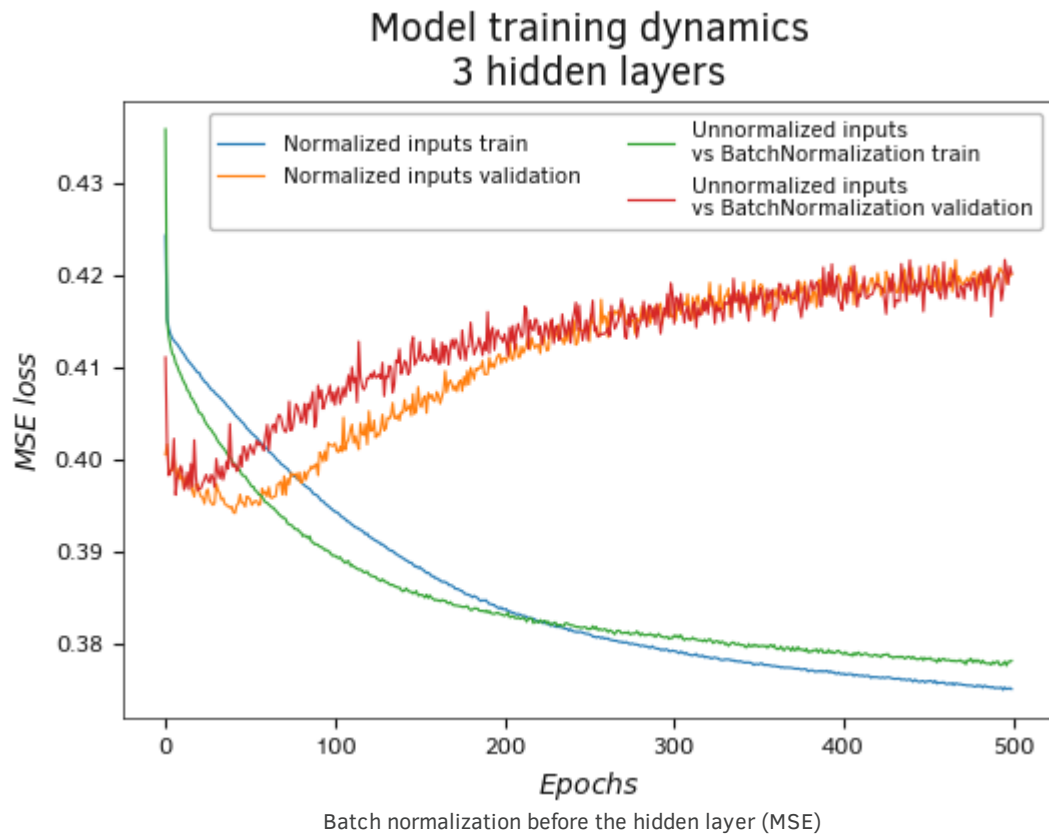


Batch normalization before the hidden layer

With a closer look, it becomes clear that the model with three hidden layers and batch normalization layers before each hidden layer even performs better on non-normalized input data. At the same time, its error dynamics graph decreases at a faster rate compared to the rest of the models.



Batch normalization before the hidden layer

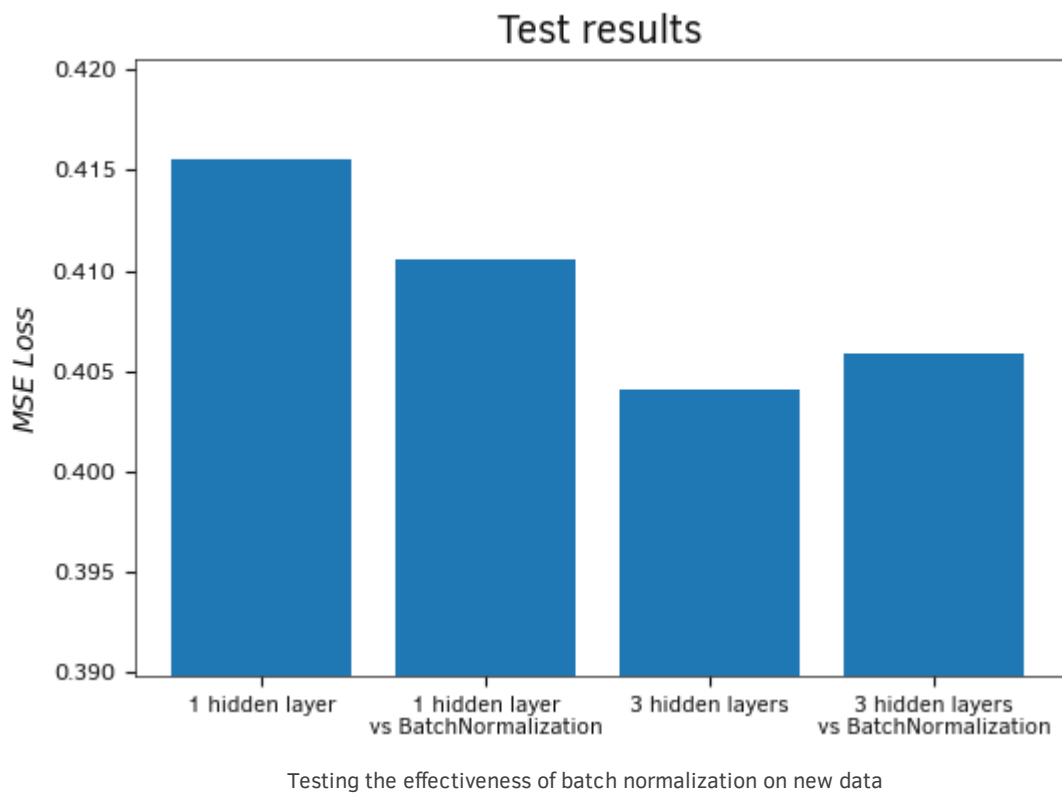
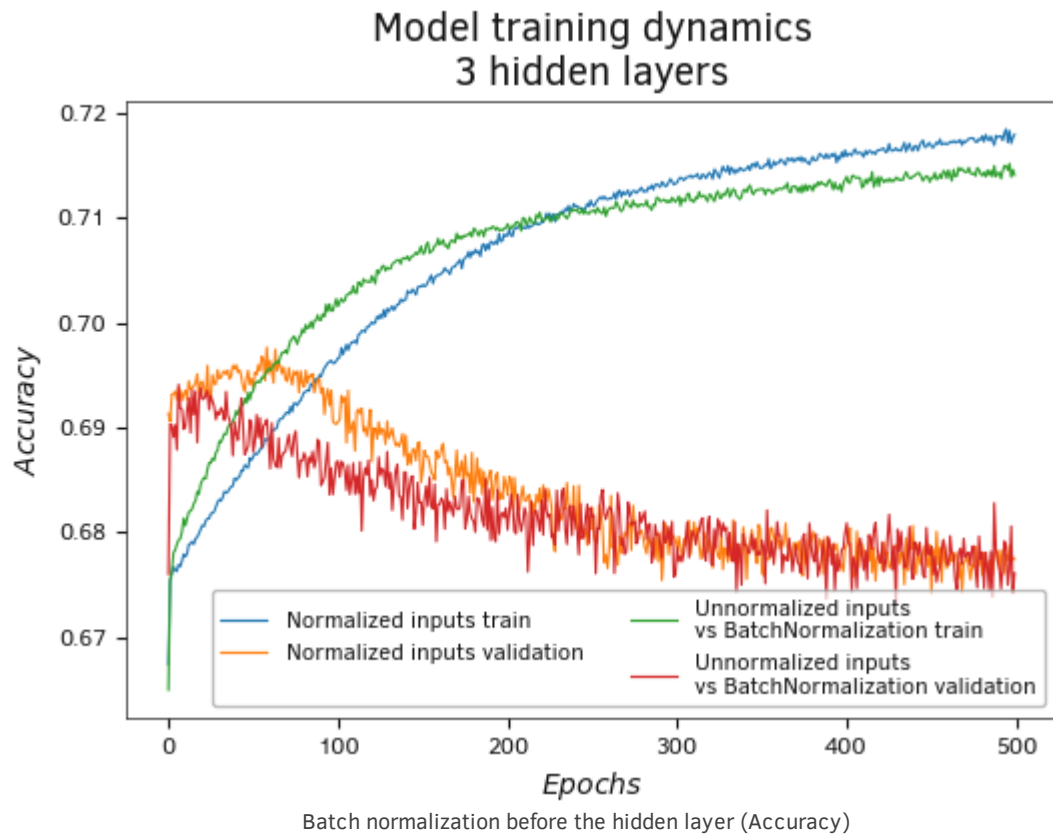


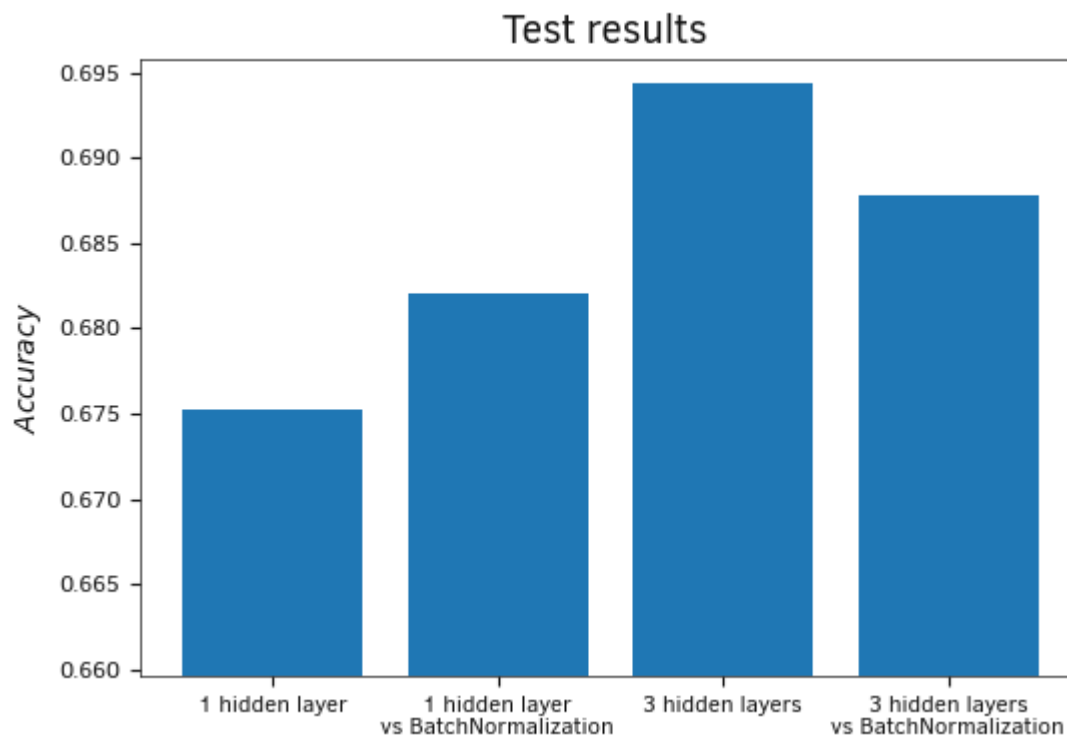
Conducting a similar experiment with models created in Python also confirms that models using batch normalization layers before each hidden layer, under otherwise equal conditions, train faster and are less prone to overfitting.

The dynamics of changes in the *Accuracy* metric value also confirms the earlier conclusions.

Additionally, we validated the models on a test dataset to evaluate the performance using new data. The results obtained showed a fairly smooth performance of all four models. The divergence of the RMS error of the models did not exceed $5 \cdot 10^{-3}$. Only a slight advantage was shown by models with three hidden layers.

Evaluation of the models using the *Accuracy* metric showed similar results.



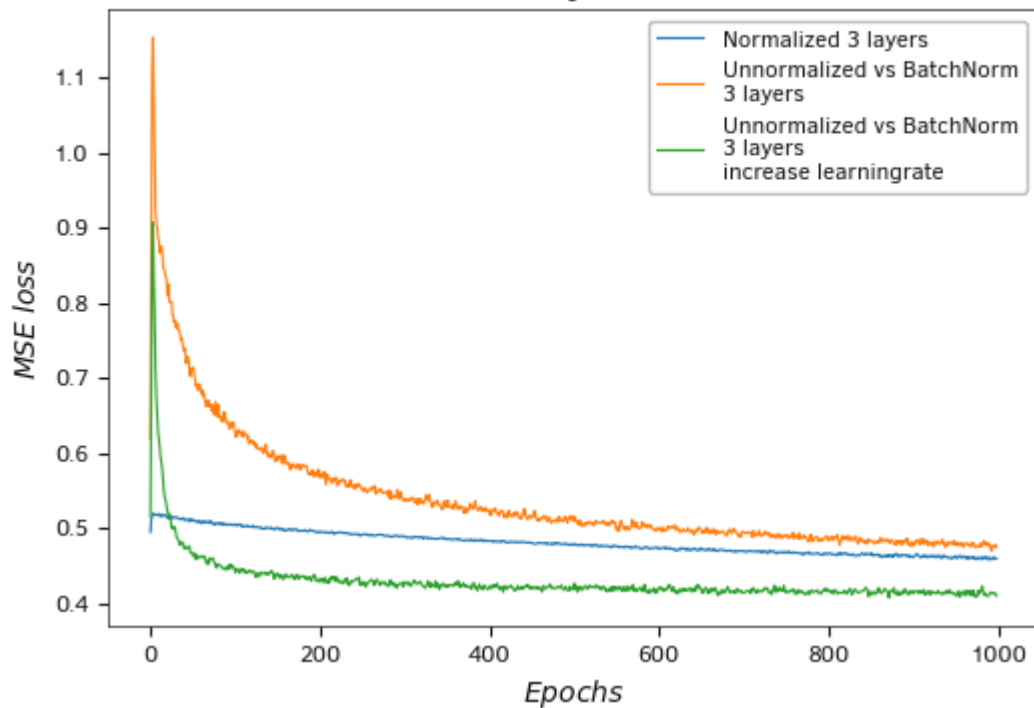


Testing the effectiveness of batch normalization on new data

To conclude, I decided to perform one more test. The authors of the method claim that the use of a batch normalization layer can increase the learning rate to speed up the process. Let's test this statement. We will run the *perceptron_test_norm2.mq5* script again, but this time increase the learning rate by 10 times.

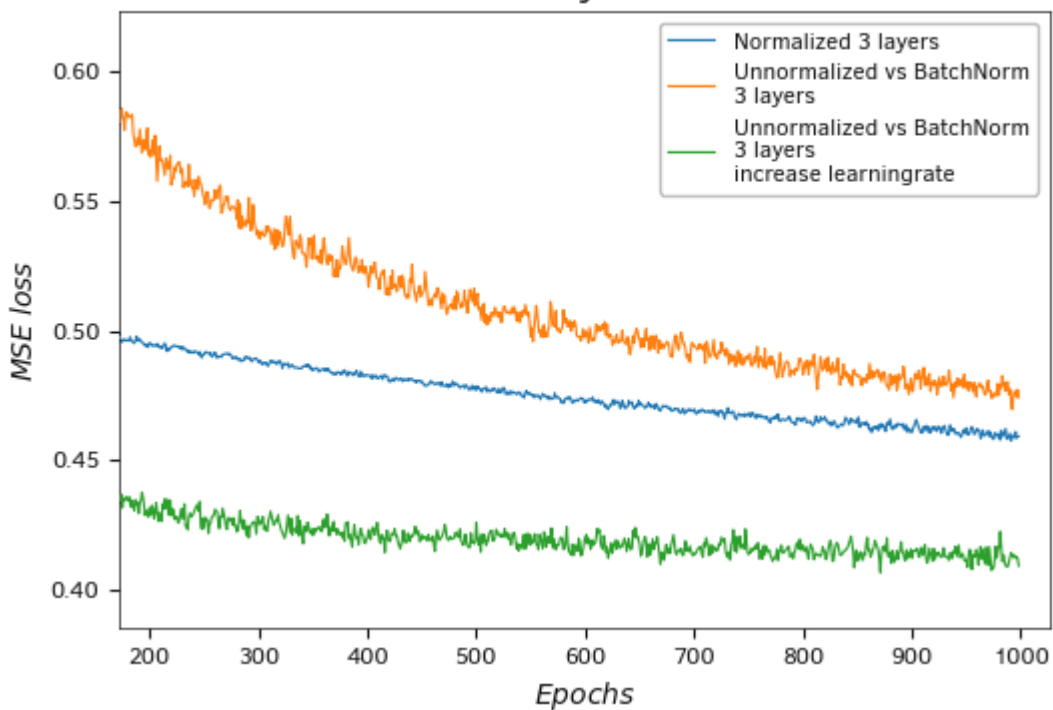
Testing has shown the potential effectiveness of this approach. In addition to a faster learning process, we got a better learning result than the previous ones.

Batch normalization 3 layers



Batch normalization before the hidden layer with increased learning rate

Batch normalization 3 layers



Batch normalization before the hidden layer with increased learning rate

In this section, we conducted a series of training tests for various models using batch normalization layers and without them. The obtained results demonstrated that a batch normalization layer after the input data can replace the normalization process at the data preparation stage for training. This

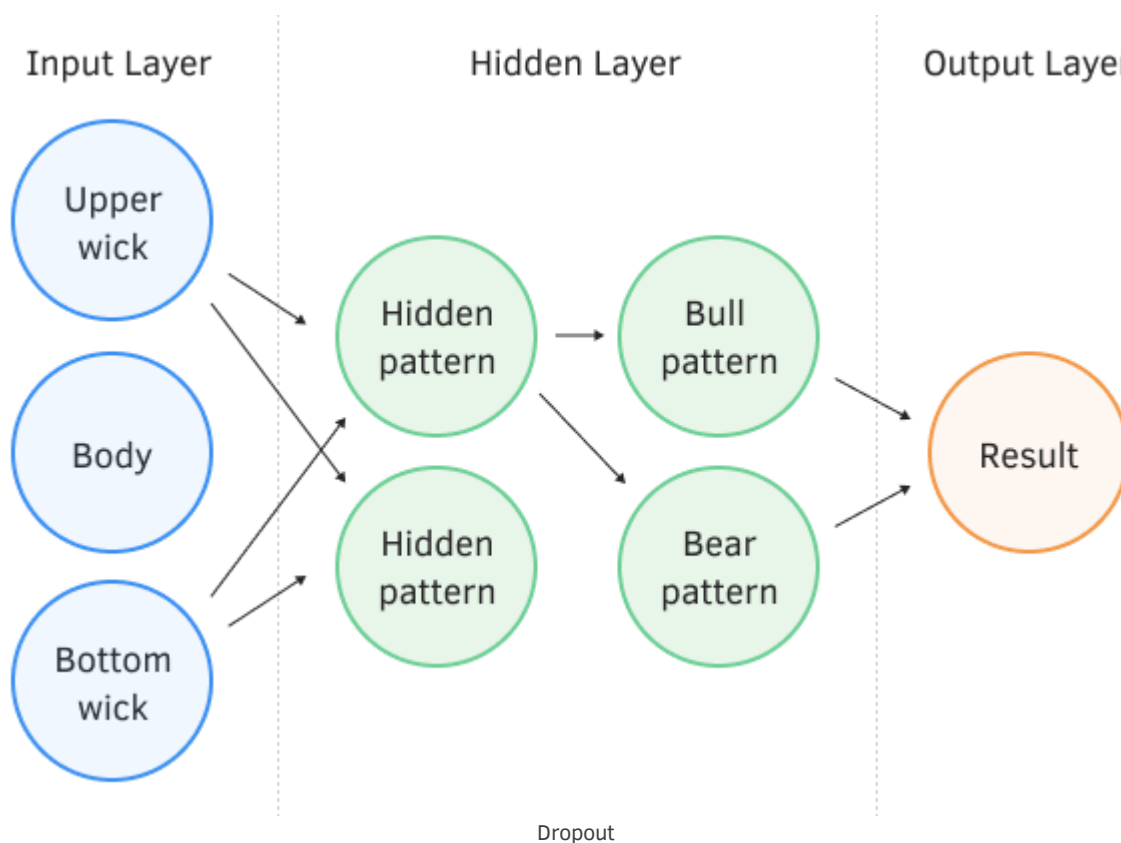
approach allows the data normalization process to be built into the model and tuned during model training. In this way, we can process the initial data in real-time during the operation of the model without complicating the overall decision-making program.

In addition, using a batch normalization layer before the hidden model layers can speed up the learning process, all other things being equal.

6.2 Dropout

As we continue discussing ways to increase the convergence of models, let's consider the *Dropout* method.

When training a neural network, a large number of features are fed into each neuron, and it is difficult to assess the influence of each of them. As a result, errors from some neurons are smoothed out by the correct values from others, and errors accumulate at the output of the neural network. As a result, training stops at a certain local minimum with a relatively large error. This effect is known as feature co-adaptation, where the influence of each feature seems to adapt to the surrounding environment. For us, it would be better to achieve the opposite effect, where the environment is decomposed into individual features, and the influence of each feature is evaluated separately.



To combat the complex co-adaptation of features, in July 2012, a group of scientists from the University of Toronto in the article [Improving neural networks by preventing co-adaptation of feature detectors](#) proposed randomly excluding some of the neurons in the learning process. Reducing the number of features during training increases the significance of each one, and constant variation in the quantitative and qualitative composition of features reduces the risk of their co-adaptation. This method is called *Dropout*. Some compare the application of this method to decision trees because, by

excluding some neurons, we get a new neural network with its own weights at each training iteration. According to the rules of combinatorics, the variability of such networks is quite high.

During the operation of the neural network, all attributes and neurons are evaluated. Thus, we get the most accurate and independent assessment of the current state of the environment under consideration.

The authors of the method in their paper mention the possibility of using it to improve the quality of pre-trained models as well.

Describing the proposed solution from the mathematics point of view, we can say that each individual neuron is dropped out of the process with a certain given probability p , or the neuron will participate in the process of training a neural network with probability q .

$$q = 1 - p$$

To determine the list of neurons to be dropped out, a pseudo-random number generator with a normal distribution is used. This approach provides the most uniform exclusion of neurons possible. In practice, we will generate a vector with a size equal to the input sequence. For the features used in the vector, we will set 1, and for the excluded elements, we will use 0.

However, excluding analyzed features undoubtedly leads to a reduction in the sum at the input of the neuron activation function. To compensate for this effect, we will multiply the value of each feature by a factor of $1/q$. It is obvious that this coefficient will increase the values since the probability of q will always be in the range from 0 to 1.

$$d_i = \frac{1}{q} x_i n_i$$

where:

- d_i = elements of the *Dropout* result vector
- q = probability of using a neuron in the learning process
- x_i = elements of the masking vector
- n_i = elements of the input sequence

During the backpropagation in the training process, the error gradient is multiplied by the derivative of the above-mentioned function. In the case of *Dropout*, the backpropagation pass will be similar to the feed-forward pass using the masking vector from the feed-forward pass.

$$\left(\frac{1}{q} x_i n_i \right)' = \frac{1}{q} x_i$$

During the operation of the neural network, the masking vector is filled with ones, allowing values to be transmitted in both directions without hindrance.

In practice, the coefficient of $1/q$ is constant throughout the training, so we can easily calculate this coefficient once and write it instead of one in the masking tensor. This way, we combine the coefficient recalculation and multiplication by 1 in each training iteration.

6.2.1 Building Dropout in MQL5

After discussing the theoretical aspects, I suggest moving on to studying the implementation of this method in our library.

To implement the *Dropout* algorithm, we will create a new class called *CNeuronDropout*, which we will include in our model as a separate layer. The new class will inherit directly from the *CNeuronBase* neural layer base class.

```
class CNeuronDropout      : public CNeuronBase
{
protected:
    TYPE          m_dOutProbability;
    int           m_iOutNumber;
    TYPE          m_dInitValue;

    CBufferType    m_cDropOutMultiplier;

public:
    CNeuronDropout(void);
    ~CNeuronDropout(void);

    //---
    virtual bool   Init(const CLayerDescription *desc) override;
    virtual bool   FeedForward(CNeuronBase *prevLayer) override;
    virtual bool   CalcHiddenGradient(CNeuronBase *prevLayer) override;
    virtual bool   CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
                                                override { return true; }

    virtual bool   UpdateWeights(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda) override { return true; }

    //--- methods of working with files
    virtual bool   Save(const int file_handle) override;
    virtual bool   Load(const int file_handle) override;
    //--- object identification method
    virtual int     Type(void) override      const { return defNeuronDropout; }
};
```

The first thing we encounter is the implementation of two different algorithms: one for the training process and another for testing and application. Therefore, we need to explicitly specify to the neural layer which algorithm it should use in each specific case. To do this, we introduce the *m_bTrain* flag which we will set to *true* during training and to *false* during testing.

To control the values of the flag, we will create a helper overload method *TrainMode*. In one version, when specifying a parameter, it will set a flag, and in the other variant, when called without parameters, it will return the current value of the *m_bTrain* flag.

```
virtual void      TrainMode(bool flag)        { m_bTrain = flag; }
virtual bool      TrainMode(void)             const { return m_bTrain; }
```

While working with the library, we built a mechanism for overriding the methods of all classes. By doing so, we created a versatile class architecture, allowing the dispatcher class of our model to work uniformly with any neural layer, without spending time on checking the type of the neural layer and branching algorithms based on the type of the neural layer used. To support this concept, we will

introduce a flag variable and methods for working with it at the level of the *CNeuronBase* base neural layer.

In the *protected* block of our class, we declare the following variables:

- *m_dOutProbability* – specified probability for dropping out neurons
- *m_iOutNumber* – number of neurons to be dropped out
- *m_dInitValue* – value for initializing the masking vector, in the theoretical part of this article we denoted this coefficient as $1/q$

Also, we will declare a pointer to the data buffer object for the *m_cDropOutMultiplier* masking vector.

The list of class methods is quite familiar. They all override the methods of the parent class.

Note that our new layer does not have weight matrices. The override of the *CalcDeltaWeights* and *UpdateWeights* methods which are responsible for distributing the error gradient to the weight matrix and updating the model parameters, is designed to maintain the overall architecture of the neural layers and the model as a whole. We cannot use methods from the parent class because the absence of corresponding objects would lead to a critical error. The creation of additional unused objects is an irrational waste of resources. Therefore, we override the methods. However, we create them as empty methods and they will simply always return a positive value.

```
virtual bool CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
                                     override { return true; }

virtual bool UpdateWeights(int batch_size, TYPE learningRate,
                          VECTOR &Beta, VECTOR &Lambda) override { return true; }
```

Now let's proceed with the class methods. We will start, as always, with the class constructor. In this method, we specify the default value of the variables. Using a static object for the mask vector buffer allows us to skip the operation of creating it in the constructor and deleting it in the destructor.

```
CNeuronDropout::CNeuronDropout(void) : m_dInitValue(1.0),
                                       m_dOutProbability(0),
                                       m_iOutNumber(0)
{
    m_bTrain = false;
}
```

Note that the values of the *m_bTrain* class mode flag, unlike other variables, are specified in the body of the method. This is due to the declaration of a variable in the parent class.

The method destructor remains empty.

Next comes the initialization method of the *CNeuronDropout::Init* class. In the parameters, the method receives a pointer to an object of the class describing the created neural layer. In the body of the method, we immediately check the validity of the received pointer as well as the compatibility of the dimensions of the created neural layer and the previous one. The only role of the *Dropout* layer is to mask neurons, while the size of the tensor does not change in any way.

```
bool CNeuronDropout::Init(const CLayerDescription *description)
{
    //--- control block
    if(!description || description.count != description.window)
        return false;
}
```

After successfully passing the control block, we reset the size of the input data window and call the initialization method of the parent class. Resetting the size of the input data window will instruct the parent class method not to create a weight matrix and other objects related to training the neural layer parameters. As always, we remember to check the results of the operations.

```
//--- calling a method of a parent class
CLayerDescription *temp = new CLayerDescription();
if(!temp || !temp.Copy(description))
    return false;
temp.window = 0;
if(!CNeuronBase::Init(temp))
    return false;
delete temp;
```

After the successful execution of the parent class method, we save the main parameters of the neural layer operation, including the dropout probability, the number of neurons to exclude, and the initialization value of the masking matrix. We obtain the first parameter from the user, while the other two parameters should be calculated.

```
//--- calculation of coefficients
m_dOutProbability = (TYPE)MathMin(description.probability, 0.9);
if(m_dOutProbability < 0)
    return false;
m_iOutNumber = (int)(m_cOutputs.Total() * m_dOutProbability);
m_dInitValue = (TYPE)(1.0 / (1.0 - m_dOutProbability));
```

After that, we initialize the masking buffer with the initial values and set the training flag to *true*.

```
//--- initiate the masking buffer
if(!m_cDropOutMultiplier.BufferInit(m_cOutputs.Rows(), m_cOutputs.Cols(),
                                     m_dInitValue))

    return false;
m_bTrain = true;
//---
return true;
}
```

This completes the work with the class initialization methods and proceeds to the actual creation of the algorithm of the *Dropout* method.

But first, let's recall that we don't have access to the neural layer directly from the main program. Now we have introduced a flag for the neural layer operation mode. Therefore, we need to go back to the dispatcher class of the model and add a method for changing the state of the flag.

```
void CNet::TrainMode(bool mode)
{
    m_bTrainMode = mode;
    int total = m_cLayers.Total();
    for(int i = 0; i < total; i++)
    {
        if(!m_cLayers.At(i))
            continue;
        CNeuronBase *temp = m_cLayers.At(i);
        temp.TrainMode(mode);
    }
}
```

6. Architectural solutions for improving model convergence

```
}  
}
```

In this method, we will save the flag value into a local variable and iterate through all the neural layers of the model in a loop, calling a similar method for each neural layer of the model.

6.2.1.1 Feed-forward method

The feed-forward pass is traditionally implemented in the *FeedForward* method. This method is declared virtual in the neural layer base class. It is overridden in each new class to build a specific algorithm for the class. We will do the same for this particular class, that is, we will also override this method.

In the parameters, the *CNeuronDropout::FeedForward* method receives a pointer to the object of the previous layer of our model. Within the method body, we immediately set up control blocks to check the pointers to objects used in this method. As usual, here we check not only pointers to external objects received in parameters but also to internal objects of the class. In this case, we will check the pointers to the previous layer object and its result buffer. We will also check the validity of the pointer to the result buffer of the current layer.

```
bool CNeuronDropout::FeedForward(CNeuronBase *prevLayer)
{
    //--- control block
    if(!prevLayer || !prevLayer.GetOutputs() || !m_cOutputs)
        return false;
```

After successfully passing the control block, we proceed to execute the algorithm of the *Dropout* method.

To execute the algorithm in training mode, we prepare a masking buffer. First, we fill the entire buffer with increasing coefficients $1/q$, which we stored in the *m_dInitValue* variable at the class initialization stage.

After that, we create a loop with the number of iterations equal to the number of elements to be dropped out. In the loop body, we generate random values from the range between 0 and the number of elements of the sequence. For randomly selected elements, we replace the multiplier in the masking buffer with 0.

Although lightning never strikes twice in the same place, let's provide an algorithm for the case when the same element falls out twice. Before writing 0 to the masking buffer, we first check the current coefficient for the dropped element. If it is equal to zero, then we decrease the value of the loop iteration counter and move on to selecting the next element. This approach will allow us to exclude the specified number of elements precisely.

```
//--- generate a data masking tensor
ulong total = m_cOutputs.Total();
if(!m_cDropOutMultiplier.m_mMatrix.Fill(m_dInitValue))
    return false;
for(int i = 0; i < m_iOutNumber; i++)
{
    int pos = (int)(MathRand() * MathRand() / MathPow(32767.0, 2) * total);
    if(m_cDropOutMultiplier.m_mMatrix.Flat(pos) == 0)
    {
        i--;
        continue;
    }
    if(!m_cDropOutMultiplier.m_mMatrix.Flat(pos, 0))
        return false;
}
```

After generating the masking vector, we only need to apply it to the initial data. To do this, we multiply two buffers, element by element: the initial data and the masking.

According to our library building concept, in each method of the class, we create two execution branches whenever possible: one using standard MQL5 means and the other one using OpenCL for multi-threaded computations. Therefore, next, we create a branching of the algorithm depending on the selected device for computing operations.

As always, now we will look at the implementation of the algorithm using MQL5. We will return to implementing the algorithm in multi-threaded operations using OpenCL a little later. In the block for implementing the algorithm using MQL5, we use matrix operations.

```
//--- branching of the algorithm depending on the execution device
if(!m_cOpenCL)
{
```

As you remember, the method has two modes: training and operation. Therefore, before executing the algorithm, we check the current operating mode. If the class runs in operational mode, we simply copy the contents of the result buffer from the previous layer into the result buffer of the current layer. In the case of the training process, we multiply the tensor of the original data by the masking tensor.

```
//--- checking the operating mode flag
if(!m_bTrain)
    m_cOutputs.m_mMatrix = prevLayer.GetOutputs().m_mMatrix;
else
    m_cOutputs.m_mMatrix = prevLayer.GetOutputs().m_mMatrix *
                           m_cDropOutMultiplier.m_mMatrix;
}
else // OpenCL block
{
    return false;
}
//---
return true;
}
```

So, as a result of the operations described above, the result buffer of our layer contains the masked data from the previous layer. The task set for the feed-forward method has been completed, and we can complete the method execution. Let's also add a temporary stub in place of the multi-threaded calculation algorithm.

Next, we move on to organizing the backpropagation process.

6.2.1.2 Backpropagation methods for Dropout

Traditionally, after implementing the feed-forward algorithm, we move on to organizing the backpropagation process. As you know, in the base class of the neural layer, the backpropagation algorithm is implemented by four virtual methods:

- **CalcOutputGradient** for calculating the error gradient at the output of a neural network
- **CalcHiddenGradient** for propagating a gradient through a hidden layer
- **CalcDeltaWeights** for the required calculation of weight correction values
- **UpdateWeights** for updating the weight matrix

All the above methods are overridden in new classes as needed. As mentioned earlier, our *Dropout* layer does not contain trainable parameters. As a consequence, it does not contain a weight matrix. Thus, the last two methods are not relevant to our class. At the same time, we will have to override these methods to maintain the integrity of our model architecture because, during training, it will call these methods for all the neural layers used. If we do not override them, then when these methods are called, the operations of the inherited parent method will be performed. In this case, the absence of a buffer of the weight matrix and related objects can lead to critical errors. In the best-case scenario, as a result of our control operation, we will terminate the method with a false result, which will lead to the interruption of the training process. Therefore, we override these methods and replace them with empty methods that will always return a positive result.

```
virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
                                     override { return true; }

virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda) override { return true; }
```

The **CalcOutputGradient** method is used only for the results layer. *Dropout* operation principles do not imply its use as a results layer. Therefore, we do not override it.

Thus, we only have one method left to override: the **CalcHiddenGradient** method that propagates the gradient through the hidden layer. This method, like most of the previous ones, is declared as virtual in the base neural network class and is overridden in all new classes to establish the specific algorithm of the neural layer operation. In the parameters, the method receives a pointer to the object of the previous layer. Right within the method body, we set up a control block to verify the validity of pointers to objects used by the method. As in the feed-forward method, we check pointers to all used objects, both external and internal.

```
bool CNeuronDropout::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    //---control block
    if(!prevLayer || !prevLayer.GetGradients() || !m_cGradients)
        return false;
```

After successfully passing the block of controls, we must create a branching of the algorithm depending on the computing device. As always, in this section, we will consider the implementation of the algorithm using MQL5 tools and will return to the multi-threaded implementation of the algorithm in the next section.

```
//--- branching of the algorithm depending on the execution device
ulong total = m_cOutputs.Total();
if(!m_cOpenCL)
{
```

In the implementation block using MQL5, we check the class operating mode. During operational use mode, we simply copy the data from the error gradient buffer of the current layer into a similar buffer of the previous layer.

```
//--- check the operating mode flag
if(!m_bTrain)
    prevLayer.GetGradients().m_mMatrix = m_cGradients.m_mMatrix;
else
    prevLayer.GetGradients().m_mMatrix = m_cGradients.m_mMatrix *
                                         m_cDropOutMultiplier.m_mMatrix;
}
else // OpenCL block
{
    return false;
}
//---
return true;
}
```

If the method operates in model training mode, according to the *Dropout* algorithm, we need to multiply the error gradient buffer of the current layer element-by-element by the masking vector buffer. The matrix multiplication operation allows us to do this literally in one line of code.

As you can see, at this stage we have passed the error gradient into the buffer of the previous layer. Therefore, the task set for this method is completed, and we can finish the method execution. Now we add a stub in the block for organizing multi-threaded operations. We will return to it in one of the subsequent sections.

Thus, we have fully implemented the *Dropout* algorithm using standard MQL5 tools. At this stage, you can already create a model and obtain initial results using this approach. However, as we have discussed before, it is equally important to have the capability to restore the previously trained model functionality at any convenient time for the full functionality of any neural layer within the model. Therefore, in the next section, we will look at methods for saving neural layer data and restoring the functioning of the layer from previously saved data.

6.2.1.3 File operations

In the previous sections, we explored the operating algorithm of the *Dropout* approach and even managed to create the *CNeuronDropout* class to implement it within our library. Within the framework of this class, we have implemented the *Dropout* feed-forward and backpropagation algorithms. Now, for the full implementation of this class, we need to add file methods that will allow us to save and restore the operation of a previously trained model at any required time. This provides the opportunity to quickly restore the functionality of the model when needed.

Again, when starting this work, we critically evaluate the variables and objects of our class to decide whether to save them to a file in whole or in part or to restore them according to some parameters.

```
class CNeuronDropout    : public CNeuronBase
{
protected:
    TYPE                m_dOutProbability;
    int                 m_iOutNumber;
    TYPE                m_dInitValue;
    CBufferType         m_cDropOutMultiplier;

public:
                                CNeuronDropout(void);
                                ~CNeuronDropout(void);

    //---
    virtual bool            Init(const CLayerDescription *desc) override;
    virtual bool            FeedForward(CNeuronBase *prevLayer) override;
    virtual bool            CalcHiddenGradient(CNeuronBase *prevLayer) override;
    virtual bool            CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
                                                override { return true; }

    virtual bool            UpdateWeights(int batch_size, TYPE learningRate,
                                           VECTOR &Beta, VECTOR &Lambda) override { return true; }

    //--- methods for working with files
    virtual bool            Save(const int file_handle) override;
    virtual bool            Load(const int file_handle) override;
    //--- object identification method
    virtual int             Type(void) override      const { return(defNeuronDropout); }
};
```

In addition to the objects inherited from the parent class, we create only one data buffer and three variables. These three variables have mathematical relationships between them. The masking vector buffer is redefined on each feed-forward pass. Thus, to restore the functionality of the *Dropout* layer, it is sufficient to save the objects of the parent class and one variable.

Therefore, the data-saving method will be quite simple and short. In parameters, the method receives a pointer to a file handle for saving. In the method body, we call a similar method from the parent class, in which all the controls and the saving of parent class objects are already implemented. After the successful execution of the parent class method, we will only write the dropout probability to the file, which represents the probability of dropping out neurons from processing. This particular variable was chosen because it is the parameter specified by the user, while the others are secondary and are calculated during class initialization.

```
bool CNeuronDropout::Save(const int file_handle)
{
```


6. Architectural solutions for improving model convergence

```
//--- call the method of the parent class
if(!CNeuronBase::Save(file_handle))
    return false;
//--- save the probability constant of dropping out elements
if(FileWriteDouble(file_handle, m_dOutProbability) <= 0)
    return false;
//---
return true;
}
```

The method for restoring the functionality of the *CNeuronDropout::Load* layer looks a little more complicated than the saving method. Just like the data-saving method, the data-loading method receives a file handle with data to load in its parameters. We remember the fundamental rule of data loading: data is loaded from the file in strict accordance with the sequence in which it was written. Therefore, in the method body, we first call a similar method from the parent class, where all the controls and loading of data inherited from the parent class objects are already implemented.

```
bool CNeuronDropout::Load(const int file_handle)
{
//--- call the method of the parent class
if(!CNeuronBase::Load(file_handle))
    return false;
}
```

We must always check the result of the parent class method execution because it confirms not only the data loading but also the passing of all implemented controls.

After the successful execution of the parent class method, we read the probability of dropping out neurons from the file. Based on the obtained value, we calculate the number of neurons to be dropped out on each feed-forward iteration and initialize the values of the masking buffer elements.

```
//--- read and restore constants
m_dOutProbability = (TYPE)FileReadDouble(file_handle);
m_iOutNumber = (int)(m_cOutputs.Total() * m_dOutProbability);
m_dInitValue = (TYPE)(1.0 / (1.0 - m_dOutProbability));
```

Finally, at the end of the method for restoring the functionality of our layer, we initialize the buffer for recording the masking vector.

```
//--- initializing the data masking buffer
if(!m_cDropOutMultiplier.BufferInit(m_cOutputs.Rows(), m_cOutputs.Cols(),
                                     m_dInitValue))
    return false;
//---
return true;
}
```

After successfully loading data and initializing objects in our layer, we exit the method with a positive result.

At this stage, we are completing work on the *Dropout* layer class using standard MQL5 tools. In the next section, we will look at implementing a multi-threaded algorithm using OpenCL.

6.2.2 Organizing multi-threaded operations in Dropout

We continue to implement the *Dropout* technology. In the previous sections, we have already fully implemented the algorithm for the operation of this technology using standard MQL5 capabilities. Now we move on to implementing the algorithm using the multi-threading capability on the GPU using OpenCL. Within the framework of this book, we performed this operation many times before. However, I would like to repeat that in order to implement it, we need to work in two directions. First, we will create an OpenCL program, and then we need to do the work on the side of the main program to implement data exchange between the main program and the OpenCL context in which the program will run and call the OpenCL program.

As always, this work begins with the creation of the OpenCL program. In this case, we don't have to write much code on the OpenCL side. Moreover, we will use the same kernel to implement both feed-forward and backpropagation passes. How did that happen? Let's recall what operations we need to implement.

In the feed-forward pass, we perform data masking. The vector mask is created using MQL5 on the main program side. Here we need to mask the initial data. To do this, we element-wise multiply the initial data tensor by the vector mask.

$$d_i = \frac{1}{q} x_i n_i$$

Therefore, for the feed-forward pass, we need to create a kernel for element-wise multiplication of two tensors of the same size.

During the backpropagation process, an error gradient must be propagated through the masking operation. Let's take a closer look at the formula for the masking operation. $1/q$ is a constant that is defined at the class initialization stage and does not change throughout the model training and operation process. x_i is a masking vector element that can only take two values: 1 or 0. Therefore, the entire masking process can be represented as multiplying a certain original value by a constant. As you know, the derivative of such an operation is the constant by which multiplication is performed.

$$\left(\frac{1}{q} x_i n_i \right)' = \frac{1}{q} x_i$$

In our case, to adjust the error gradient, we need to element-wise multiply the gradient of error of the current layer by the masking vector.

Thus, in the feed-forward and backpropagation passes, we element-wise multiply various tensors by the masking vector. Therefore, to implement both passes on the OpenCL side, it is sufficient to create one kernel of element-wise multiplication of two vectors. This is actually a fairly simple task. Using vector variables to optimize the process does not complicate the task.

To do this, we create the *MaskMult* masking kernel. In the parameters, this kernel receives pointers to three data buffers, two of which contain the input data, and the third one is used to write the results. Also, since vector operations are implied, the total number of threads will be smaller than the number of operations. So we won't be able to determine the size of the initial data tensors from the number of threads running. Therefore, to determine the dimensions of the tensors, we will transmit the necessary dimension information in kernel parameters.

In the body of the kernel, we define the ID of the current thread and transfer the necessary data from the buffers to local vector variables. Let's multiply two vector variables. The result obtained will be returned from the local vector variable to the scalar data buffer.

```
__kernel void MaskMult(__global TYPE *inputs,
                      __global TYPE *mask,
                      __global TYPE *outputs,
                      int outputs_total)
{
    const int n = get_global_id(0) * 4;
    //---
    TYPE4 out = ToVect4(inputs, n, 1, outputs_total, 0) *
                ToVect4(mask, n, 1, outputs_total, 0);
    D4ToArray(outputs, out, n, 1, outputs_total, 0);
}
```

As you can see, the entire kernel code fits into three lines. Of course, this was made possible by using the previously created [functions that translate the data](#) of the scalar buffer to and from a local vector variable.

Once the OpenCL kernel is created, we proceed to implement the functionality on the main program side. First, we need to create constants to refer to OpenCL program elements. To do this, we open the [defines.mqh](#) file and specify constants for the kernel and its parameters.

```
#define def_k_MaskMult          40

//--- data masking
#define def_mask_inputs        0
#define def_mask_mask          1
#define def_mask_outputs       2
#define def_mask_total         3
```

Then we move on to the model dispatcher class. In the OpenCL context initialization method, we change the total number of kernels and then create a kernel in the context.

```
bool CNet::InitOpenCL(void)
{
    .....
    if(!m_cOpenCL.SetKernelsCount(41))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        return false;
    }
    .....

    if(!m_cOpenCL.KernelCreate(def_k_MaskMult, "MaskMult"))
    {
        m_cOpenCL.Shutdown();
        delete m_cOpenCL;
        return false;
    }
    //---
    return true;
}
```

```
}
```

Once the preparatory work has been completed, we move on to working directly with the methods of our *CNeuronDropout* class. As always, let's start with the *CNeuronDropout::FeedForward* method and implement the following processes in this method:

- Pass information to the OpenCL context.
- Pass parameters to the OpenCL kernel.
- Place the kernel in the run queue.
- Download the kernel results.
- Clear context memory.

Moving on to the forward pass method. Changes will only affect the multi-threaded operation block, and the rest of the method code will remain unchanged.

The *Dropout* class can operate in two modes: training and production use. We have created a kernel for training mode, but have not prepared a kernel for the second case. For example, the operation of copying data from buffer to buffer is easy, and we can perform it with MQL5 tools. However, we have minimized data exchange between the OpenCL context and the main program. So, on the main program side, the content of the buffers will be irrelevant. To perform a data copy operation, you must first load the data from the OpenCL context into the main program memory and then copy the data from one buffer to another. You then need to return the data to the OpenCL context in another buffer for subsequent operations. This is totally inconsistent with our policy of minimizing data exchange between the OpenCL context and the main program.

We consider the second option: the use of a single kernel in two operation modes. In production use mode, the masking buffer is filled with units. It's also a working method. At the same time, we prepare the masking buffer on the side of the main program. OpenCL does not provide a pseudo-random number generator. So, before executing the kernel, we should pass the contents of the masking buffer from the main program to the OpenCL context. But in training mode, it's a coercive measure. Why waste time on this unnecessary operation in the use mode? Can we take a step back and prepare another kernel?

I found another solution. We already have a kernel to perform a linear activation function. Below is its mathematical representation.

$$f(x) = ax + b$$

If we consider the special case at $a=1$ and $b=0$, we get a simple copy of the data.

$$f(x) = 1x + 0 = x$$

You do not need to load additional buffers into the OpenCL context memory. Instead, we will only pass two integer values into the parameters.

The algorithm for working with the kernel remains the same: check the presence of buffers in the context's memory, pass the kernel parameters, and enqueue the kernel.

```
bool CNeuronDropout::FeedForward(CNeuronBase *prevLayer)
{
    .....
    //-- branching of the algorithm depending on the execution device
    if(!m_cOpenCL)
    {
        .....
    }
```

```

    }
else // OpenCL block
{
    //--- operation mode flag check
    if(!m_bTrain)
    {
        //--- check data buffers
        CBufferType *inputs = prevLayer.GetOutputs();
        if(inputs.GetIndex() < 0)
            return false;
        if(m_cOutputs.GetIndex() < 0)
            return false;
        //--- pass parameters to the kernel
        if(!m_cOpenCL.SetArgumentBuffer(def_k_LineActivation,
                                         def_activ_inputs, inputs.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgumentBuffer(def_k_LineActivation,
                                         def_activ_outputs, m_cOutputs.GetIndex()))
            return false;
        if(!m_cOpenCL.SetArgument(def_k_LineActivation,
                                   def_activ_param_a, (TYPE)1))
            return false;
        if(!m_cOpenCL.SetArgument(def_k_LineActivation,
                                   def_activ_param_b, (TYPE)0))
            return false;
        uint offset[] = {0};
        uint NDRange[] = {(uint)m_cOutputs.Total()};
        if(!m_cOpenCL.Execute(def_k_LineActivation, 1, offset, NDRange))
            return false;
    }
}

```

To organize work during training, we will repeat the algorithm mentioned above by enqueueing a new kernel.

```

else
{
    //--- check data buffers
    CBufferType *inputs = prevLayer.GetOutputs();
    if(inputs.GetIndex() < 0)
        return false;
    if(!m_cDropOutMultiplier.BufferCreate(m_cOpenCL))
        return false;
    if(m_cOutputs.GetIndex() < 0)
        return false;
    //--- pass parameters to the kernel
    if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,
                                     def_mask_inputs, inputs.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,
                                     def_mask_mask, m_cDropOutMultiplier.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,

```

```

                                def_mask_outputs, m_cOutputs.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_MaskMult, def_mask_total, total))
        return false;
    //--- enqueueing
    int off_set[] = {0};
    int NDRange[] = { (int)(total + 3) / 4};
    if(!m_cOpenCL.Execute(def_k_MaskMult, 1, off_set, NDRange))
        return false;
    }
}
//---
return true;
}

```

This concludes the feed-forward kernel. Let's proceed to implement similar operations for the *CNeuronDropout::CalcHiddenGradient* backpropagation method. Let me remind you that we will use the same kernels for the backpropagation pass in this case. The call algorithm does not change. Changes will only affect the specification of buffers used.

```

bool CNeuronDropout::CalcHiddenGradient(CNeuronBase *prevLayer)
{
    .....
    //--- branching of the algorithm depending on the execution device
    ulong total = m_cOutputs.Total();
    if(!m_cOpenCL)
    {
        .....
    }

    else // OpenCL block
    {
        //--- operation mode flag check
        if(!m_bTrain)
        {
            //--- checking data buffers
            CBufferType *grad = prevLayer.GetGradients();
            if(grad.GetIndex() < 0)
                return false;
            if(m_cGradients.GetIndex() < 0)
                return false;
            //--- passing parameters to the kernel
            if(!m_cOpenCL.SetArgumentBuffer(def_k_LineActivation,
                                           def_activ_inputs, m_cGradients.GetIndex()))
                return false;
            if(!m_cOpenCL.SetArgumentBuffer(def_k_LineActivation,
                                           def_activ_outputs, grad.GetIndex()))
                return false;
            if(!m_cOpenCL.SetArgument(def_k_LineActivation,
                                     def_activ_param_a, (TYPE)1))
                return false;
            if(!m_cOpenCL.SetArgument(def_k_LineActivation,

```

```

def_activ_param_b, (TYPE)0))

    return false;
uint offset[] = {0};
uint NDRange[] = {(uint)m_cOutputs.Total()};
if(!m_cOpenCL.Execute(def_k_LineActivation, 1, offset, NDRange))
    return false;
}

```

Operation mode during training.

```

else
{
    //--- check data buffers
    CBufferType* prev = prevLayer.GetGradients();
    if(prev.GetIndex() < 0)
        return false;
    if(m_cDropOutMultiplier.GetIndex() < 0)
        return false;
    if(m_cGradients.GetIndex() < 0)
        return false;
    //--- pass parameters to the kernel
    if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,
                                    def_mask_inputs, m_cGradients.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,
                                    def_mask_mask, m_cDropOutMultiplier.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,
                                    def_mask_outputs, prev.GetIndex()))
        return false;
    if(!m_cOpenCL.SetArgument(def_k_MaskMult, def_mask_total, total))
        return false;
    //--- enqueueing
    int off_set[] = {0};
    int NDRange[] = { (int)(total + 3) / 4 };
    if(!m_cOpenCL.Execute(def_k_MaskMult, 1, off_set, NDRange))
        return false;
}
}
//---
return true;
}

```

Note that in the backpropagation process, we no longer load masking data into the OpenCL context. We expect it to remain in context with the feed-forward method.

Congratulations, we have completed the work on the methods of the *Dropout* algorithm implementation class. We've done quite a lot of work and implemented the *Dropout* algorithm with MQL5 and multi-threaded operations using OpenCL. Now we can test the models. But first, I suggest considering the implementation of this approach in Python in the *TensorFlow* library.

6.2.3 Implementing Dropout in Python

To build models in Python, we previously used the *Keras* library for *TensorFlow*. This library already has a ready-made implementation of the *Dropout* layer.

```
tf.keras.layers.Dropout(
    rate, noise_shape=None, seed=None, **kwargs
)
```

The *Dropout* layer randomly sets the input units to 0 at a certain frequency equal to *rate* at each iteration during the training process. This helps prevent the model from overfitting. Initial data that is not set to 0 is scaled by $1/(1 - \text{rate})$. Therefore, the sum of all initial data transmitted remains unchanged.

Note that the *Dropout* layer is only applied if its *training* field is set to *True*. Otherwise, no values are masked. When training the model, the *training* flag will be automatically set to *True*. In other cases, the user can explicitly set *training* to *True* when calling the layer.

This is different from setting *trainable* = *False* for the *Dropout* layer. In this case, the value of the *trainable* flag does not affect the behavior of the layer, since *Dropout* does not have any weights that could be frozen during training.

The *Dropout* layer constructor has the following arguments:

- *rate* – a floating point number in the range from 0 to 1, which represents the proportion of elements of the initial data that are masked during the training process.
- *noise_shape* – a one-dimensional integer tensor representing the shape of a binary exception mask as (*batch_size*, *timesteps*, *features*). The shape will be multiplied by the tensor of the initial data. For example, if the initial data has a shape and you want the exclusion mask to be the same for all time steps, you can use *noise_shape*=(*batch_size*, 1, *features*).
- *seed* – an integer to use as a random seed.

When calling a layer, two arguments are allowed:

- *inputs* – a tensor of the source data, it is possible to use a tensor of any rank.
- *training* – a Boolean flag indicating the operating mode of the layer.

To test the effectiveness of using *Dropout* technology, we will create a script and train several models using this layer. We will not create overly complex models. Instead, let's take the script [batch_norm.py](#), which was used when testing batch normalization. We will create a copy of this script in a file *dropout.py* and add *Dropout* layers to each model.

First, we add two *Dropout* layers to the model with one hidden layer without using batch normalization. We will insert new layers before each fully connected layer.

```
# Adding a Dropout to a model with one hidden layer
model1do = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                             keras.layers.Dropout(0.3),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dropout(0.3),
                             keras.layers.Dense(targets, activation=tf.nn.tanh)
                             ])
model1do.compile(optimizer='Adam',
                 loss='mean_squared_error',
```


6. Architectural solutions for improving model convergence

```
metrics=['accuracy'])
model1do.summary()
```

Please note that in all *Dropout* layers, we will be masking 30% of the neurons of the previous layer.

Then, in a similar manner, we will add two *Dropout* layers to the model with one hidden layer and batch normalization of the initial data. Please note that we are being a little disingenuous here. It is currently not recommended to use batch normalization and *Dropout* simultaneously within this model, as this will only reduce the overall result of the model. Let's test this statement with practical examples.

```
# Adding Dropout to the model with batch normalization of the initial data
# and one hidden layer
model1bndo = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                                keras.layers.BatchNormalization(),
                                keras.layers.Dropout(0.3),
                                keras.layers.Dense(40, activation=tf.nn.swish),
                                keras.layers.Dropout(0.3),
                                keras.layers.Dense(targets, activation=tf.nn.tanh)
                                ])
model1bndo.compile(optimizer='Adam',
                   loss='mean_squared_error',
                   metrics=['accuracy'])
model1bndo.summary()
```

Similarly, we add *Dropout* batches to models with three hidden layers.

```
# Adding a Dropout to a model with three hidden layers
model2do = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                              keras.layers.Dropout(0.3),
                              keras.layers.Dense(40, activation=tf.nn.swish),
                              keras.layers.Dropout(0.3),
                              keras.layers.Dense(40, activation=tf.nn.swish),
                              keras.layers.Dropout(0.3),
                              keras.layers.Dense(40, activation=tf.nn.swish),
                              keras.layers.Dropout(0.3),
                              keras.layers.Dense(targets, activation=tf.nn.tanh)
                              ])
model2do.compile(optimizer='Adam',
                 loss='mean_squared_error',
                 metrics=['accuracy'])
model2do.summary()

# Adding Dropout to the model with batch normalization of the initial data
# and three hidden layers
model2bndo = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                                keras.layers.BatchNormalization(),
                                keras.layers.Dropout(0.3),
                                keras.layers.Dense(40, activation=tf.nn.swish),
                                keras.layers.BatchNormalization(),
                                keras.layers.Dropout(0.3),
                                keras.layers.Dense(40, activation=tf.nn.swish),
                                keras.layers.BatchNormalization(),
                                keras.layers.Dropout(0.3),
                                keras.layers.Dense(targets, activation=tf.nn.tanh)
                                ])
```

6. Architectural solutions for improving model convergence

```
        keras.layers.Dense(40, activation=tf.nn.swish),
        keras.layers.Dropout(0.3),
        keras.layers.Dense(targets, activation=tf.nn.tanh)
    ])
model2bndo.compile(optimizer='Adam',
                   loss='mean_squared_error',
                   metrics=['accuracy'])
model2bndo.summary()
```

After creating the models, we add code to start the new model training process.

```
history1do = model1do.fit(train_data, train_target,
                          epochs=500, batch_size=1000,
                          callbacks=[callback],
                          verbose=2,
                          validation_split=0.1,
                          shuffle=True)
model1do.save(os.path.join(path, 'perceptron1do.h5'))

history1bndo = model1bndo.fit(train_nn_data, train_nn_target,
                              epochs=500, batch_size=1000,
                              callbacks=[callback],
                              verbose=2,
                              validation_split=0.1,
                              shuffle=True)
model1bndo.save(os.path.join(path, 'perceptron1bndo.h5'))

history2do = model2do.fit(train_data, train_target,
                          epochs=500, batch_size=1000,
                          callbacks=[callback],
                          verbose=2,
                          validation_split=0.1,
                          shuffle=True)
model2do.save(os.path.join(path, 'perceptron2do.h5'))

history2bndo = model2bndo.fit(train_nn_data, train_nn_target,
                              epochs=500, batch_size=1000,
                              callbacks=[callback],
                              verbose=2,
                              validation_split=0.1,
                              shuffle=True)
model2bndo.save(os.path.join(path, 'perceptron2bndo.h5'))
```

We also add the ability to run models on a test dataset.

```
test_loss1do, test_acc1do = model1do.evaluate(test_data, test_target,
                                              verbose=2)
test_loss1bndo, test_acc1bndo = model1bndo.evaluate(test_nn_data,
                                                    test_nn_target,
                                                    verbose=2)
test_loss2do, test_acc2do = model2do.evaluate(test_data, test_target,
                                              verbose=2)
test_loss2bndo, test_acc2bndo = model2bndo.evaluate(test_nn_data,
                                                    test_nn_target,
```

```
verbose=2)
```

In addition to changes in terms of training and testing models, we will also add a block for rendering model results. First, let's change the code that creates dynamics graphs for the mean square error and *Accuracy* during the training process. The changes here are not global, as we are just adding new variables to the graph.

```
# Rendering the results of training models with one hidden layer
plt.figure()
plt.plot(history1.history['loss'], label='Normalized inputs train')
plt.plot(history1.history['val_loss'], label='Normalized inputs validation')
plt.plot(history1do.history['loss'], label='Normalized inputs\nvs Dropout train')
plt.plot(history1do.history['val_loss'],
          label='Normalized inputs\nvs Dropout validation')
plt.plot(history1bn.history['loss'],
          label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history1bn.history['val_loss'],
          label='Unnormalized inputs\nvs BatchNormalization validation')
plt.plot(history1bndo.history['loss'],
          label='Unnormalized inputs\nvs BatchNormalization and Dropout train')
plt.plot(history1bndo.history['val_loss'],
          label='Unnormalized inputs\nvs BatchNormalization and Dropout validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n1 hidden layer')
plt.legend(loc='upper right',ncol=2)

plt.figure()
plt.plot(history1.history['accuracy'], label='Normalized inputs trin')
plt.plot(history1.history['val_accuracy'], label='Normalized inputs validation')
plt.plot(history1do.history['accuracy'],
          label='Normalized inputs\nvs Dropout train')
plt.plot(history1do.history['val_accuracy'],
          label='Normalized inputs\nvs Dropout validation')
plt.plot(history1bn.history['accuracy'],
          label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history1bn.history['val_accuracy'],
          label='Unnormalized inputs\nvs BatchNormalization validation')
plt.plot(history1bndo.history['accuracy'],
          label='Unnormalized inputs\nvs BatchNormalization and Dropout train')
plt.plot(history1bndo.history['val_accuracy'],
          label='Unnormalized inputs\nvs BatchNormalization and Dropout validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n1 hidden layer')
plt.legend(loc='lower right',ncol=2)

# Rendering the results of training models with three hidden layers
plt.figure()
plt.plot(history2.history['loss'], label='Normalized inputs train')
plt.plot(history2.history['val_loss'], label='Normalized inputs validation')
plt.plot(history2do.history['loss'], label='Normalized inputs\nvs Dropout train')
plt.plot(history2do.history['val_loss'],
```

```

        label='Normalized inputs\nvs Dropout validation')
plt.plot(history2bn.history['loss'],
        label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history2bn.history['val_loss'],
        label='Unnormalized inputs\nvs BatchNormalization validation')
plt.plot(history2bndo.history['loss'],
        label='Unnormalized inputs\nvs BatchNormalization and Dropout train')
plt.plot(history2bndo.history['val_loss'],
        label='Unnormalized inputs\nvs BatchNormalization and Dropout validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n3 hidden layers')
plt.legend(loc='upper right',ncol=2)

plt.figure()
plt.plot(history2.history['accuracy'], label='Normalized inputs train')
plt.plot(history2.history['val_accuracy'], label='Normalized inputs validation')
plt.plot(history2do.history['accuracy'], label='Normalized inputs\nvs Dropout train')
plt.plot(history2do.history['val_accuracy'],
        label='Normalized inputs\nvs Dropout validation')
plt.plot(history2bn.history['accuracy'],
        label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history2bn.history['val_accuracy'],
        label='Unnormalized inputs\nvs BatchNormalization validation')
plt.plot(history2bndo.history['accuracy'],
        label='Unnormalized inputs\nvs BatchNormalization and Dropout train')
plt.plot(history2bndo.history['val_accuracy'],
        label='Unnormalized inputs\nvs BatchNormalization and Dropout validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n3 hidden layers')
plt.legend(loc='lower right',ncol=2)

```

The last changes in the script concern the display of model performance results on the test dataset. Here, in addition to adding new data, we split the graphs: we will separately show the results of models with one hidden layer, and we will place the results of models with three hidden layers on a new graph.

```

plt.figure()
plt.bar(['Normalized inputs','\n\nNormalized inputs\nvs Dropout',
        'Unnormalized inputs\nvs BatchNormalization',
        '\n\nUnnormalized inputs\nvs BatchNormalization and Dropout'],
        [test_loss1,test_loss1do,
        test_loss1bn,test_loss1bndo])
plt.ylabel('$MSE$ $loss$')
plt.title('Test results\n1 hidden layer')

plt.figure()
plt.bar(['Normalized inputs','\n\nNormalized inputs\nvs Dropout',
        'Unnormalized inputs\nvs BatchNormalization',
        '\n\nUnnormalized inputs\nvs BatchNormalization and Dropout'],
        [test_loss2,test_loss2do,
        test_loss2bn,test_loss2bndo])
plt.ylabel('$MSE$ $loss$')

```

```
plt.title('Test results\n3 hidden layers')

plt.figure()
plt.bar(['Normalized inputs','\n\nNormalized inputs\nvs Dropout',
        'Unnormalized inputs\nvs BatchNormalization',
        '\n\nUnnormalized inputs\nvs BatchNormalization and Dropout'],
        [test_acc1,test_acc1do,
         test_acc1bn,test_acc1bndo])
plt.ylabel('$Accuracy$')
plt.title('Test results\n1 hidden layer')

plt.figure()
plt.bar(['Normalized inputs','\n\nNormalized inputs\nvs Dropout',
        'Unnormalized inputs\nvs BatchNormalization',
        '\n\nUnnormalized inputs\nvs BatchNormalization and Dropout'],
        [test_acc2,test_acc2do,
         test_acc2bn,test_acc2bndo])
plt.ylabel('$Accuracy$')
plt.title('Test results\n3 hidden layers')

plt.show()
```

The rest of the script code remained unchanged.

We will learn about the results of testing the models in the next section.

6.2.4 Comparative testing of models with Dropout

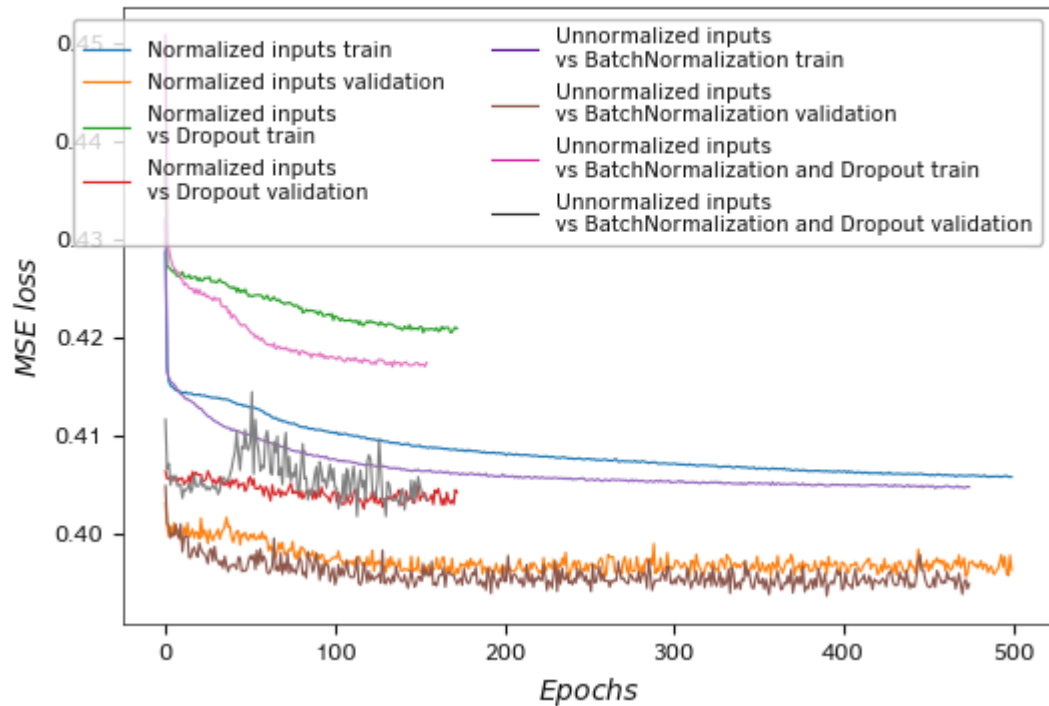
Another stage of work with our library has been completed. We have studied the *Dropout* method, which combats the issue of feature co-adaptation and have built a class to implement this algorithm in our models. In the previous section, we assembled a *Python* script for the comparative testing of models using this method and without. Let's look at the results of such testing.

First, we look at the test training schedule for models with one hidden layer. The dynamics of the mean square error of the models using *Dropout* was worse than that of models without it. This applies to both the model trained on normalized data and the model using batch normalization layers for preprocessing the input data. You can see that both models using the *Dropout* layer worked synchronously. Their lines on the graph are practically overlapping, both during the training and validation phases.

Similar conclusions can be drawn when analyzing the dynamics of *Accuracy* metrics. However, unlike the *MSE*, accuracy values in the validation process are close to those of other models.

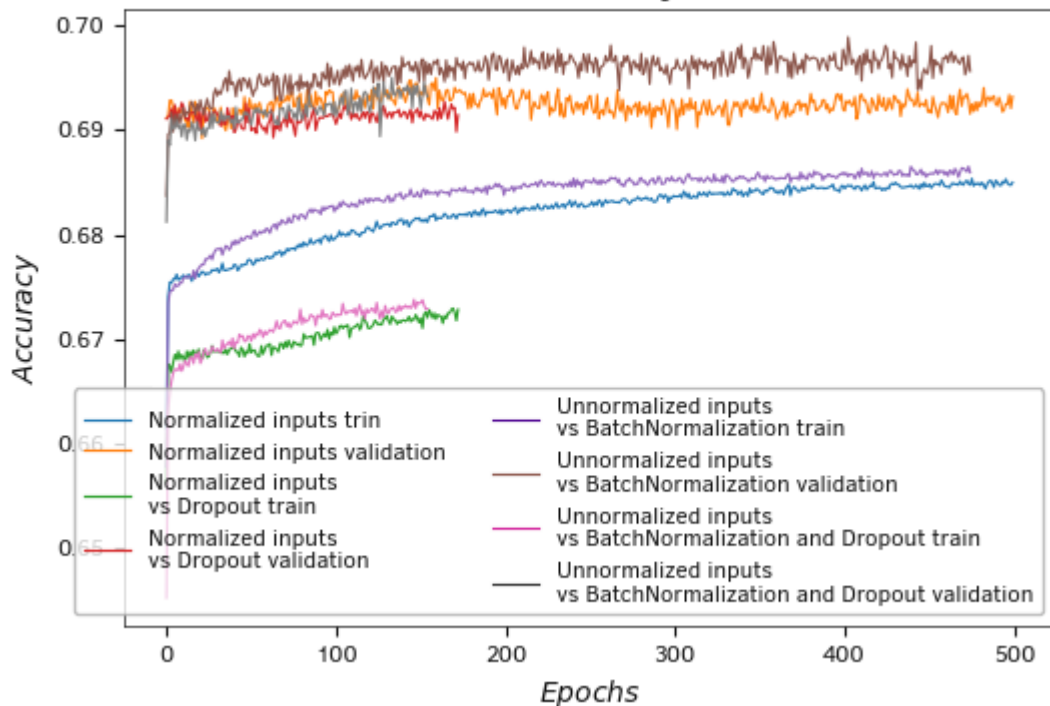
The evaluation of models on the test dataset also showed deterioration in model performance when using the *Dropout* layer, both for mean square error and for *Accuracy*. The reasons for such a phenomenon can only be speculated upon. One of the possible reasons can be attributed to the use of models that are too simple. The models didn't have too many neurons, and masking some of them reduces the capabilities of the model, which are already limited by the small number of neurons being used.

Model training dynamics 1 hidden layer



Comparative model testing with Dropout

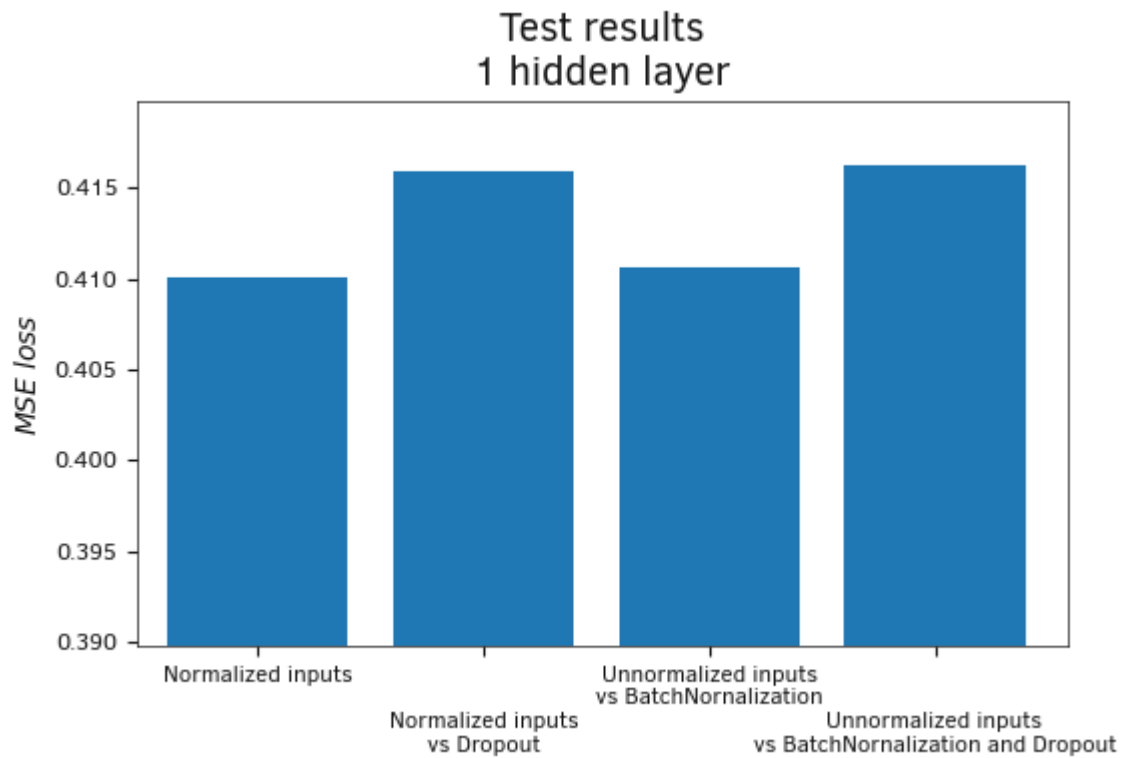
Model training dynamics 1 hidden layer



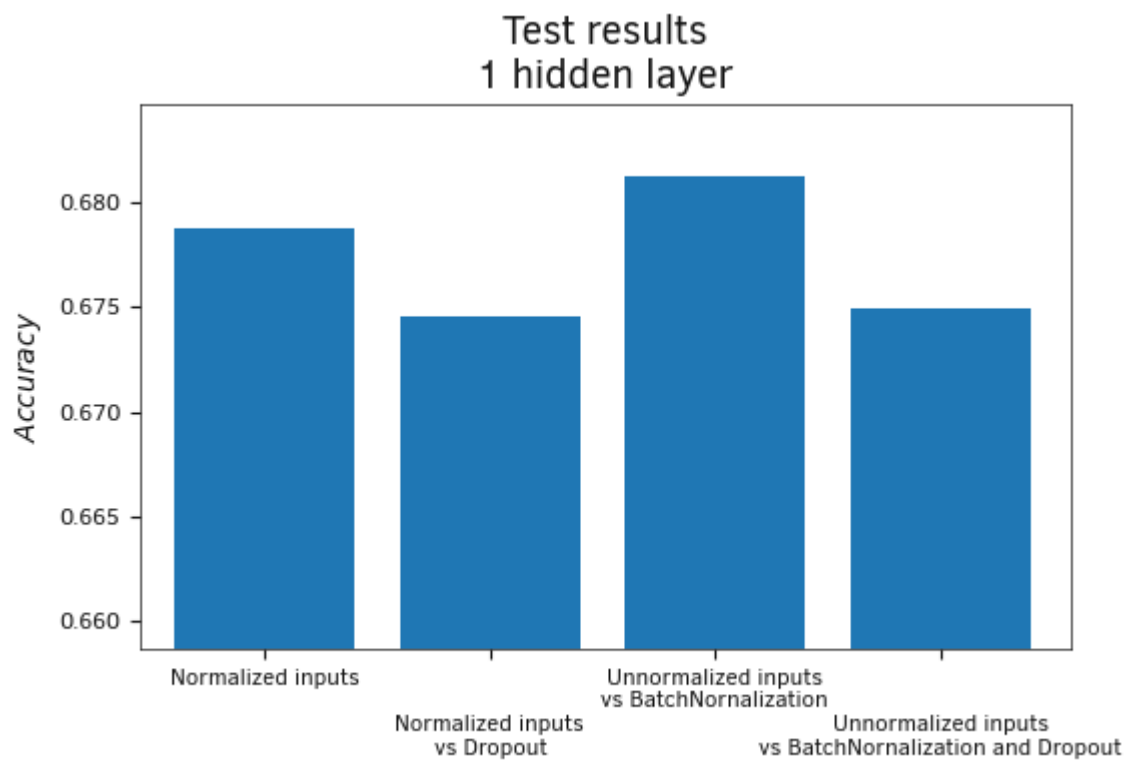
Comparative model testing with Dropout

On the other hand, we chose uncorrelated variables at the data selection stage. Probably due to the small number of features being used and the absence of strong correlations between them, co-adaptation might not be highly developed in our models. As a result, the negative impact of using

Dropout in terms of degrading the model performance may have outweighed the positive effects of the method.

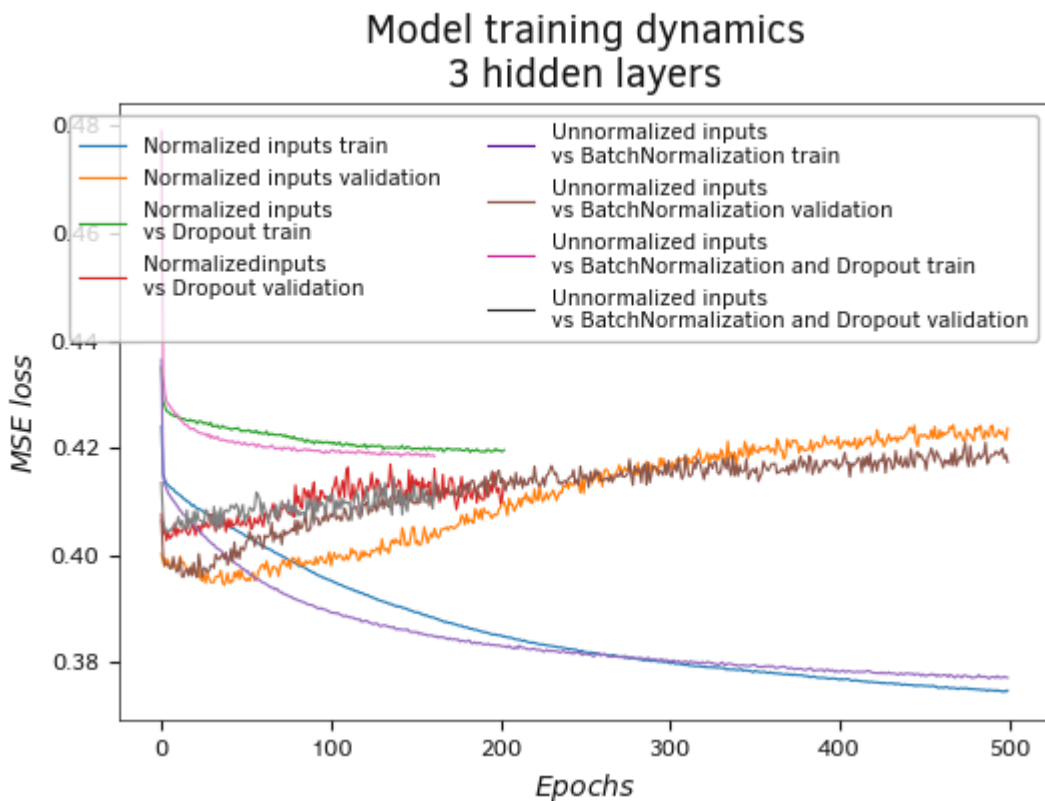


Comparative model testing with Dropout (test sample)



Comparative model testing with Dropout (test sample)

That is just my guess. I do not have enough information to draw certain conclusions. Additional tests will be required. However, it is more the focus of scientific work, while our goal is the practical use of models. We conduct experiments with various architectural solutions and choose the best one for each specific task.



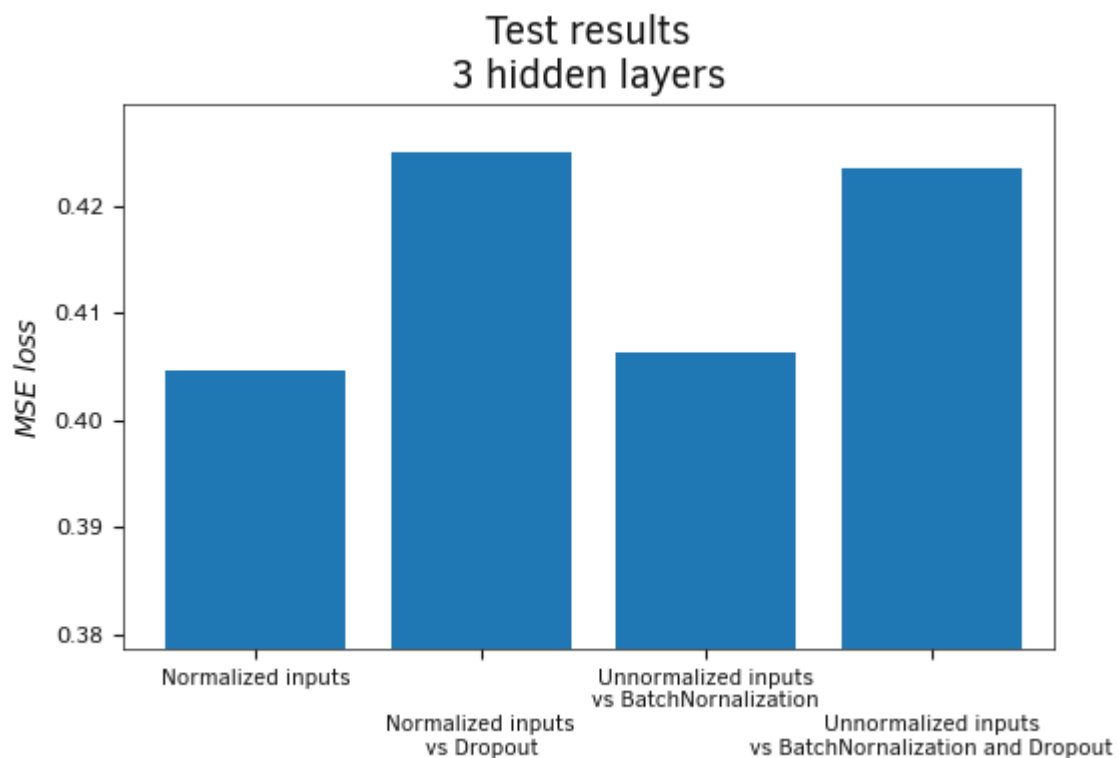
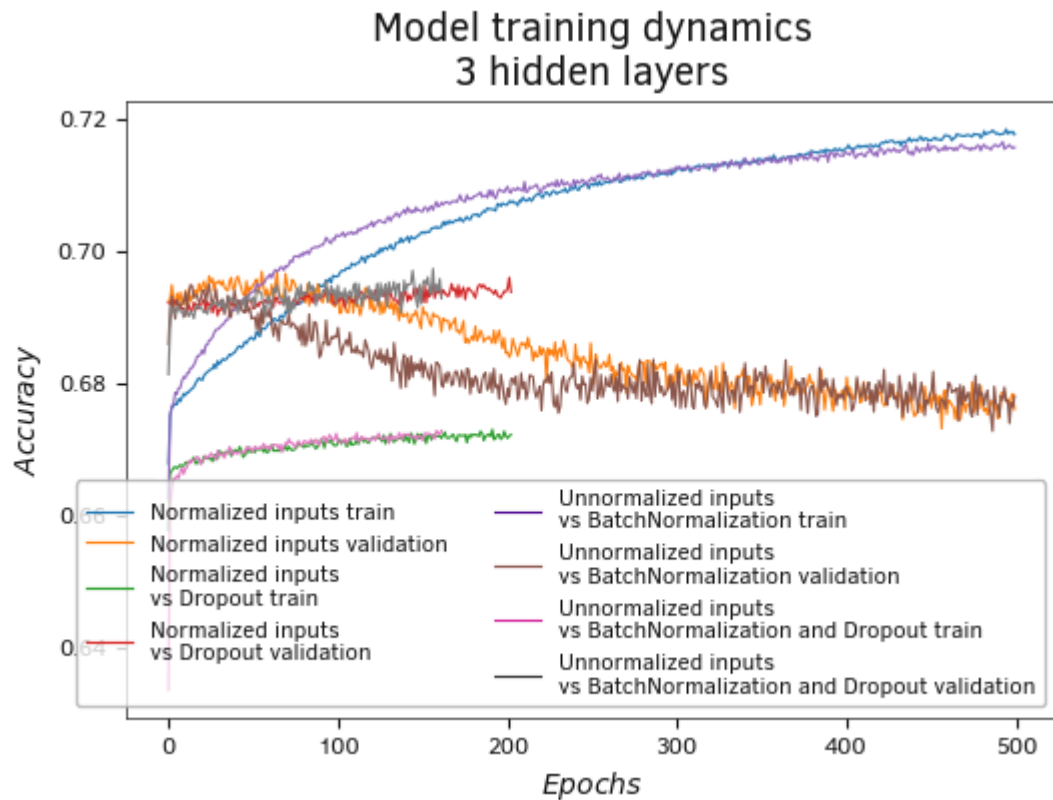
Comparative model testing with Dropout

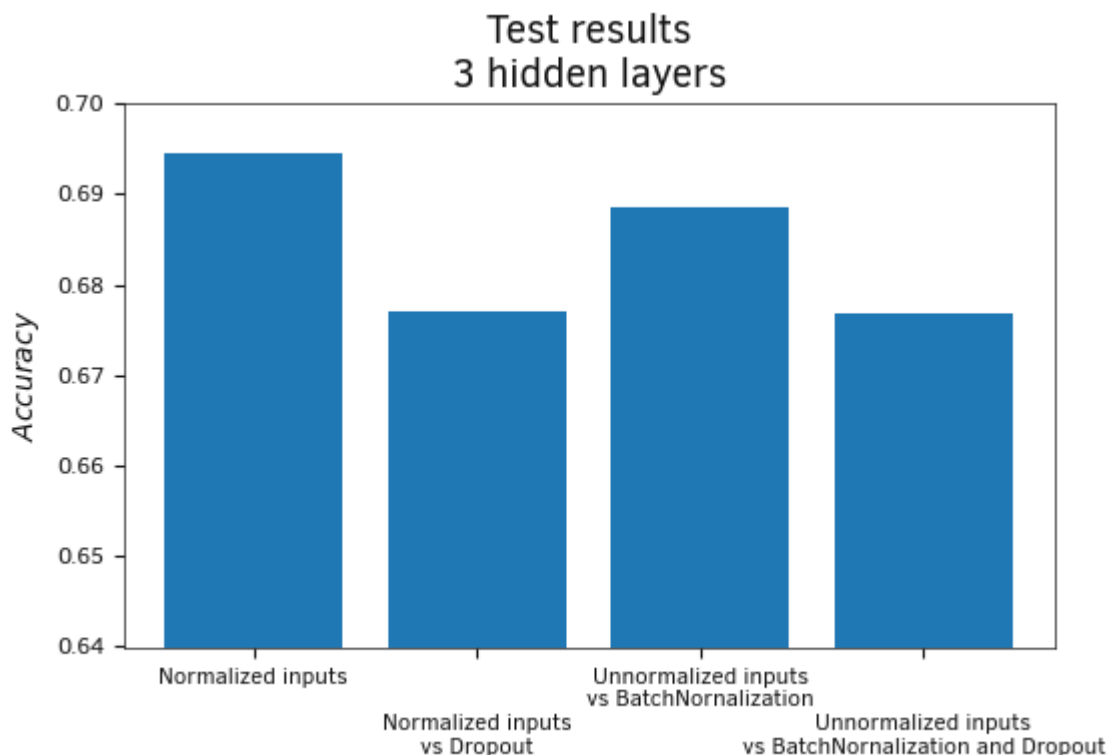
The second test was performed using a *Dropout* layer before each fully connected layer in models with three hidden layers. Again, models with the *Dropout* layer performed worse than other models in the learning process. However, during the validation process, the situation tends to change somewhat. While models without the use of *Dropout* layers tend to decrease their performance on the validation stage with an increase in the number of training epochs, models using this technology slightly improve their positions or remain at a similar level.

This suggests that the use of a *Dropout* layer reduces the likelihood of model overfitting.

The graph showing the dynamics of *Accuracy* values during training confirms the conclusion made earlier. With the increasing number of model training epochs, without the use of the *Dropout* layer, there is a widening gap between the values in the training and validation phases. This indicates a retraining of the model. For models using *Dropout* technology, the gap is narrowing. This supports the earlier conclusion that the use of a *Dropout* layer reduces the tendency of the model to overfit.

On the test dataset, models using the *Dropout* layer showed worse results in both metrics.





Comparative model testing with Dropout (test sample)

In this section, we have run comparative model testing with *Dropout* technology and without it. The following conclusions can be drawn from the tests:

- The use of the Dropout technology helps reduce the risk of model overfitting.
- The effectiveness of the Dropout technology increases as the model size grows.

7. Testing trading capabilities of the model

We have done quite a bit of work studying various architectural solutions for organizing neural networks. We have created a library for building various neural layers, and now with its help, we can create different neural network models to find the best solution for our tasks. This is all very good and useful, of course. However, we are doing this not just for the sake of science or self-enlightenment, although that is certainly not a bad reason to study something. In this case, we embarked on the study of the organization of neural networks and their architectural solutions with a practical purpose to find a solution for use in the financial markets. There are two visions for such a solution:

- Creating an indicator based on a neural network model.
- Creating an Expert Advisor capable of executing trading operations based on the signals of the neural network model.

We will not discuss which of the above options is preferable. In fact, this is a rhetorical question because it depends on the user's personal preferences. In any case, we need to organize the correct operation of the model and the interpretation of its signals.

At the same time, we would like to assess the expected profitability of our model. To conduct such work, the *MetaTrader 5* terminal offers the use of the *Strategy Tester*.

In this chapter, we move from the theoretical study and creation of neural networks to the practical application of the developed models in the financial sector. Our goal is to evaluate the effectiveness of neural networks for creating indicators and Expert Advisors capable of performing trading operations in financial markets. We'll start by examining the [functionality of the MetaTrader 5 Strategy Tester](#), which is a key tool for evaluating the performance of our models.

Next, we will move on to creating an [Expert Advisor template](#) using the MQL5 programming language. This will allow us to apply our models in real trading conditions. Then we will focus on [creating a model for testing](#). In this part, we will see how to properly prepare and configure the model to produce the most accurate and useful results.

After that, we will discuss the [definition of Expert Advisor parameters](#), which includes setting various parameters and options that optimize the Expert Advisor operation in accordance with the user's trading strategies and goals. Finally, we will [test the model](#) using new data, which is a critical step in assessing the model's ability to adapt to changing market conditions and predict future trading signals.

This chapter focuses on the practical application of the developed neural networks in real-world trading strategies, covering model testing and optimization stages.

7.1 Introduction to MetaTrader 5 Strategy Tester

MetaTrader 5 provides a built-in strategy tester which enables the validation of trading robot performance. This tool allows you to evaluate the Expert Advisor effectiveness and select the best input parameters before deploying it on a live trading account.

The entire operation of the strategy tester is based on the historical quotes of currencies and stocks. The tester automatically downloads tick history from the brokerage company's trading server and takes into account contract specifications. Therefore, the developer doesn't need to do anything manually. This allows for easy and highly accurate reproduction of all trading environment conditions, down to millisecond intervals between ticks on different symbols. The robot analyzes the accumulated quotes and executes virtual trades according to the algorithm embedded in it. This allows the evaluation of how well the strategy would have performed in the past.

Moreover, the *MetaTrader 5* strategy tester is multi-currency. All robots tested in it can receive information about all financial instruments available on the registered account in the terminal and can trade on them. Thus, the tool allows testing even complex Expert Advisors capable of analyzing multiple currencies or stocks and their correlation.

The main advantage of such testing is the evaluation of a trading robot under conditions very close to real without its actual operation in the market. Moreover, it takes much less time since historical ticks are generated by the tester much faster than the real market. This is an undeniable advantage of the strategy tester, but far from its only capability.

The *MetaTrader 5* Strategy Tester offers several testing modes. They allow selecting the optimal balance between speed and quality according to the user's needs. The 'Every tick' mode is intended for the most accurate testing; in this case, the simulated conditions will be closest to the real ones. The '1 minute OHLC' mode allows testing a strategy faster with a sufficient level of accuracy. If a very quick and rough estimate is needed, choose the 'Open prices only' mode, in which testing is conducted using only bar opening prices. The highest quality is offered by the 'Every tick based on real ticks' mode, but it also requires the maximum time investment.

7. Testing trading capabilities of the model

The capabilities of the tester are not limited to just testing. It can also be used to solve mass optimization tasks. In the mathematical calculations mode, trading history is not used, and market conditions are not modeled, while only the mathematical calculations embedded in the Expert Advisor are executed.

Stress testing is an opportunity to further approximate the conditions of testing a trading robot to real ones. The mode of arbitrary execution delays simulates network delays in transmitting and processing trading requests, as well as simulates execution delays by dealers during real trading.

One of the main features of the strategy tester is the presentation of Expert Advisor testing results. It's not just dry figures, such as the profit generated by the trading robot during testing. The presentation also includes a wealth of statistical performance metrics:

- Profit and loss percentage ratio
- Number of winning and losing trades
- Risk factor
- Expected payoff

And this is far from a complete list. Additionally, the results of strategy testing are also provided in graphical form, making the analysis of the trading strategy even more convenient and clear.

The existing visual testing mode allows real-time tracking of the robot's trading on historical price data. All Expert Advisor trades are displayed on the chart, making them easy to analyze. The testing process can be slowed down or paused to observe how trading is conducted at specific time intervals.

Visualization mode is not only an opportunity to see how the robot trades. In addition, it allows checking the performance of custom technical indicators. For example, before purchasing through the [Market](#), you can assess its behavior on historical data.

An important function of the strategy tester is the optimization of the trading robot, which allows you to find the best input parameters for a specific Expert Advisor. Various optimization modes allow finding optimal parameters to make the trading robot as profitable and robust as possible, with minimal risk, and so on.

During optimization, one trading robot is tested with different input parameters. After testing is completed, the results of the runs can be compared, and the settings that best meet the requirements placed on the robot can be selected.

The number of input parameter combinations during optimization can reach tens or hundreds of thousands. As a result, optimization can become a very time-consuming process, which can still be significantly reduced using genetic algorithms. This feature disables the sequential enumeration of all input parameter combinations and selects only those that best meet the optimization criteria. In subsequent stages, the optimal combinations are crossbred until the results stop improving. This reduces the number of combinations and the overall optimization time many times over.

In addition, the strategy tester works in a multi-thread mode and allows the utilization of all CPU cores. This will run an Expert Advisor on each core with its own set of parameters. Furthermore, for a large pool of tasks, the strategy tester provides the ability to connect to cloud computing through the use of the *MQL5 Cloud Network*. This is a network of cloud computing resources that combines thousands of computers worldwide. The strategy tester can use its practically limitless computational power. With the *MQL5 Cloud Network*, optimization that would take months in regular mode can be completed in just a few hours.

The strategy tester provides powerful tools for visual analysis of optimization results in both 2D and 3D modes. For example, in a two-dimensional representation, you can analyze the dependencies of the final result on two parameters simultaneously, while in 3D, you can see the entire picture of finding the best result during optimization.

In addition to the built-in capabilities, you can use your own visualization methods. There is no need to prepare, export, or process data in an external application. Simply display the optimization results on the screen in real time during its execution.

The built-in forward testing feature helps eliminate "over-optimization" or parameter fitting. With this option enabled, the history of currency and stock quotes is divided into two parts. Optimization occurs in the first segment of the history, and the second segment is used only to confirm the results. If the trading robot efficiency is equally high in both segments, it means that the trading system has the best parameters and parameter fitting is practically eliminated.

The strategy tester is an indispensable tool for Expert Advisor developers. Without it, it is practically impossible to write an efficient trading robot. It saves time and assists in creating a truly profitable tool for use in financial markets.

7.2 Developing an Expert Advisor template using MQL5

To effectively assess the performance of our model in the strategy tester, we need to encapsulate it in a trading robot. Hence, in this section, I decided to present a small template of an Expert Advisor utilizing a neural network as the primary and sole decision-making block. I must clarify that this is just a template, aimed at demonstrating implementation principles and approaches. Its code is considerably simplified and is not intended for use on real accounts. Nevertheless, it is fully functional and can serve as a foundation for constructing a working Expert Advisor. Additionally, I want to caution you that financial market trading implies high-risk investments. You perform all your operations at your own risk and under your full responsibility, including if you use Expert Advisors on your accounts. Of course, unless the creators of such trading robots offer explicit guarantees, subject to your individual agreements.

Regarding Expert Advisors, before installing them on your real trading accounts and entrusting them with your funds, carefully study their parameters and configuration options. Also, validate their performance across various modes in the strategy tester and on demo accounts.

I hope this clarification is comprehensible to everyone. Now, let's proceed to the implementation of the template. Primarily, as I mentioned earlier, the presented template is significantly simplified, omitting several essential functions required for Expert Advisors that are not related to the operation of our model. In particular, the Expert Advisor completely lacks a money management block. For simplicity, we use a fixed *Lot*. We also use a fixed *StopLoss* and set the range for take profit between *MinTarget* and *MaxTP*. This approach to setting the take profit stems from the fact that in the models we are testing, the second target variable precisely represented the distance to the nearest future extreme point.

```

input string      Model = "our_model.net";
input int         BarsToPattern = 40;
input bool        Common = true;
input ENUM_TIMEFRAMES TimeFrame = PERIOD_M5;
input double      TradeLevel=0.9;
input double      Lot = 0.01;
input int         MaxTP= 500;

```

7. Testing trading capabilities of the model

```
input double      ProfitMultiply = 0.8;
input int         MinTarget=100;
input int         StopLoss=300;
```

Additionally, I have opted for a simplified approach to model usage. Rather than creating and training the model within the Expert Advisor, I approached it from a different angle. In all the scripts we created to test the architectural solutions of neural layers, we saved the trained models. So why not simply load one of the trained models? You can create and train your own model, and then just specify the file name of the trained model in the external *Model* parameter and use it. All that remains is to specify the storage location of the *Common* file, the number of bars describing one pattern *BarsToPattern*, and the *TimeFrame* used. Also, to make a decision, we will indicate the minimum predicted probability of profit *TradeLevel*.

To increase the probability of closing a trade at the take profit level, we add the *ProfitMultiply* parameter in which we indicate the coefficient of confidence in the predicted movement strength. In other words, when specifying the take profit level for an open position, we will adjust the size of the expected movement by this coefficient.

Using the *Common* parameter to specify the location of the trained model file is quite important, as strange as it may seem. The reason is that access to files in *MetaTrader 5* is restricted within its sandbox. Each terminal installed on the computer has its own sandbox. So, each of the two terminals installed on the same computer works in its own sandbox and does not interfere with the second. For cases where data exchange is needed between terminals on the same computer, a separate common folder is used. So, the *true* value of the *Common* parameter indicates the use of this common folder.

When using the strategy tester optimization mode, each testing agent works in its own separate sandbox, even within the same trading terminal. Therefore, to provide equal access to the trained model for all testing agents, you need to place it in the common terminal folder and specify the corresponding flag value.

After declaring the external parameters of our Expert Advisor, we include our library for working with neural network models *neuronnet.mqh* and the standard library for trading operations *Trade\Trade.mqh* in the global space.

```
#include "..\..\Include\NeuroNetworksBook\realization\neuronnet.mqh"
#include <Trade\Trade.mqh>

CNet *net;
CTrade *trade;
datetime lastbar = 0;
int h_RSI;
int h_MACD;
```

Next, we declare global variables:

- *net* – pointer to the model object
- *trade* – pointer to the object of trade operations
- *lastbar* – time of the last analyzed bar, used to check the new candlestick opening event
- *h_RSI* – handle of the *RSI* indicator
- *h_MACD* – handle of the *MACD* indicator

Our template will contain a minimum set of functions. But this does not mean that your Expert Advisor should contain exactly the same number of them.

In the *OnInit* function, we initialize the Expert Advisor. At the beginning of the function, we create a new instance of a neural network object and immediately check the result of the operation. If the creation of a new object is successful, we load the model from the specified file. Of course, we verify the result of these operations.

```
int OnInit()
{
    //---
    if(!(net = new CNet()))
    {
        PrintFormat("Error creating Net: %d", GetLastError());
        return INIT_FAILED;
    }
    if(!net.Load(Model, Common))
    {
        PrintFormat("Error loading mode %s: %d", Model, GetLastError());
        return INIT_FAILED;
    }
    net.UseOpenCL(UseOpenCL);
}
```

After loading the model, we load the required indicators. Within the framework of this book, we trained models on [historical datasets](#) from two indicators: *RSI* and *MACD*. As always, we check the result of the operation.

```
h_RSI = iRSI(_Symbol, TimeFrame, 12, PRICE_TYPICAL);
if(h_RSI == INVALID_HANDLE)
{
    PrintFormat("Error loading indicator %s", "RSI");
    return INIT_FAILED;
}
h_MACD = iMACD(_Symbol, TimeFrame, 12, 48, 12, PRICE_TYPICAL);
if(h_MACD == INVALID_HANDLE)
{
    PrintFormat("Error loading indicator %s", "MACD");
    return INIT_FAILED;
}
```

The next step is to create an instance of an object to perform trading operations. Again, we check the object creation result and set the order execution type.

```
void OnDeinit(const int reason)
{
    if(!net)
        delete net;
    if(!trade)
        delete trade;
    IndicatorRelease(h_RSI);
    IndicatorRelease(h_MACD);
}
```

At the end of the function, we set the initial value for the time of the last bar and exit the function.

Immediately after the initialization function, we create the *OnDeinit* deinitialization function, in which we delete the objects created in the program. We also close the indicators.

7. Testing trading capabilities of the model

```
void OnDeinit(const int reason)
{
    if(CheckPointer(net) == POINTER_DYNAMIC)
        delete net;
    if(CheckPointer(trade) == POINTER_DYNAMIC)
        delete trade;
    IndicatorRelease(h_RSI);
    IndicatorRelease(h_MACD);
}
```

We write the entire algorithm of the Expert Advisor in the *OnTick* function. The terminal calls this function when a new tick event occurs on the chart with the program running. At the beginning of the function, we check if a new bar has opened. If the candlestick has already been processed, we exit the function and wait for a new tick. The essence of this action is simple: we feed our model with information only from closed candlesticks, and to ensure the information is as up-to-date as possible, we do this at the opening of a new candlestick.

```
void OnTick()
{
    if(lastbar >= iTime(_Symbol, TimeFrame, 0))
        return;
    lastbar = iTime(_Symbol, TimeFrame, 0);
}
```

There are no functions in our template that process every tick, so we will only perform actions at the opening of a new candlestick. If you include functions in your program that need to process every tick, such as trailing stops, moving orders to breakeven, or anything else, you will need to call these functions before checking for the new candlestick event.

When a new candlestick event occurs, we load information from our indicators into local dynamic arrays. Here we need to be sure to check the result of the operations.

```
double macd_main[], macd_signal[], rsi[];
if(h_RSI == INVALID_HANDLE || CopyBuffer(h_RSI, 0, 1, BarsToPattern, rsi) <= 0)
{
    PrintFormat("Error loading indicator %s data", "RSI");
    return;
}
if(h_MACD == INVALID_HANDLE || CopyBuffer(h_MACD, MAIN_LINE, 1, BarsToPattern, macd_main) <= 0 || CopyBuffer(h_MACD, SIGNAL_LINE, 1, BarsToPattern, macd_signal) <= 0)
{
    PrintFormat("Error loading indicator %s data", "MACD");
    return;
}
```

Once the indicator data is loaded, we create an instance of a data buffer object to collect the current state. Also, we run a loop to fill the data buffer with the current state of the indicators. Here we should organize exactly the same sequence of values describing the current state, as we filled in the training dataset file. Otherwise, the result of the model will be unpredictable.

```
CBufferType *input_data = new CBufferType();
if(!input_data)
{
    PrintFormat("Error creating Input data array: %d", GetLastError());
    return;
}
```



```

    }
    if(!input_data.BufferInit(BarsToPattern, 4, 0))
        return;

    for(int i = 0; i < BarsToPattern; i++)
    {
        if(!input_data.Update(i, 0, (TYPE)rsi[i]))
        {
            PrintFormat("Error adding Input data to array: %d", GetLastError());
            delete input_data;
            return;
        }

        if(!input_data.Update(i, 1, (TYPE)macd_main[i]))
        {
            PrintFormat("Error adding Input data to array: %d", GetLastError());
            delete input_data;
            return;
        }

        if(!input_data.Update(i, 2, (TYPE)macd_signal[i]))
        {
            PrintFormat("Error adding Input data to array: %d", GetLastError());
            delete input_data;
            return;
        }

        if(!input_data.Update(i, 3, (TYPE)(macd_main[i] - macd_signal[i])))
        {
            PrintFormat("Error adding Input data to array: %d", GetLastError());
            delete input_data;
            return;
        }
    }
    if(!input_data.Reshape(1,input_data.Total()))
        return;

```

When we have fully gathered the description of the current state in the data buffer, we proceed to work on our model. First, we validate the model pointer and then call the feed-forward method. After a successful completion of the feed-forward method, we obtain its results in a local buffer. We do not create a new instance of an object for the results buffer; instead, we use the input data buffer.

```

    if(!net)
    {
        delete input_data;
        return;
    }
    if(!net.FeedForward(input_data))
    {
        PrintFormat("Error of Feed Forward: %d", GetLastError());
        delete input_data;
        return;
    }

```

7. Testing trading capabilities of the model

Next comes the decision-making block based on signals from our model. As a result of the feed-forward pass, the model returns two numbers. The first number is trained to determine the direction of the upcoming movement, while the second one determines the distance to the nearest extreme point. Thus, to execute operations, we will rely on both signals, which should be aligned.

First, we check the buy signal. The parameter responsible for the direction of movement must be positive. We also immediately check for open positions. If there are open long positions, we refrain from opening a new position and exit the function until the next tick.

Please note that we do not check for the presence of an open sell position. In our simplified version of the EA, we trust the forecasts of our model and expect all open positions to be closed by the take profit or stop loss. Consequently, we excluded the position management block from our Expert Advisor. As a result, we expect the possibility of simultaneously holding two opposite positions, which is only possible with position hedging. Therefore, testing such an Expert Advisor is possible only on the corresponding accounts.

This approach allows us to assess the effectiveness of forecasts made by our model. But when building Expert Advisors for real market usage, I would recommend considering and adding a position management block to the Expert Advisor.

```
if(!net.GetResults(input_data))
{
    PrintFormat("Error of Get Result: %d", GetLastError());
    delete input_data;
    return;
}
if(input_data.At(0) > 0.0)
{
    bool opened = false;
    for(int i = 0; i < PositionsTotal(); i++)
    {
        if(PositionGetSymbol(i) != _Symbol)
            continue;
        if(PositionGetInteger(POSITION_TYPE) == POSITION_TYPE_BUY)
            opened = true;
    }

    if(opened)
    {
        delete input_data;
        return;
    }
}
```

If there are no open long positions, we check the strength of the signal (probability of movement in the desired direction) and the expected movement to the upcoming extreme point. If at least one of the parameters does not meet the requirements, we exit the function until the next tick.

```
if(input_data.At(0) < TradeLevel ||
   input_data.At(1) < (MinTarget * SymbolInfoDouble(_Symbol, SYMBOL_POINT)))
{
    delete input_data;
    return;
}
```

7. Testing trading capabilities of the model

If, however, a decision is made to open a position, we determine the stop loss and take profit levels and send a buy order.

```
double tp = SymbolInfoDouble(_Symbol, SYMBOL_BID) + MathMin(input_data.At(1) *
    ProfitMultiply, MaxTP * SymbolInfoDouble(_Symbol, SYMBOL_POINT));
double sl = SymbolInfoDouble(_Symbol, SYMBOL_BID) -
    StopLoss * SymbolInfoDouble(_Symbol, SYMBOL_POINT);
trade.Buy(Lot, _Symbol, 0, sl, tp);
}
```

The algorithm for making a sell decision is organized in a similar way.

```
if(input_data.At(0) < 0)
{
    bool opened = false;
    for(int i = 0; i < PositionsTotal(); i++)
    {
        if(PositionGetSymbol(i) != _Symbol)
            continue;
        if(PositionGetInteger(POSITION_TYPE) == POSITION_TYPE_SELL)
            opened = true;
    }

    if(opened)
    {
        delete input_data;
        return;
    }

    if(input_data.At(0) > -TradeLevel ||
        input_data.At(1) > -(MinTarget * SymbolInfoDouble(_Symbol, SYMBOL_POINT)))
    {
        delete input_data;
        return;
    }

    double tp = SymbolInfoDouble(_Symbol, SYMBOL_BID) + MathMax(input_data.At(1) *
        ProfitMultiply, -MaxTP * SymbolInfoDouble(_Symbol, SYMBOL_POINT));
    double sl = SymbolInfoDouble(_Symbol, SYMBOL_BID) +
        StopLoss * SymbolInfoDouble(_Symbol, SYMBOL_POINT);
    trade.Sell(Lot, _Symbol, 0, sl, tp);
}
delete input_data;
}
```

After performing all the operations according to the described algorithm, we delete the buffer of the current state and exit the function.

The Expert Advisor has been made very simplified, but it will also allow you to test the operation of our model in the MetaTrader 5 strategy tester.

7.3 Creating a model for testing

In the previous section, we created a template for an Expert Advisor to test the feasibility of using our neural network models for conducting trading operations in financial markets. This is a universal template that can work with any model. However, it has limited parameters for the description of one candlestick and for the configuration of the results layer. As a result of the model operation, it should return a tensor of values that the decision-making block in the template can unambiguously interpret.

For testing purposes, I decided to build a new model that involves multiple types of neural layers. We will create and train the model using a script. The script format is familiar to us from the numerous tests that we examined in this book. We will create a new script in the file *gpt_not_norm.mq5*. We will save the new script file in the *gpt* subdirectory of our book in accordance with the [file structure](#).

At the script's global level, we will declare two constants:

- *BarsInHistory* – number of bars in the training dataset
- *ModelName* – file name to save the trained model

Next, we define the external parameters of the script. First of all, this is the name of the file with the training dataset *StudyFileName*. Please note that we are using a dataset without prior data normalization. In the previous section, in our Expert Advisor template, we did not configure data preprocessing, so the entire calculation relies on using batch normalization layers. The tests we conducted earlier confirm the possibility of such a replacement.

The external parameter *OutputFileName* contains the name of the file for writing the dynamics of changes in the model error during the training process.

We plan to use a block with the *GPT* architecture. For such an architecture, it's common to use a parameter to specify the length of the internal buffer sequence for the pattern. To request this parameter from the user, we will create an external parameter *BarsToLine*.

Next comes the set of parameters that has become standard for such scripts:

- *NeuronsToBar* – number of input layer neurons per bar
- *UseOpenCL* – flag for using OpenCL
- *BatchSize* – batch size between weight matrix updates
- *LearningRate* – learning rate
- *HiddenLayers* – number of hidden layers
- *HiddenLayer* – number of neurons in the hidden layer
- *Epochs* – number of iterations for updating the weight matrix before the training process stops.

```
#define HistoryBars          40
#define ModelName            "gpt_not_norm.net"
//+-----+
//| External parameters for the script |
//+-----+
// Name of the file with the training sample
input string StudyFileName = "study_data_not_norm.csv";
// File name for recording error dynamics
input string OutputFileName = "loss_study_gpt_not_norm.csv";
// Depth of the analyzed history
input int BarsToLine = 60;
```

7. Testing trading capabilities of the model

```
// Number of input layer neurons per 1 bar
input int      NeuronsToBar    = 4;
// Use OpenCL
input bool     UseOpenCL       = false;
// Batch size to update the weight matrix
input int      BatchSize       = 10000;
// Learning factor
input double   LearningRate    = 0.0003;
// Number of hidden layers
input int      HiddenLayers    = 2;
// Number of neurons in one hidden layer
input int      HiddenLayer     = 60;
// Number of iterations to update the weight matrix
input int      Epochs          = 5000;
```

After declaring external parameters, we add our neural network model library to the script.

```
//+-----+
//| Connecting the neural network library |
//+-----+
#include "..\..\..\Include\NeuroNetworksBook\realization\neuronnet.mqh"
```

This is where the work in the global field ends. Let's continue writing the script code in the body of the *OnStart* function. In the body of the function, we use a structured approach to call individual functions, each of which performs specific actions.

```
void OnStart(void)
{
    VECTOR loss_history;
    //-- prepare a vector to store the history of network errors
    if(!loss_history.Resize(0, Epochs))
    {
        Print("Not enough memory for loss history");
        return;
    }
    CNet net;
    //-- 1. network initialization
    if(!NetworkInitialize(net))
        return;
    //-- 2. loading training sample data
    CArrayObj data;
    CArrayObj result;
    if(!LoadTrainingData(StudyFileName, data, result))
        return;
    //-- 3. network training
    if(!NetworkFit(net, data, result, loss_history))
        return;
    //-- 4. saving network error history
    SaveLossHistory(OutputFileName, loss_history);
    Print("Done");
}
```

7. Testing trading capabilities of the model

The first function in our script is the model initialization function *NetworkInitialize*. In its parameters, this function receives a pointer to the model object that needs to be initialized.

The function body provides two options for model initialization. First, we attempt to load a pre-trained model from the file specified in the external parameters of the script and check the operation result. If the model is successfully loaded, we skip the block that creates a new model and continue working with the loaded model. This capability enables us to stop and resume the learning process if necessary.

```
bool NetworkInitialize(CNet &net)
{
    if(net.Load(ModelName))
    {
        printf("Loaded pre-trained model %s", ModelName);
        net.SetLearningRates((TYPE)LearningRate, (TYPE)0.9, (TYPE)0.999);
        net.UseOpenCL(UseOpenCL);
        net.LossSmoothFactor(BatchSize);
        return true;
    }
}
```

If the loading of a pre-trained model fails, we create a new neural network. First, we create a dynamic array to store pointers to objects describing neural layers and then we immediately call the *CreateLayersDesc* function to create the architecture description of our model.

```
CArrayObj layers;
//--- create a description of the network layers
if(!CreateLayersDesc(layers))
    return false;
```

As soon as our dynamic array of objects contains the complete description of the model to be created, we call the model generation method, specifying in the parameters a pointer to the dynamic array describing the model, the loss function, and the model optimization parameters.

```
//--- initialize the network
if(!net.Create(&layers, (TYPE)LearningRate, (TYPE)0.9, (TYPE)0.999, LOSS_MSE,
              0, (TYPE)0))
{
    PrintFormat("Error of init Net: %d", GetLastError());
    return false;
}
```

We ensure to verify the result of the operation.

After creating the model, we set the user-specified flag for using OpenCL technology and the error smoothing range.

```
net.UseOpenCL(UseOpenCL);
net.LossSmoothFactor(BatchSize);
return true;
}
```

This concludes the model initialization function. Let's now consider the algorithm of the *CreateLayersDesc* function that creates the architecture description of the model. In the parameters, the function receives a pointer to a dynamic array object describing the model architecture. In the body of the function, we immediately clear the received array.

7. Testing trading capabilities of the model

```
bool CreateLayersDesc(CArrayObj &layers)
{
    layers.Clear();
```

First, we create the initial data layer. The algorithm for creating all neural layers will be the same, so we begin by initiating a new object for describing the neural layer. As always, we verify the result of the operation, that is, check the creation of a new object.

```
    CLayerDescription *descr;
    ///-- create an initial data layer
    if(!(descr = new CLayerDescription()))
    {
        PrintFormat("Error creating CLayerDescription: %d", GetLastError());
        return false;
    }
```

Once the neural layer description object is created, we fill it with data sufficient to unambiguously understand the architecture of the neural layer being created.

```
    descr.type          = defNeuronBase;
    int prev_count = descr.count = NeuronsToBar * GPT_InputBars;
    descr.window        = 0;
    descr.activation     = AF_NONE;
    descr.optimization = None;
```

We will input information about the last three candlesticks into the created model. In fact, this is not enough, both in terms of the amount of information for the neural network to make a decision and from the practical trading perspective. However, we should remember that we will use blocks with the *GPT* architecture in our model. This architecture involves the accumulation of historical data inside a block, compensating for the lack of information. At the same time, using a small amount of initial data allows for a significant reduction in computational operations at each iteration. Thus, the size of the initial data layer is determined as the product of the number of elements to describe one candlestick and the number of analyzed candlesticks. In our case, the number of description elements for one candlestick is specified in the external parameter *NeuronsToBar*, and the number of analyzed candlesticks is specified by the *GPT_InputBars* constant.

The initial data layer does not use either an activation function or parameter optimization. Note that we write the initial data directly to the results buffer.

```
    if(!layers.Add(descr))
    {
        PrintFormat("Error adding layer: %d", GetLastError());
        delete descr;
        return false;
    }
```

Once we have filled the architecture description object of the neural layer with the necessary set of data, we add it to our dynamic array of pointers to objects.

I would like to remind you that we did not pre-process the initial data. Therefore, in the neural network architecture, we have included the creation of a batch data normalization layer immediately after the initial data layer. According to the above algorithm, we instantiate a new object describing the neural layer. It is important to verify the result of the object creation operation, as in the next stage, we will be populating the elements of this object with the necessary description of the architecture of the

7. Testing trading capabilities of the model

created neural layer. Attempting to access the object elements with an invalid pointer will result in a critical error.

```
//--- create a data normalization layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
```

In the description of the neural layer being created, we specify the type of the neural layer *defNeuronBatchNorm*, which corresponds to the batch normalization layer. We set the sizes of the neural layer and the window of initial data to be equal to the size of the previous input data neural layer.

We will indicate the batch size at the batch size level between updates of the weight matrix, which the user specified in the external parameter *BatchSize*.

Similar to the previous layer, the batch normalization layer does not employ an activation function. However, it introduces the *Adam* optimization method for trainable parameters.

```
descr.type          = defNeuronBatchNorm;
descr.count         = prev_count;
descr.window        = prev_count;
descr.activation     = AF_NONE;
descr.optimization   = Adam;
descr.batch          = BatchSize;
```

After specifying all the necessary parameters for describing the neural layer to be created, we add a pointer to the object to the dynamic array of pointers describing the architecture of our model.

```
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

As we have discussed previously, four parameters in the description of one candlestick might be insufficient. Therefore, it would be beneficial to add a few more parameters. To use machine learning methods in conditions of a shortage of parameters, a number of approaches have been developed that have been combined into the field of *Feature Engineer*. One such approach involves the use of convolutional layers, in which the number of filters exceeds the size of the input window. The logic of this approach is that the description vector of one element is considered as the coordinates of a certain point representing the current state in an N -dimensional space, where N is the length of the description vector of one element. By performing convolution, we project this point onto the convolution vector. We use exactly this property when compressing data and reducing its dimensionality. The same property will be used to increase data dimensionality. As you can see, there is no contradiction here with the previously studied approach to using convolutional layers. We simply use the number of filters exceeding the description vector of one element and thereby increase the space dimension. Let's use the described method and create the next convolutional layer with the number of filters being twice the number of elements in the description of one candlestick. It should be noted that in this case we are making a convolutional layer within the description of one candlestick, so the size of the initial data window and its step size will be equal to the size of the description vector of one candlestick.

7. Testing trading capabilities of the model

The algorithm for creating the description of the neural layer remains the same. First, we create a new instance of the neural layer description object and check the result of the operation.

```
//--- Convolutional layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
```

Then we fill in the required information.

```
descr.type = defNeuronConv;
prev_count = descr.count = prev_count / NeuronsToBar;
descr.window = NeuronsToBar;
int prev_window = descr.window_out = 2 * NeuronsToBar;
descr.step = NeuronsToBar;
descr.activation = AF_SWISH;
descr.optimization = Adam;
descr.activation_params[0] = 1;
```

We pass a pointer to the populated instance of the object into the dynamic array describing the architecture of the model being created.

```
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

After the information passes through the convolutional layer, we expect to obtain a tensor with eight elements describing the state of one candlestick. However, we know that fully connected models do not evaluate the dependence between elements, whereas such dependencies are typically strong when analyzing time series data.

Hence, at the next stage, we aim to analyze such dependencies. We discussed such analysis during our introduction to convolutional networks. Despite seeming peculiar, we employ the same type of neural layers to address two seemingly different tasks. In fact, we are performing a similar task but with different data. In the preceding convolutional layer, we decomposed the description vector of a single candlestick into a larger number of elements. We can look at this task from another perspective. As we discussed during the study of the convolutional layer, the convolution process involves determining the similarity between two functions. That is, in each filter, we identify the similarity of the original data with some reference function. Each filter uses its own reference function. By conducting convolution operations on the scale of a single bar, we sought the similarity of each bar with some reference.

Now we want to analyze the dynamics of changes in candlestick parameters. To do this, we need to perform convolution between identical elements of description vectors for different candles. After convolution, the previous layer returned three values sequentially (the number of analyzed candlesticks) from each filter. So, the next step is to create a convolutional layer with a window of initial data and a step equal to the number of analyzed candlesticks. In this convolutional layer, we will also use eight filters.

Let's create a description for the convolutional neural layer following the algorithm mentioned earlier.

7. Testing trading capabilities of the model

```
//--- Convolutional layer 2
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}

descr.type = defNeuronConv;
descr.window = prev_count;
descr.step = prev_count;
prev_count = descr.count = prev_window;
prev_window = descr.window_out = 8;
descr.activation = AF_SWISH;
descr.optimization = Adam;
descr.activation_params[0] = 1;

if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
```

Thus, after preprocessing the data in one batch normalization layer and two consecutive convolutional layers, we obtained a tensor with 64 elements ($8 * 8$). Let me remind you that we fed a tensor of 12 elements to the input of the neural network: 3 candlesticks with 4 elements each.

Next, we will process the signal in a block with the *GPT* architecture. In it, we will create four sequential neural layers with eight attention heads in each. We have exposed the size of the depth of analyzed data in the external parameters of the script. This will allow us to conduct training with different depths and choose the optimal parameter based on the trade-off between training costs and model performance. The algorithm for creating a description of the neural layer remains the same.

```
//--- GPT layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}

descr.type = defNeuronGPT;
descr.count = BarsToLine;
descr.window = prev_count * prev_window;
descr.window_out = prev_window;
descr.step = 8;
descr.layers = 4;
descr.activation = AF_NONE;
descr.optimization = Adam;
descr.activation_params[0] = 1;

if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
}
```

```

    return false;
}

```

After the *GPT* block, we will create a block of fully connected neural layers. All layers in the block will be identical. We included the number of layers and neurons in each into the external parameters of the script. According to the algorithm proposed above, we create a new instance of the neural layer description object and check the result of the operation.

```

/-- Hidden fully connected layers
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}

```

After successfully creating a new object instance, we populate it with the necessary data. As mentioned above, we will take the number of neurons in the layer from the external parameter *HiddenLayer*. I chose the activation function *Swish*. Certainly, for greater script flexibility, more parameters can be moved to external settings, and you can conduct multiple training cycles with different parameters to find the best configuration for your model. This approach will require more time and expense for training the model but will allow you to find the most optimal values for the model parameters.

```

descr.type = defNeuronBase;
descr.count = HiddenLayer;
descr.activation = AF_SWISH;
descr.optimization = Adam;
descr.activation_params[0] = 1;

```

Since we plan to create identical neural layers, we then create a loop with a number of iterations equal to the number of neural layers to be created. In the body of the loop, we will add the created neural layer description to the dynamic array of architecture descriptions for the model being created. And, of course, we check the result of the operations at each iteration of the loop.

```

for(int i = 0; i < HiddenLayers; i++)
{
    if(!layers.Add(descr))
    {
        PrintFormat("Error adding layer: %d", GetLastError());
        delete descr;
        return false;
    }
}

```

To complete the model, we will create a results layer. This is a fully connected layer that contains two neurons with the *tanh* activation function. The choice of this activation function is based on the aggregate assessment of the target values of the trained model:

- The first element of the target value takes 1 for buy targets and -1 for sell targets, which is best configured by the hyperbolic tangent function *tanh*.
- We trained the models on the *EURUSD* pair, therefore, the value of the expected movement to the nearest extremum should be in the range from -0.05 to 0.05 . In this range of values, the graph of the hyperbolic tangent function *tanh* is close to linear.

7. Testing trading capabilities of the model

If you plan to use the model on instruments with an absolute value of the expected movement to the nearest extremum of more than 1, you can scale the target result. Then use reverse scaling when interpreting the model signal. You might also consider using a different activation function.

We use the same algorithm to create a description of the neural layer in the architecture of the created model. First, we create a new instance of the neural layer description object and check the result of the operation.

```
//--- Results layer
if(!(descr = new CLayerDescription()))
{
    PrintFormat("Error creating CLayerDescription: %d", GetLastError());
    return false;
}
```

We then populate the created object with the necessary information: the type of neural layer, the number of neurons, the activation function, and the optimization method for the model parameters.

```
descr.type          = defNeuronBase;
descr.count         = 2;
descr.activation    = AF_TANH;
descr.optimization  = Adam;
```

We add a pointer to the populated object to the dynamic array describing the architecture of the model being created and check the operation result.

```
if(!layers.Add(descr))
{
    PrintFormat("Error adding layer: %d", GetLastError());
    delete descr;
    return false;
}
return true;
}
```

We completed the function.

Following in the script algorithm is the *LoadTrainingData* function that loads the training dataset. In the parameters, the function receives a string variable with the name of the file to load and pointers to two dynamic array objects: *data* for patterns and *result* for target values.

```
bool LoadTrainingData(string path, CArrayObj &data, CArrayObj &result)
{
```

Let me remind you that we will load the training sample without preliminary normalization of the initial data from the file [study_data_not_norm.csv](#) since we plan to use the model in real-time, and we will use a batch normalization layer to prepare the initial data.

The algorithm for loading the source data will completely repeat what we previously considered while performing the same task in the [GPT](#) architecture testing script. Let's briefly recap on the process. To load the training dataset, we declare two new variables to store pointers to data buffers, in which we will read patterns and their target values one by one from the file (*pattern* and *target* respectively). We will create the object instances later. This is because we will need new object instances to load each pattern. Therefore, we will create objects in the body of the loop before the actual process of loading data from the file.

7. Testing trading capabilities of the model

```
CBufferType *pattern;  
CBufferType *target;
```

After completing the preparatory work, we open the file with the training sample to read the data. When opening a file, among other flags, we specify *FILE_SHARE_READ*. This flag opens shared access to the file for data reading. That is, by adding this flag, we do not block access to the file from other applications for reading the file. This will allow us to run several scripts in parallel with different parameters, and they will not block each other's access to the file. Of course, we can run several scripts in parallel only if the hardware capacity allows it.

```
//--- open the file with the training sample  
int handle = FileOpen(path, FILE_READ | FILE_CSV | FILE_ANSI | FILE_SHARE_READ,  
                      ",", CP_UTF8);  
  
if(handle == INVALID_HANDLE)  
{  
    PrintFormat("Error opening study data file: %d", GetLastError());  
    return false;  
}
```

Make sure to check the operation result. In case of a file opening error, we inform the user about the error, delete all previously created objects, and exit the program.

After successfully opening the file, we create a loop to read data from the file. The operations in the body of the loop will be repeated until one of the following events occurs:

- The end of the file is reached.
- The user interrupts program execution.

```
//--- show the progress of loading training data in the chart comment  
uint next_comment_time = 0;  
uint OutputTimeout     = 250; // not more often than once every 250 milliseconds  
//--- organizing a training dataset loading loop  
while(!FileIsEnding(handle) && !IsStopped())  
{  
    if(!(pattern = new CBufferType()))  
    {  
        PrintFormat("Error creating Pattern data array: %d", GetLastError());  
        return false;  
    }  
    if(!pattern.BufferInit(1, NeuronsToBar * GPT_InputBars))  
        return false;
```

In the body of the loop, we first create new instances of objects to record the current pattern and its target values. Again, we immediately check the result of the operation. If an error occurs, we inform the user about the error, delete previously created objects, close the file, and exit the program. It is very important to delete all objects and close the file before exiting the program.

```
    if(!(target = new CBufferType()))  
    {  
        PrintFormat("Error creating Pattern Target array: %d", GetLastError());  
        return false;  
    }  
    if(!target.BufferInit(1, 2))  
        return false;
```

7. Testing trading capabilities of the model

After this, we organize a nested loop with the number of iterations equal to the full data pattern. We have created training samples of 40 candlesticks per pattern. Now, we need to sequentially read all the data. However, our model does not require such a large pattern description for training. Therefore, we will skip unnecessary data and will only write the last required data to the buffer.

```
int skip = (HistoryBars - GPT_InputBars) * NeuronsToBar;
for(int i = 0; i < NeuronsToBar * HistoryBars; i++)
{
    TYPE temp = (TYPE)FileReadNumber(handle);
    if(i < skip)
        continue;
    pattern.m_mMatrix[0, i - skip] = temp;
}
```

After loading the current pattern data in full, we organize a similar loop to load target values. This time the number of iterations of the loop will be equal to the number of target values in the training dataset, that is, in our case, two. Before starting the loop, we will check the state of the pattern saving flag. We enter the loop only if the pattern description has been loaded in full.

```
for(int i = 0; i < 2; i++)
    target.m_mMatrix[0, i] = (TYPE)FileReadNumber(handle);
```

After the data loading loops have been executed, we move on to the block in which pattern information is added to our dynamic arrays. We add pointers to objects to the dynamic array of descriptions of patterns and target results. We also check the results of all operations.

```
if(!data.Add(pattern))
{
    PrintFormat("Error adding study data to array: %d", GetLastError());
    return false;
}
if(!result.Add(target))
{
    PrintFormat("Error adding study data to array: %d", GetLastError());
    return false;
}
```

After successfully adding to the dynamic arrays, we inform the user about the number of loaded patterns and proceed to load the next pattern.

```
//--- show the loading progress in the chart comment (no more than 1 time every
if(next_comment_time < GetTickCount())
{
    Comment(StringFormat("Patterns loaded: %d", data.Total()));
    next_comment_time = GetTickCount() + OutputTimeout;
}
}
FileClose(handle);
Comment(StringFormat("Patterns loaded: %d", data.Total()));
return(true);
}
```

After successfully loading all the data from the training dataset, we close the file and complete the data loading function.

7. Testing trading capabilities of the model

Next, we move on to the procedure for training our model in the *NetworkFit* function. In its parameters, the function receives pointers to three objects:

- trainable model
- dynamic array of system state descriptions
- dynamic array of target results

```
bool NetworkFit(CNet &net, const CArrayObj &data, const CArrayObj &result, VECTOR &lc
{
```

In the body of the method, we first do a little preparatory work. We start by preparing local variables.

```
int patterns = data.Total();
int count = -1;
TYPE min_loss = FLT_MAX;
```

After completing the preparatory work, we organize nested loops to train our model. The external loop will count the number of updates to the weight matrices, and in the nested loop, we will iterate over the patterns of our training dataset.

Let me remind you that in the *GPT* architecture, historical data is accumulated in a stack. Therefore, for the model to work correctly, the historical sequence of data input to the model is very important, similar to recurrent models. For this reason, we cannot shuffle the training dataset within a single training batch, and we will feed the model with patterns in chronological order. However, for model training, we can use random batches from the entire training dataset. It is worth noting that when determining the training batch, its size should be increased by the size of the internal accumulation sequence of the *GPT* block, as maintaining their chronological sequence is necessary for correctly determining dependencies between elements.

Thus, before running the nested loop, we define the boundaries of the current training batch.

In the body of the nested loop, before proceeding with further operations, we check the flag for the forced termination of the program, and if necessary, we interrupt the function execution.

```
//--- loop through epochs
for(int epoch = 0; epoch < Epochs; epoch++)
{
    ulong ticks = GetTickCount64();
    //--- training in batches
    //--- selection of a random pattern
    int k = (int)((double)(MathRand() * MathRand()) / MathPow(32767.0, 2) * (patterns - 1));
    k = fmax(k, 0);
    for(int i = 0; (i < (BatchSize + BarsToLine) && (k + i) < patterns); i++)
    {
        //--- check if training stopped
        if(IsStopped())
        {
            Print("Network fitting stopped by user");
            return true;
        }
    }
}
```

First, we perform the feed-forward pass through the model by calling the *net.FeedForward* method. In the parameters of the feed-forward method, we pass a pointer to the object describing the current

7. Testing trading capabilities of the model

pattern state and check the result of the operation. If an error occurs during method execution, we inform the user about the error, delete the created objects, and exit the program.

```
if(!net.FeedForward(data.At(k + i)))
{
    PrintFormat("Error in FeedForward: %d", GetLastError());
    return false;
}
```

After the successful execution of the feed-forward method, we check the fullness of the buffer of our *GPT* block. If the buffer is not yet full, move on to the next iteration of the loop.

```
if(i < BarsToLine)
    continue;
```

The backpropagation method *net.Backpropagation* is called only after the cumulative sequence of the *GPT* block is filled. This time, in the parameters of the method, we pass a pointer to the object representing the target values. It is very important to check the result of the operation. If an error occurs, we perform the operations as if there was an error in the direct method.

```
if(!net.Backpropagation(result.At(k + i)))
{
    PrintFormat("Error in Backpropagation: %d", GetLastError());
    return false;
}
}
```

Using the feed-forward and backpropagation methods, we have executed the respective algorithms for training our model. At this stage, the error gradient has already been propagated to each trainable parameter. All that remains is to update the weight matrices. However, we perform this operation not at every training iteration but only after accumulating a batch. In this particular case, we will update the weight matrices after completing the iterations of the nested loop.

```
//--- reconfigure the network weights
net.UpdateWeights(BatchSize);
printf("Use OpenCL %s, epoch %d, time %.5f sec", (string)UseOpenCL, epoch, (Get
```

As you have seen in the model testing graphs, the model error dynamics almost never follow a smooth line. Saving the model with minimal error will allow us to save the most appropriate parameters for our model. Therefore, we first check the current model error and compare it to the minimum error achieved during training. If the error has dropped, we save the current model and update the minimum error variable.

```
//--- notify about the past epoch
TYPE loss = net.GetRecentAverageLoss();
Comment(StringFormat("Epoch %d, error %.5f", epoch, loss));
//--- remember the epoch error for saving to a file
loss_history[epoch] = loss;
if(loss < min_loss)
{
    //--- saving the model with minimal error
    if(net.Save(ModelName))
    {
        min_loss = loss;
        count = -1;
    }
}
```


7. Testing trading capabilities of the model

```
}
```

Additionally, we have introduced the *count* counter. We will use it to count the number of update iterations from the last minimum error value. If its value exceeds the specified threshold (in the example, it is set to 10 iterations), then we interrupt the training process.

```
    if(count >= 10)
        break;
    count++;
}
return true;
}
```

After completing a full training cycle, we will need to save the accumulated dynamics of the model error changes during the training process to a file. To do this, we have created the *SaveLossHistory* function. In the parameters, the function receives a string variable with the file name for storing the data and a vector of errors during the model training process.

In the function body, we open the file for writing. In this case, we use the file name that the user specified in the parameters. We immediately check the result. If an error occurs when opening the file, we inform the user and exit the function.

```
void SaveLossHistory(string path, const VECTOR &loss_history)
{
    int handle = FileOpen(OutputFileName, FILE_WRITE | FILE_CSV | FILE_ANSI,
                           ",", CP_UTF8);

    if(handle == INVALID_HANDLE)
    {
        PrintFormat("Error creating loss file: %d", GetLastError());
        return;
    }
    for(ulong i = 0; i < loss_history.Size(); i++)
        FileWrite(handle, loss_history[i]);
    FileClose(handle);
    PrintFormat("The dynamics of the error change is saved to a file %s\\MQL5\\Files\\",
                TerminalInfoString(TERMINAL_DATA_PATH), OutputFileNa
    }
}
```

If the file was opened successfully, we organize a loop in which we write, one by one, all the values of the model error accumulation vector during training. After completing the full data writing loop, we close the file and inform the user about the location of the file.

With this, our script for creating and training the model is complete, and we can begin training the model on the previously created dataset of non-normalized training data from the file [study_data_not_norm.csv](#).

The next step is to start the model training process. Here you need to be patient, as the learning process is quite long. Its duration depends on the hardware used. For example, I started training a model with the parameters shown in the screenshot below.

On my Intel Core i7-1165G7 laptop, it takes 35-36 seconds to compute one batch between weight matrix updates. So, full training of the model with 5000 iterations of weight updates will take approximately 2 days of continuous operation. However, if you notice that training has halted and the minimum error hasn't changed for an extended period, you can manually stop the model training. If the

achieved performance doesn't meet the requirements, you can continue training the model with different values for the learning rate and batch size for weight updates. The common approach to selecting parameters is as follows:

- The learning rate: training starts with a larger learning rate, and during training, we gradually decrease the learning rate.
- Weight matrix update batch size: training starts with a small batch and gradually increases.

Variable	Value
ab Training dataset file name	study_data_not_norm.csv
ab File name to write error dynamics	loss_study_gpt_not_norm.csv
01 Depth of the analyzed history	60
01 Number of input layer neurons per 1 bar	4
↕ Use OpenCL	false
01 Batch size to update the weight matrix	10000
½ Learning rate	0.0003
01 Number of hidden layers	2
01 Number of neurons in each hidden layer	60

Model training parameters

The techniques mentioned above allow initial fast and rough training of the model followed by finer tuning. If during the model training process, the error consistently increases, it indicates an excessively high learning rate. Using a large batch for weight matrix updates helps to adjust the weight matrices in the most prioritized direction, but it requires more time to perform operations between weight updates. On the other hand, a small batch leads to faster and more chaotic parameter updates, while still maintaining the overall trend. However, when using small batch sizes, it is recommended to decrease the learning rate to reduce model overfitting to specific parts of the training dataset.

7.4 Determining Expert Advisor parameters

After we have trained the model, before running it in the trading strategy, we need to learn how to use it. First and foremost, we need to understand its signals. As you know, after the feed-forward pass, our model returns two values:

- The probable direction of movement (the absolute value shows the probability of movement, and the sign shows the direction).
- The expected strength of movement (the absolute value shows the force of motion, and the sign shows the direction).

For each parameter, you need to find the decision threshold. A too-high value can filter out a large number of profitable trades or not provide any signals for trading operations at all. A too-small value can lead to a large number of false signals and even make trading unprofitable. Therefore, it is now very important to find the optimal parameters of the Expert Advisor to work with our trained model.

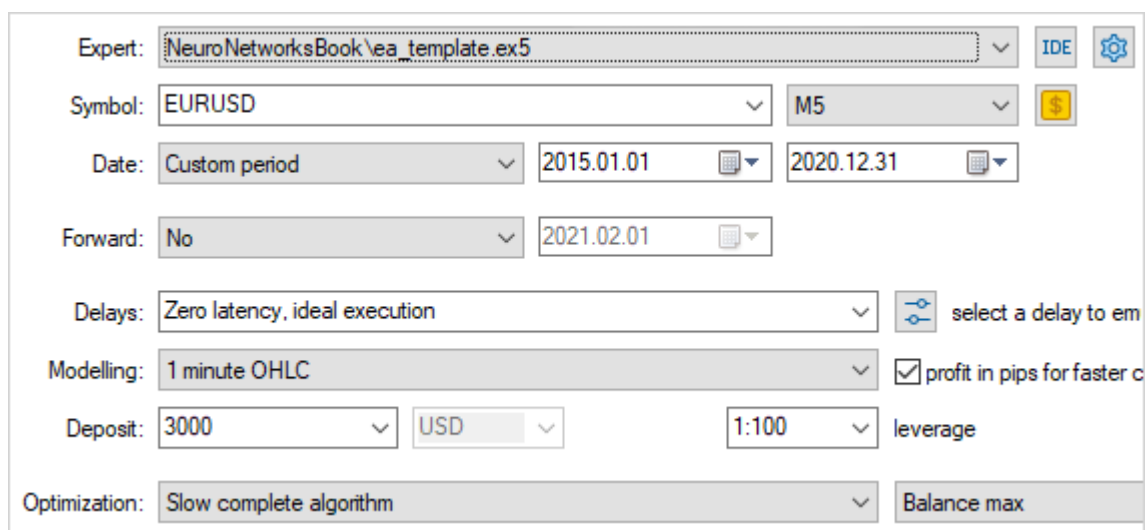
7. Testing trading capabilities of the model

The best tool to do this is the *MetaTrader 5* strategy tester. In the terminal, press *Ctrl + R* and navigate to the tester. In the *Settings* tab, in the *Expert* field, select our Expert Advisor and set the testing parameters.

We trained the model on *EURUSD* historical data from 2015 to 2020 on the M5 timeframe. We will use the same historical data to determine the optimal parameters of the Expert Advisor. According to the general neural network training rule, the performance of the model should be verified on a validation dataset. However, in this case, we simply determine the optimal parameters for the Expert Advisor when the model is running on the training dataset.

We know that our Expert Advisor analyzes entry points only at the opening of the candlestick, so to check the presence of signals from the model, it would be possible to test only at the opening prices. However, we also need to understand the quality of these signals. At the same time, we want to conduct the initial rough selection of parameters with minimal time and resources. Therefore, let's choose the testing mode based on the control points of the M1 timeframe.

We need to optimize the parameters, so we choose the optimization mode. To select the optimization mode, it's desirable to know the number of upcoming iterations. Counting them requires no effort, as they are automatically calculated when selecting Expert Advisor parameters for optimization. Let's go to the parameters tab and set the initial values of the parameters in the *Value* column. I would like to point out that parameters such as the model file name and the number of candles in the current pattern are not optimized because they should match the model being used.



Expert:	NeuroNetworksBookVea_template.ex5		IDE	Settings
Symbol:	EURUSD	M5		
Date:	Custom period	2015.01.01	2020.12.31	
Forward:	No	2021.02.01		
Delays:	Zero latency, ideal execution			select a delay to em
Modelling:	1 minute OHLC			☒ profit in pips for faster c
Deposit:	3000	USD	1:100	leverage
Optimization:	Slow complete algorithm		Balance max	

Optimization of Expert Advisor parameters

In the first stage, we will roughly optimize only one parameter: the *TradeLevel* decision-making threshold. Select the checkbox of this parameter. At the output of our model, we used the hyperbolic tangent (*tanh*) as the activation function. Therefore, the output values of neurons are normalized in the range from -1 to 1 . The sign shows the direction of movement. This means that the decision-making level can be in the range from 0 to 1 . Obviously, making trades with a probability of making a profit of less than 50% looks risky, to say the least. Therefore, let's try to choose the level of decision-making in the range from 0.5 to 1.0 . Recall that this is the first and rough selection of the parameter, so we will use step 0.05 . The strategy tester immediately counted 11 iterations for us. As you can see, there are quite a few of them. Let's go back to the *Settings* tab and select the *Slow complete algorithm* optimization type. We also select optimization for the maximum balance and start the optimization process by clicking on the *Start* button.

7. Testing trading capabilities of the model

Variable	Value	Start	Step	Stop	Steps
☒ Model	gpt_not_norm.net				
☒ BarsToPattern	3				
☒ Common	true				
☐ TimeFrame	5 Minutes	current		1 Month	
☒ TradeLevel	1	0.5	0.05	1	11
☐ Lot	0.01	0.01	0.001	0.1	
☐ MaxTP	1000	200	100	1000	
☐ ProfitMultiply	0.9	0.5	0.05	1	
☐ MinTarget	100	50	50	500	
☐ StopLoss	100	50	50	500	
					11

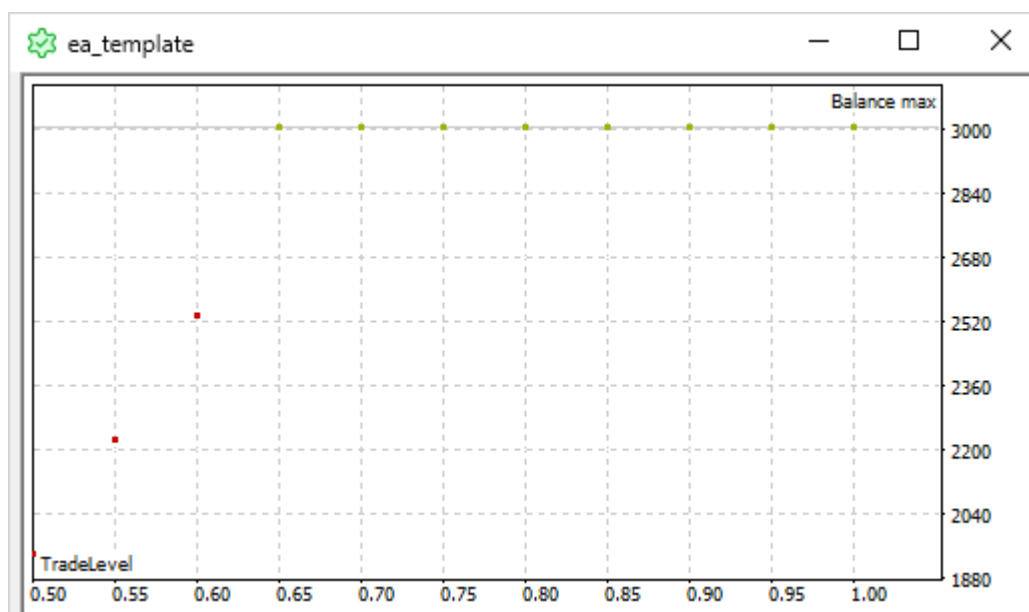
Optimization of Expert Advisor parameters

The screenshot below shows the optimization results. As you can see, with a decision threshold of 0.65 or higher, the Expert Advisor does not execute any trades. From this, we can conclude that during the training process, our neural network did not identify patterns with a probability of one-directional movement equal to or greater than 65%. You should not be alarmed by the loss incurred by the Expert Advisor at this stage, as we have only conducted preliminary rough optimization with a crude determination of the decision-making level. Next, we have to optimize a few more parameters of our Expert Advisor.

Pass	Res...	Profit	Total trad...	Profit fac...	Expected...	Drawdo...	TradeLevel
10	3000.00	0.00	0		0.00	0.00	1.00
9	3000.00	0.00	0		0.00	0.00	0.95
8	3000.00	0.00	0		0.00	0.00	0.90
7	3000.00	0.00	0		0.00	0.00	0.85
6	3000.00	0.00	0		0.00	0.00	0.80
5	3000.00	0.00	0		0.00	0.00	0.75
4	3000.00	0.00	0		0.00	0.00	0.70
3	3000.00	0.00	0		0.00	0.00	0.65
2	2532.23	-467.77	5198	0.88	-0.09	16.38	0.60
1	2222.02	-777.98	9101	0.88	-0.09	26.60	0.55
0	1935.31	-1064.69	12399	0.87	-0.09	35.79	0.50

Results of the first optimization

7. Testing trading capabilities of the model



Results of the first optimization

First, let's try to optimize the parameter of the minimum strength of the upcoming movement *MinTarget* in order to filter out minor fluctuations. The goal of this iteration is to select the strongest movements. This is because the probability of such patterns triggering in practice is higher, and minor fluctuations may not have enough momentum to reach the target or may not trigger at all. Moreover, using orders with a low level of profitability reduces the risk-reward ratio.

We will optimize the parameter in the range from 50 to 600 points in increments of 50 points. In this iteration, we need to check 12 runs of the Expert Advisor.

Variable	Value	Start	Step	Stop	Steps
☒ Model	gpt_not_norm.net				
☒ BarsToPattern	3				
☒ Common	true				
☐ TimeFrame	5 Minutes	current		1 Month	
☐ TradeLevel	0.6	0.5	0.05	1	
☐ Lot	0.01	0.01	0.001	0.1	
☐ MaxTP	1000	200	100	1000	
☐ ProfitMultiply	0.9	0.5	0.05	1	
☒ MinTarget	100	50	50	600	12
☐ StopLoss	100	50	50	500	
					12

Selection of the threshold value of decision-making

Based on the results of this parameter optimization, we can observe the emergence of the first profitable runs with a decision level of 500 and 600 points. However, with such parameter choices, the number of completed trading operations significantly decreases. Indeed, we want to extract the maximum potential from our model. It seems that values of the decision-making threshold around 350-400 pips are the most promising, with a trade count exceeding 1000 and being closest to the breakeven point. Let's take a small gamble and continue optimizing the parameters with the specified parameter range.

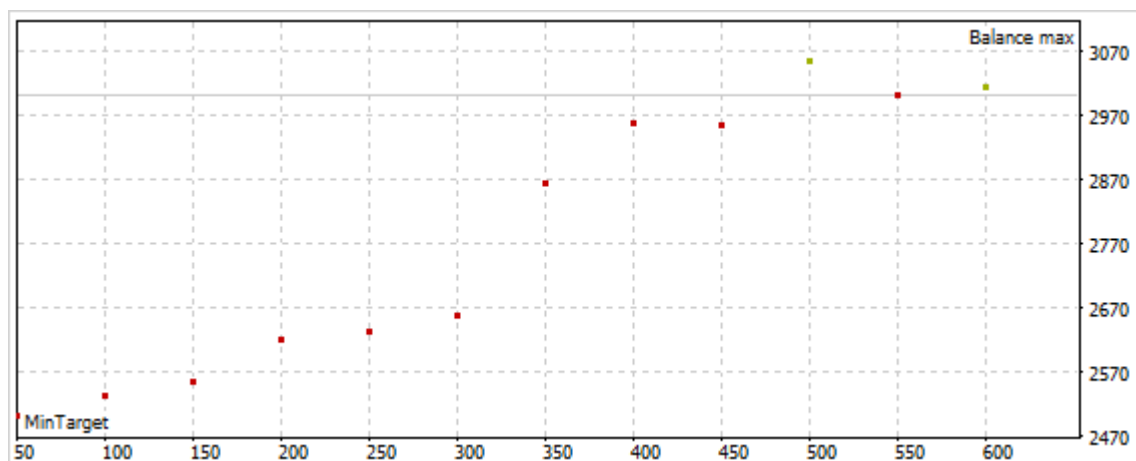
7. Testing trading capabilities of the model

Pass	Res...	Profit	Total trad...	Profit fac...	Expected...	Drawdo...	MinTarget
9	3052.69	52.69	297	1.22	0.18	0.93	500
11	3010.71	10.71	42	1.31	0.25	0.47	600
10	2999.95	-0.05	123	1.00	-0.00	0.77	550
7	2954.46	-45.54	1188	0.95	-0.04	2.21	400
8	2952.48	-47.52	635	0.91	-0.07	2.45	450
6	2862.93	-137.07	1954	0.91	-0.07	5.16	350
5	2655.56	-344.44	2763	0.84	-0.12	11.93	300
4	2630.25	-369.75	3509	0.86	-0.11	13.22	250
3	2619.80	-380.20	4181	0.88	-0.09	13.52	200
2	2553.66	-446.34	4724	0.87	-0.09	15.68	150
1	2532.23	-467.77	5198	0.88	-0.09	16.38	100
0	2500.37	-499.63	5537	0.87	-0.09	17.54	50

Results of Decision Threshold Optimization

Next, let's move on to optimizing the stop-loss parameter, which limits the risks for each trade. We will optimize this parameter in the range from 50 to 500 pips with a step size of 50 pips.

As mentioned above, we have not defined a clear value for the *MinTarget* parameter. Therefore, for the current optimization process, we will use two optimized parameters. At the same time, the parameter for the forecast strength threshold of the upcoming impulse will only take two permissible values.



Results of Decision Threshold Optimization

Thus, the strategy tester counted 20 passes of the current optimization process.

It is worth noting one more circumstance. In the upcoming optimization process, we are going to find the optimal stop-loss level. Here, it should be noted that in real trading, stop-loss and take-profit levels are handled by the broker on each tick. To get loss values as close as possible to real levels when the stop-loss is triggered, it will be necessary to optimize with each tick processed. Therefore, we go to the *Settings* tab and change the simulation mode to *Every tick based on real ticks*, which will switch the strategy tester to the mode of processing real historical ticks. We will also change the optimization mode to *Fast genetic based algorithm*. This will allow the tester to filter out passes whose results will be significantly worse than those already conducted.

7. Testing trading capabilities of the model

Variable	Value	Start	Step	Stop	Steps
☒ Model	gpt_not_norm.net				
☒ BarsToPattern	3				
☒ Common	true				
☐ TimeFrame	5 Minutes	current		1 Month	
☐ TradeLevel	0.6	0.5	0.05	1	
☐ Lot	0.01	0.01	0.001	0.1	
☐ MaxTP	1000	200	100	1000	
☐ ProfitMultiply	0.9	0.5	0.05	1	
☒ MinTarget	100	350	50	400	2
☒ StopLoss	100	50	50	500	10
					20

Optimization of stop-loss parameters

Expert: IDE ⚙️

Symbol: 💰

Date:

Forward:

Delays: ⚙️ select a delay to em

Modelling: ☒ profit in pips for faster c

Deposit: leverage

Optimization:

Optimization of stop-loss parameters

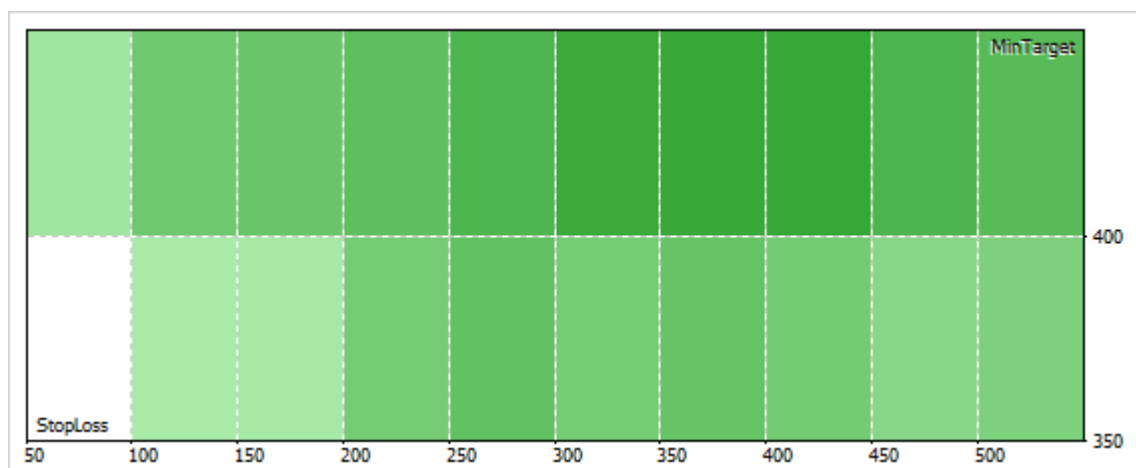
As a result of optimizing the parameters, unfortunately, we still do not see any profitable passes. However, there is a clear superiority of a larger decision-making parameter based on the strength of the upcoming *MinTarget* momentum. At the same time, fairly close results were obtained for three stop-loss levels in the range of 300-400 points.

Thus, for further optimization of parameters, we take *MinTarget* at the level of 400 points, and we will continue optimizing the stop loss in the range of 300-400 points.

7. Testing trading capabilities of the model

Pass	Result ▼	Profit	Total tr...	Profit f...	Expect...	Drawd...	MinTar...	StopLoss
15	2970.63	-29.37	718	0.98	-0.04	3.13	400	400
13	2967.65	-32.35	762	0.98	-0.04	3.74	400	350
11	2962.85	-37.15	799	0.97	-0.05	3.98	400	300
17	2928.43	-71.57	700	0.95	-0.10	3.82	400	450
9	2924.67	-75.33	860	0.95	-0.09	4.17	400	250
19	2907.33	-92.67	679	0.94	-0.14	3.98	400	500
7	2895.35	-104.65	921	0.92	-0.11	4.49	400	200
8	2889.63	-110.37	1364	0.95	-0.08	4.59	350	250
12	2884.78	-115.22	1206	0.95	-0.10	5.48	350	350
5	2876.44	-123.56	1025	0.90	-0.12	5.03	400	150
3	2868.69	-131.31	1205	0.87	-0.11	4.86	400	100
14	2858.75	-141.25	1139	0.94	-0.12	6.38	350	400
10	2855.77	-144.23	1280	0.93	-0.11	5.98	350	300

Stop-loss selection results



Stop-loss selection results

The next parameter to be optimized is the coefficient of confidence in the predicted strength of the expected momentum. This is the coefficient by which we will multiply the value of the second parameter returned by our model when calculating the take profit for the opened position. We will not overestimate the expected momentum value. Therefore, the upper limit of parameter optimization will be equal to one. We will set the lower limit of optimization at the level of 0.5, which is equivalent to 50% of the predicted momentum. With a step of 0.05, we get 11 optimization passes. Multiplying this number by 3 stop-loss options, we will get 33 passes of the upcoming parameter optimization.

7. Testing trading capabilities of the model

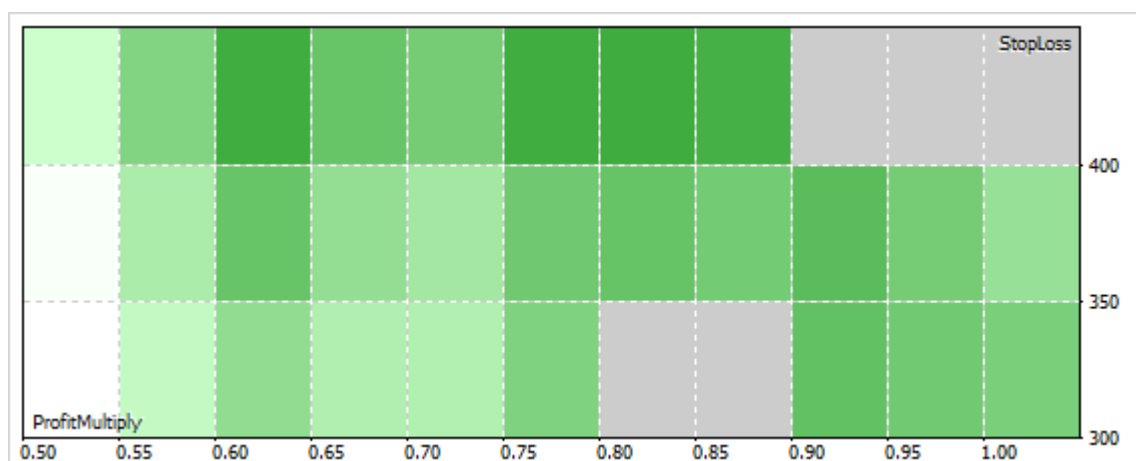
Variable	Value	Start	Step	Stop	Steps
☒ Model	gpt_not_norm.net				
☒ BarsToPattern	3				
☒ Common	true				
☐ TimeFrame	5 Minutes	current		1 Month	
☐ TradeLevel	0.6	0.56	0.01	0.64	
☐ Lot	0.01	0.01	0.001	0.1	
☐ MaxTP	1000	300	100	900	
☒ ProfitMultiply	0.5	0.5	0.05	1.0	11
☐ MinTarget	400	400	50	500	
☒ StopLoss	300	300	50	400	3
					33

Optimization of the confidence factor

As a result of optimization, we make a clear choice of a stop loss parameter at the level of 400 points and a confidence coefficient at the level of 0.8.

Pass	Result ▼	Profit	Total tr...	Profit f...	Expect...	Drawd...	Profit...	StopLoss
28	2985.39	-14.61	763	0.99	-0.02	3.44	0.80	400
24	2984.49	-15.51	873	0.99	-0.02	2.89	0.60	400
27	2984.22	-15.78	784	0.99	-0.02	3.57	0.75	400
29	2980.48	-19.52	736	0.99	-0.03	3.27	0.85	400
19	2967.65	-32.35	762	0.98	-0.04	3.74	0.90	350
8	2962.85	-37.15	799	0.97	-0.05	3.98	0.90	300
17	2960.05	-39.95	809	0.97	-0.05	4.55	0.80	350
25	2959.68	-40.32	841	0.97	-0.05	3.71	0.65	400
13	2959.35	-40.65	912	0.97	-0.04	3.34	0.60	350

Results of Confidence Factor Optimization



Results of Confidence Factor Optimization

Unfortunately, we must admit the failure of our endeavor. We never got a profitable combination of parameters. Let's go back to the *MinTarget* parameter expressing the threshold decision-making value based on the strength of the predicted momentum. During the previous optimization of this parameter, we got the maximum profit at the level of 500 points. Let's conduct a small re-optimization of this parameter in the range from 400 to 500 points with a step of 50 points.

7. Testing trading capabilities of the model

Variable	Value	Start	Step	Stop	Steps
☒ Model	gpt_not_norm.net				
☒ BarsToPattern	3				
☒ Common	true				
☐ TimeFrame	5 Minutes	current		1 Month	
☐ TradeLevel	0.6	0.56	0.01	0.64	
☐ Lot	0.01	0.01	0.001	0.1	
☐ MaxTP	1000	300	100	900	
☐ ProfitMultiply	0.8	0.5	0.05	1.0	
☒ MinTarget	400	400	50	500	3
☐ StopLoss	400	300	50	400	
					3

Re-optimization of the decision-making parameter

As a result of optimization, we get a profitable combination of parameters at the level of 500 points. I must say that the level of profit received is almost twice as much as previously received. At the same time, the number of trading operations decreased, while the profit factor remained at the level of 1.22.

Pass	Res...	Profit	Total trad...	Profit fac...	Expected...	Drawdo...	MinTarget
2	3089.06	89.06	215	1.22	0.41	2.10	500
0	2985.39	-14.61	763	0.99	-0.02	3.44	400
1	2906.30	-93.70	438	0.90	-0.21	4.30	450

Results of optimization of the decision-making parameter

Variable	Value	Start	Step	Stop	Steps
☒ Model	gpt_not_norm.net				
☒ BarsToPattern	3				
☒ Common	true				
☐ TimeFrame	5 Minutes	current		1 Month	
☐ TradeLevel	0.6	0.56	0.01	0.64	
☐ Lot	0.01	0.01	0.001	0.1	
☒ MaxTP	300	300	100	900	7
☐ ProfitMultiply	0.8	0.5	0.05	1.0	
☐ MinTarget	500	400	50	500	
☐ StopLoss	400	300	50	400	
					7

Optimization of the profitability constraint parameter

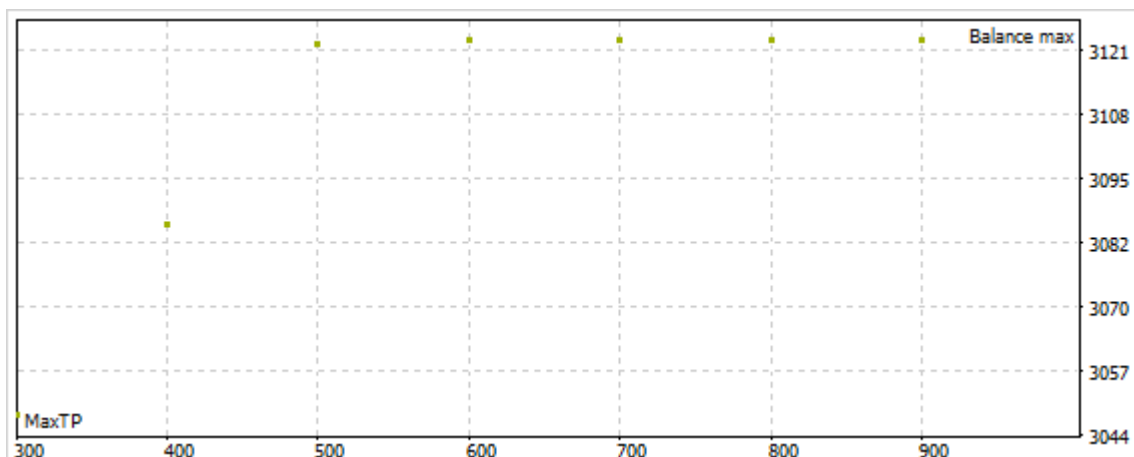
Next, we optimize the *MaxTP* parameter indicating the maximum take profit limit. This parameter will act as a safeguard against inflated expectations. If the model predicts an exaggerated movement with new values, the Expert Advisor will limit the take profit level to this value, which we will determine from the statistics of the training dataset. We optimize the value of the *MaxTP* parameter in the range from 300 to 900 points in increments of 100 points.

Based on the optimization results, it can be noticed that when the parameter is increased beyond 600, the performance of the Expert Advisor does not change. Consequently, the level of expected movement does not exceed 600 points for the entire training sample. Therefore, we can safely limit the maximum profit level to 600 points.

7. Testing trading capabilities of the model

Pass	Res...	Profit	Total trad...	Profit fac...	Expected...	Drawdo...	MaxTP
6	3122.69	122.69	215	1.32	0.57	1.46	900
5	3122.69	122.69	215	1.32	0.57	1.46	800
4	3122.69	122.69	215	1.32	0.57	1.46	700
3	3122.69	122.69	215	1.32	0.57	1.46	600
2	3121.93	121.93	215	1.32	0.57	1.46	500
1	3085.97	85.97	222	1.21	0.39	1.48	400
0	3047.89	47.89	243	1.12	0.20	1.55	300

Results of optimization of the profitability constraint parameter



Results of optimization of the profitability constraint parameter

Variable	Value	Start	Step	Stop	Steps
☒ Model	gpt_not_norm.net				
☒ BarsToPattern	3				
☒ Common	true				
☐ TimeFrame	5 Minutes	current		1 Month	
☒ TradeLevel	0.56	0.56	0.01	0.64	9
☐ Lot	0.01	0.01	0.001	0.1	
☐ MaxTP	600	300	100	900	
☐ ProfitMultiply	0.8	0.5	0.05	1.0	
☐ MinTarget	500	400	50	500	
☐ StopLoss	400	300	50	400	
					9

Fine-tuning the decision-making parameter

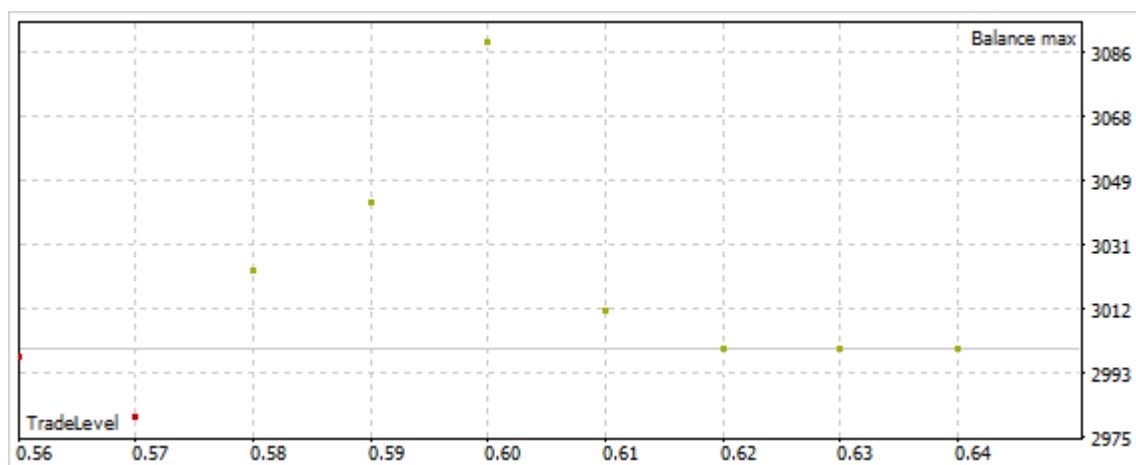
Finally, we will fine-tune the decision-making parameter based on the movement probability level *TradeLevel*. Earlier, we conducted a rough optimization of this parameter with a step of 0.05 and settled on a level of 0.6. Now we will try to optimize the parameter with a step of 0.01 in the vicinity of the previously chosen level. Thus, we will optimize the parameter in the range of 0.56–0.64.

Strangely enough, the optimization we conducted only confirmed the correctness of the previously chosen decision-making parameter value at the level of 0.6. Any deviation of the parameter from this value has a negative impact on the profitability of our Expert Advisor.

7. Testing trading capabilities of the model

Pass	Res...	Profit	Total trad...	Profit fac...	Expected...	Drawdo...	TradeLevel
4	3089.06	89.06	215	1.22	0.41	2.10	0.60
3	3042.58	42.58	365	1.06	0.12	3.66	0.59
2	3022.75	22.75	446	1.02	0.05	4.51	0.58
5	3011.26	11.26	72	1.08	0.16	1.51	0.61
8	3000.00	0.00	0		0.00	0.00	0.64
7	3000.00	0.00	0		0.00	0.00	0.63
6	3000.00	0.00	0		0.00	0.00	0.62
0	2997.80	-2.20	555	1.00	-0.00	4.11	0.56
1	2980.18	-19.82	506	0.98	-0.04	5.03	0.57

Results of fine-tuning the decision-making parameter



Results of fine-tuning the decision-making parameter

So, as a result of the optimization work, we have the following set of parameters that allow for profitability on the training dataset.

It should be noted that to determine the optimal set of parameters, we went through quite a few iterations of their optimization. At the same time, the *MetaTrader 5* strategy tester allows you to optimize any number of parameters in one run of the optimization process. However, you will have to pay for it with time and computing resources. If we calculate the total number of passes made for all iterations of optimization, we get about 95 passes. If we were to run simultaneous optimization of all the parameters mentioned above, the total number of possible parameter combinations for conducting passes would exceed 100,000. One can hope for a reduction in the number of passes through the use of genetic algorithms, but still, their number will significantly exceed what we've conducted. Consequently, it will take much more time to optimize the parameters.

7. Testing trading capabilities of the model

Variable	Value	Start	Step	Stop	Steps
☒ Model	gpt_not_norm.net				
☒ BarsToPattern	3				
☒ Common	true				
☒ TimeFrame	5 Minutes	current		1 Month	
☒ TradeLevel	0.6	0.56	0.01	0.64	9
☒ Lot	0.01	0.01	0.001	0.1	
☒ MaxTP	600	300	100	900	
☒ ProfitMultiply	0.8	0.5	0.05	1	
☒ MinTarget	500	400	50	500	
☒ StopLoss	400	300	50	400	

A set of optimized parameters

Now, after determining the optimal set of parameters, let's test the model's performance on new data.

7.5 Testing the model on new data

In the previous section, we optimized the parameters of our Expert Advisor on the training dataset and determined the optimal set of parameters. Now we need to test the performance of our model on new data. We are creating a model to potentially earn money in the financial market, aren't we? So far, we have only trained the model and optimized the EA parameters using historical data for the period from 2015 to 2020 inclusive. We have identified the optimal set of parameters that allow us to make a profit on historical data. While we cannot travel back in time and make money on historical data, we can run our Expert Advisor on a trading account and hope for a comparable return in the future. To confirm or refute the possibility of future profitability, let's test our Expert Advisor with the trained model and optimized parameters on historical data outside the training set using data for 2021. Thus, we will test the profitability of the model on new data.

As in the case of parameter optimization, we go to the *MetaTrader 5* strategy tester and in the *Settings* tab, specify the testing period 2021, select the type of modeling based on real ticks and disable parameter optimization. Also, do not forget to specify the correct financial instrument and timeframe.

After that, we will go to the EA parameters tab and specify the values of the parameters that we defined in the previous section. Start the testing process with the *Start* button.

Expert: NeuroNetworksBook\ea_template.ex5 [IDE] [Settings]

Symbol: EURUSD [M5] [USD]

Date: Custom period [2021.01.01] [2022.12.31]

Forward: No [2021.08.26]

Delays: Zero latency, ideal execution [select a delay to em...]

Modelling: Every tick based on real ticks [profit in pips for faster c...]

Deposit: 3000 [USD] [1:100] leverage

Optimization: Disabled [visual mode with the dis...]

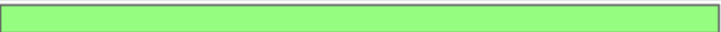
Forward testing of the model

7. Testing trading capabilities of the model

Variable	Value	Start	Step	Stop	Steps
☒ Model	gpt_not_norm.net				
☒ BarsToPattern	3				
☒ Common	true				
☒ TimeFrame	5 Minutes	current		1 Month	
☒ TradeLevel	0.6	0.56	0.01	0.64	9
☒ Lot	0.01	0.01	0.001	0.1	
☒ MaxTP	600	300	100	900	
☒ ProfitMultiply	0.8	0.5	0.05	1	
☒ MinTarget	500	400	50	500	
☒ StopLoss	400	300	50	400	

Forward testing of the model

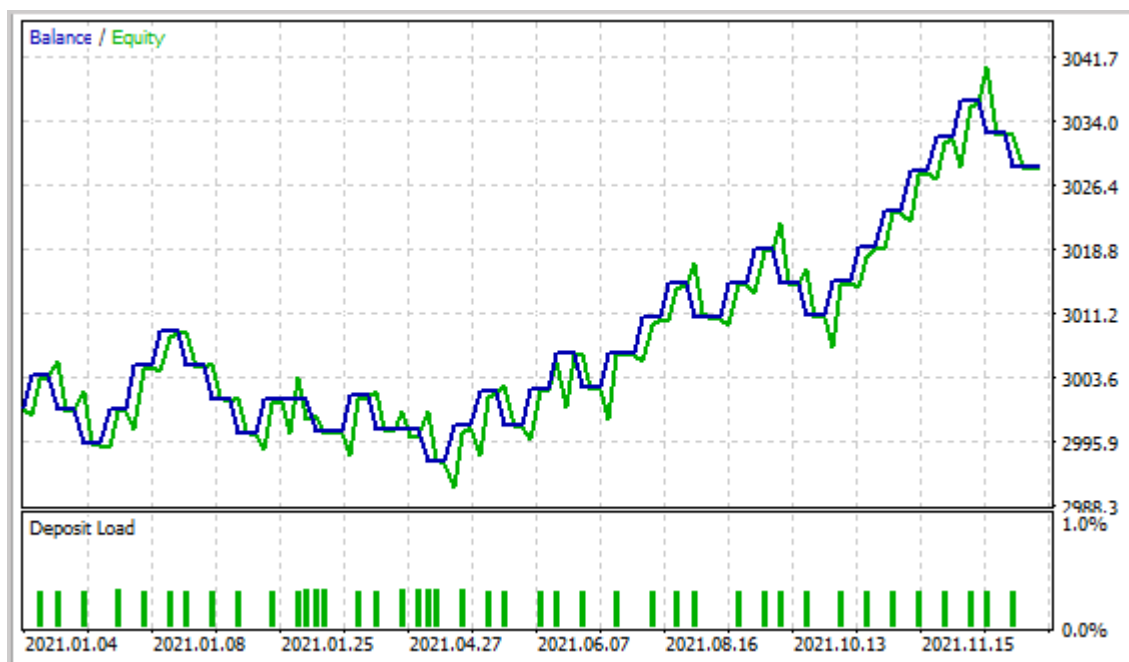
During the testing period, the Expert Advisor made a profit over a long time interval. In general, the year was closed with a positive result. It should be noted that for testing the model, we use a rather simplified Expert Advisor algorithm without the use of money management and position tracking features. But even in this version, the EA shows profit. This is indicative of the overall profitability of the trading signals generated by the model. Potentially, adding money management and position tracking features will increase the profitability of Expert Advisor performance.

History Quality	100%				
Bars	74453	Ticks	20348987	Symbols	1
Initial Deposit	3 000.00				
Total Net Profit	28.64	Balance Drawdo...	6.39	Equity Drawdown...	9.18
Gross Profit	88.74	Balance Drawdo...	15.52 (0.5...	Equity Drawdown...	18.46 (0.61%)
Gross Loss	-60.10	Balance Drawdo...	0.52% (15....	Equity Drawdown...	0.61% (18.46)

Results of forward testing of the model

The balance change chart shows sideways movement in the first half of the year, but from May, there is a clear trend of capital growth.

7. Testing trading capabilities of the model



Results of forward testing of the model

Analysis of the Expert Advisor performance on new data showed that on some variables it even surpasses the values obtained on the training set. For instance, the profit factor on the new data was 1.48, whereas, during the parameter optimization on the training set, this indicator was at 1.22. The margin level in this case is not indicative, as all trades were made with a minimal volume, which greatly inflated this indicator.

Profit Factor	1.48	Expected Payoff	0.80	Margin Level	29908.20%
Recovery Factor	1.55	Sharpe Ratio	2.35	Z-Score	0.00 (0.00%)
AHPR	1.0003 (0....	LR Correlation	0.83	OnTester result	0
GHPR	1.0003 (0....	LR Standard Error	6.53		

Results of forward testing of the model

In total, for the whole of 2021, the EA opened 36 positions, 21 of which were closed with a profit. This accounted for 58.33% of the total number of positions. The obtained value is very close to the 60% expected return from the model's signals. Let me remind you that the threshold level for conducting trading operations is a 60% probability of the price moving in the predicted direction (parameter *TradeLevel=0.6*).

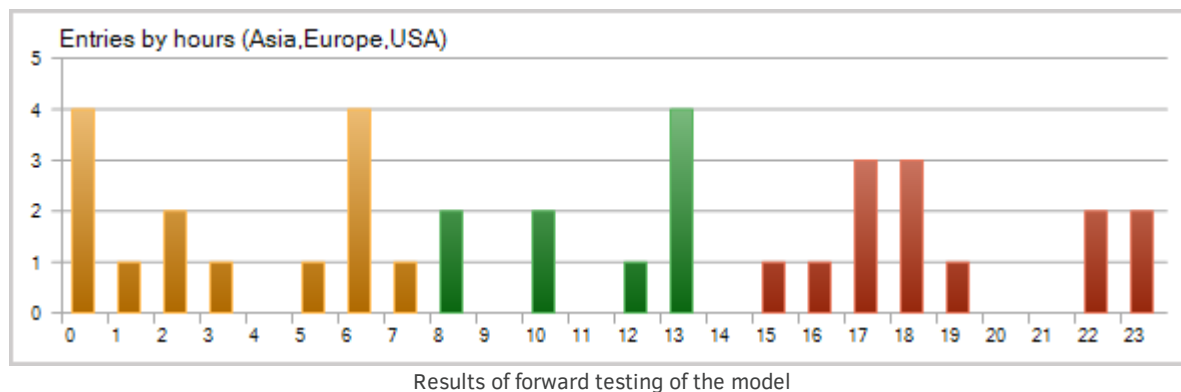
The maximum number of consecutive losing trades is three, while the maximum number of profitable trades is six.

Total Trades	36	Short Trades (wo...	34 (55.88%)	Long Trades (wo...	2 (100.00%)
Total Deals	72	Profit Trades (% ...	21 (58.33%)	Loss Trades (% o...	15 (41.67%)
	Largest	profit trade	5.05	loss trade	-4.03
	Average	profit trade	4.23	loss trade	-4.01
	Maximum	consecutive win...	6 (25.67)	consecutive loss...	3 (-12.02)
	Maximal	consecutive prof...	25.67 (6)	consecutive loss ...	-12.02 (3)
	Average	consecutive wins	2	consecutive losses	2

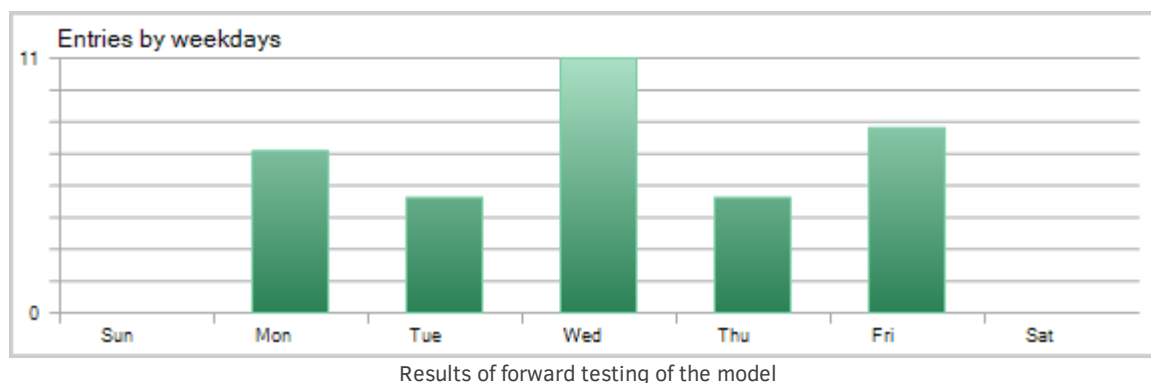
Results of forward testing of the model

7. Testing trading capabilities of the model

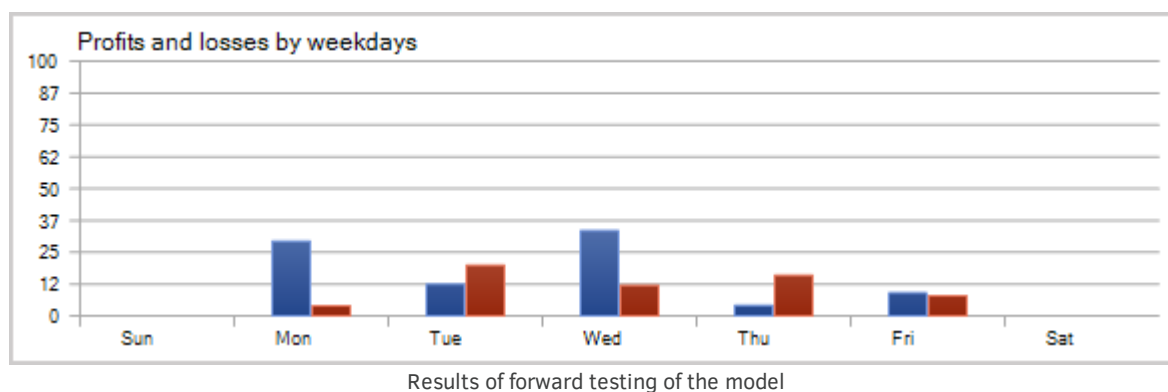
We did not integrate time-based transaction filtering into our Expert Advisor, nor did we provide time benchmarks for the training model. As a result, we see that the Expert Advisor opens positions more or less evenly throughout all trading sessions.



However, throughout the week, we see a significant advantage in opening positions on Wednesday (about 30%). Friday and Monday follow next. The fewest positions are opened on Tuesday and Thursday.



The Expert Advisor achieves the highest profitability on Wednesday and Monday. At the same time, the profit-to-loss ratio is better on Monday. On Friday, the profit and loss are balanced around the break-even point. However, on Tuesday and Thursday, the losses exceed the profits obtained. Such analysis potentially allows us to increase the profitability of the Expert Advisor by excluding inherently unprofitable trades. For instance, if we add a filter for opening positions based on the days of the week, we can increase the overall profitability of the Expert Advisor by making trades only on Monday and Wednesday.



In general, the result of the Expert Advisor profitability on new data allows for the following conclusions:

1. During technical analysis, it's possible to identify certain patterns that can generate quite stable signals for executing trades with profitability of at least 60%.
2. The use of neural network models makes it possible to identify such patterns.
3. Building an Expert Advisor based on neural networks allows for stable profitability over an extended period of time.

Conclusion

We have reached the end of the book. Throughout its pages, we have explored some of the most popular architectural solutions from various domains. These include convolutional models used in image recognition tasks, recurrent models for processing temporal sequences, and the Transformer with the *Self-Attention* mechanism, developed for solving language-related tasks.

My intention in showcasing these diverse architectural solutions wasn't merely to provide examples. It's a reminder to never fear to experiment. While it's easier to follow the beaten path, it only leads to repeating what has already been achieved, no matter how good these achievements may be. While there is nothing inherently wrong with this, true innovation and personal growth come from venturing off-road and embracing the unknown. The outcomes of such journeys are uncertain as they may lead to acclaim and success or fade into obscurity. Yet, I firmly believe that every effort contributes to our growth. As you move forward, I hope you find success in achieving your goals.

In this book, we have built a library that will assist you in implementing your own neural network models, training them on historical data, and testing their performance in the strategy tester using the provided Expert Advisor template. I wish you to find the model that will bring you profit and prosperity. It is important to remember: Make sure to thoroughly verify and comprehensively test the Expert Advisor before entrusting it with your savings.

See you soon. You can always find more information on mql5.com website.