Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Глибинна нейронна мережа для розпізнавання образів на основі М-нейронів
(тема)

Виконав:
студент 2 курсу, групи СШІм-20-2
Албасова А.І.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту
(повна назва спеціалізації)

Керівник проф. Бодянський Є.В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ В.О. Філатов
(підпис) (прізвище, ініціали)

2022 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту (СШІ)
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Албасовій Аліні Ігорівні
(прізвище, ім'я, по батькові)

1. Тема роботи Глибинна нейронна мережа для розпізнавання образів на основі М-нейронів

затверджена наказом університету від 24 березня 20 22 р. № 414 Ст.

2. Термін подання студентом роботи до екзаменаційної комісії 11 травня 20 22 р.

3. Вихідні дані до роботи наукові та науково-технічні публікації, дані Інтернет джерел та наукових проектів, що описують задачі комп'ютерного зору, інтелектуальної обробки цифрових зображень, а також проблеми навчання в реальному часі, програмно-технічна документація бібліотек глибинного навчання

4. Перелік питань, що потрібно опрацювати в роботі subject area analysis and formalized statement of the problem; neural networks: bulding blocks and learning procedures; neural networks based on matrix neurons and its learning; experimental research

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Figure 2.1 – Sigmoid function and its derivative; Figure 2.2 – Hyperbolic tangent and its derivative; Figure 2.3 – ReLU and its derivative; Figure 2.4 – Modifications of ReLU; Figure 3.1 – Architecture of M-neuron; Figure 4.1 – Example of images; Figure 4.2 – The results of experiments with the M-neuron; Figure 4.3 – The results of binary classification using a single-layer network with two M-neurons; Figure 4.4 – Graph of loss function with SGD; Figure 4.5 – Graph of loss function with Adam and ReLU; Figure 4.6 – Graph of loss function with Adam and using LeakyReLU; Figure 4.7 – Graph of loss function vs. iterations during training with Adam and PRELU; Figure 4.8 – Graph of loss function vs. iterations during training with Adam and ELU; Figure 4.9 – Graph of loss function vs. iterations during training with Adam SELU; Figure 4.10 – Figure 4.17 - Results

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз література та Інтернет джерел	28.03 – 31.03.2022	Виконано
2	Аналіз предметної області	31.03 – 03.04.2022	Виконано
3	Узгодження з керівником та постановка задачі	03.04 – 04.04.2022	Виконано
4	Аналіз поточного стану в області глибинного навчання,	04.04 – 07.04.2022	Виконано
	обробки зображень та онлайн навчання		
5	Розробка математичних моделей для матричних мереж	07.04 – 12.04.2022	Виконано
6	Підготовка та проведення експериментальних досліджень,	12.04 – 17.04.2022	Виконано
	програмна реалізація та аналіз результатів		
7	Підготовка та оформлення пояснювальної записки та	17.04 – 20.04.2022	Виконано
	презентації		
8	Рецензування, проходження нормоконтролю	20.04 – 30.04.2022	Виконано
9	Захист роботи	11.05.2022	Виконано

Дата видачі завдання 28 березня 20 22 р.

Студент _____
(підпис)

Керівник роботи _____ _____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Записка пояснювальна: 77 с., 22 рис., 4 табл., 1 дод., 44 джерела.

БАГАТОШАРОВІ НЕЙРОННІ МЕРЕЖІ, ГЛИБИННЕ НАВЧАННЯ, МАТРИЧНА НЕЙРОННА МЕРЕЖА, МАТРИЧНИЙ НЕЙРОН, РОЗПІЗНАВАННЯ ОБРАЗІВ, ШТУЧНА НЕЙРОННА МЕРЕЖА.

Об'єкт дослідження – архітектура та навчання штучних нейронних мереж.

Предмет дослідження – методи моделювання штучних нейронних мереж та алгоритми їх навчання.

Мета роботи – підвищення швидкості розпізнавання образів.

Методи дослідження – аналіз існуючих нейромережевих підходів для обробки двовимірних даних, розробка архітектури та алгоритму навчання матричного нейрону та глибинної нейронної мережі на його основі, програмна реалізація та проведення комп’ютерного експерименту, обробка та аналіз отриманих результатів.

Під час виконання кваліфікаційної роботи проведений теоретичний аналіз літературних джерел щодо методів моделювання штучних нейронних мереж та алгоритмів їх навчання, наукових публікацій щодо розробки нейронних мереж для безпосередньої роботи з двовимірними входами. Було виділено основні недоліки існуючих підходів та запропоновано новий тип штучного нейрону – М-нейрон та алгоритм його навчання. На його основі розроблена нейронна мережа. Реалізований програмний модуль для імітаційного моделювання мережі, протестований на задачі класифікації зображень. На основі результатів був проведений аналіз швидкодії запропонованого підходу та обчислені метрики для оцінки якості класифікації.

РЕФЕРАТ

Пояснительная записка: 77 с., 22 рис., 4 табл., 1 прил., 44 источника.

ГЛУБИННОЕ ОБУЧЕНИЕ, ИССКУСТВЕННАЯ НЕЙРОННАЯ СЕТЬ, МАТРИЧНАЯ НЕЙРОННАЯ СЕТЬ, МАТРИЧНЫЙ НЕЙРОН, МНОГОСЛОЙНЫЕ НЕЙРОННЫЕ СЕТИ, РАЗПОЗНАВАНИЕ ОБРАЗОВ.

Объект исследования – архитектура и обучение искусственных нейронных сетей.

Предмет исследования – методы моделирования искусственных нейронных сетей и алгоритмы их обучения.

Цель работы – повышение скорости распознавания образов.

Методы исследования – анализ существующих нейросетевых подходов для обработки двумерных данных, разработка архитектуры и алгоритма обучения матричного нейрона и глубинной нейронной сети на его основе, программная реализация и проведение компьютерного эксперимента, обработка и анализ полученных результатов.

В квалификационной работе проведен теоретический анализ литературы по методам моделирования искусственных нейронных сетей и алгоритмов их обучения, научных публикаций по разработке нейронных сетей для работы с двумерными входами. Были выделены основные недостатки существующих подходов и предложен новый тип искусственного нейрона – М-нейрон и алгоритм его обучения. На его основе разработана, программно реализована и протестирована исскуственная нейронная сеть. На основе результатов проведен анализ быстродействия предложенного подхода и вычисленны метрики для оценки качества классификации.

ABSTRACT

Explanatory note: 77 p., 22 fig., 4 tabl., 1 ann., 44 sources.

ARTIFICIAL NEURAL NETWORK, DEEP LEARNING, MATRIX NEURAL NETWORK, MARTIX NEURON, MULTILAYER NEURAL NETWORKS, PATTERN RECOGNITION.

The object of the study is architecture and learning process in artificial neural networks.

The subject of the study is artificial neural networks modelling methods and their learning algorithms.

This master's thesis aims to develop the architecture and the learning algorithm for a deep neural network based on a matrix neuron.

The methods of investigation include analysis of existing neural network-based approaches for processing 2D data, development of architecture and learning algorithm for matrix neuron and deep neural network based on it, software implementation and computer experiment, processing and analysis of the results.

A theoretical analysis of the literature on artificial neural networks modelling methods and learning algorithms, scientific publications on the development of neural networks for work with two-dimensional inputs was carried out. The main disadvantages of the existing approaches were highlighted, and a new type of artificial neuron and its learning algorithm were proposed. Based on it, an artificial neural network has been developed, programmatically implemented and tested. Based on the results, an analysis of the speed and accuracy of the proposed approach was performed.

CONTENTS

GLOSSARY, ABBREVIATIONS, ACRONYMS

ANN – Artificial Neural Network;

CNN – Convolutional Neural Network;

CV – Computer Vision;

DNN – Deep Neural Network;

ELU – Exponential Linear Unit;

MLP – Multi-Layer Perceptron;

MNIST – Modified National Institute of Standards and Technology;

NLP – Natural Language Processing;

ODL – Online Deep Learning;

ReLU – Rectified Linear Unit;

ViT – Visual Transformer.

INTRODUCTION

Several years ago, IBM analytics concluded that more than 90% of world's data were generated during last two years [1]. And today, it is probably that the actual number is much higher. As a result of global digitalization, this number has been growing exponentially, and it continues to do so. About 80-90% of these data are unstructured [2], e.g., photos and video, text documents, invoices, social media posts, audio, etc. Furthermore, the share of real-time data, such as sensor measurements, satellite data transmission, stock market and financial data, is also rising steadily. According to the Harvard Business Review, in 2017, less than 1% of unstructured data was generally used and analysed [3]. In the case of streaming data, this percentage is even smaller.

Businesses and everyday lives are data-driven and data-dependent now, so the ability to effectively process data becomes crucial. And on top of that, the speed of data processing and adaptability of used approaches play an important role. Now, most analytics and forecasting tools use static models that, once built, do not change. But the reality we live in is dynamic, and the events like Covid pandemic have already proved that systems that cannot adapt quickly will fall.

In the area of computer vision systems design, it may be not so obvious which changes may degrade the performance of such systems. It would seem that visual data has stable patterns, i.e., a dog is a dog, in terms of the common toy example of cat versus dog image classification with a neural network. But if we dive deep enough, there is room for improvement in terms of computer vision algorithms' adaptability. Domain and concept drift are issues that are relevant to all areas of data analysis and machine learning, including computer vision.

Among the most effective ways to deal with it are retraining and online learning. When computer vision systems reach a certain size, both in terms of the domain they cover and the resources expended, retraining becomes a highly inefficient or even impossible tool. But it can be useful in case of sudden domain drift when the nature of data has changed dramatically. The second approach

mentioned seems to be a prominent way to prevent concept drift. Online learning is a learning strategy when a model processes one sample at a time, allowing for real-time updates.

Unlike batch processing, the input of online learning algorithms is data streams. It makes it possible to consider the impact of new data without the necessity of a retraining model.

However, streaming data has its characteristics, which should be taken into account when developing algorithms for their processing. Among them are the following:

– elements arrive continuously and sequentially;

– data streams are generated by external resources, so the systems used to process them usually do not have direct access to the data source and cannot control it;

– the initial characteristics of the data streams are uncontrollable and usually unpredictable;

– sensitivity to changes in data distribution due to the dynamic nature of the real environment;

– data stream elements are error prone.

Due to these features of data streams, traditional machine learning algorithms are not suitable for processing such data due to computational and running time resource limitations. In the case of unstructured data, examples of which are images, it becomes even more difficult. The de facto standard in computer vision is complex artificial neural networks, the training of which is computationally expensive and time-consuming. Usually, ANNs work with batched data, and it is supposed that all training data is available before training. For streaming data, this is not the case.

The development of neural network architectures and learning algorithms that can quickly process unstructured streaming data is a logical step towards increasing the proportion of data that can be analysed and used by organizations to optimize business processes.

1 SUBJECT AREA ANALYSIS AND FORMALIZED STATEMENT OF THE PROBLEM

1.1 Image recognition

Image recognition is one of the most common techniques in Computer vision that allows computers to interpret and classify information presented in the form of images or videos (i.e., series of images). It is also often refferred as image classification and is used as a core component in many machine learning systems that solve Computer vision problems.

In simple terms, image recognition is a process of tuning machine learning model using a known and, in most cases, labeled dataset (i.e., training dataset) for future use of tuned model to predict class labels of previously unknown objects that belong to one of the classes on which the model was tuned (i.e., testing dataset).

The goal of image classification is to build machine learning model that, given input data X, approximates the true relation between objects in images (i.e., X) and their respective class labels (i.e., Y) and, therefore, is capable to predict class labels of new inputs. More formally, image classification goal is to build $h: X \rightarrow Y$ such that $h(X_t) = y_t$ [4].

Classification is a common task in machine learning where we have feature vectors xs and class labels ys. However, in the case of computer vision, building feature vectors is an essential step, which largely determines the model success. Both classical machine learning algorithms (e.g., kNN, decision trees) and the first neural networks that were applied to the problem of image classification used a simple flattening operation to transform a 2D image into a feature vector. From an image processing point of view, this approach obviously does not work since images have strong relationships between columns and rows, and simple flattening results in the loss of a huge amount of useful information. Moreover, in the case of neural networks, this approach is quickly becoming problematic as the

number of tuning parameters increases dramatically.

For that reason, we first need to extract relevant and low-dimensional features. There are two approaches to extracting features from an image: handcraft and learned feature extraction. Examples of the former approach are Local Binary Patterns and Principal Component Analysis. However, since this work is devoted to the development of a modification of the neural network approach to image classification, the second type of feature extraction deserves special attention. And this is where convolutional neural networks come in.

Traditionally, the history of CNNs starts from LeNet-5 architecture proposed by Yann Le Cun in a paper [5] published in 1998. The idea behind LeNet-5 is to pass the input image through a series of convolutions and subsampling blocks to extract low-dimensional and meaningful features and then transform those features into a vector fed into the dense layer to perform image classification. Convolutional layers are used to extract meaningful features while subsampling purpose is to reduce the dimensionality of the features number of parameters (i.e. reduce the number of parameters). The building blocks of LeNet-5 have obviously been known and used by mathematicians long before it was introduced. But the main contribution of this idea was to combine these building blocks in a way that allow the construction of relevant features through learning, rather than handcrafting them. Architecture was quite simple and had many drawbacks, but it marked the beginning of convolutional neural networks era in computer vision. The next milestone was winning the ILSVRC image classification competition by AlexNet, in 2012. This work covered a batch of key ideas:

– first, authors used ReLU activation function instead of more traditional funtions (i.e., sigmoids and hyperbolic tangent). It had positive impact on the efficiency of gradient proparagion;

– second, a dropout technique was introduced. Dropout is a regularization technique that makes the network randomly forget things to prevent the network from overfitting;

– also, paper introduced data augmentation.

With the advent of each new architecture, networks went deeper, and this trend continues to the present day. Going deeper required developing techniques to make training of such networks feasible. In VGG Network [6], authors abandoned the use of filters larger than 3x3 to going deeper. They proved that three layers with 3x3 filters arranged sequentially could replace 7x7 filters, and in this case, 55% fewer parameters are used. Similarly, two layers with 3x3 filters stacked sequentially can replace a 5x5 filter layer, which saves 22% of network parameters.

Inception modules first proposed in paper [7] allowed us to go deeper by applying convolutions of different sizes to the same input and then concatenating the resulting feature maps. In the same paper concept of 1x1 convolution was introduced. It allowed reducing the dimensionality of features while preserving beneficial representation.

At some point, researchers implemented that simply stacking more layers does not lead to better performance. Moreover, it turned out that there was a point where an increase in depth causes accuracy to degrade. And to overcome this residual learning was introduced [8]. The fundamental idea of ResNet is introducing a so-called “shortcut connection” that skips one or more layers, i.e., instead of directly approximating the underlying mapping $H(x)$, we learn a residual function $H(x) - x$. This is done by making the output of a stack of layers be $y = F(x) + x$, where $F(x)$ is the output of the layers (before the activation function of the last layer), and then the original input x is element-wise added.

For a long time, different variations of convolutional neural networks have been the actual standard for computer vision tasks. More recently, however, Visual Transformers (ViT) [9] attained remarkable performance on a range of computer vision tasks. Unlike CNNs, which have powerful spatial bias, ViTs relies more on using model regularization or data augmentation for training, especially on small datasets. Transformers are networks that process sequences of data. These sequences are first tokenized and then fed into the transformers. As

transformers use self-attention layers that have quadratic complexity, it is computationally intractable to apply them to images directly as a number of operations would scale quadratically with the number of pixels per image if applied at the level of each pixel. Hence, the input of ViTs is represented as a sequence of image patches. All the following operations are similar to those used for NLP tasks. Using transformers in computer vision is an active research area now, and there are still many open questions to be addressed.

In the paper [10] researchers from Google Brain team questioned the need of convolutions and attention for computer vision tasks. They introduced so-called MLP-Mixer, an architecture based on multi-layer perceptrons. As the authors stated, being mathematically and technically simpler, it achieved comparative performance in terms of a trade-off between accuracy and computational cost.

The authors of paper [11] introduced simple architecture meant to support the hypothesis that the usage of patches as the input representation is primarily responsible for ViT performance. ConvMixer outperforms ViTs, MLP-Mixers, and traditional vision models, according to the study. While there is not enough theoretical support on the effectiveness and efficiency of ConvMixers, the paper questioned some of the design conventions for ANN architectures that we have taken for granted for years without having a clear answer to the "why?" question.

Another thought-provoking paper when the authors applied techniques used in one ANN architecture design to another is [12]. The authors attempted to improve ResNet-like architecture by applying Transformers concepts.

1.2 Online and batch learning in machine learning

Batch learning is a de-facto standard in machine learning, especially in deep learning. When using this approach, it is assumed that the model is built offline on the entire training set and is never updated in the future. In addition, deep neural networks are usually trained in a multi-epoch learning mode that takes a long time.

However, in the case of streaming data processing, this approach is not possible for several reasons. First, it is usually assumed that the data streams are sequences of infinite length. And only a small part of this sequence can be stored in memory. Second, streams require each element to be processed in real-time; otherwise, that element will be lost forever due to the dynamic nature of the data stream. Third, the distribution of data underlying the data stream may change over time. Consequently, old data may be irrelevant or even harmful to modelling the current concept.

It should be noted that in contrast to batch learning, when processing data in a stream, it is expected that the characteristics of the new observed data change compared to past data.

1.2.1 Current state of online machine learning

Online learning approaches, like traditional (offline) machine learning algorithms, can be used to tackle a variety of tasks in a wide range of real-world application fields. These tasks include, but are not limited to:

– supervised machine learning tasks. Classification is one of the most popular tasks in this subgroup. It aims to predict the categories to which a new data instance belongs based on previous training data examples with labeled categories. Another popular supervised learning tasks is regression which aims to estimate relationships among one (but in recent times this is not a mandatory limitation) dependent and one or more independent variables. For regression analysis applications, such as time series analysis in financial markets, where data examples arrive in a sequential order, online learning approaches are naturally utilized;

– unsupervised machine learning tasks. The most common task here is clustering – a method for arranging items so that objects in the same group ("cluster") are more similar than those in different clusters. The online cluster analysis is actual for the data stream mining;

– other machine learning tasks. Other types of machine learning tasks that can be done online include learning for recommender systems, learning to rank, and reinforcement learning. For example, by learning to improve collaborative filtering jobs progressively from continuous streams of ratings/feedback information from users, collaborative filtering with online learning can be used to improve the performance of recommender systems [13].

In the paper [13] a lot of online learning algorithms are overviewed. However, most of the existing online learning algorithms are designed to learn shallow models with online convex optimization, which means they cannot learn complex nonlinear functions in complicated application settings. While now deep neural networks are used to solve most real problems. This creates a big gap between the theoretical research of online learning and the real needs of productional machine learning, which for the most part today is represented by a wide range of different neural networks, which are mostly cumbersome and data hungry.

1.2.2 Online deep learning

In the paper [14], the authors attempted to fill the gap between theoretical online learning research and deep learning reality by introducing a separate direction in online learning called Online Deep Learning (ODL), the task of which is to train Deep Neural Networks in a real-time setting.

Online learning techniques applicable to deep neural networks training can be roughly divided into two groups:

– in the first case, the DNNs train using traditional backpropagation (or its modifications), however, only one sample is processed at each time point. This approach has a number of shortcomings. One of them is that it is not a trivial task to find the optimal model capacity before starting training;

– alternatively, authors proposed an approach where model architecture is evolving starting from a simple shallow network and increasing capacity as the

concepts become more complex. In order to do this, they, firstly, attach separate output classifier for each hidden layer. Then they apply a technique known as Hedge backpropagation, which examines the performance of each such output classifier at each step and extends the backpropagation algorithm to train DNN online by utilizing classifiers of various depths. According to the authors, this makes it possible to build a DNN of optimal capacity in real time setting, while maintaining knowledge sharing between shallow and deep networks.

As both approaches are areas of active research now it seems promising not only develop them separately, but to combine them to get even more powerfull techniques for online deep learning.

Returning to the topic of this graduation project, the the first approach is closest to the solution proposed within it. Proposed neural network and its learning algorithm aimed at speeding up the process of image processing, making online learning possible. Why this is possible will be described in the main section.

1.2.3 Use Case: processing visual information in real time

The share of visual information in the total amount of information is growing every year. Image analysis (also known as "computer vision") allows computers to recognize concepts expessed visually in images. Areas of application of image analysis have long gone beyond photo editors and social networks. Now businesses are also using images as a means of extracting more informative data for their business needs. For example, innovations in the field of semiconductor production are contributing to the size of computers becoming smaller and smaller as their capacity increases. However, the ability to optimize workflows is also important. When digital printing of semiconductor components, the failure rate of "one drop per billion" may seem acceptable. However, given that up to 50 million drops per second can be expelled, this leads to an unacceptable level of defects – "1 drop every 20 seconds".

One very large semiconductor company uses image processing and

machine learning to automatically detect defective plates at the beginning of the production process in order to avoid low profitability.

Another promising area of image processing is healthcare. Some researchers estimate that medical images, which are becoming the largest and fastest-growing data sources in the industry, account for at least 90 percent of all medical data, and that number continues to grow. From daily X-rays in the dentist's office to three-dimensional magnetic resonance imaging in hospitals, imaging has become the biggest driver of healthcare data, generating millions of terabytes of data each year in the United States alone. Whether to automate the work of radiologists or to detect diseases that would not be detected by the human eye, the analysis of biological images using deep learning methods and artificial intelligence will have a huge impact on medical imaging in the near future.

The considered examples of the use of image analysis can be considered as typical examples of visual data streams. These data are processed continuously, at high speed, require instant decision-making and have a high error rate. Therefore, the key to solving such problems is the development of algorithms that can be configured and used in real time and at the same time have high accuracy and reliability. To date, the most popular in the field of image processing are approaches using deep neural networks, including convolutional neural networks and transformers. They have shown good results in solving a variety of computer vision problems, including classification. However, the bottleneck of such systems is that almost all of them are:

- cumbersome;
- data hungry;
- unable to work in online setting;
- computational expensive;
- time consuming.

1.3 Analysis of neural network approaches based on direct processing of matrix inputs

1.3.1 Definitions and prospects of using matrix neural networks

As mentioned above, convolutional neural networks (CNNs) is a common approach for solving a wide range of image processing problems. The input signal of CNNs is usually an image represented as a high dimensional matrix. CNN trasforms this matrix with a sequence of convolutional and subsampling layers into a relatively low dimensional vector. Then this vector goes through a sequence of fully connected layers (i.e., multilayered perceptron) whose role, in case of image classification, is to predict the final class scores. All this complex system is tuned using the well-known learning rule based on backpropagation of errors. Almost all modern CNN architectures follow this approach with various modifications related to selecting the type of activation function and protection from unwanted effects of vanishing and exploding gradients. So, CNNs process vector signals during their training in one way or another. Moreover, the adjusted weights in the perceptron layers are also grouped in a vector form.

It is more natural from the image processing point of view not to convert them into a vector form, but to analyse them in a matrix form of reduced dimension, as a result of which the connections between rows and the connections between the columns of the initial image signal are preserved.

Aforementioned approach is implemented in so-called matrix neural networks [15], [16], [17], [18], [19], [20], [21] which are based on a bilinear transformation of the following form [22], [23]:

$$\hat{Y} = \{\hat{y}_{j_1 j_2}\} = AXB, \tag{1.1}$$

where $\hat{Y} \epsilon R^{m_1 \times m_2}; j_1 = 1,2,\dots,m_1, j_2 = 1,2,\dots,m_2$ is a predicted output of the proposed system;

$X = \{x_{i_1 i_2}\} \epsilon\, R^{n_1 \times n_2}, i_1 = 1,2,\dots,n_1,\ i_2 = 1,2,\dots,n_2$ is an input matrix signal;

A and B are $(m_1 \times n_1), (n_2 \times m_2)$ -transformation operators, the parameters of which need to be determined.

Based on (1.1) in [24], [25] the following adaptive matrix bilinear model was introduced:

$$\hat{Y}(k) = A(k-1)X(k)B(k-1), \tag{1.2}$$

(here k is a number of current iteration) as well as reccurent algorithm for tuning parameters matrices $A(k)$ and $B(k)$.

1.3.2 Neural networks based on adaptive matrix bilinear model

Based on adaptive linear model (1.2) y [19], [20], [21] the first versions of matrix neural networks were introduced as well as backpropagation based learning algorithms for its training. Each layer of such a network relizes the following non-linear transformation:

$$\hat{Y}(k) = \Psi \odot \big(A(k-1)X(k)B(k-1)\big), \tag{1.3}$$

$\Psi -$ $(m_1 \times m_2)$-operator, formed by $m_1 m_2$ non-linear activation functions.

To tune such networks, a modified delta learning rule was proposed, for the implementation of which it was proposed to introduce three types of errors, which in scalar form can be written as follows:

$$e_{j_1 j_2}(k) = y_{j_1 j_2}(k) - \psi\Big(A_{j_1 j_2}(k-1)X(k)B_{j_1 j_2}(k-1)\Big), \tag{1.4}$$

$$e_{A_{j_1 j_2}}(k) = y_{j_1 j_2}(k) - \psi\Big(A_{j_1 j_2}(k) X(k) B_{j_1 j_2}(k-1)\Big), \tag{1.5}$$

$$e_{B_{j_1 j_2}}(k) = y_{j_1 j_2}(k) - \psi\Big(A_{j_1 j_2}(k) X(k) B_{j_1 j_2}(k)\Big), \tag{1.6}$$

and, relatively, two learning criteria, that in the case of using the quadratic criterion can be written as follows:

$$E^*_{A_{j_1 j_2}}(k) = \frac{1}{2} e_{j_1 j_2}(k), \tag{1.7}$$

$$E^*_{B_{j_1 j_2}}(k) = \frac{1}{2} e_{A_{j_1 j_2}}(k). \tag{1.8}$$

Or in the case of using the cross-entropy criterion:

$$\begin{aligned} E^*_{A_{j_1 j_2}}(k) = \frac{1}{2}\Big(1 + y_{j_1 j_2}(k)\Big) \ln \frac{1 + y_{j_1 j_2}(k)}{1 + \hat{y}_{j_1 j_2}(k)} + \\ + \frac{1}{2}\Big(1 - y_{j_1 j_2}(k)\Big) \ln \frac{1 - y_{j_1 j_2}(k)}{1 - \hat{y}_{j_1 j_2}(k)}, \end{aligned} \tag{1.9}$$

$$\begin{aligned} E^*_{B_{j_1 j_2}}(k) = \frac{1}{2}\Big(1 + y_{j_1 j_2}(k)\Big) \ln \frac{1 + y_{j_1 j_2}(k)}{1 + \hat{y}_{A\, j_1 j_2}(k)} + \\ + \frac{1}{2}\Big(1 - y_{j_1 j_2}(k)\Big) \ln \frac{1 - y_{j_1 j_2}(k)}{1 - \hat{y}_{A\, j_1 j_2}(k)}. \end{aligned} \tag{1.10}$$

Regardless of the choice of the activation function and the learning criterion, the transformation implemented by the layer of the matrix neural network has the form:

$$O^{[l]}(k) = \Psi\odot(A^{[l]}(k-1) O^{[l-1]}(k) B^{[l]}(k-1)), \tag{1.11}$$

where $O^{[l]}(k)$ та $O^{[l-1]}(k)$ – output matrix of l-th and $l-1$-th layers, respectively;

$A^{[l]}(k-1)$ – weights matrix A of l-th layer;

$B^{[l]}(k-1)$ – weights matrix B of l-th layer;

$l = \overline{1,L}$ – layer number, where L is the overall number of layers in the network.

And the overall transformation performed by the network can be written as follows:

$$\hat{Y}(k) = \Psi\odot(A^{[L]}(k-1)\,\Psi\odot(A^{[L-1]}(k-1)\times\ldots\times \times B^{[L-1]}(k-1))B^{[L]}(k-1)). \tag{1.12}$$

The algorithm for training a network built on such layers using backpropagation of errors may be formulated as follows:

– for the final layer:

$$\Delta a^{[L]}_{j_1 j_2 i_1}(k) = \eta_A(k)\delta^{[L]}_{j_1 j_2}(k)\hat{o}^{[L-1]}_{i_1}(k), \tag{1.13}$$

$$\Delta b^{[L]}_{j_1 j_2 i_2}(k) = \eta_B(k)\delta^{[L]}_{A_{j_1 j_2}}(k)\hat{o}^{[L-1]}_{A_{i_2}}(k), \tag{1.14}$$

where

$$\delta^{[L]}_{j_1 j_2}(k) = e_{j_1 j_2}(k)\,\psi'\left(u^{[L]}_{j_1 j_2}(k)\right), \tag{1.15}$$

$$\delta^{[L]}_{A_{j_1 j_2}}(k) = e_{A_{j_1 j_2}}(k)\,\psi'\left(u^{[L]}_{A_{j_1 j_2}}(k)\right), \tag{1.16}$$

and

$$\hat{o}_{i_1}^{[L-1]}(k) = \sum_{i_2=1}^{n_2} b_{j_1 j_2 i_2}^{[L]}(k-1) o_{i_1 i_2}^{[L-1]}(k), \tag{1.17}$$

$$\hat{o}_{A_{i_2}}^{[L-1]}(k) = \sum_{i_1=1}^{n_1} a_{j_1 j_2 i_1}^{[L]}(k) o_{i_1 i_2}^{[L-1]}(k)\,; \tag{1.18}$$

– or intermediate layers:

$$\Delta a_{j_1 j_2 i_1}^{[l]}(k) = \eta_A(k) \delta_{j_1 j_2}^{[l]}(k) \hat{o}_{i_1}^{[l-1]}(k), \tag{1.19}$$

$$\Delta b_{j_1 j_2 i_2}^{[l]}(k) = \eta_B(k) \delta_{A_{j_1 j_2}}^{[l]}(k) \hat{o}_{A_{i_2}}^{[l-1]}(k), \tag{1.20}$$

where

$$\delta_{j_1 j_2}^{[l]}(k) = \psi'\left(u_{j_1 j_2}^{[l]}(k)\right) \sum_{i_1=1}^{n_1} \delta_{j_1 j_2}^{[l+1]}(k)\, a_{j_1 j_2 i_1}^{[l+1]}, \tag{1.21}$$

$$\delta_{A_{j_1 j_2}}^{[l]}(k) = \psi'\left(u_{A_{j_1 j_2}}^{[l]}(k)\right) \sum_{i_2=1}^{n_2} \delta_{A_{j_1 j_2}}^{[l+1]}(k)\, b_{j_1 j_2 i_2}^{[l+1]}, \tag{1.22}$$

and

$$\hat{o}_{i_1}^{[l-1]}(k) = \sum_{i_2=1}^{n_2} b_{j_1 j_2 i_2}^{[l]}(k-1) o_{i_1 i_2}^{[l-1]}(k), \tag{1.23}$$

$$\hat{o}_{A_{i_2}}^{[l-1]}(k) = \sum_{i_1=1}^{n_1} a_{j_1 j_2 i_1}^{[l]}(k) o_{i_1 i_2}^{[l-1]}(k). \quad (1.24)$$

1.4 The formal statement of the research problem

From the artificial neural network architectures point of vantage, all systems proposed in [15], [16], [17], [18], [19], [20], [21] are so-called stacked neural networks [26]. The stacking approach complicates the analysis of the system in general and makes the training procedures somewhat cumbersome in terms of their optimization.

In order to address the aforementioned drawbacks, the so-called matrix neuron is introduced into consideration in this research work. Proposed M-neuron is a generalization of the popular F. Rosenblatt perceptron. Along with the M-neuron, its learning algorithm, based on the widely used delta rule, with its additional optimization is proposed.

The main task of this master's project is the following: based on the introduced M-neuron and its learning algorithm, develop the architecture and learning algorithm of a DNN. To do this, the following set of tasks was defined:

– to study the literature and Internet sources on the topic of learning algorithms and neural networks optimization;

– to formulate a mathematical model of a matrix neuron and to develop an algorithm for its learning based on the delta rule;

– to develop a matrix neural network layer, its learning algorithm;

– to build a deep matrix neural network and modify back propagation algorithm to make it possible to use it for training this type of network;

– to programmatically implement a matrix neuron, a layer based on it, and a neural network based on matrix layers as well as data preparation pipeline;

– to conduct experiments with introduced systems;

– to evaluate performance of the proposed systems and analyse the results.

2 NEURAL NETWORKS: BUILDING BLOCKS AND LEARNING PROCEDURES

Artificial neural network may be described as a computational system that is able to approximate function of any complexity. Initially inspired by the power of human brain, now this complex computational systems bear little resemblance to their biological mastermind. And in order to present a new neural network architecture, it is worth starting with the study of the components of these powerful computational models.

An artificial neural network can be described in terms of its architecture and learning algorithm.

2.1 Architecture of artificial neural networks: from basic to complex

2.1.1 Artificial neuron as smallest atomic unit of neural network

2.1.1.1 Models of artificial neuron

The first computational model of neuron was proposed by neuroscientists Warren McCulloch and Walters Pitts in the paper [27]. It was oversimplified and accepted only binary values as inputs and outputs. Its weights and bias were fixed so there was no possibility for learning.

The next step was improvements proposed by Donald Hebb who proposed learning rule for updating the weights.

Based on McCulloch and Pitts neuron and Hebb’s findings, Frank Rosenblatt proposed the artificial neuron model, which significantly influenced the development of modern deep learning theory called Rosenblatt's perceptron. Its idea is pretty simple while still powerful. It formed the foundation of dense or fully connected layer that can be found in almost every modern deep neural network architecture.

The transformation performed by Rosenblatt's perceptron is the following:

$$out = \sum_{i=1}^{m} w_i x_i + b, \tag{2.1}$$

where x_i – inputs;

w_i – learnable weights;

b – bias term;

m – input size.

The goal of such a simple neuron was to classify the set of stimuli x_i into one of two classes.

Another interesting but not so popular models of neurons include:

– Fukushima neuron;

– Hopfield neuron;

– Grossberg neuron.

2.1.1.2 Activation functions

The non-linear activation function is one of the elements that make neural networks universal approximators. The primary role of the activation function is to transform the aggregated input from the neuron into a scalar output value to be fed to the next hidden layer or used as output. Without using it, any neural network, no matter how deep, is transfigured into a simple linear regression model that is not able to process highly non-linear data.

There are dozens of different activation functions, and the choice largely depends on the problem being solved.

In historical terms, the first activation functions were so-called S-shaped functions: sigmoid function and hyperbolic tangent. They were highly inspired by the behaviour of biological neurons and have been used in the early days.

The sigmoid activation function and its derivative are given as:

$$S(z) = \frac{1}{1 + e^{-z}}, \tag{2.2}$$

$$S'(z) = S(z) \cdot (1 - S(z)). \tag{2.3}$$

The sigmoid takes a real scalar value as input and returns a value in range [0, 1]. The graph of sigmoid and its derivative are illustrated in figure 2.1. The sigmoid affected by the problem of vanishing gradients since the derivative tends to 0 at the ends. As a result, the training process is paralyzed. Furthermore, the lack of a zero-centric output results in poor convergence.

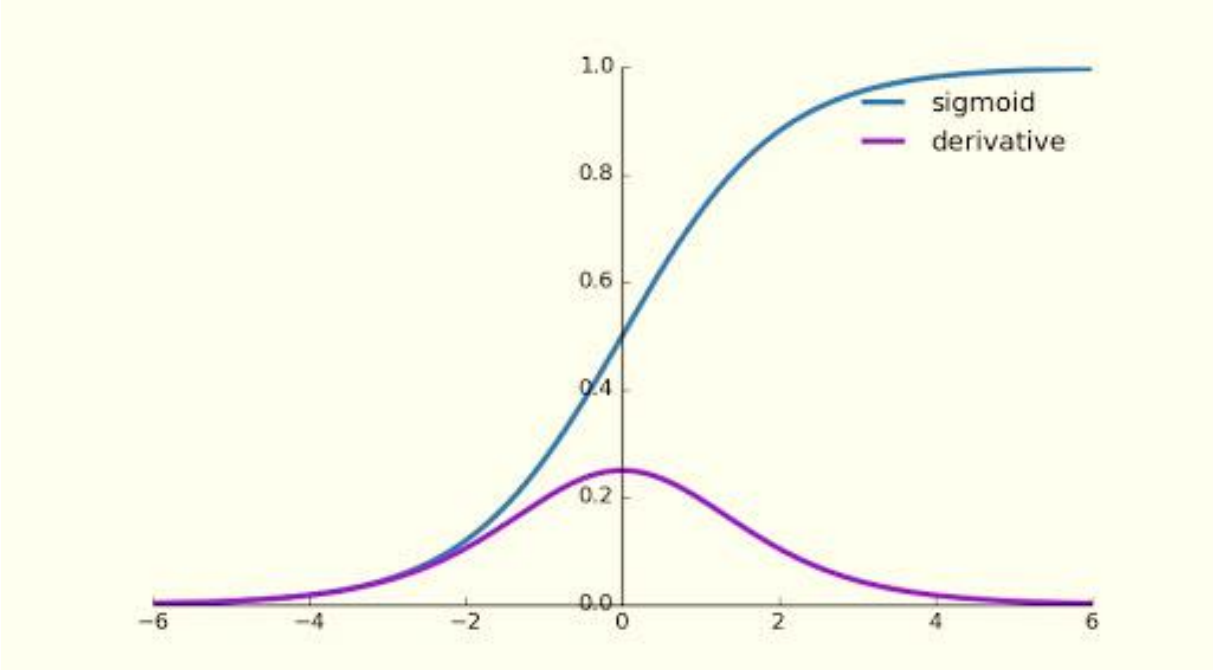

Figure 2.1 – An illustration of sigmoid funciton and its derivative

The hyperbolic tangent is similar to the sigmoid function, however it is zero-centred, as shown in figure 2.2. It is written as:

$$Tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}, \tag{2.4}$$

$$Tanh'(z) = 1 - Tanh^2(z). \tag{2.5}$$

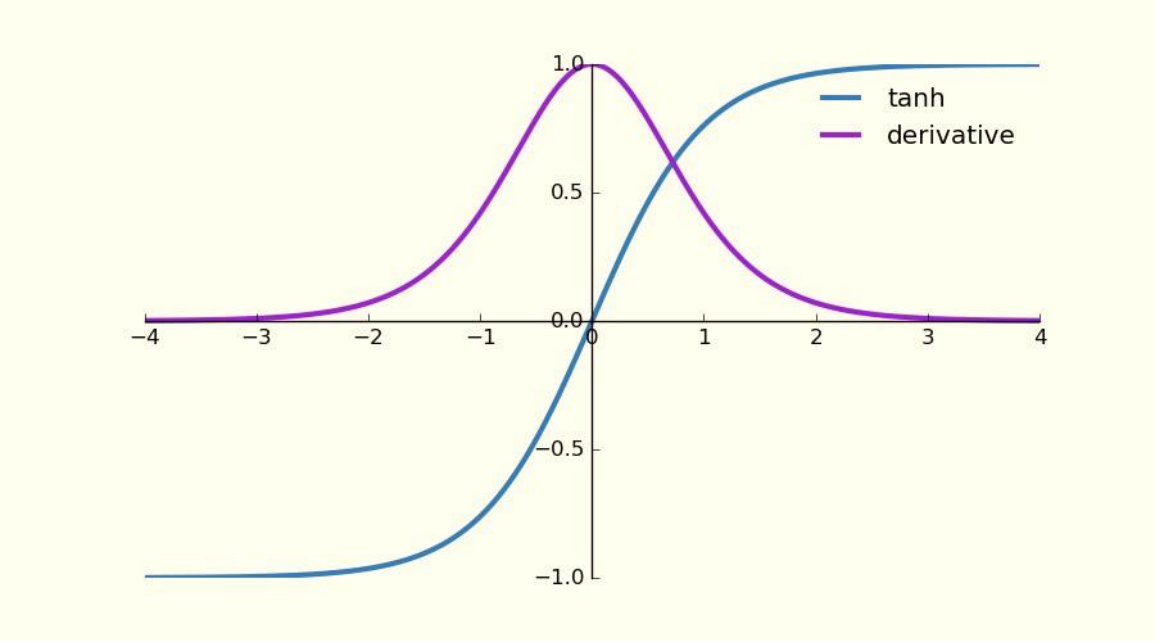

Figure 2.2 – An illustration of hyperbolic tangent and its derivative

The hyperbolic tangent suffers the same problems as sigmoid, so it is also rarely used outside of research papers.

In order to overcome aforementioned drawbacks of S-shaped activation functions, rectified linear unit activation function (ReLU) was introduced [28]. Graphs of ReLU and its derivative are shown in the figure 2.3.

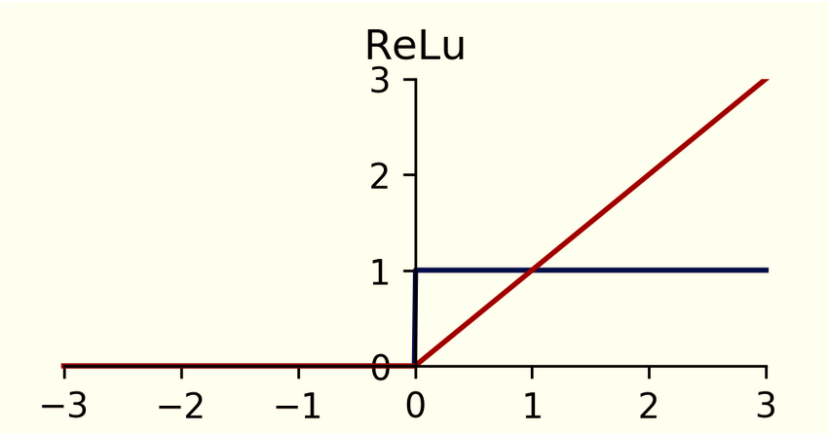

Figure 2.3 – An illustration of ReLU and its derivative

The ReLU function and its derivative are given as:

$$ReLU(z) = \max(0, z), \tag{2.6}$$

$$ReLU'(z) = \begin{cases} 0, & if\ z < 0, \\ 1, & if\ z > 0. \end{cases} \tag{2.7}$$

ReLU quickly became popular because of its simplicity and improved efficiency. However, it was not without shortcomings, among which were the 'dead' ReLU problem and lack of protection from the exploding gradient. To avoid the drawbacks of vanilla ReLU, its numerous modifications were introduced (figure 2.4).

One of the problems of ReLU is the underutilization of its negative values. In order to overcome this problem, several modifications were introduced, e.g.:

– Leaky ReLU [29]. The idea is to multiply negative values on some fixed coefficient usually from the range [0.1, 0.3]. The main problem of this function is to find right value of this coefficient;

– Parametric ReLU [30]. The idea is to make the coefficient introduced in Leaky ReLU a trainable parameter;

– Concatenated ReLU [31]. The idea is to concatenate outputs of ReLU over original input and negative input.

Comparing to S-shaped activation functions, ReLU-like activation has more limited non-linearity. In order to address this issue some other modifications were introduces, e.g. S-shaped ReLU [32], defined as a combination of three linear functions with four trainable parameters.

It follows from the formula (2.5) that ReLU's output is not upper-bounded that causes instabilities in network training procedures. Bounded ReLU [33] is a attempt to solve this problem by setting upper bound.

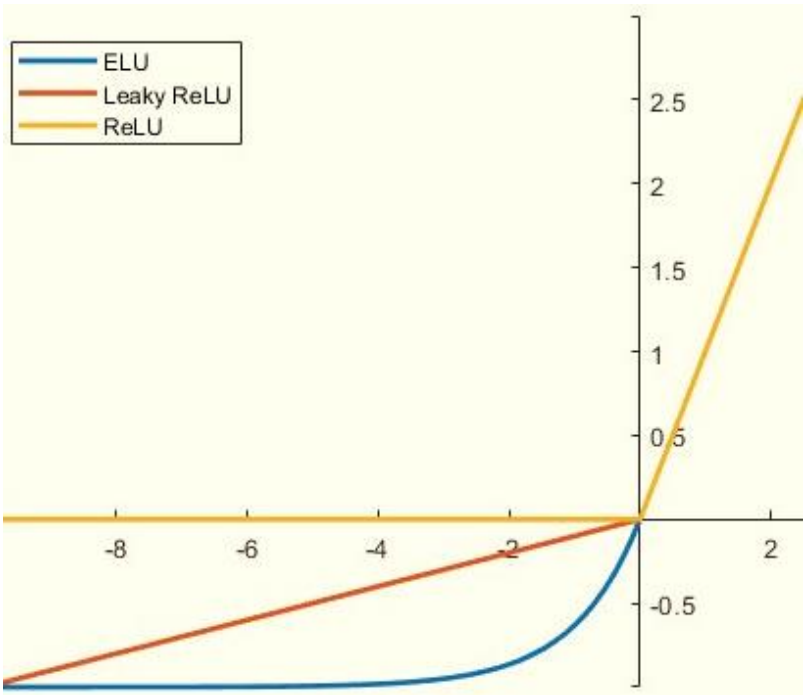

Figure 2.4 – Illustration of some popular modifications of ReLU

To some extent, the separate direction of ReLU-inspired activation functions is exponential linear unit (ELU) activation functions. ELU-like activation functions address the gradient vanishing problem of ReLU. The ELU function [34] and its derivative are given as:

$$ELU(z) = \begin{cases} z, & if\ z > 0, \\ \alpha \cdot (e^z - 1), & if\ z \leq 0.' \end{cases} \tag{2.8}$$

$$ELU'(z) = \begin{cases} 1, & if\ z > 0, \\ \alpha \cdot e^z, & if\ z \leq 0, \end{cases} \tag{2.9}$$

where α is trainable parameter.

ELU has all the advantages of ReLU and also solves some of its problems, namely being more noise tolerant, handling negative values better. There are several modifications of ELU, including scaled ELU [35], parametric ELU [36], etc.

Also recently, the so-called adaptive activation functions deserve special

attention. The idea is that activation function is also a learnable part that can be tuned during training. As a base function, any of the previously mentioned activation functions can be used, or it can even be constructed from scratch. For example, Mexican ReLU [37] is adaptive modification of ReLU. It is given as:

$$MeLU(z) = PReLU(z) + \sum_{k=1}^{K} c_k \times \max\left(\lambda_k - |x - a_k|, 0\right), \qquad (2.10)$$

where c_k is the learnable parameter;

λ_k and a_k are the real numbers.

In general, it is impossible to answer which activation function is the best option as each of them has both advantages and disadvantages. The choice of the right activation function is determined by the specific task that is being addressed. In addition, it is usually necessary to conduct dozens of experiments to define the suitable activation function as with all the other network hyperparameters.

2.1.2 Concept of neural network layer

The layer in an artificial neural network is a structural element of its architecture that unites individual neurons and usually acts as a unified transformation function that converts its input into some new representation. In that sense, the neuron may be seen like its particular case whose output representation is scalar. Each layer has its input, output, and transformation function that converts input to output. A layer is defined by its type, size, input type, and the activation function it uses.

Each neural network with more than three layers consists of 3 types of layers:

– input layer. This layer has direct access to the network input. It passes input to the rest of the layers;

– hidden layer. This layer has no direct access to either inputs or output

errors. Hidden layers are either one or more in number for a neural network. There is usually more than one hidden layer. In modern deep neural networks, there are dozens or even hundreds of them;

– output layer. This layer holds the output of the network. It has direct access to the network error.

The above classification of layer refers to their location in the network. The second important classification of layers is by their tasks. Different layers transform their inputs in different ways, and some layers are more suited for certain tasks than others. The common network layers are:

– dense layer. It is the most general-purpose neural network layer. Every neuron in one layer is coupled to every neuron in the next layer in dense layers. As input size grows, this type of layer can become computationally expensive. These layers are usually used in the classification heads of the networks;

– convolutional layer. This layer was firstly introduced in convolutional neural networks. It is used for extracting features in images data;

– recurrent layer. It is used in recurrent neural networks. It processes inputs as a sequences, its input usually consists of both the input data as well as the output from a previous transformation performed by this layer. Its modification is LSTM layer that have better long-term memory;

– attention layer. This layer is inspired by cognitive attention. When processing a sequence, we should choose which part of the sequence to collect information from. This is a basic yet powerful concept. The concept is straightforward: some elements of a sequence are more significant than others.

2.2 Neural network learning

2.2.1 Learning criteria

The learning criterion or loss function is an important neural network concept. The learning criterion function measures how well this model performs,

or, in other words, how expensive errors are.

Nowadays there are a huge number of learning criteria, and the choice of a particular one depends largely on the type of task. Moreover, within the task, there are more than one option for the loss function. What to choose is often determined by the subject area specifics and project priorities.

In general, learning criterion is defined as:

$$\mathrm{E} = \Phi(y_{out}, y_{pred}), \tag{2.11}$$

where y_{out} is target;

y_{pred} is model output;

Φ is some function that measure the quality of model output.

The examples of learning criteria include:

– mean square error (MSE) is a squared difference between targets and predictions. It is one of the most popular loss funcitons in regression tasks;

– cross-entropy. It evaluates a classification model's output, which is a probability value between 0 and 1.

2.2.2 Learning procedure

Most machine learning algorithms, in particular, neural networks use optimization algorithms to define optimal models for learning problems. Given the learning criterion, which shows how well the model performs, we need to find parameters set under which the model shows the best result. This process is called model training, and optimization techniques are usually used here. In the vast majority of cases, optimization models are presented in minimization form. The basic idea is to find such parameter space where the rate of change of the loss function in any direction is 0. In practice, such a condition is impossible, so it is often enough to find the local minimum of the loss function.

Most of the time, neural networks use first-order optimization techniques,

inter alia, Gradient Descent and its variations. The training process for a single-layer neural network is quite simple because the loss function is a direct function of weights, allowing easy gradient computation. The problem with a multi-layer network is that the loss is a complex composition of the weights from previous layers. Error backpropagation filled this gap, applying simple chain-rule of differential calculus. The high-level concept is to computer error gradients in terms of summations of local-gradient products over the various pathways from a neuron to the output [38]. The backpropagation process consists of two steps:

– forward path. It is required to compute network outputs and loss that shows how good prediction is. On this step, loss derivative with respect to output layer weights is computed;

– backward path. It is required to propagate the error back from the output layer neurons to the neurons of previous layers that don’t have direct access to it and update model weight.

3 NEURAL NETWORKS BASED ON MATRIX NEURONS AND ITS LEARNING

The layer of feed-forward network is based on the Rosenblatt's perceptrons that are merged into a vector. Similar to this, matrix layer can be defined, but before the matrix neuron, by analogy with The Rosenblatt's Perceptron, should be introduced into consideration.

3.1 Matrix neuron architecture

In the paper [39], we introduced matrix neuron. The transformation performed by matrix neuron can be written as follows:

$$\hat{y}_{j_1 j_2}(\tau) = \psi_{j_1 j_2}\left(w_{L_{j_1 j_2}}(\tau-1)\cdot X(\tau)\cdot w_{R_{j_1 j_2}}(\tau-1) + \theta_{j_1 j_2}(\tau-1)\right), \qquad (3.1)$$

where τ is the index number of the current iteration;

$\hat{y}_{j_1 j_2}(\tau)$ is an output of matrix neuron at the iteration τ;

$\psi_{j_1 j_2}$ is a non-linear activation function of M-neuron;

$X(\tau)$ is a $R^{n_1 \times n_2}$ - neuron input at the iteration τ;

$w_{L_{j_1 j_2}}(\tau-1)$ is a raw left set of M-neuron parameters adjusted after previous iteration $\tau-1$;

$w_{R_{j_1 j_2}}(\tau-1)$ is a column right set of M-neuron parameters adjusted after previous iteration $\tau-1$;

$\theta_{j_1 j_2}(\tau-1)$ is a bias term adjusted after previous iteration $\tau-1$.

It is important to note that the input to an activation function (i.e., pre-activation) is computed in 3 steps:

– on the first step, the original input $X(\tau)$ is multiplied by a row vector of parameters $w_{L_{j_1 j_2}}(\tau-1)$. As a result of it, an intermediane raw vector signal is computed;

– on the second step, intermediate raw vector signal is multiplied be a column vector of parameters $w_{R_{j_1 j_2}}(\tau-1)$. As a result of this operation, an intermediate scalar output signal is formed;

– on the third step, a scalar output is summed with the bias term $\theta_{j_1 j_2}(\tau-1)$. The result of it is a preactivation that will be denoted by $u_{j_1 j_2}(\tau)$.

Given the above, the equation (3.1) can be rewritten in even more compact form:

$$\hat{y}_{j_1 j_2}(\tau) = \psi_{j_1 j_2}\left(u_{j_1 j_2}(\tau)\right). \tag{3.2}$$

In order to simplify mathematical formulations, the bias term may be pushed in the input by adding additional constant dimensions. This convention may be useful in a case of writing each step of computing pre-activation as an independent transformation. Under this convention, at first, we consider $w_{R_{j_1 j_2}}(\tau-1)$ fixed, so $X(\tau)\cdot w_{R_{j_1 j_2}}(\tau-1)$ can be written as $X^{w_R}(\tau)$. After this reassigning and pushing bias term in the input, equation (3.1) takes the following form:

$$\hat{y}_{j_1 j_2}(\tau) = \psi_{j_1 j_2}\left(\widetilde{w}_{L_{j_1 j_2}}(\tau-1)\cdot \tilde{X}^{w_R}(\tau)\right), \tag{3.3}$$

where $\widetilde{w}_{L_{j_1 j_2}}(\tau-1) \equiv [\theta_{j_1 j_2}(\tau-1), w_{L_{j_1 j_2}}(\tau-1)]$;

$\tilde{X}^{w_R}(\tau) \equiv [1, X^{w_R}(\tau)]^T$.

Next, we consider $w_{L_{j_1 j_2}}(\tau-1)$ fixed , so $w_{L_{j_1 j_2}}(\tau-1)\cdot X(\tau)$ can be written as $X^{w_L}(\tau)$. Equation (3.1) takes the following form:

$$\hat{y}_{j_1 j_2}(\tau) = \psi_{j_1 j_2}\left(\tilde{X}^{w_L}(\tau)\cdot \widetilde{w}_{R_{j_1 j_2}}(\tau-1)\right), \tag{3.4}$$

where $\widetilde{w}_{R_{j_1 j_2}}(\tau-1) \equiv [\theta_{j_1 j_2}(\tau-1), w_{R_{j_1 j_2}}(\tau-1)]$;

$\tilde{X}^{w_L}(\tau) \equiv [1, X^{w_L}(\tau)]^T$.

But equations (3.3) and (3.4) are only applicable in a case when there is no sequence of M-neurons. In the case of sequence, we need to perform operations step by step without the possibility to fix one or another parameter set. In addition, in the case of such notation, the bias term is adjusted twice.

The M-neuron architecture is shown in figure 3.1.

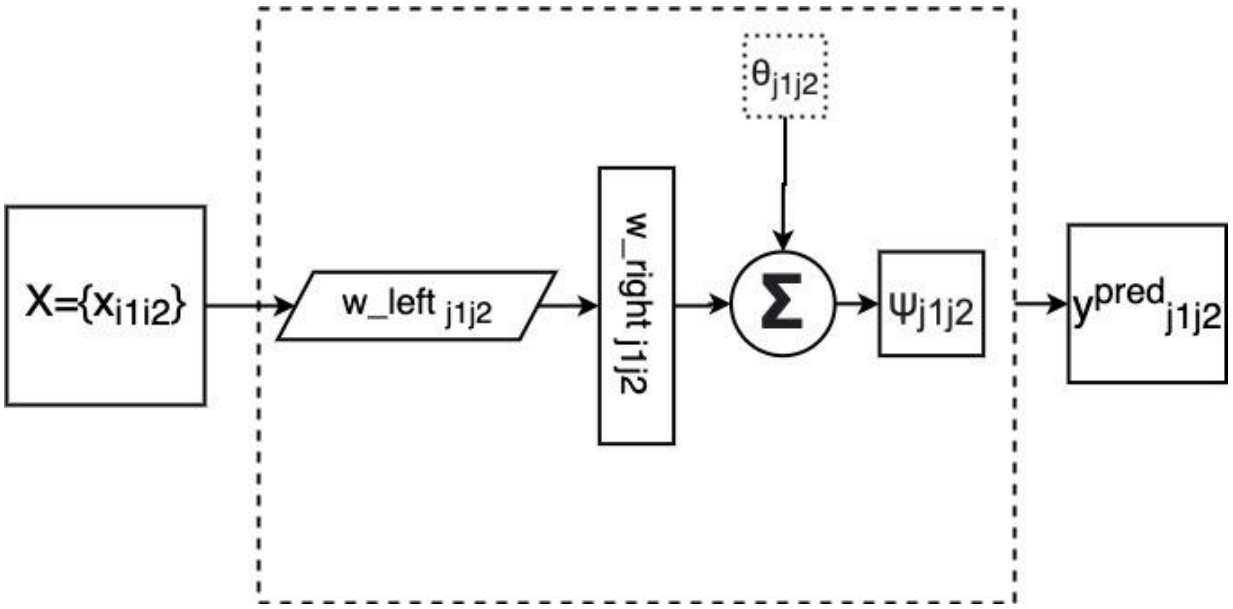

Figure 3.1 – High-level architecture of matrix neuron

3.2 Adaptive algorithm for learning a matrix neuron

In the paper [39] we proposed adaptive learning algorithm for adjusting parameters of matrix neuron. It is a modification of conventional δ-rule for the case of matrix output [40]. However, as matrix neuron has two sets of parameters the learning rule for each neuron consists of two steps. The updating of bias term is performed independently as in case of any other artificial neuron model, so it is not considered as separate step that is needed to be mentioned.

In a case of one neuron there is no principal difference in an order of updating the parameter sets. But in the general case, building the computational

graph, we need firstly update $w_{R_{j_1 j_2}}(\tau)$ and then $w_{L_{j_1 j_2}}(\tau)$.

The general formula for updating the $w_{R_{j_1 j_2}}(\tau)$ can be written in the following way:

$$w_{R_{j_1 j_2}}(\tau) = w_{R_{j_1 j_2}}(\tau - 1) + \eta_{w_R}(k)\delta_{j_1 j_2}^{w_L}(k)X^{w_L\,T}(k), \tag{3.5}$$

where η_{w_R} is a learning rate.

The delta $\delta_{j_1 j_2}^{w_R}$ is computed in the following way:

$$\delta_{j_1 j_2}^{w_L}(\tau) = e_{j_1 j_2}^{w_L}(\tau)\,\psi'\left(u_{j_1 j_2}^{w_L}(\tau)\right). \tag{3.6}$$

The error signal $e_{j_1 j_2}^{w_R}$ in the case of squared error learning criterion is computed in the following way:

$$e_{j_1 j_2}^{w_L}(k) = y_{j_1 j_2}(\tau) - \hat{y}_{j_1 j_2}^{w_L}(\tau) = y_{j_1 j_2}(\tau) - \psi_{j_1 j_2}\left(u_{j_1 j_2}^{w_L}(\tau)(\tau)\right). \tag{3.7}$$

The parameters $w_{L_{j_1 j_2}}(\tau)$ is updated in similar way:

$$w_{L_{j_1 j_2}}(\tau) = w_{L_{j_1 j_2}}(\tau - 1) + \eta_{w_L}(k)\delta_{j_1 j_2}^{w_R}(\tau)X^{w_R\,T}(\tau), \tag{3.8}$$

where η_{w_R} is a learning rate.

$$\delta_{j_1 j_2}^{w_R}(\tau) = e_{j_1 j_2}^{w_R}(\tau)\,\psi'\left(u_{j_1 j_2}^{w_R}(\tau)\right), \tag{3.9}$$

$$e_{j_1 j_2}^{w_R}(\tau) = y_{j_1 j_2}(\tau) - \hat{y}_{j_1 j_2}^{w_R}(\tau) = y_{j_1 j_2}(\tau) - \psi_{j_1 j_2}\left(u_{j_1 j_2}^{w_R}(\tau)(\tau)\right). \tag{3.10}$$

As is known, the activation functions of neural networks must be non-

linear, since they must form non-linear decision boundaries using non-linear combinations of weights and input data. Up to this point, the activation function was denoted by the letter ψ, and its derivative - ψ′, respectively.

In the case of ReLU (formula 2.5), equations (3.5 – 3.10) will be simplified as parameters will be updated only if activation is more than zero. Moreover, as the derivative of ReLU is constant (formula 2.6), even in the case of positive output, updating equations will have a simple form. As described in paper [41], using ReLU as activation for hidden units gives the network universal approximation capabilities.

In order to optimize learning algorithm by speed and protect neuron from exploding gradient problem, the exponentially weighted modification of Kaczmarz-Widrow-Hoff algorithm [42] may be used:

$$w_{L_{j_1 j_2}}(\tau) = w_{L_{j_1 j_2}}(\tau - 1) + r_{w_L}{}^{-1}\delta_{j_1 j_2}^{w_R}(\tau)X^{w_R\,T}(\tau) \tag{3.11}$$

where $r_{w_L}(\tau) = \alpha\, r_{w_L}(\tau - 1) + \|X^{w_R}(\tau)\|^2$;

α is a coefficient in range [0, 1].

$$w_{R_{j_1 j_2}}(\tau) = w_{R_{j_1 j_2}}(\tau - 1) + r_{w_R}{}^{-1}\delta_{j_1 j_2}^{w_L}(k)X^{w_L\,T}(k), \tag{3.12}$$

where $r_{w_R} = \beta\, r_{w_R}(\tau - 1) + \|X^{w_L}(\tau)\|^2$;

β is a coefficient in range [0, 1].

3.3 Single-layer neural network based on matrix neurons and its training

After the introduction of the matrix neuron, its mathematical formulation and learning algorithm, we consider the architecture and procedure for learning a single-layer neural network based on matrix neurons.

In the case of binary classification, a single-layer network based on matrix

neurons is no different from the mathematical formulation of a single neuron described in the previous section. In the simplest case, it is sufficient to have one matrix neuron with a sigmoid activation function at the output.

More interesting from the point of view of further consideration and mathematical formulation is the problem of multiclass classification. In this case, the network with a single layer can be constructed as a vector of matrix neurons. The length of this layer is equal to the number of classes in the dataset. The SoftMax or similar activation function should be applied to the output of such a network to solve the classification task. The SoftMax allows representing the predictions as probabilities of belonging to a particular class.

From a mathematical point of view, transformations of this kind can be written as follows:

$$\hat{y}_{final}(k) = \underset{c=1,\dots,C}{\operatorname{argmax}} \left(softmax \left(\begin{bmatrix} M_{00} \\ M_{00} \\ \vdots \\ M_{0j} \\ \vdots \\ M_{0C} \end{bmatrix} \right) \right), \tag{3.13}$$

where M_{0j} is a final layer matrix neuron;

C is a number of classes.

It should be noted that in this case, we assume that the layer has a vector shape of size (1, C). However, in the general case, a matrix neurons-based layer is two-dimensional. It will be described in detail in the following chapter. Here, all equations correspond to the simpler vector case.

In order to describe the learning procedure for such a system we initially need to consider the following machine learning task: let we have training sample X and suppose that for it the j1-t neuron of single-layer matrix network has output: $\hat{y}_{j_1 0}(\tau)$, target $y_{j_1 0}(\tau)$ with parameters $w_{L_{j_1 0}}(\tau - 1)$, $w_{R_{j_1 0}}(\tau - 1)$, and bias term $\theta_{j_1 0}(\tau - 1)$.

We want to compute how we need to change the values of parameters $w_{L_{j_1 0}}(\tau-1)$, $w_{R_{j_1 0}}(\tau-1)$, and bias term $\theta_{j_1 0}(\tau-1)$ for each unit of the layer given $\hat{y}_{final}$ in order to decrease the final error for the given training sample.

The softmax function is used as activation function:

$$softmax\left(z_{j_1 0}(\tau)\right) = \frac{e^{z_{j_1 0}(\tau)}}{\sum_c e^{z_{c0}(\tau)}}. \tag{3.14}$$

Cross-entropy is used as loss function in the task of multiclass classification:

$$E = -\sum_{c}^{C} y_{c0} \log(\hat{y}_{c0}). \tag{3.15}$$

The equations for updating rules for parameter sets $w_{L_{j_1 0}}(\tau)$ and $w_{R_{j_1 0}}(\tau)$, can be illustrated on the case of one neuron.

We describe the transformation carried out by one neuron of the final layer as follows:

$$\hat{y}_{c0} = softmax\left(w_{L_{j_1 0}}(\tau-1)\cdot X\right)\cdot w_{R_{j_1 0}}(\tau) + \theta_{j_1 0}(\tau-1)). \tag{3.16}$$

For convenience, we now omit the bias term and introduce auxiliary notation:

$$\begin{aligned} x_{b_{j_1 0}} &= \left(x_{w_{L_{j_1 0}}}\right)\cdot w_{R_{j_1 0}}(\tau), \\ x_{a_{j_1 0}} &= w_{L_{j_1 0}}(\tau-1)\cdot X. \end{aligned} \tag{3.17}$$

Taking into account the auxiliary notation (3.17), we can rewrite

equation (3.16) as follows:

$$\hat{y}_{c0} = softmax\left(x_{w_{R_{j_1 0}}}\right). \tag{3.18}$$

Depending on the selected configurations, we will write equations for the adjustment of the neuron parameters in a layer.

The equation to update the vector $w_{R_{j_1 0}}(\tau)$ can be written in the following form:

$$w_{R_{j_1 0}}(\tau) = w_{R_{j_1 0}}(\tau - 1) + w_R(\tau) \cdot \frac{\partial E}{\partial e_{j_1 0}(\tau)} \cdot \frac{\partial e_{j_1 0}(\tau)}{\partial\, \psi(x_{w_{R_{j_1 0}}})} \cdot \frac{\partial\, \psi(x_{w_{R_{j_1 0}}})}{\partial x_{w_{R_{j_1 0}}}} \frac{\partial x_{w_{R_{j_1 0}}}}{\partial w_{R_{j_1 0}}(\tau)}, \tag{3.19}$$

or in a more convenient form:

$$w_{R_{j_1 0}}(\tau) = w_{R_{j_1 0}}(\tau - 1) + \eta_{w_R}(k) \cdot \left(y_{j_1 0}(\tau) - \hat{y}_{j_1 0}(\tau)\right) \cdot \psi'\left(x_{w_{R_{j_1 0}}}\right) \cdot x_{w_{L_{j_1 0}}}. \tag{3.20}$$

The equation to update the vector $w_{L_{j_1 0}}(k)$ can be written in the following form:

$$w_{L_{j_1 0}}(k) = w_{L_{j_1 0}}(\tau - 1) + \eta_{w_L}(\tau) \cdot \frac{\partial E}{\partial e_{j_1 0}(\tau)} \cdot \frac{\partial e_{j_1 0}(\tau)}{\partial\, \psi\left(x_{w_{R_{j_1 0}}}\right)} \cdot \frac{\partial\, \psi\left(x_{w_{R_{j_1 0}}}\right)}{\partial x_{w_{R_{j_1 0}}}} \frac{\partial x_{w_{R_{j_1 0}}}}{\partial x_{w_{L_{j_1 0}}}} \cdot \frac{\partial x_{w_{L_{j_1 0}}}}{\partial w_{L_{j_1 0}}}, \tag{3.21}$$

or in a more convenient form:

$$w_{L_{j_1 0}}(\tau) = w_{L_{j_1 0}}(\tau - 1) + \eta_{w_L}(\tau) \cdot \left(w_{L_{j_1 0}}(\tau) - \hat{y}_{j_1 0}(\tau)\right) \cdot \psi'\left(x_{w_{R_{j_1 0}}}\right) \cdot x_{w_{R_{j_1 0}}}. \tag{3.22}$$

The modifications of the traditional learning rule proposed in the previous section can be easily applied to the case of a single-layer network.

It is easy to see that when using the cross-entropy criterion with the softmax

activation function, the equations for weight adjustment do not differ at all from the equations considered for the case with one neuron.

For the bias term $\theta_{j_1 0}(\tau)$, the adjustment will be performed using the following equation:

$$\theta_{j_1 0}(\tau) = \theta_{j_1 0}(\tau - 1) + \eta_{w_L}(\tau) \cdot \left(y_{j_1 0}(\tau) - \hat{y}_{j_1 0}(\tau)\right) \cdot \psi'\left(x_{w_{R_{j_1 0}}}\right). \quad (3.23)$$

3.4 Multilayer neural network based on M-neurons

The most common and complex case, both for mathematical formulation and for software implementation, is a multilayer neural network based on M-neurons.

A feature of multilayer networks compared to single-layer, which was discussed in the previous section, is that each of the inner layers does not have direct access to the error signal. So, to train such a network, we need to reconstruct the error of each inner layer based on the errors of the next layers connected with them. This approach is called error backpropagation. And the purpose of this section is to describe the procedure of backpropagation for a multilayer neural network based on matrix neurons.

As it is easy to determine, the weights of the neurons of the last layer for multiclass classification will be updated according to formulas (3.19 – 3.23). More interesting is the description of the procedure for adjusting the weights of the inner layers. Their features are the following:

– firstly, it is assumed that the outputs of the inner layers remain matrices, just reduced dimensionality;

– secondly, the neurons of the inner layers do not have direct access to the error signal.

In this case, an error backpropagation algorithm is applied. The idea is the same as for any other deep neural network.

Again, as we have two sets of parameters, we need to first update $w_{R_{j_1 j_2}}(\tau)$ and then $w_{L_{j_1 j_2}}(\tau)$. And bias term is updated independently.

Mathematicaly, having L layers, we have the following updating rules for each neuron in the layer:

– for the last layer (L-th):

$$w^{[L]}_{L_{j_1 j_2 i_1}}(\tau) = w^{[L]}_{L_{j_1 j_2}}(\tau - 1) + r_{w_L}{}^{-1}\psi'\left(u^{[L]}_{j_1 j_2}(\tau)\right) o^{[L-1]w_R}_{i_1}(\tau), \quad (3.24)$$

where $o^{[L-1]w_R}_{i_1}$ is an output signal of (L-1)-th layer;

$i_1 = 0,1,\dots,n_1^{[L]}$.

$$w^{[R]}_{L_{j_1 j_2 i_2}}(\tau) = w^{[R]}_{L_{j_1 j_2}}(\tau - 1) + r_{w_R}{}^{-1}\psi'\left(u^{[L]\,w_L}_{j_1 j_2}(\tau)\right) o^{[L-1]w_L}_{i_2}(\tau), \quad (3.25)$$

where $o^{[L-1]w_L}_{i_2}$ is an output signal of (L-1)-th layer;

$i_2 = 0,1,\dots,n_2^{[L]}$.

$$\theta^{[L]}_{j_1 j_2}(\tau) = \theta^{[L]}_{j_1 j_2}(\tau - 1) + r_{w_R}{}^{-1}\psi'\left(u^{[L]\,w_L}_{j_1 j_2}(\tau)\right). \quad (3.26)$$

– for the hidden layers, $l = 1,\dots,L-1$:

$$w^{[l]}_{L_{j_1 j_2 i_1}}(\tau) = w^{[l]}_{L_{j_1 j_2}}(\tau - 1) + r_{w_L}{}^{-1}\delta^{[L]}_{j_1 j_2}(\tau) o^{[l-1]w_R}_{i_1}(\tau), \quad (3.27)$$

where

$$\delta_{j_1 j_2}^{[l]}(\tau) = \psi'\left(u_{j_1 j_2}^{[l]}(\tau)\right) \sum_{i_1=0}^{n_1} \delta_{j_1 j_2}^{[l+1]}(\tau) w_{L_{j_1 j_2 i_1}}^{[l+1]}(\tau)\,. \tag{3.28}$$

$$w_{L_{j_1 j_2 i_2}}^{[l]}(\tau) = w_{L_{j_1 j_2}}^{[l]}(\tau - 1) + {r_{w_R}}^{-1} \delta_{j_1 j_2}^{[L]\,w_L}(\tau) o_{i_2}^{[l-1]w_L}, \tag{3.29}$$

where

$$\delta_{j_1 j_2}^{[l]}(\tau) = \psi'\left(u_{j_1 j_2}^{[l]w_L}(\tau)\right) \sum_{i_1=0}^{n_1} \delta_{j_1 j_2}^{[l+1]w_L}(\tau) w_{L_{j_1 j_2 i_2}}^{[l+1]}(\tau)\,. \tag{3.30}$$

$$\theta_{j_1 j_2}^{[l]}(\tau) = \theta_{j_1 j_2}^{[l]}(\tau - 1) + {r_{w_R}}^{-1} \psi'\left(u_{j_1 j_2}^{[l]}(\tau)\right) \sum \psi'\left(u_{j_1 j_2}^{[l+1]}(\tau)\right). \tag{3.31}$$

It is easy to see that training procedures for multi-layer matrix networks are not very different from the training process of a conventional multi-layer perceptron. The only principal difference is that the tuning procedure is two-step. Another important detail is that such a network contains individual neurons that process input independently, which could be a potential room for further experiments and improvements.

4 EXPERIMENTAL RESEARCH

4.1 Problem statement

In the experimental part of this master thesis, the matrix neurons and deep neural networks based on them software implementation, training, and evaluation are described. The purpose of the research is to perform image classification with the methods introduced in the theoretical part and evaluate the effectiveness of these methods.

The experiments covered in this section could be divided into three main blocks, according to the subject of the experiment:

– matrix neuron, its training and evaluation,
– single-layer matrix network, its training and evaluation,
– deep matrix neural network, its training and evaluation.

For each experiment, the setup, including:

– data preprocessing,
– software implementation of the architecture and learning procedure,
– model training with different configurations,
– model/models performance evaluation,
– analysis of results,

is described. The results will be listed for each block separately. In the end, there will be a brief summing up and conclusion.

It should be mentioned that the matrix neuron and the networks based on it are capable of processing only monochrome images but can potentially be used to process color images.

Furthermore, interactive demo app was prepared to demonstrate introduced approach in action.

4.2 Implementation tools

For software implementation, the Python 3 programming language was used. Python is an open-source high-level dynamic programming language. Python advanced machine learning ecosystem and relative ease of coding were the motivation for using it as a language for this study.

The following libraries were used for software implementation:

– Sklearn is an open-source machine learning library in python. It contains implementation of most popular machine learning algorithms as well as a set of useful data preprocessing methods, e.g. data scaling, splitting data into train and test sets, etc.

– PyTorch is an open-source deep learning framework created by the Facebook AI Research team. Its goal is to simplify the construction and training of complex neural networks. PyTorch is a highly customizable framework, so it is a great option for this research, as it allows you to easily create custom blocks, layers, models, learning algorithms, etc. What is important for this research is the concept of a neural network block, which in PyTorch can correspond to a neuron as well as a layer or model. In Tensorflow2, another great deep learning framework, the smallest block is the layer, which is not suitable for our study. Moreover, defining custom blocks in PyTorch is clearer than in Tensorflow.

– matplotlib is an open-source Python library for data visualization that simplifies creation of 2D graphs from data stored in different data structures, including Python lists, Numpy array, Pandas dataframes, etc.

– Streamlit is an open-source python framework for building web apps for Machine Learning and Data Science. It allows quick prototyping of interactive web iterfaces for demonstating results.

– NumPy is a core scientific computing package in Python that allows effectively performing vectorized operations. The main object in the NumPy is a multidimensional array. And the package implements a wide set of operations that can be effectively performed with these arrays. This library allows to quickly

build and test prototype of matrix neuron and its learning procedure. Built-in operation vectorization allows to effectively perform operations on vectors and matrices while keeping code readability and avoiding unnecessary looping.

4.3 Software implementation and experiments with matrix neuron

4.3.1 Matrix neuron implementation

The main building block introduced within the framework of this qualification work is a matrix neuron. For experiments and evaluation a performance of proposed matrix neuron, it was implemented in software in the form of Python class. This class implements the following functionality:

– matrix neuron __init()__ method that allows to initialize weights $w_{L_{j_1 j_2}}, w_{R_{j_1 j_2}}$ and specify activation function type;

– methods that implementfeedforward propagation steps for weights $w_{L_{j_1 j_2}}$ and $w_{R_{j_1 j_2}}$, correspondingly;

– methods that implementbackward propagation steps for weights $w_{L_{j_1 j_2}}$ and $w_{R_{j_1 j_2}}$, correspondingly;

– methods that implementtraining step for weights $w_{L_{j_1 j_2}}$ and $w_{R_{j_1 j_2}}$, correspondingly;

– method that implements online two-phase training procedure for matrix neuron;

– method that implements test and evaluation of trained matrix neuron on the test set.

4.3.2 Experiments with matrix neuron

The matrix neuron is a central element proposed in this work. It was proposed primary as a building block of a neural network but can be used as a

separate model.

For experiments with matrix neuron and evaluate its performance, binary image classification task was considered. The dataset used in experiments with matrix neuron is a subset of handwritten digits dataset from UCI repository [25]. This dataset contains 5620 grayscale images of digits from 0 to 9, 8x8 pixels in size (figure 4.1).

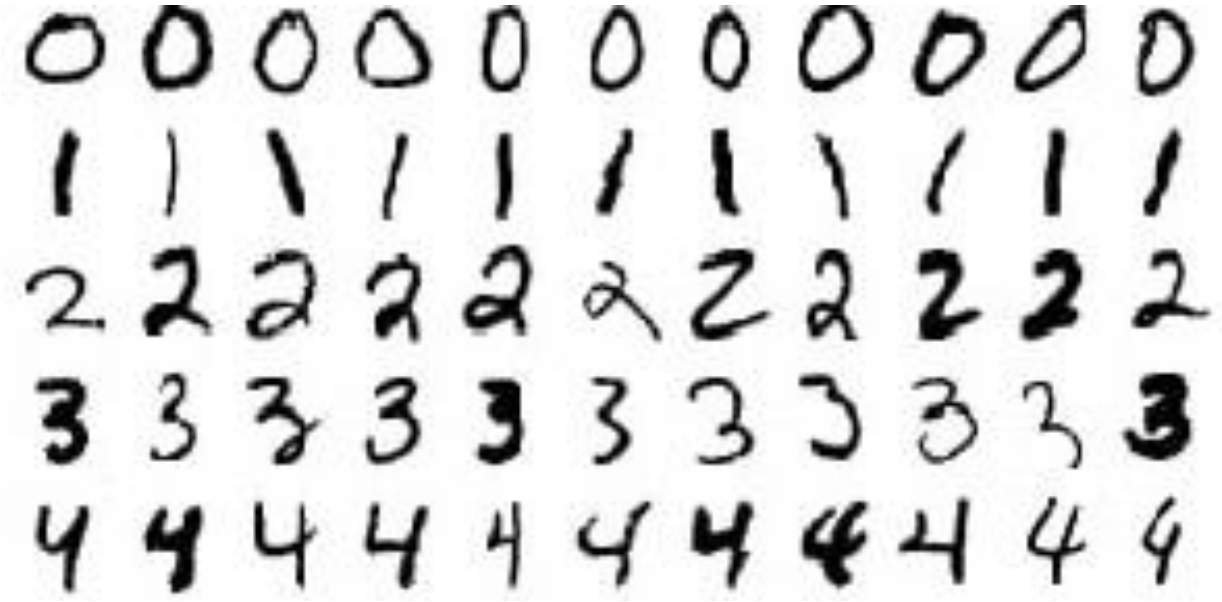

Figure 4.1 – Example of images from a set of handwritten numbers

Before feeding dataset to the input of matrix neuron, we preprocessed it following the next sequence of steps:

– normalize the values in a dataset on interval [0, 1];

– select a set of data subsets for binary image classification;

– randomly split dataset into train and test.

After data preprocessing, we initialized matrix neuron parameters using random weights initialization approach and feed training data to the matrix neuron. Through all experiments with matrix neuron we use online learning approach, so it “saw” each image only once.

To explore the effect of visual similarities between pairs of digits (e.g., 6

and 9), we conducted experiments with different pairs of digits (figure 4.2). The results of the experiments are in table 4.1.

Table 4.1 – The results of the experiments with matrix neuron

Digits classes	Classification accuracy, %
0 | 1	53,5
6 | 9	50
2 | 3	48
0 | 7	57

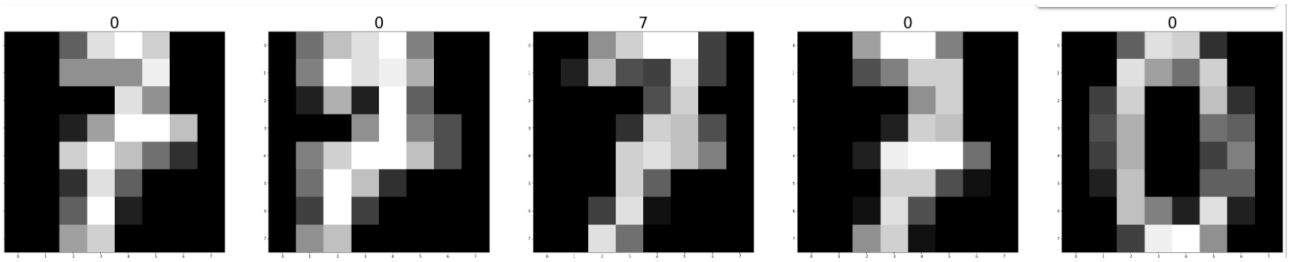

Figure 4.2 – The results of experiments with the matrix neuron on the tasks of binary images classification (digits classes: 0 and 7)

As expected, one neuron is not enough to approximate the mapping of a 8×8 matrix into a vector of size 1. But at the same time, the results show that in the case when classes are different enough, the neuron begins to learn to distinguish between these classes.

4.4 Software implementation and experiments with single-layer network based on matrix neurons

4.4.1 Single-layer neural network based on matrix neurons implementation

For experiments with single-layer network based on matrix neurons we

implemented it in the form of Python class that implements the following two methods:

– method that implements online two-phase training procedure for single-layer matrix network,

– method that implements test and evaluation of trained single-layer matrix network on the test set.

4.4.2 Experiments with single-layer network based on matrix neurons

As noted above, a single neuron is not enough to approximate a function even for binary digit image classification. Hence we combined several matrix neurons into a vector to perform image classification. Since several neurons can approximate more complex functions, we performed experiments with a different number of classes (from 2 to 10).

The experiment settings are:

– training set is a subset of preprocessed images from the dataset [25],

– training regime is online learnings,

– tasks are both binary classification and multiclass classification.

For the binary classification, we used visually similar and dissimilar pairs of digits to estimate the impact of visual similarity between digits classes. The results of the experiments are in table 4.2.

Table 4.2 – The results of experiments with single-layer network based on matrix neurons in the task of binary image classification

Digits classes	Classification accuracy, %
0 | 1	99.9
6 | 9	87.8
2 | 3	89.5
0 | 7	98.5

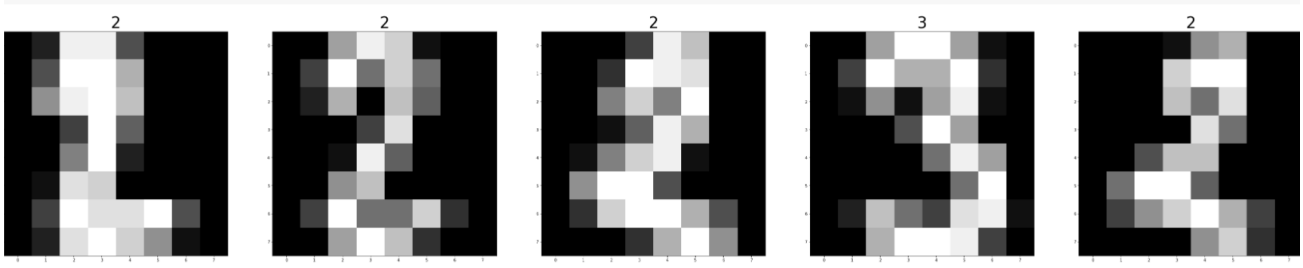

Figure 4.3 – The results of binary classification of images of digits 2 and 3 using a single-layer network with two matrix neurons and softmax function at the output

As can be seen from the result, the network is quite good at distinguishing numbers in the problem of binary classification (figure 4.3). In order to assess how the number of classes affects the accuracy of the model, experiments were conducted with different numbers of classes. The results are shown in table 4.3.

Table 4.3 – The results of experimental studies of the influence of the number of classes on the performance of a single-layer network based on matrix neurons

Number of classes	Classification accuracy, %
2	93.25
3	97.40
4	93.50
5	92.20
6	79.60
7	85.00
8	78.70
9	75.94
10	57.90

As shown in table 4.3, with increasing the number of classes in

classification task, the performance of single-layer network is dropping. It is because the complexity of task is increasing, and one layer becomes insufficient to generalize data well enough. It should be mentioned that in this experiment the complexity of classes was not taken in consideration. The influence of various hyperparameters, such as the activation function, optimization algorithm, etc., was not taken into account.

The main goal of the series of experiments described in subsections 4.3 and 4.4 was to show the general tendency and prove the efficiency of the proposed approach.

4.5 Software implementation and experiments with multi-layer network based on matrix neurons

This subsection is devoted to multi-layer networks based on matrix neurons. Multi-layer network is the most general and complex case that have which is of significant practical importance.

While the models in the previous subsections were all implemented in NumPy, the code for this subsection is written using the PyTorch deep learning framework. However, this approach is not scalable because, as the complexity of the network grows, it becomes difficult both to write efficient error backpropagation algorithms and to train complex networks on the CPU. Deep learning frameworks address these problems with auto-differentiation, an effective way to compute gradients, and software compatibility with graphics processing units (GPUs). It allows you to focus on tuning and optimizing your model without having to build everything from scratch.

4.5.1 Implementation of model using Tensorflow2

Initially, the Tensorflow2 framework was selected as a tool for multi-layer network implementation. The choice of the framework was primarily determined

by the author's personal experience.

In order to implement network, the custom layer, definition of which is given in listing 4.1, was implemented as a subclass of TensorFlow layer.

Listing 4.1 – Program code for matrix layer definition in Tensorflow

```
class MatrixLayer(Layer):
    def __init__(self, rs=1, cs=1, act=None):
        super(MatrixLayer, self).__init__()
        self.rs = rs
        self.cs = cs
        self.act = activations.get(act)

    def _matrix_transform(self, ins, wr, wl, b):
      return self.act(matmul(matmul(wr,ins),wl) +  b)

    def build(self, in_s):
        self.b=Variable(b_init((self.rs,self.cs,1)),
trainable=True)
        self.r=Variable(wr_init((self.rs,   self.cs,   1,
in_s[1])), trainable=True)
        self.l=Variable(w_left_init((self.rs,self.cs,
in_s[-1], 1)), trainable=True)
        super().build(in_s)

    def call(self, ins):
        res = []
        for i in range(self.rs):
          res_w = []
          for j in range(self.cs):
               res_w.append((self._matrix_transform(ins,
               self.r[i][j], self.l[i][j], self.b[i][j])))
          res.append(res_w)
        return res
```

After creation of custom layer, it can be used to build a model and train it using TensorFlow's auto grad solution. The example of model definition is shown in listing 4.2.

Listing 4.2 – Program code for model definition in Tensorflow

```
input_shape = Input(shape=(8, 8))
x_1 = MatrixLayer(8, 8,'relu')(input_shape)
x_2 = MatrixLayer(4, 4, 'relu')(x_1)
x_3 = MatrixLayer(10, 1,'linear')(x_2)
model = Model(input_shape, x_3)
```

The next step is to compile the model and start training. In Tensorflow, there are methods for train model, but in this case the custom training loop was developed. The code for training is given in listing 4.3.

Listing 4.3 – Custom training loop for our model in Tensorflow

```
    for      step,      (x_train_ex,      y_train_ex)      in
enumerate(zip(x_train, y_train_one_hot)):
            with tf.GradientTape() as tape:
                logits=model(x_train_ex, 0), training=True)
                loss=softmax_ce_with_logits(y_train_ex,logits)
            grads=tape.gradient(loss, model.trainable_weight)
            opt.apply_gradients(zip(grads,
    model.trainable_weights))
```

For testing purposes, the MNIST dataset from scikit-learn library was used.

After first iteration, it was noticed that loss was not decreasing. Debugging showed that there are some unconnected gradients in the model that resulted in error was not backpropagated to earlier model. The analysis of possible causes showed that because of non-trivial matrix structure of layer, when we need to stack neurons in the form of 2D array, we need a special trainable data structure supported by TensorFlow auto grad. But at the time of the experiments, such solution was not implemented in TensorFlow. To keep things from getting too complicated and focus on experiments, it has been decided to switch to other deep learning framework – PyTorch – which is more customizable.

4.5.2 Implementation of model using PyTorch

In PyTorch, the building block of the model is nn.Module. It allows not only to combine layers to create models but to define layer elements as separate blocks. The concept of the module proved beneficial for implementing the network based on M-neurons proposed in this work. As each layer consists of independent neurons which perform individual operations on inputs, the possibility to define separate neurons and then stack them into a 2D array made

the overall architecture definition clearer.

The first step is to implement matrix neuron defined in equation (3.2). The matrix neuron has two parameters set and bias term. The code for defining matrix neuron is given in listing 4.4.

Listing 4.4 – Code for defining matrix neuron in PyTorch

```
class MNeuron(nn.Module):
  def __init__(self, input_shape):
    super().__init__()
    self.input_shape = input_shape
    ws_l = torch.Tensor(1, input_shape)
    self.ws_l = nn.Parameter(ws_r, requires_grad=True)
    ws_r = torch.Tensor(input_shape, 1)
    self.ws_r = nn.Parameter(ws_r, requires_grad=True)
    b = torch.Tensor(1)
    self.b = nn.Parameter(b, requires_grad=True)

    nn.init.normal_(self.ws_l, mean=0, std=1)
    nn.init.normal_(self.ws_r, mean=0, std=1)
    nn.init.normal_(self.b, mean=0, std=1)

  def forward(self, x):
    w_l_x_w_r = torch.mm(torch.mm(self.ws_l, x), self.ws_r)
    return torch.add(w_l_x_w_r, self.b)
```

As usual, we defined parameters of the neuron and forward pass. Here, the hyperparameters we can experiment with include weights initialization. We do not apply activation function here as it will be a separate layer.

The next step is to define matrix layer. Unlike traditional feed-forward network layers, we need to stack independent neurons in grid-like form. The code for defining matrix neuron is given in listing 4.5.

Listing 4.5 – Code for defining matrix layer in PyTorch

```
class MLayer(nn.Module):
  def __init__(self, in_x, in_y, out_x, out_y):
    super().__init__()
    self.in_x = in_x
    self.in_y = in_y
    self.out_x = out_x
    self.out_y = out_y
    self.mneurons = nn.ModuleList(
```

```
        [MNeuron(self.in_x) for i in range(out_x * out_y)])
        self.out = nn.Unflatten(2, (self.out_x, self.out_y))

    def forward(self, x):
        outputs = [l(x) for l in self.mneurons]
        out_stack = torch.stack(outputs, dim=-1)
        out = self.out(out_stack)
        return torch.squeeze(out)
```

There are several key moments in the implementation of matrix layer that worth to be noted:

– we used ModuleList to stack neurons. ModuleList is one of the modules containers implemented in PyTorch. It holds modules (in our case, matrix neurons) in a list. The difference between ModuleList and conventional Python List is that Pytorch is “aware” that there are modules inside ModuleList, which is not true for Python List. If using a conventional List while training the model, PyTorch will not be able to propagate errors to update the parameters of modules in a list because the optimizer will not find parameters for the layers stored in it. It will cause an error. In the case of ModuleList, there will not be any error;

– PyTorch modules containers support only flat structure, but neurons in a matrix layer form a grid-like structure. In order to address this issue, we first create a flat ModuleList. It is possible because each neuron processes inputs independently. After performing all operations by neurons, their input is transformed into a matrix and passed to the next layer. It is an additional operation that slightly increases training time. But this increase in time is not critical for the current research.

The layer has configurable number of neurons.

The last module needed to be defined is a model. Model consist of the matrix layers. It is possible to define:

– number of layers;

– number of neurons in a layer and their spatial configuration;

– activation function types for hidden and final layer.

All mentioned hyperparameters are the subject of the experiment.

An example of one of the possible definitions of a multilayer neural network model based on matrix neurons is shown in Listing 4.6.

Listing 4.6 – Code for defining multi-layer nerwork based on M-neurons in PyTorch (example of model)

```
class MyModel(nn.Module):
  def __init__(self, in_feats, nb_classes,
      hidden_size, act=nn.LeakyReLU):
      super(MyModel, self).__init__()
      self.act = act()
      self.final_act = nn.Softmax()

      self.ml1 = MLayer(in_feats, in_feats, hidden_s,
hidden_s)
      self.ml2 = MLayer(hidden_s, hidden_s, hidden_s-2,
hidden_s-2)
      self.out = MLayer(hidden_s-2, hidden_s-2, nb_classes,
1)

  def forward(self, x):
      x = x.squeeze()
      x = self.act(self.ml1(x))
      x = self.act(self.ml2(x))
      x = self.out(x)
      return x.unsqueeze(0)
```

After defining a model, its parameters can be adjusted using the PyTorch training loop. PyTorch training loop is highly configurable that giving a huge space for experiments with different hyperparameters, including optimization algorithm, loss function, etc.

The following paragraphs are devoted to experiments with a multi-layer network based on matrix neurons and its parameters and hyperparameters.

4.5.3 MNIST digits classification using multi-layer network based on matrix neurons

The goal of this experiment is to build MNIST digits classificator based on matrix neurons. The MNIST dataset used for experiments in paragraph 4.5.3 has 28×28 greyscale images.

Firstly, use ‘classical’ settings for building simple image classificator:

– dataset: MNIST;

– training regime: online;

– hidden layers activations: ReLU;

– final layer activation: SoftMax;

– learning criterion: cross-entropy;

– optimizer: stochastic gradient descent with momentum;

– number of hidden layers: 2.

Important things to note about SGD is that using it can cause an exploding gradient problem. The gradient clipping technique was used to avoid this.

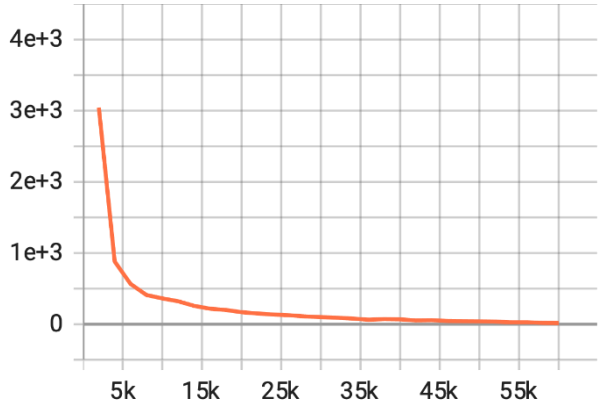

Figure 4.4 – Graph of loss function vs. iterations during training with SGD

This experiment showed not so good results with accuracy on the test set is around 20 % (figure 4.4).

In the following experiments, Adam was selected as an optimizer as it showed good performance in many cases.

Experiment 2 was conducted with the following setting:

– dataset: MNIST;

– training regime: online;

– final layer activation: SoftMax;

– learning criterion: cross-entropy;

– optimizer: Adam;

– number of hidden layers: 2;

– hidden layers activations: varying.

The goal of this experiment is to explore the influence of activation function choice on the performance of simple matrix neural net.

As it was described in the chapter 3, ReLU and its modifications are the most popular choice of activation function in the modern neural networks. But the choice of activation function is always an experimental step, as it depends on many parameters of particular task. The figures 4.5 – 4.10 show how the loss functions for the experiment change depending on the activation.

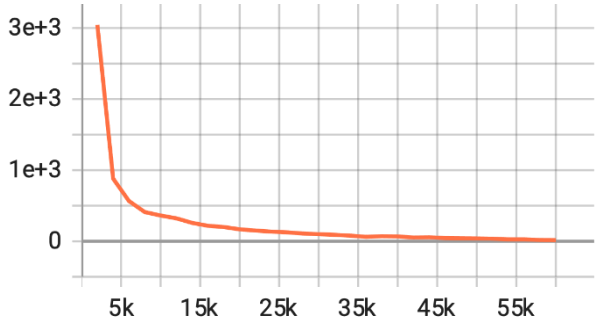

Figure 4.5 – Graph of loss function vs. iterations during training with Adam and using ReLU as hidden layers activation function

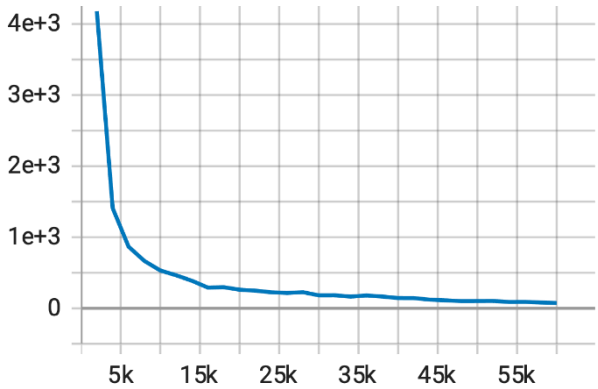

Figure 4.6 – Graph of loss function vs. iterations during training with Adam and using LeakyReLU as hidden layers activation function

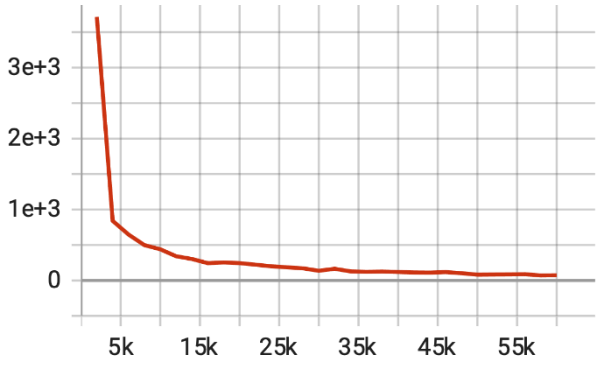

Figure 4.7 – Graph of loss function vs. iterations during training with Adam and using PReLU as hidden layers activation function

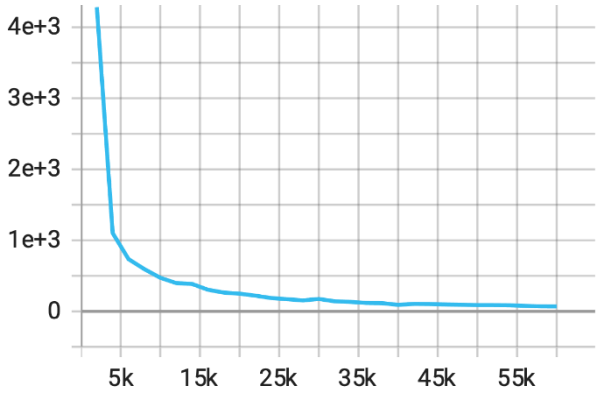

Figure 4.8 – Graph of loss function vs. iterations during training with Adam and using ELU as hidden layers activation function

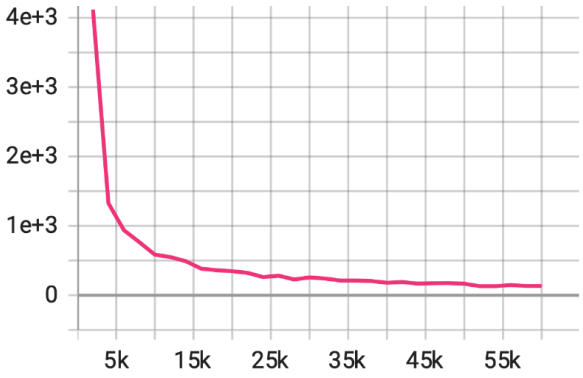

Figure 4.9 – Graph of loss function vs. iterations during training with Adam and using SELU as hidden layers activation function

Results in table 4.4 showed that a matrix network with only two hidden layers was efficient enough to solve the multilabel image classification task with an accuracy of 85 %.

Table 4.4 – The results of evaluation of the performance of the matrix neural network depending on the choice of the activation function of the hidden layer

Activation function	Training loss	Test accuracy, %
ReLU	16.32	85.9
LeakyReLU	74.62	80.8
PReLU	73.36	85.7
ELU	68.92	84.4
SELU	133.8	83.4

4.5.4 CIFAR10 images classification using multi-layer network based on matrix neurons

MNIST dataset contains relatively simple objects. In order to explore how matrix network perform on more complex objects the Cifar10 dataset was choosen for the next experiment. The graph of loss function is illustrated in figure 4.10.

The experiment settings is the following:

– dataset: Cifar10;
– training regime: online;
– final layer activation: SoftMax;
– learning criterion: cross-entropy;
– optimizer: Adam;
– number of hidden layers: 4;
– hidden layers activations: ReLU.

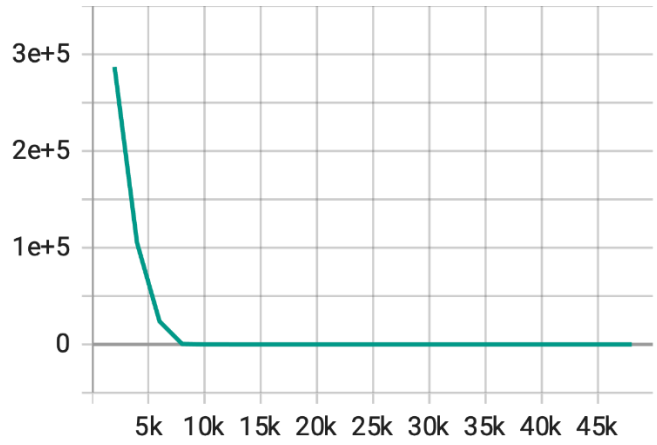

Figure 4.10 – Graph of loss function vs. iterations during training with Adam and using ReLU as hidden layers activation function in task of CIFAR-10 image classification

Despite the decreasing trend in training loss, in the evaluation step model outputed only one class. There are many possible reasons of this behaviour. Possible solutions include:

– decrease learning rate;

– change activation function from ReLU to some other modification of it as using ReLU may result in dying gradient;

– create deeper network as objects in CIFAR10 are more complex comparing to objects from MNIST;

Tuning aforementioned hyperparameters did not result in improving performance of used in this experiment matrix neural network.

The images in CIFAR10 are of a complex nature and to build image classification algorithm that performs well enough on this data we need deeper neural networks and powerful feature extraction approach. The leaderboard [43] shows that good results in CIFAR10 image classification are obtained using big models, often with feature extractor with convolutional layers.

The goal of the first experiment is to build a simple neural network with several convolutional layers as a feature extraction tool and several matrix layers as a classifier. The idea is to repeat simple CNN architecture with matrix layers

instead of dense layers in the classification head. In these settings, we train the whole network from scratch. The results of this experiment showed that with these settings, we did not get any noticeable improvements other than a better learning speed. It is very likely that it is not enough to train the feature extraction part that consists of convolutional layers in online settings, as it usually requires hundreds of epochs to train CNN.

An extension of the previous experiment is as follows: given the previously trained backbone, add several matrix layers in the head to perform classification. In this case, the overall model performance largely depends on the performance of the previously trained feature extractor. But using a previously trained feature extractor seems like a powerful preprocessing step for our network. By analogy with feed-forward networks, it is difficult to build an image processing system on raw images since important features are often not obvious for networks without the conductive biases of CNNs. The Resnet50 backbone trained on CIFAR10 dataset was used as a feature extractor in this experiment [44].

The graphs of training and validation loss for this experiment are shown in figures 4.11 and 4.12.

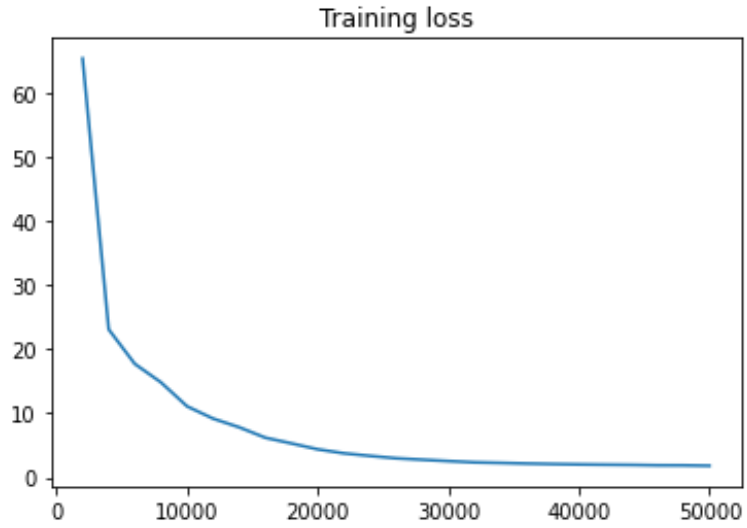

Figure 4.11 – Graph of training loss function vs. iterations during training in task of CIFAR-10 image classification

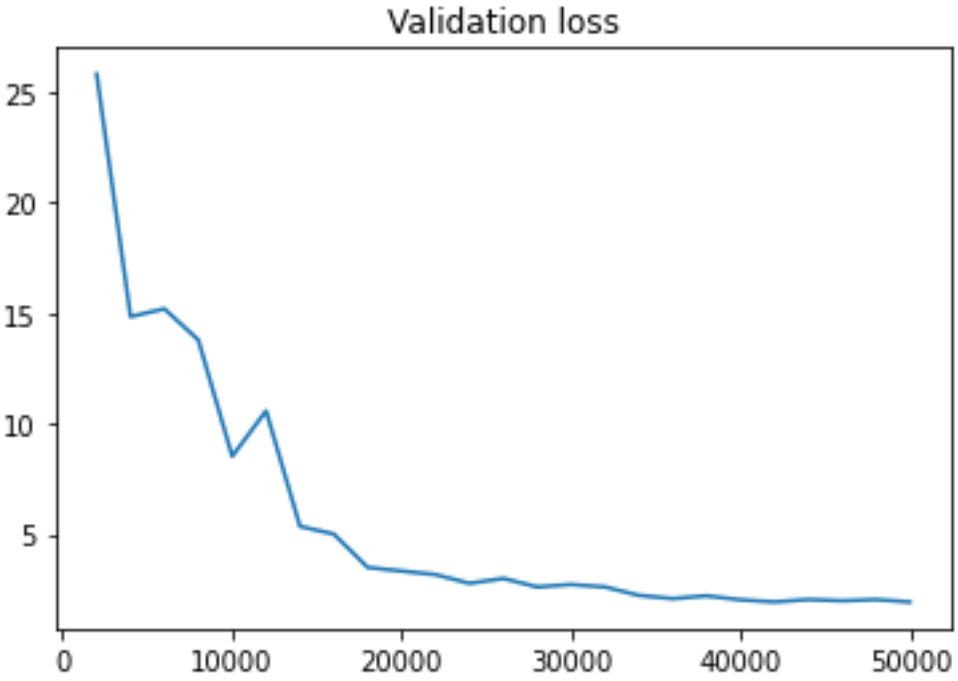

Figure 4.12 – Graph of validation loss function vs. iterations during training in task of CIFAR-10 image classification

As computational resources available for these experiments are very limited, it is not possible to train very large models, even if it is matrix networks. By comparing the number of operations, matrix networks require far fewer computations to achieve results compared with dense layers. But as matrix layers are not supported now by used deep learning frameworks, they are not optimized well on low-level, unlike common layers such as dense and convolutional layers.

Compared to the approaches used in previous experiments with the CIFAR10 dataset, the use of previously trained feature extractors showed better performance. In previous experiments, the proposed matrix network could not learn to display CIFAR10 images. The results of the latest experiment show that the network has begun to find useful patterns.

The final accuracy of the model reached 30% on multi-class classification. The number of classes is 10, so results are not just random guessing. But there is still a room for further improvements. Some results of classification are shown in the figure 4.13.

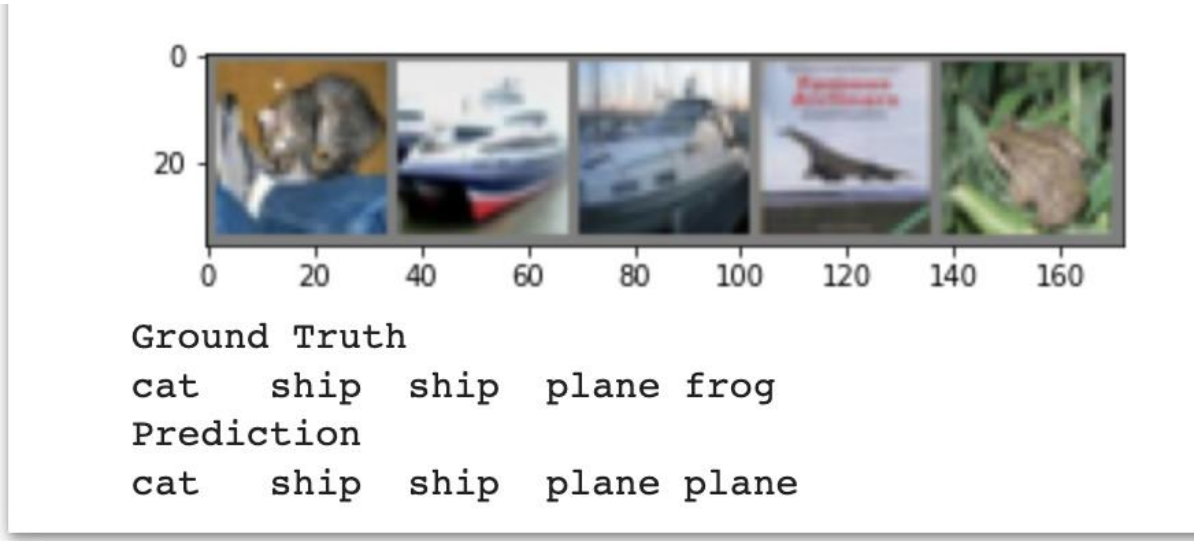

Figure 4.13 – Example of predictions on CIFAR10 dataset

The classification report with scores calculated during model evaluation is shown in figure 4.14. It shows how well a model with a classification head consisting of matrix layers performs on different classes and in general.

	precision	recall	f1-score	support
plane	0.46	0.28	0.35	1000
car	0.49	0.44	0.47	1000
bird	0.23	0.04	0.07	1000
cat	0.20	0.29	0.24	1000
deer	0.25	0.39	0.30	1000
dog	0.20	0.45	0.28	1000
frog	0.40	0.02	0.04	1000
horse	0.46	0.40	0.43	1000
ship	0.55	0.37	0.44	1000
truck	0.34	0.52	0.41	1000
accuracy			0.32	10000
macro avg	0.36	0.32	0.30	10000
weighted avg	0.36	0.32	0.30	10000

Figure 4.14 – Classification report

4.6 Demo web-application

In order to demonstrate the results obtained during experiments with matrix networks, the demo web application was implemented. The Streamlit library was used for it. The demo application consists of two modules:

– the first module is a demo of a network trained on the CIFAR10 dataset. It allows uploading images of the size and returns predictions for them. As the network was trained on images from the CIFAR10 dataset, it is possible to load only images in such resolution and images that belong to one of ten classes presented in the used dataset;

– the second module is a quick theoretical overview of the proposed matrix neuron. It contains the graph of matrix neuron architecture and its brief introduction.

The illustration of demo web application functionality is shown in figures 4.15, 4.16, and 4.17.

Figure 4.15 – Screenshot 1 of Demo web-app

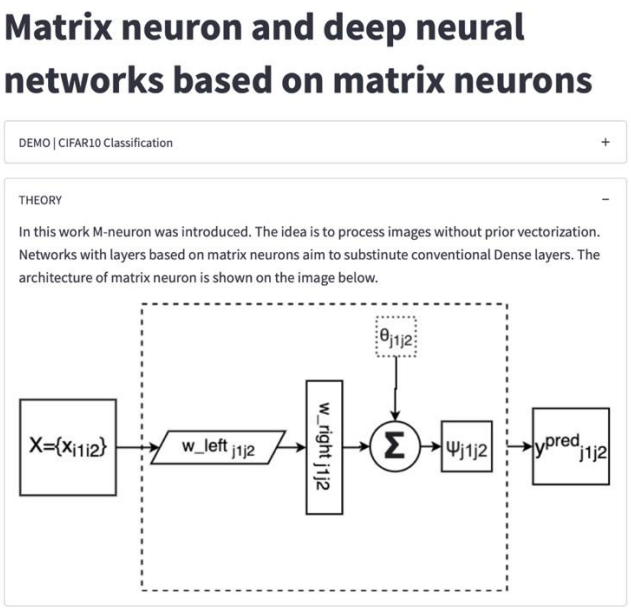

Figure 4.16 – Screenshot 2 of Demo web-app

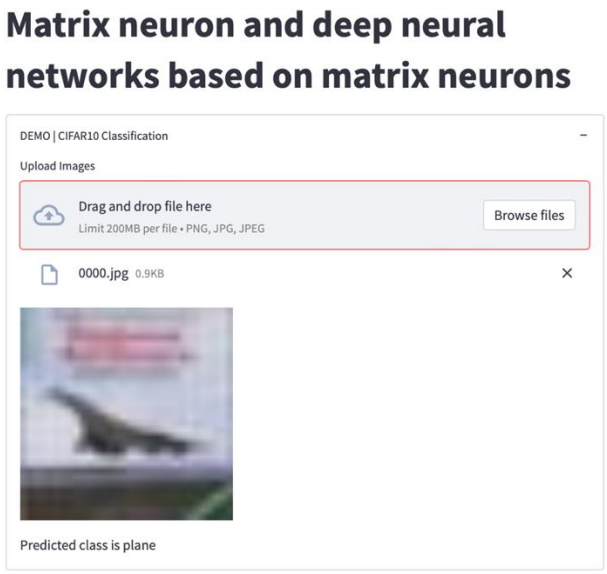

Figure 4.17 – Example of prediction in demo app

This simple web-app allows to get predictions with convolutional neural network that has matrix layers in had. Such configuration gave the best results on images more complex than digits and have potential for further research and development.

CONCLUSION

Within the framework of this research work the matrix neuron and network based on it was proposed and explored. Proposed approach is further work on the developing matrix neural networks that are able to process matrix inputs without previous vectorization.

The results obtained in bachelor's work proved that matrix neural networks are the perspective direction in online deep learning. But at the same time previously explored matrix neural networks build on monolithic matrix layers were not effective enough to solve the objectives. To address the problem of monolithic architecture that complicates training of matrix networks and very likely negatively affects its performance, the new type of neuron called matrix neuron was introduced.

M-neurons allow building matrix layers and networks in a more flexible way. In this paper M-neuron architecture and training procedure were formulated. M-neuron served as a building block of more complex structures, such as matrix layers and networks. The mathematical formulation of which was described in detail in theoretical chapter. The image classification task was selected as an application area for this research, but it can be potentially extended to any other machine learning task.

Given the mathematical formulation, the proposed approach was implemented in software. The experimental research included study of both the matrix neuron and networks based on it. The influence of some hyperparameters, including activation functions, on the performance of the proposed neural network was studied.

The results of experimental research showed that:

– using model based on matrix neurons trained in online settings allows to achieve accuracy comparable with simple traditional CNNs trained in batch settings;

– the experiments showed that training such networks can take a long time,

but it is mainly because deep learning frameworks used in the work does not support some operations that used in proposed models. In particular, in order to make it possible to stack neurons in a layer it was needed to introduce additional vectorize-devectorize operations;

– in image processing, the high-quality feature extraction is an important step. As matrix networks do not have inductive biases of CNNs, we cannot just replace whole CNN with a Matrix network. More promising is the combination of CNN feature layer layers with classification matrix layers;

– as in any other neural network architecture, the hyperparameters search step affects the performance of neural networks based on matrix layers.

REFERENCES

1. IBM Watson. *IBM – Deutschland | IBM*. URL: https://www.ibm.com/watson (date of access: 18.03.2022).

2. AI unleashes the power of unstructured data. *CIO*. URL: https://www.cio.com/article/220347/ai-unleashes-the-power-of-unstructured-data.html (date of access: 18.03.2022).

3. The 2 types of data strategies every company needs. *Harvard Business Review*. URL: https://hbr.org/2017/05/whats-your-data-strategy (date of access: 18.03.2022).

4. Challenges in benchmarking stream learning algorithms with real-world data / V. M. A. Souza et al. *Data Mining and Knowledge Discovery*. 2020. Vol. 34, no. 6. P. 1805–1858. URL: https://doi.org/10.1007/s10618-020-00698-5 (date of access: 18.04.2022).

5. Gradient-based learning applied to document recognition / Y. Lecun et al. *Proceedings of the IEEE*. 1998. Vol. 86, no. 11. P. 2278–2324. URL: https://doi.org/10.1109/5.726791 (date of access: 18.04.2022).

6. Simonyan K., Zisserman A. Very deep convolutional networks for large-scale image recognition. 2014. (Preprint. arXiv:1409.1556).

7. Going deeper with convolutions / C. Szegedy et al. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA, 7–12 June 2015. 2015. URL: https://doi.org/10.1109/cvpr.2015.7298594 (date of access: 18.04.2022).

8. Deep residual learning for image recognition / K. He et al. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 27–30 June 2016. 2016. URL: https://doi.org/10.1109/cvpr.2016.90 (date of access: 18.04.2022).

9. An image is worth 16x16 words: transformers for image recognition at scale / A. Dosovitskiy et al. *arXiv.org*. URL: https://arxiv.org/abs/2010.11929 (date of access: 18.04.2022).

10. MLP-Mixer: An all-MLP Architecture for Vision / I. Tolstikhin et al. *arXiv.org*. URL: https://arxiv.org/abs/2105.01601 (date of access: 18.04.2022).

11. Trockman A., Kolter J. Z. Patches are all you need?. *arXiv.org*. URL: https://arxiv.org/abs/2201.09792 (date of access: 18.04.2022).

12. A ConvNet for the 2020s / Z. Liu et al. *arXiv.org*. URL: https://arxiv.org/abs/2201.03545 (date of access: 18.04.2022).

13. Online learning: a comprehensive survey / S. C. H. Hoi et al. *Neurocomputing*. 2021. Vol. 459. P. 249–289. URL: https://doi.org/10.1016/j.neucom.2021.04.112 (date of access: 18.04.2022).

14. Online deep learning: learning deep neural networks on the fly / D. Sahoo et al. *arXiv.org*. URL: https://arxiv.org/abs/1711.03705 (date of access: 18.04.2022).

15. Daniušis P., Vaitkus P. Neural network with matrix inputs. *Informatica*. 2008. Vol. 19, no. 4. P. 477–486.

16. Mohamadian M., Afarideh H., Babapour F. New 2D matrix-based neural network for image processing applications. *IAENG International Journal of Computer Science*. 2015. Vol. 42, no. 3. P. 265–274. URL: http://www.iaeng.org/IJCS/issues_v42/issue_3/IJCS_42_3_11.pdf (date of access: 18.04.2022).

17. Gao J., Guo Y., Wang Z. Matrix neural networks. *Advances in Neural Networks* : Proceedings of the14th International Symposium on. Neural Networks (ISNN), Part II, Sapporo. 2017. P. 1–10.

18. Do K., Tran T., Venkatesh S. Learning Deep Matrix Representations. *arXiv.org*. URL: https://arxiv.org/abs/1703.01454 (date of access: 18.04.2022).

19. Matrix deep neural network and its rapid learning in data science tasks / I. Pliss et al. "Advanced computer information technologies – ACIT 2018 : Proceedings of the conference. Ceske Budejovice, Czech Republic, 2018. P. 141–144.

20. Deep 2d-neural network and its fast learning / Y. Bodyanskiy et al. 2018 IEEE second international conference on data stream mining & processing (DSMP), Lviv, Ukraine, 21–25 August 2018. 2018. URL: https://doi.org/10.1109/dsmp.2018.8478578 (date of access: 25.03.2022)

21. Bodyanskiy Y., Boiko O., Pliss I., Kopaliani D., Volkova V. 2D-Deep Neural Network and Its Online Rapid Learning. Proceedings of the 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Metz, France, 2019, P. 304-307, 2019.

22. Kuntsevich V., Lychak M. Synthesis of optimal and adaptive control systems: the game approach. Naukova dumka. 1985.

23. Kuntsevych V. On a solving of the problem of two-dimensional discrete filtration (synthesis of matrix filters). Automatica i telemekhanika. 1987. No. 6. P. 68–78.

24. Bodyanskiy Y., Pliss I. On a solving of the problem of a matrix object controlling under uncertainty conditions. Automatika i telemekhanika. 1990. No. 2. P. 175–178.

25. Bodyanskiy Y., Pliss I., Timofeev V. Discrete adaptive identification and extrapolation of two-dimensional fields. Pattern recognition and image analysis. 1995. Vol. 5, no. 3. P. 410–416.

26. Wolpert D. H. Stacked generalization. *Neural Networks*. 1992. Vol. 5, no. 2. P. 241–259. URL: https://doi.org/10.1016/s0893-6080(05)80023-1 (date of access: 18.04.2022).

27. McCulloch W. S., Pitts W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*. 1990. Vol. 52, no. 1-2. P. 99–115. URL: https://doi.org/10.1007/bf02459570 (date of access: 18.04.2022).

28. Nair V., Hinton G. E. Rectified linear units improve restricted boltzmann machines. *ICML* : Proceedings of the 27th International Conference

on Machine Learning, Haifa, 21 June 2010. Madison, WI, USA, 2010. P. 807–814.

29. Maas A. L., Hannun A. Y., Ng A. Y. Rectifier nonlinearities improve neural network acoustic models. *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing (WDLASL 2013)*, Atlanta, 16 June 2013.

30. Delving deep into rectifiers: surpassing human-level performance on imagenet classification / K. He et al. *2015 IEEE International Conference on Computer Vision (ICCV)*, Santiago, Chile, 7–13 December 2015. 2015. URL: https://doi.org/10.1109/iccv.2015.123 (date of access: 18.04.2022).

31. Understanding and improving convolutional neural networks via concatenated rectified linear units / W. Shang et al. *arXiv.org*. URL: https://arxiv.org/abs/1603.05201 (date of access: 18.04.2022).

32. Deep Learning with S-Shaped Rectified Linear Activation Units / Proceedings of the AAAI Conference on Artificial Intelligence / X. Jin et al. *AAAI Publications*. URL: https://ojs.aaai.org/index.php/AAAI/article/view/10287 (date of access: 18.04.2022).

33. Liew S. S., Khalil-Hani M., Bakhteri R. Bounded activation functions for enhanced training stability of deep neural networks on visual pattern recognition problems. *Neurocomputing*. 2016. Vol. 216, c. P. 718–734. URL: https://doi.org/10.1016/j.neucom.2016.08.037 (date of access: 18.04.2022).

34. Clevert D.-A., Unterthiner T., Hochreiter S. Fast and accurate deep network learning by exponential linear units (elus). *arXiv.org*. URL: https://arxiv.org/abs/1511.07289 (date of access: 18.04.2022).

35. Self-Normalizing neural networks / G. Klambauer et al. *arXiv.org*. URL: https://arxiv.org/abs/1706.02515 (date of access: 18.04.2022).

36. Trottier L., Giguere P., Chaib-draa B. Parametric exponential linear unit for deep convolutional neural networks. *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Cancun, 18–21

December 2017. 2017. URL: https://doi.org/10.1109/icmla.2017.00038 (date of access: 18.04.2022).

37. Maguolo G., Nanni L., Ghidoni S. Ensemble of convolutional neural networks trained with different activation functions. *arXiv.org*. URL: https://arxiv.org/abs/1905.02473 (date of access: 18.04.2022).

38. Aggarwal C. C. Neural Networks and Deep Learning: A Textbook. Springer, 2018. 497 p.

39. 2-D neural network based on m-neurons and its learning / A. Albasova et al. *2021 11th International Conference on Advanced Computer Information Technologies (ACIT)* : Proceedings of the International Conference, Deggendorf, 15 September 2021. 2021. P. 700–703. URL: https://doi.org/10.1109/ACIT52158.2021.9548532 (date of access: 18.04.2022).

40. Scherer A. Neuronale Netze: Grundlagen und Anwendungen. Vieweg+Teubner Verlag, 2012. 260 p.

41. Huang C. ReLU networks are universal approximators via piecewise linear or constant functions. Neural computation. 2020. Vol. 32, no. 11. P. 2249–2278. URL: https://doi.org/10.1162/neco_a_01316 (date of access: 25.03.2022)

42. Bodyanskiy Y., Kolodyazhniy V., Stephan A. An adaptive learning algorithm for a neuro-fuzzy network. Computational intelligence. theory and applications. Berlin, Heidelberg, 2001. P. 68–75. URL: https://doi.org/10.1007/3-540-45493-4_11 (date of access: 25.03.2022).

43. Papers with code - CIFAR-10 benchmark (image classification). *The latest in Machine Learning | Papers With Code*. URL: https://paperswithcode.com/sota/image-classification-on-cifar-10 (date of access: 21.04.2022).

44. GitHub - huyvnphan/PyTorch_CIFAR10: Pretrained TorchVision models on CIFAR10 dataset (with weights). *GitHub*. URL: https://github.com/huyvnphan/PyTorch_CIFAR10 (date of access: 23.04.2022).