Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра математики факультету інформатики

Імпульсні нейронні мережі для задач класифікації зображень

Текстова частина до курсової роботи
за спеціальністю „Прикладна математика ” 6.040301

Керівник курсової роботи
к.ф.-м.н., ст. в. Швай Н.О.

_____
(підпис)

“____” _____ 2021 р.

Виконав
студент 1 курсу
факультету інформатики
Кравченко І. В.

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри математики,
проф., д.ф.-м.н.
_____Б. В. Олійник
(підпис)
„____”_____2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Кравченко І. В. факультету інформатики 1-го курсу МП
ТЕМА Імпульсні нейронні мережі для задач класифікації зображень
Зміст ТЧ до курсової роботи:

1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Вступ
5. РОЗДІЛ 1: Огляд штучних нейронних мереж
6. РОЗДІЛ 2: Огляд імпульсних нейронних мереж
7. Висновки
8. Список використаних джерел

Дата видачі „___” _____ 2020 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи.	15.10.2020	
2.	Огляд задачі, підбір вхідних даних	04.05.2021	
3.	Аналіз побудови нейронних мереж	05.05.2021	
4.	Написання перших двох розділів	06.05.2021	
5.	Створення моделей	07.05.2021	
6.	Написання останнього розділу	08.05.2021	
8.	Корегування роботи згідно із зауваженнями керівника	09.05.2021	

Студенту Кравченко І. В.

Керівник Швай Н.О.

“_____”_____

Contents

Introduction……..6
1. Artificial Neural Networks Overview……....7
1.1. Neurons and Perceptron……...7
1.2. Multilayer Perceptron……..8
1.3. Activation Functions……..10
1.4. Feedforward Process……..11
1.5. Error Functions……...12
1.6. Optimization Algorithms……14
1.7. Backpropagation……16
2. Spiking Neural Networks Overview……18
2.1. Spiking Neural Network……18
2.2. Performance of ANN and SNN on MNIST dataset……...19
3. Conclusion……..…….25
References……26
Appendices……...27

Annotation

This work is dedicated to the discussion of the main features of the classical fully-connected feed-forward Artificial Neural Network(ANN) and the Spiking Neural Network(SNN) based on their performance on the MNIST dataset and a few of its variations. Firstly, models' architectures are discussed, then they are trained on the MNIST dataset. Finally, their performances are compared, and from the results of the experiment, the conclusion is stated.

Introduction

Artificial Neural Networks(ANN) have emerged in 1958 thanks to Frank Rosenblatt, who invented the Perceptron, and they have come a long and complex way full of indeterminacy and difficulties. Nowadays, Artificial Neural Networks are a hot topic in scientific research and have many applications in different business applications, and are referred to as the next Industrial revolution by some field experts which is a sign of the bright future of this technology and its predicted high impact on the future of the entire humanity. There are many different ANN architectures that excel at doing certain types of tasks such as Image Recognition or Natural Language Processing, each having its own strengths and weaknesses. Among this huge variety of network architectures, there is one which is discussed in this paper, namely, the Spiking Neural Network(SNN). Its main feature is that it is more inspired by the design of the human brain and brains of other biological creatures on the contrary to the more common ANN architectures which are just briefly influenced by the structure of the brain at the highest level by the artificial neuron, however, they do not aim to replicate the behavior of an actual living creature's brain.

The goal of this paper is to discuss architectural differences between the common ANN architecture represented by the classic feed-forward fully-connected neural network and SNN, to train them and compare their performance on the MNIST dataset and its modified variations, and to sum up the discussion by comparing each model's strengths and weaknesses.

The work consists of three parts: in the first, we introduce and briefly review key artificial neural network components and their training process. In the second part, we introduce the spiking neural network architecture and discuss how it is different from more common ANN architectures. In the third final part, we train both models on the MNIST dataset and compare their performance of the MNIST itself and its different variations. After a comparison of the experiment's results, we make a concluding statement about differences between ANNs and SNNs and where ones are better than others.

1. Artificial Neural Networks Overview

1.1 Neurons and Perceptron

The human brain is a complex biological system, the elementary unit of which is known as a neuron. Neurons are gathered into a complex network where they exchange electrical signals with each other in very difficult patterns that remain unknown till today. The neuron gets by dendrites electric signals from other neurons as an input, combines them in a certain way, and if the total amount of electric charge it has accumulated is bigger than a certain threshold this neuron has, it fires an output signal via its axon to the next neurons that receive this signal as their input, and this process continues.

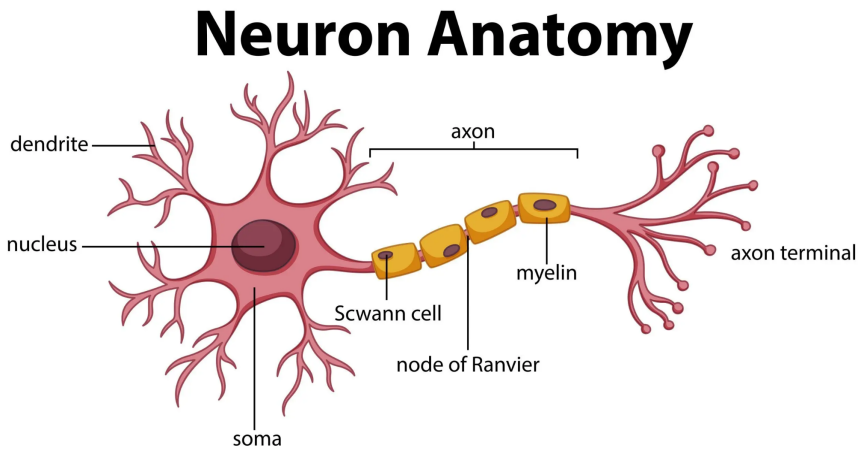

Figure 1.1 - Biological Neuron Structure

The first mathematical model to represent such behavior was developed in 1958 by Frank Rosenblatt and was named Perceptron. The perceptron receives several numerical inputs, takes their weighted sum, and applies a step function to the result of this computation. The step function fires if the weighted sum is greater than or equal to the threshold

value of this neuron, and does not fire otherwise. Mathematically it is defined in the following way:

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

In this case, the input of the step function is a weighted sum of inputs:

$$z = \sum_i x_i w_i + b$$

In the formula above, x_i corresponds to the i-th input and w_i, respectively, corresponds to the weight of the i-th input, and b is called a bias term, which is added in order to give a model more flexibility and express more complex relationships in the data. In order to simplify notation, the vector of inputs **x** is usually extended by an artificial input $x_0 = 1$, which allows to add b to the weights vector as the 0-th coordinate $w_0 = b$, and thus, consider the input of neuron as the dot-product of the vectors of weights and inputs: $z = \mathbf{x} \cdot \mathbf{w}$.

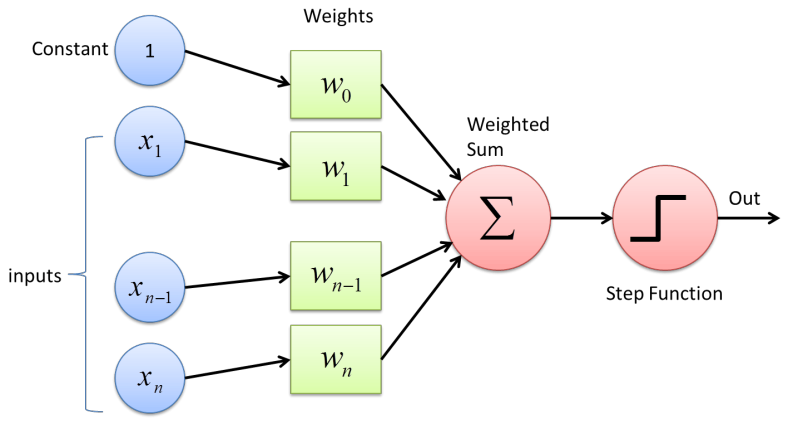

Figure 1.2 - The Perceptron

1.2 Multilayer Perceptron

A single Perceptron produces a linear output, n+2-dimensional linear surface in particular if an input of the neuron has n features and 1 bias

term. This linear kind of surface is often not flexible enough to express the complexity hidden in the dataset fed to the network. To be precise, the data that can be split with a linear surface is called linear, and nonlinear otherwise. Nowadays, the vast majority of datasets are nonlinear, and thus require a more nontrivial neural network that would be able to capture this nonlinearity.

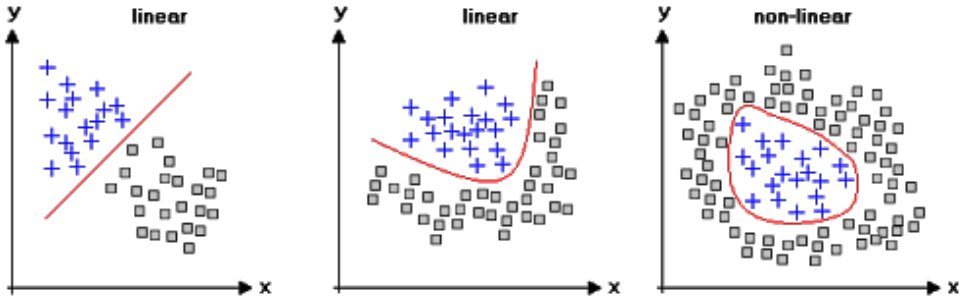

Figure 2 - examples of linear and nonlinear data

To split a nonlinear dataset, there is more than just one line needed, which means that we need to use a more complex architecture that would split this data with tens or hundreds or even thousands lines, and thus, have many neurons, since a single neuron produces a single line.

In order to achieve this goal, neurons are often stacked into several layers: the input one which consists of neurons that get original data as the input, hidden layers, that are connected to the neurons from the previous layer and send their output to the neurons from the next layer, and the output layer, which produces the final output of the model. The principle behind this architecture lies in the behavior of different neurons from different layers: in general, the neurons from one layer capture different data features from the input, while neurons from different layers are able to capture features of different abstraction levels. For example, if one is building a model that would classify human faces, the neurons in the first layer would find some low-level features such as line segments,

and the neurons from the last hidden layer would likely find more high-level features, such as nose, lips, brows and so on.

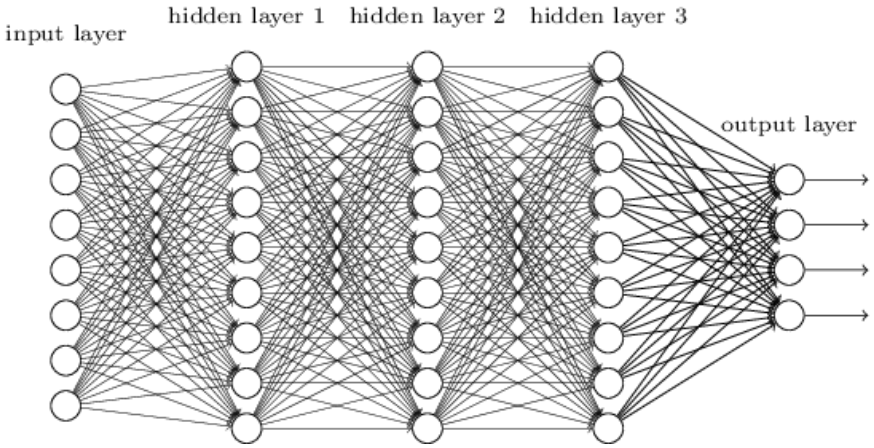

Figure 3 - Multilayer Neural Network

1.3 Activation Functions

Activation functions are an important part of any neural network architecture as they are said to add *nonlinearity*. Since there is always a linear transformation equivalent to any sequence of other linear transformations no matter how long it is, if the network has a linear activation function, its output will always be a simple linear transformation of the input data no matter how complex the network's architecture is. In order to build a network that would be able to express complicated relations in the data, its neurons need to have *activation function* which is the function applied to the output of a neuron before transferring it to the next layer. The Heaviside function or step-function discussed before is good in terms of modelling an actual behavior of the biological neuron, however, it is very hard to optimize or train such a model since this function is not continuous, and optimizing non-continuous functions is way harder than the continuous ones. There are many other options to choose for an activations function, however in the

majority of cases, the most used ones are the sigmoid(logistic function), softmax(generalization of sigmoid on multiclass cases), hyperbolic tangent(tanh), and rectified linear unit(ReLU). Current state-of-the-art solution status is fixed after the ReLU function as it is very easy to compute and doesn't have problems with vanishing gradients and saturation which are common for sigmoid and hyperbolic tangent functions.

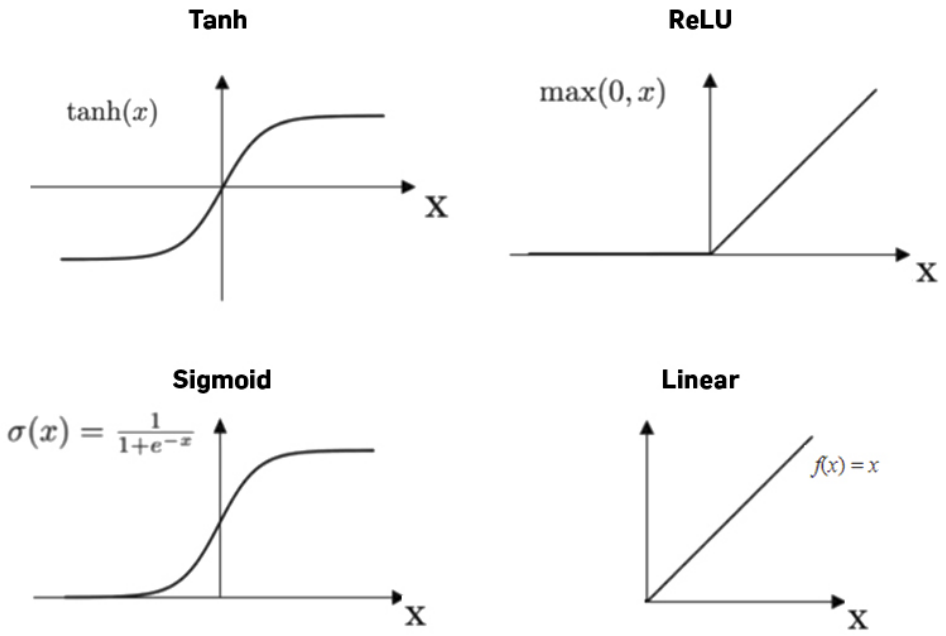

Figure 4 - Activation Functions

1.4 Feedforward Process

The process of computing a weighted sum and applying an activation function to it is called *feedforward*. The feedforward process generally consists of the following steps:

1. Inputs are stacked into one vector **x**.
2. Input vector **x** is multiplied by the weights matrix from layer 1 ($W^{(1)}$) and the result is fed to the activation function, for example, sigmoid: $\sigma \cdot W^{(1)}$.

3. The result from the previous step is multiplied by the weights matrix of the second layer, and again is fed to the activation function: $\sigma \cdot \mathrm{W}^{(1)} \cdot \sigma \cdot \mathrm{W}^{(2)}$.
4. The process continues till the output layer.

If the model has three layers with sigmoid activations, its output $\hat{y}$ would look like the following:

$$\hat{y} = \sigma \cdot W^{(3)} \cdot \sigma \cdot W^{(2)} \cdot \sigma \cdot W^{(1)} \cdot \mathbf{x}$$

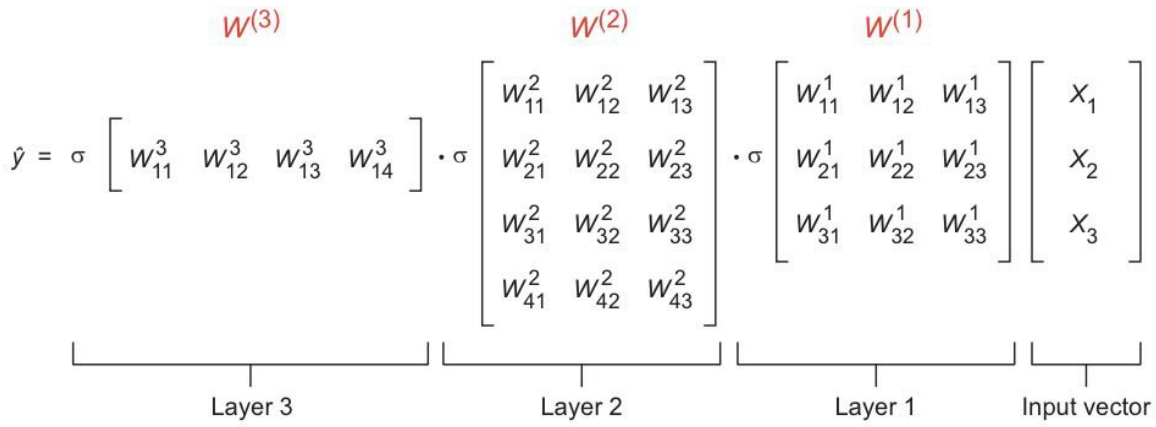

Figure 5 - Feedforward Process

1.5 Error Functions

In order to train a neural network that would do something useful, one has to optimize its parameters so that they produce the best output. To optimize parameters, a network needs a certain type of measure how good or bad its predictions are. This role is played by the *error function*. One of the most commonly used error functions in deep learning is called the mean squared error (MSE) and is defined as the average sum of squared errors:

$$MSE(W,b) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

MSE is a good choice because it is always non-negative and equals 0 only if all the predictions were correct. Also, the derivatives of MSE are easy to compute and optimize which is a plus. However, MSE is quite sensitive to the outliers because it squares the errors, and thus might not be the best choice depending on the context of the problem. For example, sensitivity to outliers might be useful when analyzing stock market, but it would harm the model if it would be trying to predict house prices. In that case, the model should be more focused on the average tendency rather than exceptional cases. For this purpose, mean average error (MAE) function is often used:

$$MAE(W,b) = \frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|$$

For the classifications problem, cross-entropy (CE) error function is currently considered to be the state-of-the art solution.
For one example, cross-entropy is defined in the following way:

$$CE(W,b) = -\sum_{i=1}^{m}\hat{y}_i \log(p_i)$$

By analogy, for the entire dataset cross-entropy is defined as:

$$CE(W,b) = -\sum_{i=1}^{n}\sum_{j=1}^{m}\hat{y}_{ij} \log\left(p_{ij}\right)$$

Roughly speaking, cross-entropy is measuring the difference between two probability distributions. For example, if the neural network is solving multiclass classification problem, and certain input object belongs to the second class, it corresponds to the probability distribution [0 1 0]. If the network produces its own distribution [0.2 0.3 0.5], the cross-entropy loss is going to be 1.2 on this example. But the loss on a closer to the original distribution like [0.3 0.5 0.2] where the network is more confident about its prediction, is going to be smaller, in particular 0.69.

1.6 Optimization Algorithms

As discussed in the previous chapter, the goal of the error function is to measure how wrong the network's predictions are. This information is very important for the training process, as its goal is to minimize this error. The straightforward method from calculus to take every parameter's derivative and make it equal zero and solve this system of equations, but in practice it wouldn't work because even relatively small neural networks might have tens of thousands trainable parameters, which are namely weights of neurons and biases. Modern software is not advanced enough to solve systems of this kind, thus another approach to optimization is required. The classic algorithmic approach to this problem is known as the Gradient descent, and even though it is currently far from the nest algorithms, all of them have emerged as modifications of the gradient descent. The key figure in this algorithm is the notion of gradient. It is very useful since the gradient of the error function $\nabla E(W, b)$ is a vector that points in the direction of the function's fastest increase, which means that the opposite direction is where the function decreases the fastest. The main point of gradient descent lies in iteratively

tweaking each weight by a small amount such that the error function decreases in general. The amount by which the weight w_i is changed is determined by the partial derivative of the error function $E(W, b)$ with respect to w_i and the value λ known as learning rate.

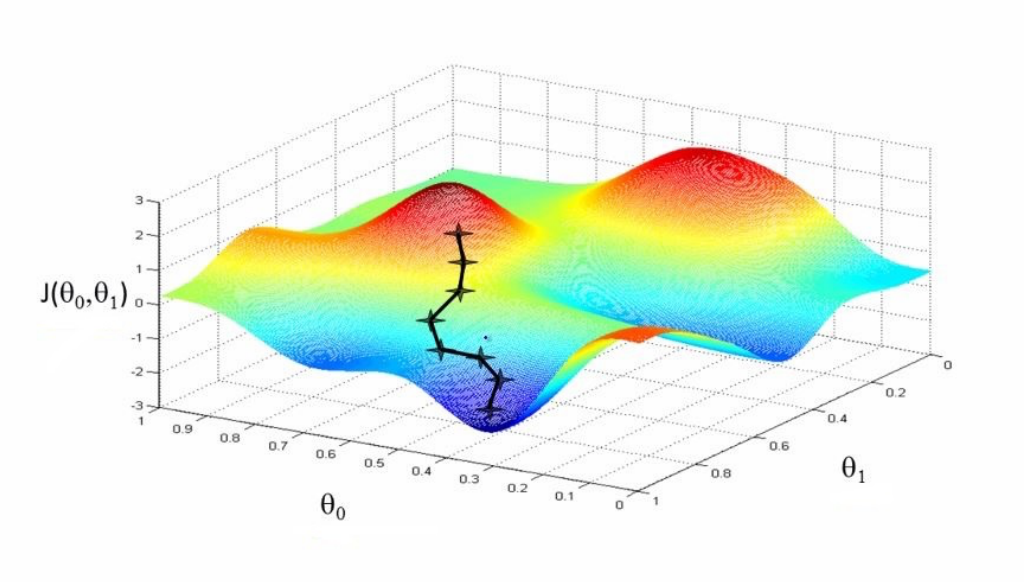

Figure 6 - Gradient Descent

In mathematical notation, the change of weights is defined as follows:

$$\Delta w_i = -\alpha \frac{\mathrm{dE}}{\mathrm{dw_i}}$$

$$w_{i+1} = w_i + \Delta w_i$$

There are a few important issues with this version of gradient descent. Firstly, it is prone to stopping at local minima and thus not finding better solutions. Also, it is very computationally expensive procedure, since the computer has to compute gradients for every weight and update it. This problem is especially important for deep neural networks with hundreds of thousands and more trainable parameters, where only a single step of gradient descent will require computation of this enormous number of

updates. Although gradient descent is almost never used nowadays, there are many of its modifications that are successfully applied to both research and business problems and allow much faster convergence with less hardware resources. The current state-of-the-art optimization solution is considered to be Adam.

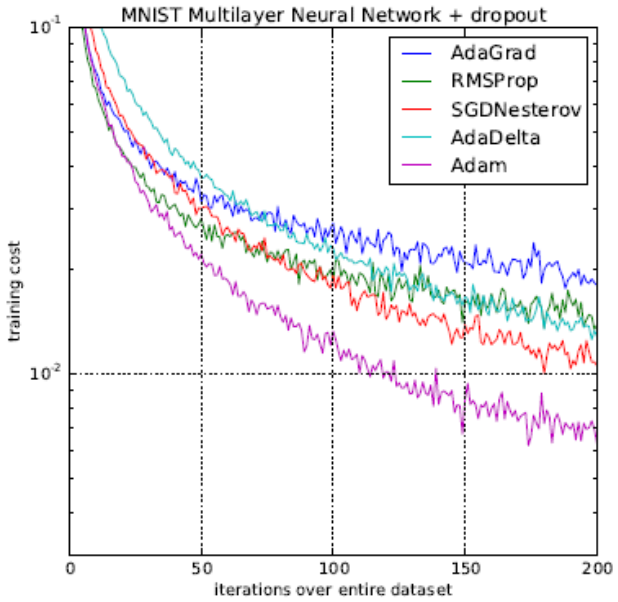

Figure 7 - Optimizers Performance

1.7 Backpropagation

Backpropagation, or *backward pass*, is a process of propagation of partial derivatives of the error function $E(W, b)$ with respect to each specific weight w_i from the last(output) layer to the first layer(input) to adjust weights. The thing is that partial derivatives are easy to compute for a one-layer neural network where each weight has a direct impact on the output of the model, however, as the network gets more complex gaining more layers, the impact of each weight on the final output is less clear. In order to compute the partial derivative $\frac{dE}{dw_i}$ of

the error function with respect to each weight, we use the chain rule.

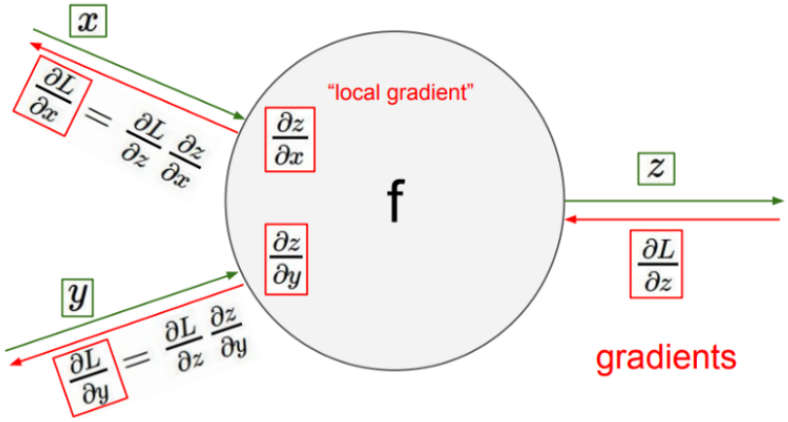

Figure 8 - Backpropagation Scheme

2. Spiking Neural Networks Overview

2.1 Spiking Neural Network Architecture

Although the original artificial neural network based on the Perceptron was inspired by the biological model of the neuron, it does not actually represent its properties and behavior in biological systems. The key difference between current(second) generation of neural networks and spiking neural networks (third generation) is in the character of interaction between the neurons: in classical models the neurons continuously interchange information between each other, and the data permanently flows forward and backward during the processes of feedforward and backpropagation respectively. In biological models, neurons interact in other way: they do not transmit the impulses at each propagation cycle. Instead, they have a cumulative quality similar to the one originally suggested by Frank Rosenblatt and his Perceptron model: biological neurons accumulate an electric potential, and once this accumulated potential becomes bigger than a certain threshold of this neuron, it fires, and generates an electric signal that travels to other neurons by axons and dendrites and makes their accumulated potential smaller or bigger according to the signal they receive.

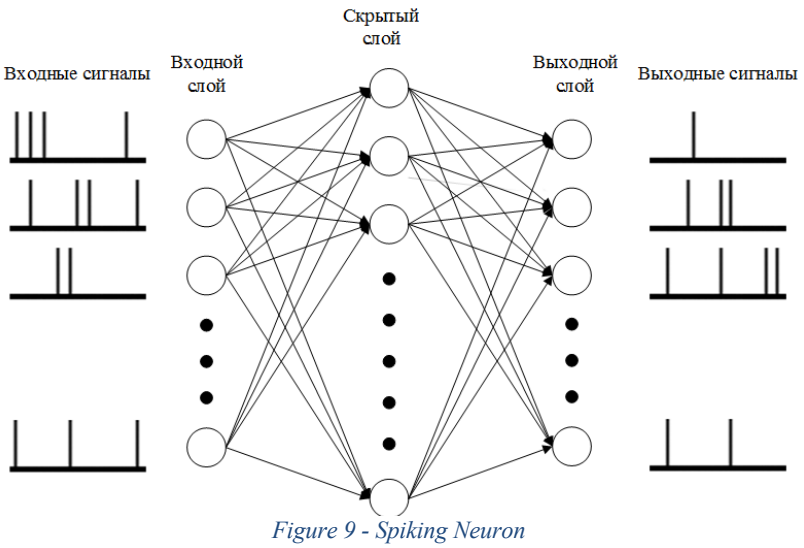

Figure 9 - Spiking Neuron

The working principle of the spiking neural network is the following: the network gets as inputs a series of impulses and outputs a series of impulses as well. Each neuron at any moment has a certain value analogous to the potential energy in biological neurons, and if this value surpasses a certain threshold, the neuron spikes and sends a signal to the other neurons after what its value drops to the level below average for a short period of time, approximately 2-30ms.

2.2 Comparison of ANN and SNN on MNIST dataset

The experimental part of this paper is the comparison of trained classical fully-connected feedforward artificial neural network and spiking neural network on the MNIST dataset and its variations. MNIST is a dataset of handwritten digits often used for testing various concepts and neural networks architectures in the problem of computer vision.

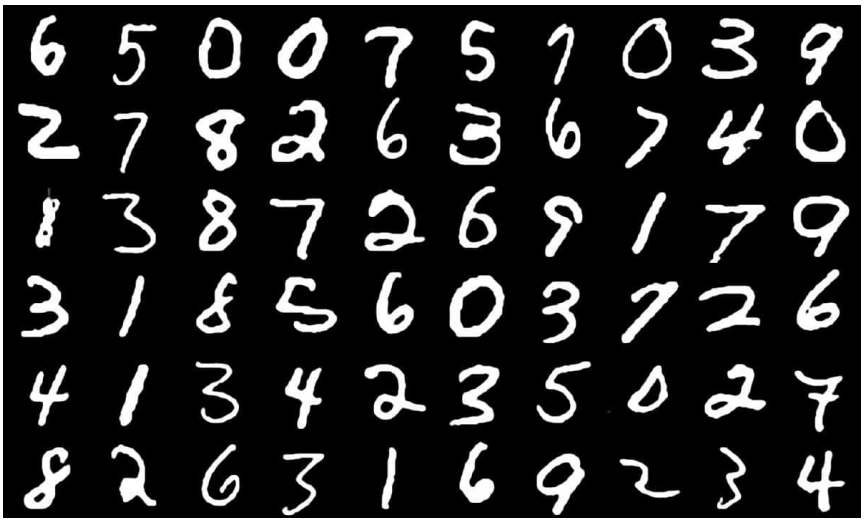

Figure 10 - MNIST Dataset

For the comparison two models with different activation functions and identical layers architectures were chosen.

The results of training on the classical MNIST were the following:

```
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4248 - accuracy: 0.8847
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1180 - accuracy: 0.9652
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0822 - accuracy: 0.9758
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0592 - accuracy: 0.9819
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0400 - accuracy: 0.9882
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0329 - accuracy: 0.9899
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0252 - accuracy: 0.9924
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0218 - accuracy: 0.9935
Epoch 9/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0173 - accuracy: 0.9947
Epoch 10/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0145 - accuracy: 0.9959
313/313 - 0s - loss: 0.0811 - accuracy: 0.9779

Test accuracy: 0.9779000282287598
```

Figure 11 - ANN Results

```
Epoch 1/10
1875/1875 [==============================] - 15s 7ms/step - loss: 0.4360 - accuracy: 0.8762
Epoch 2/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.1196 - accuracy: 0.9654
Epoch 3/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0848 - accuracy: 0.9746
Epoch 4/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0593 - accuracy: 0.9820
Epoch 5/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0404 - accuracy: 0.9877
Epoch 6/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0330 - accuracy: 0.9905
Epoch 7/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0249 - accuracy: 0.9928
Epoch 8/10
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0217 - accuracy: 0.9934
Epoch 9/10
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0180 - accuracy: 0.9946
Epoch 10/10
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0151 - accuracy: 0.9952
313/313 - 1s - loss: 18.5753 - accuracy: 0.1870

Test accuracy: 0.18700000643730164
```

Figure 12 - SNN Results

It can be seen that although SNN's performance on the train set is very good, it does not perform well on the train set. This is due to a unique feature of SpikingActivation: it automatically swaps the behavior of the spiking neurons during training. Because spiking neurons are in general non-differentiable, it is not possible to directly use the spiking activation function during training. Instead, SpikingActivation will use the base

non-spiking activation during training, and the spiking version during inference. So, during training above we are seeing the performance of the non-spiking model, but during evaluation we are seeing the performance of the spiking model. In order to understand the reasons of such a poor performance, it is useful to visualize the results.

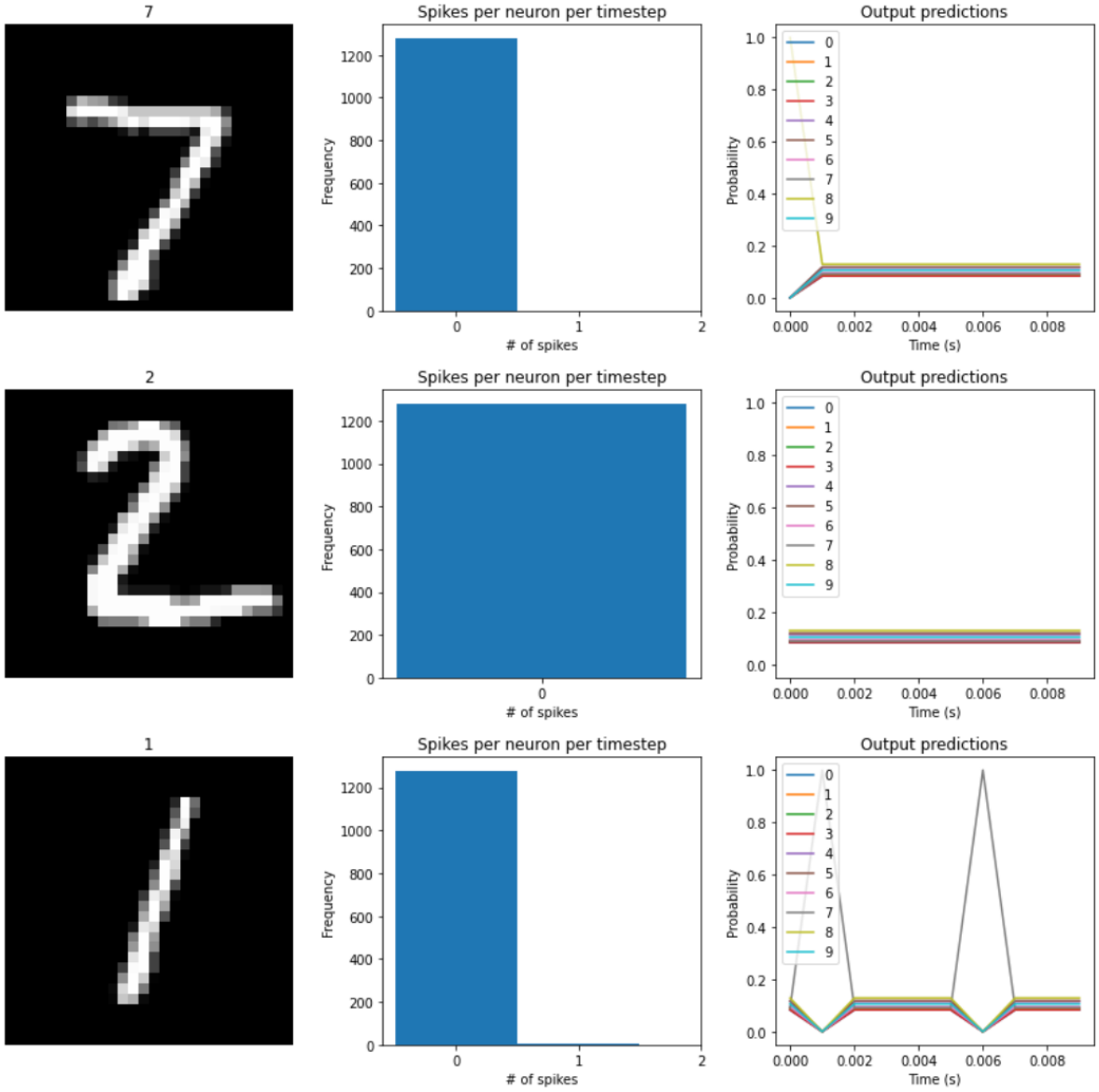

Figure 13 - SNN Spikes Visualization

It is immediately obvious where the problem is: the neurons are barely spiking due to the short length of the time interval of the simulation, to be precise, it is equal to 0.01s in this case. If the neurons do not spike very rapidly during this time interval, they do not have enough time to gain

enough potential and spike. With the increase of time interval for the simulation, the accuracy on the test set increases dramatically.

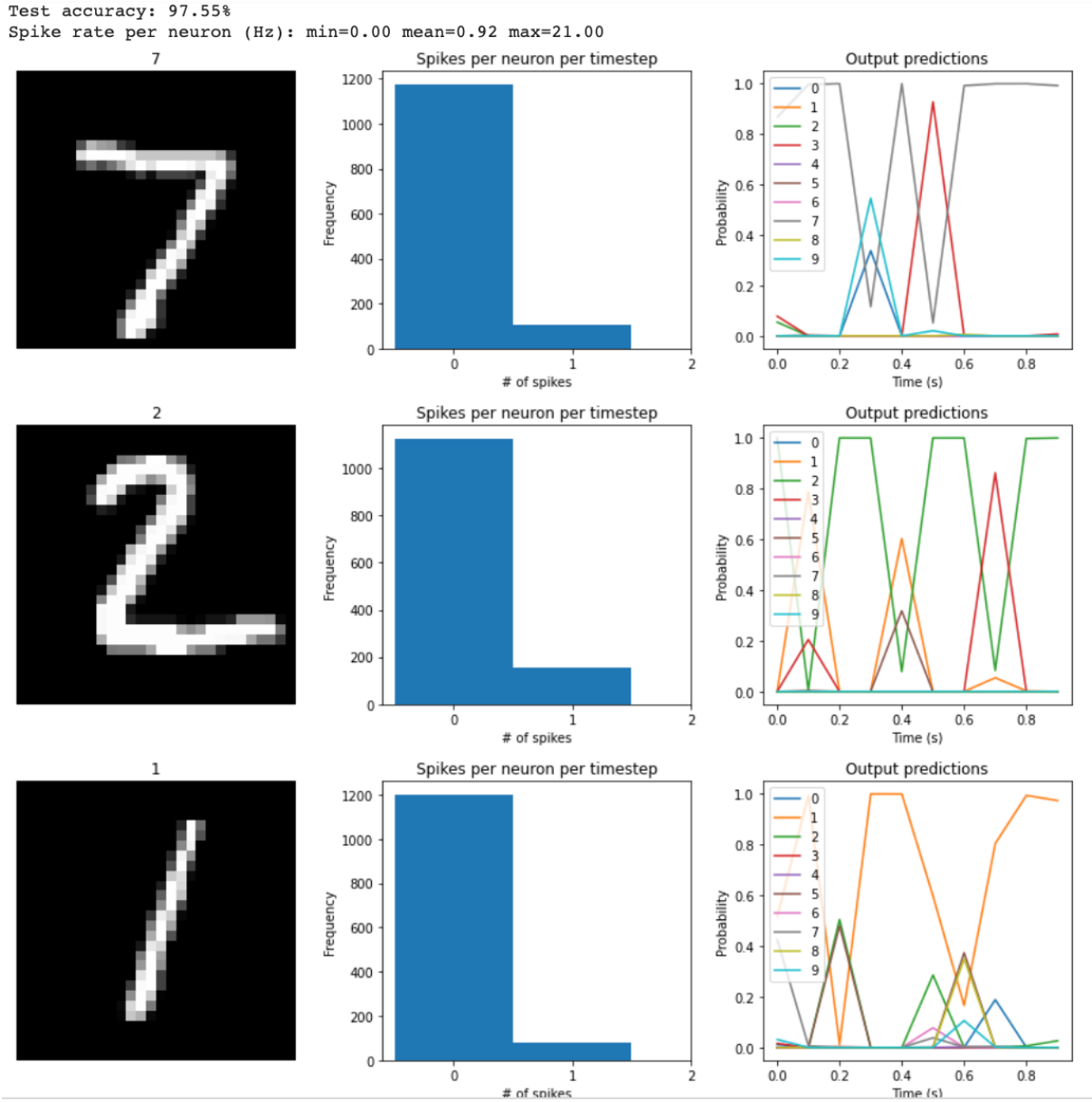

Figure 14 - SNN Performance with 1s time interval

One might be tempted to simply increase the length of the time interval for the increase in the SNN's performance, however, it is important to remember about advantages of SNNs over other ANNs. One of the key advantages is spiking neural network's ergonomics: instead of supporting the flow of data through the entire network with its huge number of neurons, it is way easier to track only a few neurons interchanging spikes. This kind of SNNs behavior is way more computationally effective than

feedforward and backpropagation used in the networks of second generation.

The following part of the work is dedicated to the comparison of pre-trained on the original MNIST models on its two variations: one with rotated numbers and the other with random noise at the background and see if there is any significant difference between them.

Firstly we shall consider the rotated dataset.

The models have shown no difference and both performed quite poorly:

```
1 score = model.evaluate(rot_imgs, rot_labels, verbose=0)
2 print("Test loss:", score[0])
3 print("Test accuracy:", score[1])
```

```
Test loss: 7.814118385314941
Test accuracy: 0.34275001287460327
```

```
1 spiking_score = spiking_model.evaluate(rot_imgs, rot_labels, verbose=0)
2 print("Test loss:", score[0])
3 print("Test accuracy:", score[1])
```

```
WARNING:tensorflow:Model was constructed with shape (None, None, 28, 28) fo
Test loss: 7.814118385314941
Test accuracy: 0.34275001287460327
```

Figure 15 - ANN and SNN results on rotated MNIST

With the increase of the time interval, the number of spikes increased but did not dramatically influence the accuracy of the SNN:

```
Test accuracy: 33.53%
Spike rate per neuron (Hz): min=0.00 mean=0.83 max=19.00
```

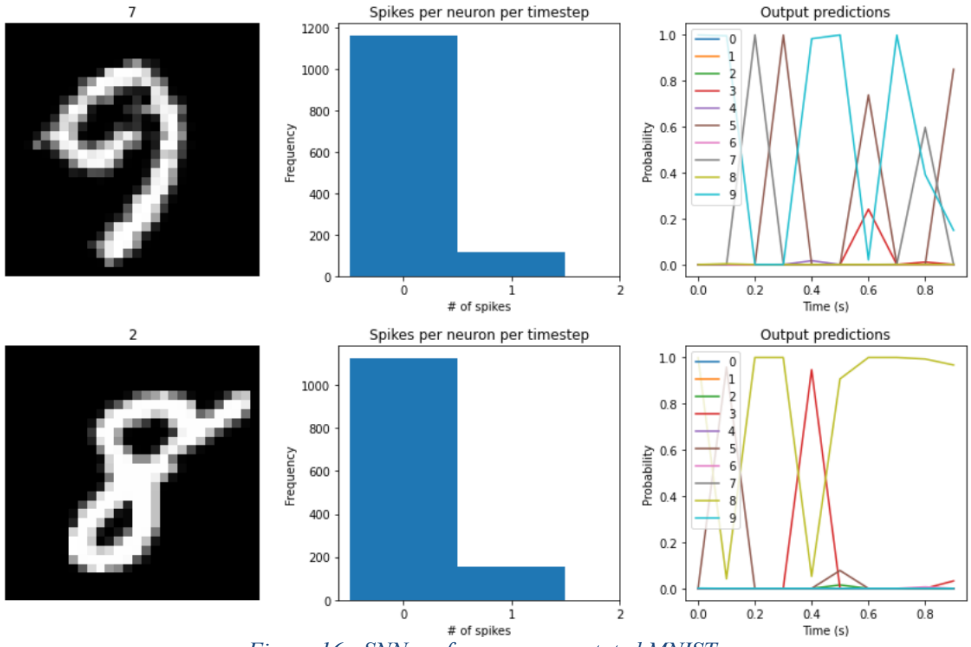

Figure 16 - SNN performance on rotated MNIST

Moving on to the noisy dataset, the accuracy of the SNN was slightly worse.

```
score = model.evaluate(rand_imgs, rand_labels, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
Test loss: 17.880184173583984
Test accuracy: 0.18774999678134918
```

```
spiking_score = spiking_model.evaluate(rand_imgs, rand_lab
print("Test loss:", spiking_score[0])
print("Test accuracy:", spiking_score[1])
```

```
Test loss: 82.02574157714844
Test accuracy: 0.09958333522081375
```

Figure 17 - ANN and SNN performance on the noisy MNIST

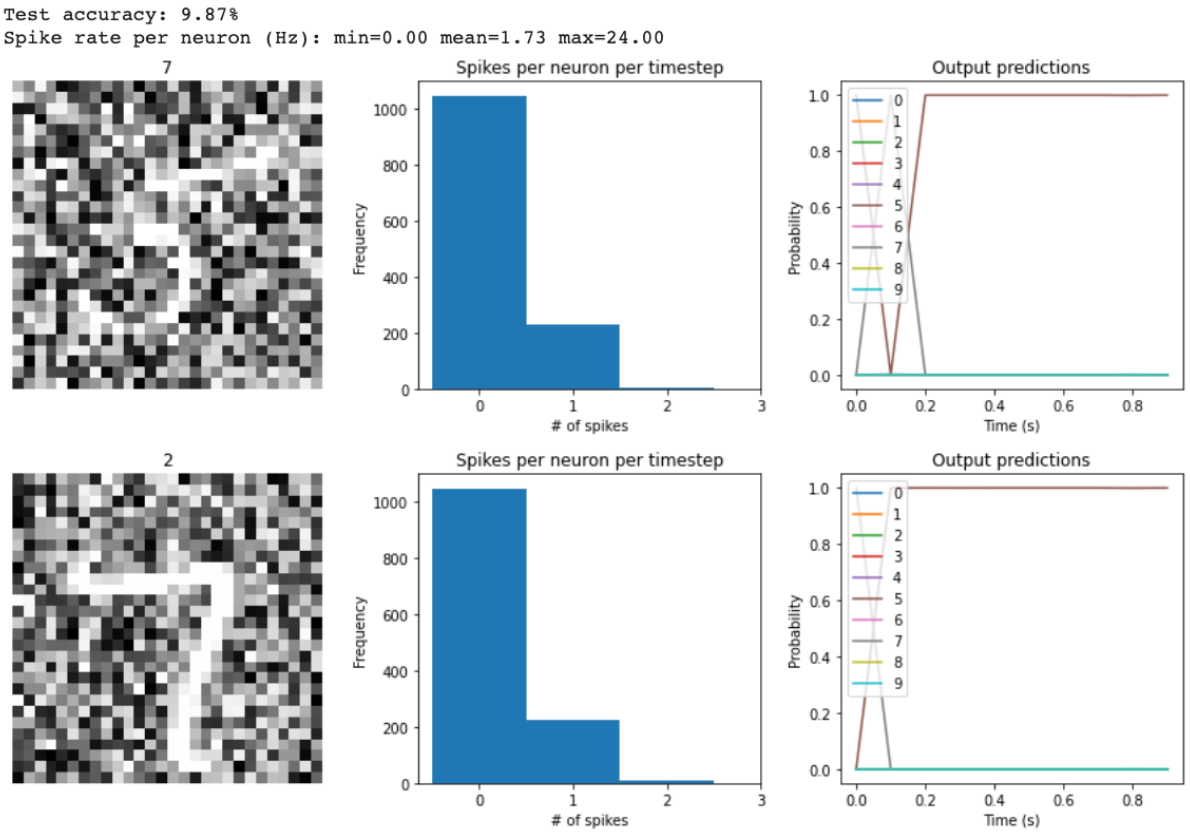

Figure 18 - SNN performance on the noisy MNIST

To sum up, neither of the networks has shown an ability to adapt to the distorted or uncommon data.

Conclusion

In the first part of this paper, we have covered key components of modern artificial neural networks and how they work. The second part of the work represents the notion of spiking neural networks, their relation to the biological neurons and principles of their work. In the practical part, we have built, trained and compared the performance of an ordinary fully connected feedforward artificial neural network and spiking neural network. To sum up the results of the experiment, there are no strong differences between the performance of ANNs and SNNs on the unseen previously data. In addition, the performance of the SNN might be regulated by changing the dt parameter that corresponds to the length of the simulation. The longer this time interval is – the more actively neurons spike and the better model's predictions are. On the other side, SNN's advantage is their ergonomics which is more computationally efficient, but at the cost of model's accuracy.

References

1. Chollet F. Deep Learning with Python; – Manning Publications [2017] – 384p.
2. Géron A. Hands-On Machine Learning with Scikit-Learn and TensorFlow; – O'Reilly [2017] – 542p.
3. Hastie T. The Elements of Statistical Learning. Data Mining, Inference, and Prediction / Hastie T., Tibshirani R., Friedman J. – Springer [2017] – 747p.
4. Goodfellow I. Deep Learning / Goodfellow I., Bengio Y., Courville A. – Cambridge MA : MIT Press [2017] – 777p.
5. Maas, Wolfgang. Networks of spiking neurons: The third generation of neural network models // Neural Networks : journal. — 1997. — Vol. 10. — P. 1659—1671. — doi:10.1016/S0893-6080(97)00011-7.

Appendices

The code of the program:

```
pip install keras_spiking

"""## Завантаження та обробка даних"""

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

import keras_spiking

tf.random.set_seed(0)
np.random.seed(0)

(train_images, train_labels), (test_images, test_labels) =
tf.keras.datasets.mnist.load_data()

# normalize images so values are between 0 and 1
train_images = train_images / 255.0
test_images = test_images / 255.0

class_names = [str(i) for i in range(10)]
num_classes = len(class_names)

plt.figure(figsize=(5, 5))
for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.axis("off")
    plt.title(class_names[train_labels[i]])

"""## Класична Feedforward мережа"""

model = tf.keras.Sequential(
    [
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(128, activation="relu"),
        tf.keras.layers.Dense(10),
    ]
)


def train(input_model, train_x, test_x):
    input_model.compile(
        optimizer="adam",

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=["accuracy"],
    )

    input_model.fit(train_x, train_labels, epochs=10)

    _, test_acc = input_model.evaluate(test_x, test_labels, verbose=2)

    print("\nTest accuracy:", test_acc)
```

```
train(model, train_images, test_images)

"""## Spiking мережа"""

# repeat the images for n_steps
n_steps = 10
train_sequences = np.tile(train_images[:, None], (1, n_steps, 1, 1))
test_sequences = np.tile(test_images[:, None], (1, n_steps, 1, 1))

spiking_model = tf.keras.Sequential(
    [
        # add temporal dimension to the input shape; we can set it to None,
        # to allow the model to flexibly run for different lengths of time
        tf.keras.layers.Reshape((-1, 28 * 28), input_shape=(None, 28, 28)),
        # we can use Keras' TimeDistributed wrapper to allow the Dense
layer
        # to operate on temporal data
        tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(128)),
        # replace the "relu" activation in the non-spiking model with a
        # spiking equivalent
        keras_spiking.SpikingActivation("relu",
spiking_aware_training=False),
        # use average pooling layer to average spiking output over time
        tf.keras.layers.GlobalAveragePooling1D(),
        tf.keras.layers.Dense(10),
    ]
)

# train the model, identically to the non-spiking version,
# except using the time sequences as inputs
train(spiking_model, train_sequences, test_sequences)

spikeaware_model = tf.keras.Sequential(
    [
        tf.keras.layers.Reshape((-1, 28 * 28), input_shape=(None, 28, 28)),
        tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(128)),
        # set spiking_aware_training and a moderate dt
        keras_spiking.SpikingActivation("relu", dt=0.01,
spiking_aware_training=True),
        tf.keras.layers.GlobalAveragePooling1D(),
        tf.keras.layers.Dense(10),
    ]
)

train(spikeaware_model, train_sequences, test_sequences)

"""## Перевірка Spiking мережі"""

# Commented out IPython magic to ensure Python compatibility.
def check_output(seq_model, input=test_sequences, labels=test_labels,
modify_dt=None):
    # rebuild the model with the functional API, so that we can
    # access the output of intermediate layers
    inp = x = tf.keras.Input(batch_shape=seq_model.layers[0].input_shape)

    has_global_average_pooling = False
    for layer in seq_model.layers:
        if isinstance(layer, tf.keras.layers.GlobalAveragePooling1D):
            # remove the pooling so that we can see the model's
```

```
            # output over time
            has_global_average_pooling = True
            continue

        if isinstance(layer, (keras_spiking.SpikingActivation,
keras_spiking.Lowpass)):
            cfg = layer.get_config()
            # update dt, if specified
            if modify_dt is not None:
                cfg["dt"] = modify_dt
            # always return the full time series so we can visualize it
            cfg["return_sequences"] = True

            layer = type(layer)(**cfg)

        if isinstance(layer, keras_spiking.SpikingActivation):
            # save this layer so we can access it later
            spike_layer = layer

        x = layer(x)

    func_model = tf.keras.Model(inp, [x, spike_layer.output])

    # copy weights to new model
    func_model.set_weights(seq_model.get_weights())

    # run model
    output, spikes = func_model.predict(input)

    if has_global_average_pooling:
        # check test accuracy using average output over all timesteps
        predictions = np.argmax(output.mean(axis=1), axis=-1)
    else:
        # check test accuracy using output from only the last timestep
        predictions = np.argmax(output[:, -1], axis=-1)
    accuracy = np.equal(predictions, labels).mean()
    print("Test accuracy: %.2f%%" % (100 * accuracy))

    time = input.shape[1] * spike_layer.dt
    n_spikes = spikes * spike_layer.dt
    rates = np.sum(n_spikes, axis=1) / time

    print(
        "Spike rate per neuron (Hz): min=%.2f mean=%.2f max=%.2f"
#         % (np.min(rates), np.mean(rates), np.max(rates))
    )

    # plot output
    for i in range(4):
        plt.figure(figsize=(12, 4))

        plt.subplot(1, 3, 1)
        plt.title(class_names[test_labels[i]])
        if input is None:
          plt.imshow(test_images[i], cmap="gray")
        else:
          plt.imshow(input[i][0], cmap="gray")
        plt.axis("off")

        plt.subplot(1, 3, 2)
        plt.title("Spikes per neuron per timestep")
```

```
        bin_edges = np.arange(int(np.max(n_spikes[i])) + 2) - 0.5
        plt.hist(np.ravel(n_spikes[i]), bins=bin_edges)
        x_ticks = plt.xticks()[0]
        plt.xticks(
            x_ticks[(np.abs(x_ticks - np.round(x_ticks)) < 1e-8) & (x_ticks
> -1e-8)]
        )
        plt.xlabel("# of spikes")
        plt.ylabel("Frequency")

        plt.subplot(1, 3, 3)
        plt.title("Output predictions")
        plt.plot(
            np.arange(test_sequences.shape[1]) * spike_layer.dt,
            tf.nn.softmax(output[i]),
        )
        plt.legend(class_names, loc="upper left")
        plt.xlabel("Time (s)")
        plt.ylabel("Probability")
        plt.ylim([-0.05, 1.05])

        plt.tight_layout()

"""### Класичний MNIST"""

check_output(spiking_model)

check_output(spiking_model, modify_dt=0.01)

check_output(spiking_model, modify_dt=0.1)

check_output(spiking_model, modify_dt=0.9)

"""### Датасети із зашумленими даними"""

# Конвертація .amat файлу в датасет для моделі

def create_dataset(array):
  images = []
  labels = []
  for i in range(len(array)):
    img = array[i][:-1].reshape(28,28).T
    label = int(array[i][-1])
    images.append(img)
    labels.append(label)
  imgs = np.array(images)
  lbls = np.array(labels)
  return imgs, lbls

"""### Датасет із повернутими зображеннями"""

rot_img_test = np.loadtxt('/content/mnist_all_rotation_train.amat',
dtype=np.float32)
rot_imgs, rot_labels = create_dataset(rot_img_test)

plt.imshow(rot_imgs[0], cmap='gray')

rot_labels[0]

rot_test_sequences = np.tile(rot_imgs[:, None], (1, n_steps, 1, 1))
```

```
score = model.evaluate(rot_imgs, rot_labels, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])

spiking_score = spiking_model.evaluate(rot_imgs, rot_labels, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])

check_output(spiking_model, input=rot_test_sequences, labels=rot_labels,
modify_dt=0.1)

"""### Датасет з випадковим шумом """

rand_img_test = np.loadtxt('/content/mnist_background_random_train.amat',
dtype=np.float32)
rand_imgs, rand_labels = create_dataset(rand_img_test)

plt.imshow(rand_imgs[0], cmap='gray')

rot_labels[0]

rand_test_sequences = np.tile(rand_imgs[:, None], (1, n_steps, 1, 1))

score = model.evaluate(rand_imgs, rand_labels, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])

spiking_score = spiking_model.evaluate(rand_imgs, rand_labels, verbose=0)
print("Test loss:", spiking_score[0])
print("Test accuracy:", spiking_score[1])

aware_score = spikeaware_model.evaluate(rand_imgs, rand_labels, verbose=0)
print("Test loss:", aware_score[0])
print("Test accuracy:", aware_score[1])

rand_test_sequences[0][0].shape

check_output(spiking_model, input=rand_test_sequences, labels=rand_labels,
modify_dt=0.1)

check_output(spikeaware_model, input=rand_test_sequences,
labels=rand_labels, modify_dt=0.1)
```