МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА 33

ОТЧЕТ ЗАЩИЩЕН С ОЦЕНКОЙ _____

ПРЕПОДАВАТЕЛЬ

ассистент		Н.С.Красников
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ № 2

РАЗРАБОТКА ОДНОСЛОЙНОЙ НЕЙРОНОЙ СЕТИ ПРЯМОГО РАСПРОСТРАНЕНИЯ СИГНАЛА

по курсу: ОСНОВЫ МАШИННОГО ОБУЧЕНИЯ

СТУДЕНТ ГР. № 3031	_____	М.В. Вдовин
номер группы	подпись, дата	инициалы, фамилия

Санкт-Петербург
2023

1. **Цель работы:**

Изучение свойств нейрона (персептрона Розенблатта). Приобретение навыков разработки однослойных нейронных сетей в объектно-ориентированных средах программирования

2. **Задание:**

1. Изучить свойства персептрона Розенблатта и нейронной сети прямого распространения сигнала.
2. Создать входные данные (файл с весовыми коэффициентами нейрона) для обучения нейрона со случайными значениями.
3. В программной среде MS Visual Studio (или иной объектноориентированной IDE) создать нейрон, реализующий функцию распознавания заданного объекта датасета в соответствии со своим вариантом из лабораторной работы №1.
4. Обучить нейрон, используя правило Хебба. Выполнить обучение нейрона для классификации векторов (изображения объекта датасета) на две категории (верно/не верно).
5. Создать слой нейронов и выполнить имитацию работы однослойной нейронной сети для всех классов датасета, созданного в лабораторной работе №1.
6. График изменения loss-функции в ходе обучения сети следует выводить на отдельной форме приложения и, при необходимости, иметь возможность его со хранить.
7. Рассчитать метрики качества классификации, созданной однослойной сети как на обучающей, таки и на тестовых выборках.
8. Изучить возможности однослойных нейронных сетей решать линейно несепарабельные задачи.
9. Реализовать (при необходимости) многослойный персептрон, обеспечивающий после обучения значение метрики Accuracy не ниже 0,95 для каждого класса.
10. Оформить отчет по лабораторной работе.

3. Ход работы

3.1. Теоретические сведения

Перцептрон – это простейший вид искусственной нейронной сети, предложенный Френком Розенблаттом в 1957 году. Он является базовым строительным блоком для более сложных нейронных сетей. Перцептрон представляет собой математическую модель нейрона. Он принимает входные сигналы, умножает их на соответствующие веса, суммирует результаты и передает полученное значение через функцию активации – Рисунок 1. Веса в перцептроне определяются в процессе обучения, когда сеть настраивается на определенный набор данных. Простой перцептрон обычно используется для решения задач бинарной классификации, где целью является разделение двух классов на основе входных данных.

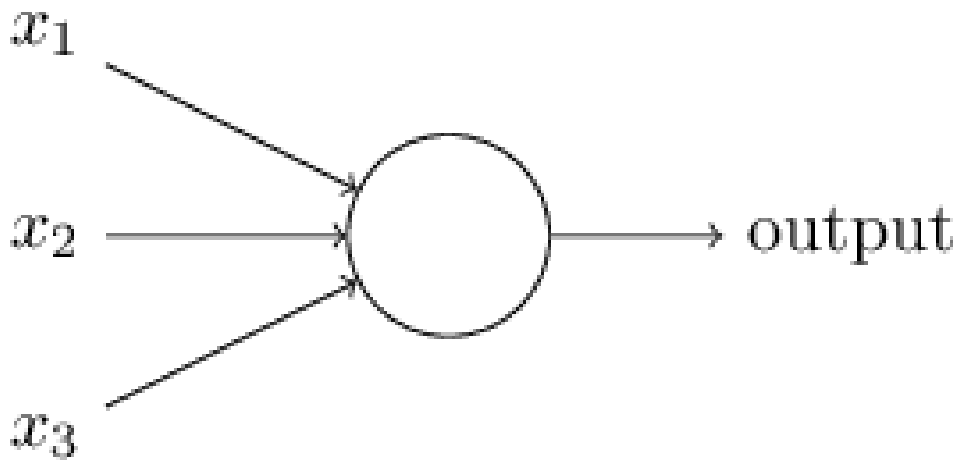

Рисунок 1. Перцептрон

Выход перцептрона можно представить либо в виде системы двух неравенств, где левая часть представляет сумму всех произведений весов и выходов нейрона:

$$output = \begin{cases} 0 \; if \;\; \sum_j w_j x_j \leq threshold \\ 1 \; if \;\; \sum_j w_j x_j > threshold \end{cases}$$

Где threshold – некоторый порог. Либо через скалярное произведение вместо суммы произведений и смещение перцептрона вместо порога:

$$output = \begin{cases} 0 \ if \ w \cdot x + b \leq 0 \\ 1 \ if \ w \cdot x + b > 0 \end{cases}$$

Сегмоидный нейрон (или нейрон с сигмоидной функцией активации) — это тип нейрона в искусственных нейронных сетях, который использует сигмоидную функцию активации для преобразования взвешенной суммы входных сигналов в выходной сигнал.

Сигмоидная функция активации обычно представляет собой S-образную кривую и принимает входные значения из любого диапазона и преобразует их в диапазон между 0 и 1.

Нейронная сеть прямого сигнала, также известная как feedforward neural network, представляет собой архитектуру искусственной нейронной сети, где данные перемещаются вперед, от входного слоя к выходному слою, без обратных связей. Это означает, что информация передается только в одном направлении, от входа к выходу, без циклических соединений.

Основные компоненты нейронной сети прямого распространения сигнала включают:

1. Входной слой (Input Layer): на этом слое находятся нейроны, которые принимают входные данные. Каждый нейрон в этом слое представляет собой отдельный признак входных данных.

2. Скрытые слои (Hidden Layers): эти слои находятся между входным и выходным слоями. Каждый нейрон в скрытом слое связан с каждым нейроном предыдущего и следующего слоев. Многослойные нейронные сети могут иметь несколько скрытых слоев.

3. Выходной слой (Output Layer): этот слой представляет собой результаты работы сети. Каждый нейрон в выходном слое обычно представляет собой

определенный класс или значение, в зависимости от типа задачи (классификация, регрессия и т. д.).

4. Веса и смещения (Weights and Biases): каждая связь между нейронами имеет свой вес, который отражает важность этой связи. Также каждый нейрон имеет свое смещение (bias), что позволяет учесть некоторую степень нелинейности в данных.

5. Функции активации: нейроны в каждом слое обычно применяют функции активации к своему выходу. Эти функции добавляют нелинейность в сеть, что позволяет ей моделировать сложные зависимости в данных.

Правило Хебба для нейронной сети прямого распространения (feedforward neural network) определяет, как изменяются веса связей между нейронами в процессе обучения. Это правило говорит о том, что вес связи увеличится, если активация нейрона *i* и *j* происходит одновременно (синхронно), и уменьшится, если активации происходят несинхронно. В результате обучения веса связей настраиваются таким образом, чтобы учитывать структуру входных данных и улучшать производительность сети на задаче, которую она решает.

3.2. Практическая часть

Создадим нейронную сеть, состоящую из 4 слоев: входной слой, который состоит из 784 нейрона, что равняется количеству пикселей на картинке размером 28 на 28, два промежуточных слоев размерами 128 и 32, и выходной слой, который состоит из 10 нейронов, что равняется цифрами от 0 до 9.

У нас имеется готовый датасет с прошлой лабораторной работы (Рисунок 2) и разметка для него (Рисунок 3).

Каждый сигмоидный нейрон в нашей нейронной сети выдает значение относительно сигмоидной (логической) функции:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}$$

В нашем случае значение сигмоидного нейрона с входными данными x_1, x_2 … , весами w_1, w_2 ... и смещением b будет считаться, как:

$$\frac{1}{1 + exp(-\sum_j w_j x_j - b)}$$

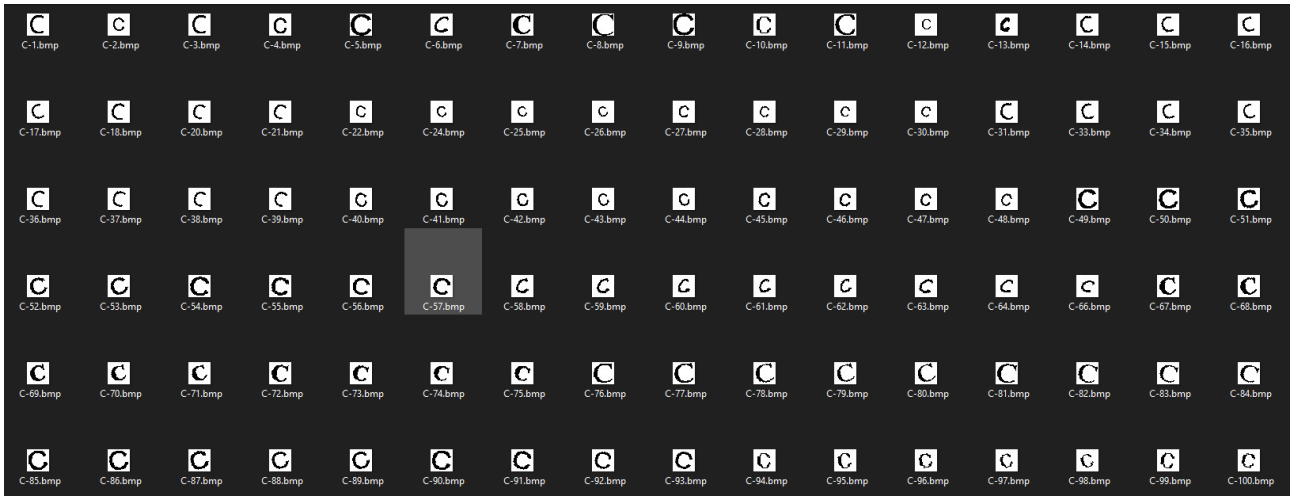

Рисунок 2. Датасет

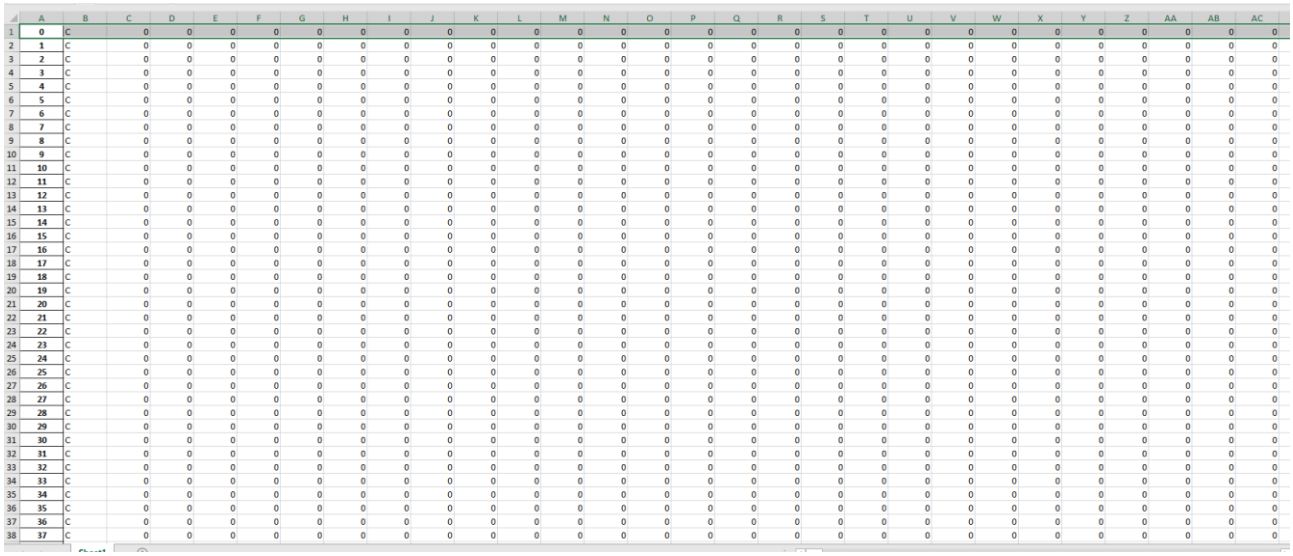

Рисунок 3. Разметка для датасета

Определим функцию стоимости или функцию потерь:

$$C(w,b) = \frac{1}{2n} \sum_x ||y(x)\check{}a||^2$$

Теперь наша цель в обучении нейронной сети сводится к поиску весов и смещений, минимизирующих квадратичную функцию стоимости.

Для минимизации будем использовать такое понятие, как градиентный спуск, который работает через последовательное вычисление градиента, и

последующее смещение в противоположном направлении, что приводит к «падению» по склону долины, которую образует наша функция от многих переменных.

Для ускорения обучения, можно использовать стохастический градиентный спуск работает через случайную выборку небольшого количества m обучающих входных данных. Мы назовём эти случайные данные X_1, X_2,.., X_m, и назовём их мини-пакетом. Если размер выборки m будет достаточно большим, среднее значение ∇C_{Xj} будет достаточно близким к среднему по всем ∇Cx, то есть:

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C$$

где вторая сумма идёт по всему набору обучающих данных. Поменяв части местами, мы получим:

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}$$

что подтверждает, что мы можем оценить общий градиент, вычислив градиенты для случайно выбранного мини-пакета.

Обучение происходит по методу прямого распространения, когда процесс передачи данных распространяется от входного нейрона через скрытые слои до выходного нейрона, при этом отсутствуют петли.

Обучение происходит в несколько эпох, которые представляют из себя один полный цикл обучения на всем наборе данных. В течение каждой эпохи веса и смещения обновляются на основе градиентного спуска.

После каждой эпохи вычисляется среднеквадратичная ошибка (MSE) и выполняется обратное распространение для обновления весов. Среднеквадратичная ошибка вычисляется по формуле:

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

где:

- N - количество примеров в наборе данных,
- y_i - фактическое значение для i-го примера,
- $\hat{y}_i$ - предсказанное значение для i-го примера.

Графики зависимости MSE от количества эпох приведен на Рисунок 5, а график зависимости точности от количества эпох на Рисунок 4.

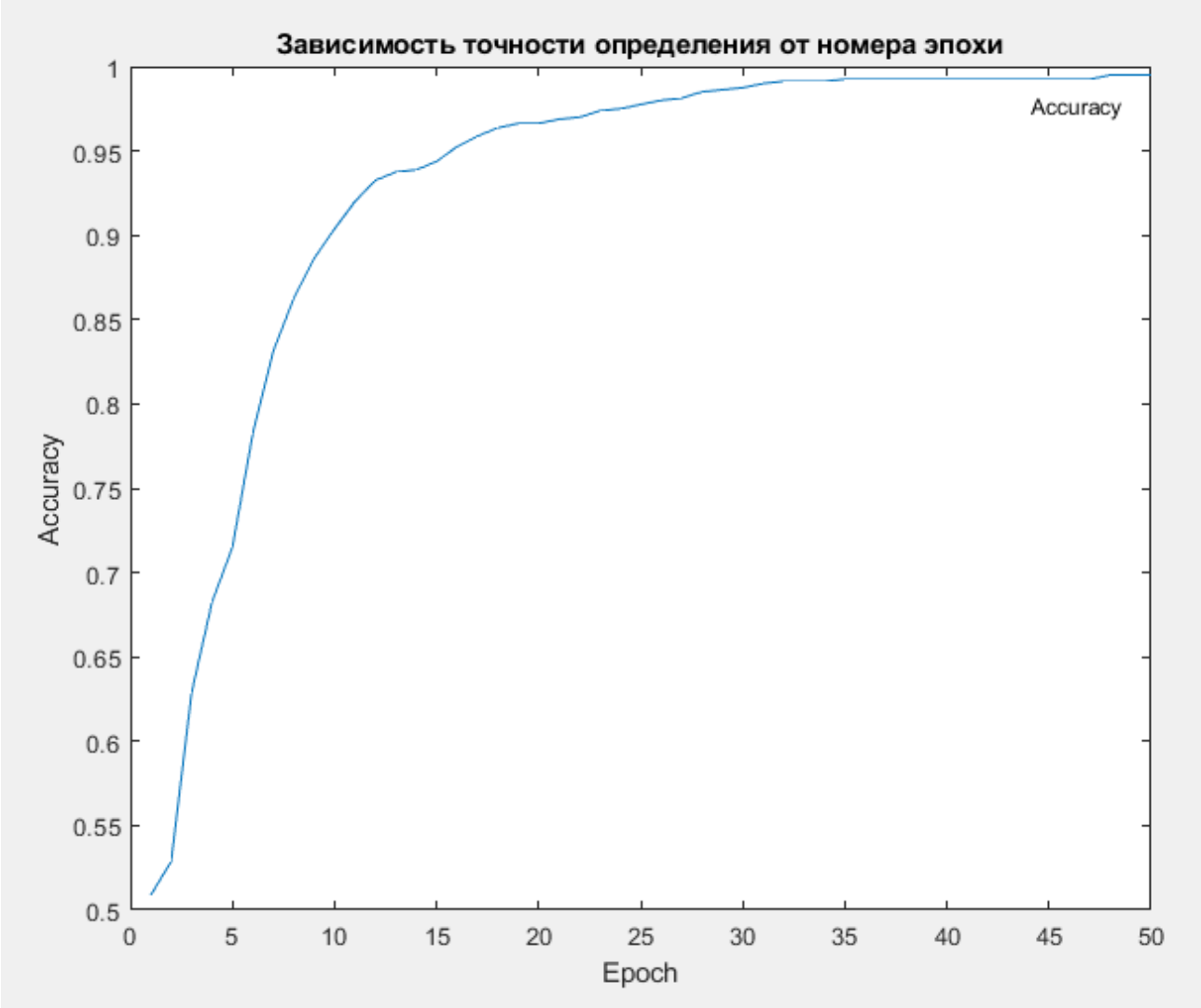

Рисунок 4. График зависимости точности от количества пройденных эпох

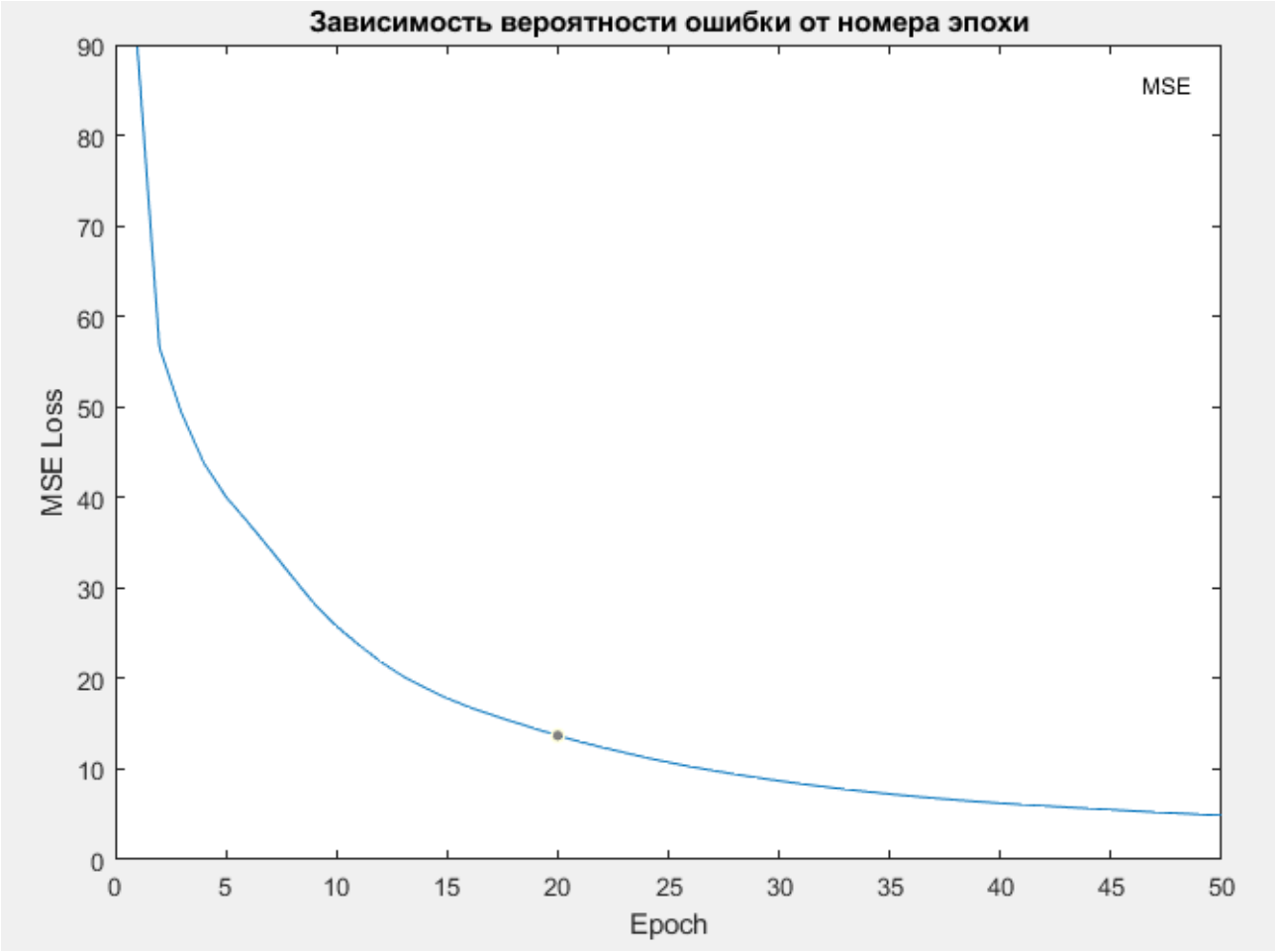

Рисунок 5. График зависимости вероятности ошибки от количества пройденных эпох

Вывод

В ходе лабораторной работы были изучены свойства перцептрона и сигмоидного нейрона. Была построена многослойная нейронная сеть в объектно-ориентированной среде программирования. По итогам испытаний 50 эпох, нейронная сеть научилась определять римские цифры с точностью, близкой к 1, а вероятность ошибки составляла меньше 5%.

Листинг кода

```
package com.ml.lab2;

import com.ml.lab2.network.NNetwork;
import com.ml.lab2.utils.Pair;

import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.function.UnaryOperator;

public class Main {
    public static final Integer COUNT_NEURONS_INPUT_LAYER = 784;
    public static final String datasetNameFile = "Dataset.csv";
    public static final String graphAccuracyFileName = "graphAccuracy.txt";
    public static final String graphMSELossFileName = "graphMSELoss.txt";

    public static UnaryOperator<Double> sigmoid = x -> 1 / (1 + Math.exp(-x));
    public static UnaryOperator<Double> dsigmoid = y -> y * (1 - y);

    public static void main(String[] args) {
        try (FileInputStream dataFromDataset = new FileInputStream(datasetNameFile);
             DataInputStream inputDataset = new DataInputStream(dataFromDataset);
             PrintWriter writerAccuracy = new PrintWriter(graphAccuracyFileName);
             PrintWriter writerMSE = new PrintWriter(graphMSELossFileName)) {

            NNetwork nnetwork = new NNetwork(0.001, sigmoid, dsigmoid, COUNT_NEURONS_INPUT_LAYER, 512, 128,
32, 10);

            /*          int samples = 1000;
            Pair[] arrValObj = new Pair[samples];
            double[][] inputs = new double[samples][NEURONS];

            String currLine;
            String[] lines;
            while ((currLine = inputDataset.readLine()) != null) {
                lines = currLine.split(",");
                int sampInd = Integer.parseInt(lines[0]);
                if (sampInd != samples) {
                    int digitValObj = (int) (Math.ceil(sampInd / 100)) + 1;
                    Pair<Integer, String> validObject = new Pair(digitValObj, lines[1]);
                    arrValObj[sampInd] = validObject;
                    for (int j = 0; j < NEURONS; j++) {
                        inputs[sampInd][j] = Integer.parseInt(lines[j + 2]);
                    }
                }
            }

            int epochs = 30;
            for (int i = 1; i <= epochs; i++) {
                double right = 0;
                double errorSum = 0;
                double accuracy = 0;
                int batchSize = 1000;

                for (int j = 0; j < batchSize; j++) {
                    int imgIndex = j;
                    double[] targets = new double[10];
                    int digit = (int) arrValObj[imgIndex].getFirst();
                    targets[digit] = 1;

                    double[] outputs = nnetwork.feedForward(inputs[imgIndex]);
                    int maxDigit = 0;
                    double maxDigitWeight = -1;
                    for (int k = 0; k < 10; k++) {
                        if (outputs[k] > maxDigitWeight) {
                            maxDigitWeight = outputs[k];
```

```
                    maxDigit = k;
                }
            }
            if (digit == maxDigit) right++;
            for (int k = 0; k < 10; k++) {
                errorSum += (targets[k] - outputs[k]) * (targets[k] - outputs[k]);
            }
            nnetwork.backpropagation(targets);
            accuracy = right / batchSize;
        }
        System.out.println("Epoch: " + i + ". Accuracy: " + accuracy + ". MSE Loss: " + errorSum / 10);
        writerAccuracy.println(accuracy);
        writeMSE.println(errorSum / 10);*/

    int samples = 800;
    Pair[] arrValObj = new Pair[samples];
    double[][] inputs = new double[samples][COUNT_NEURONS_INPUT_LAYER];

    String thisLine;
    String[] line;
    while ((thisLine = inputDataset.readLine()) != null) {
        line = thisLine.split(",");
        int sampInd = Integer.parseInt(line[0]);

        if (sampInd == 800) break;
        int digitValObj = (int) (Math.ceil(sampInd / 100)) + 1;
        Pair<Integer, String> validObject = new Pair(digitValObj, line[1]);
        arrValObj[sampInd] = validObject;
        for (int j = 0; j < COUNT_NEURONS_INPUT_LAYER; j++) {
            inputs[sampInd][j] = Integer.parseInt(line[j + 2]);
        }
    }

    int epochs = 50;
    for (int i = 1; i <= epochs; i++)  {
        double right = 0;
        double errorSum = 0;
        double accuracy = 0;
        int batchSize = 800;

        for (int j = 0; j < batchSize; j++) {
            int imgIndex = j;
            double[] targets = new double[10];
            int digit = (int) arrValObj[imgIndex].getFirst();
            targets[digit] = 1;

            double[] outputs = nnetwork.feedForward(inputs[imgIndex]);
            int maxDigit = 0;
            double maxDigitWeight = -1;
            for (int k = 0; k < 10; k++) {
                if (outputs[k] > maxDigitWeight) {
                    maxDigitWeight = outputs[k];
                    maxDigit = k;
                }
            }
            if (digit == maxDigit) right++;
            for (int k = 0; k < 10; k++) {
                errorSum += (targets[k] - outputs[k]) * (targets[k] - outputs[k]);
            }
            nnetwork.backpropagation(targets);
            accuracy = right / batchSize;
        }
        //System.out.println("Epoch: " + i + ". Accuracy: " + accuracy + ". MSE Loss: " + errorSum/10);
        System.out.println(i + " " + errorSum/10);
//          System.out.println(i + " " + errorSum / 10);

        writerAccuracy.println(accuracy);

        writerMSE.println(errorSum / 10);
    }
```

```
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
package com.ml.lab2.network;

import java.util.function.UnaryOperator;

public class NNetwork {
    private double learningRate;
    private Layer[] layers;
    private UnaryOperator<Double> fActivation;
    private UnaryOperator<Double> dActivation;

    public NNetwork(double learningRate,
                    UnaryOperator<Double> fActivation,
                    UnaryOperator<Double> dActivation,
                    int... sizes) {
        this.learningRate = learningRate;
        this.fActivation = fActivation;
        this.dActivation = dActivation;
        this.layers = new Layer[sizes.length];
        initBiasesAndWeights(sizes);
    }

    public double[] feedForward(double[] inputs) {
        System.arraycopy(inputs, 0, layers[0].neurons, 0, inputs.length);
        for (int i = 1; i < layers.length; i++) {
            Layer l = layers[i - 1];
            Layer l1 = layers[i];
            for (int j = 0; j < l1.size; j++) {
                l1.neurons[j] = 0;
                for (int k = 0; k < l.size; k++) {
                    l1.neurons[j] += l.neurons[k] * l.weights[k][j];
                }
                l1.neurons[j] += l1.biases[j];
                l1.neurons[j] = fActivation.apply(l1.neurons[j]);
            }
        }
        return layers[layers.length - 1].neurons;
    }

    public void backpropagation(double[] targets) {
        double[] errors = new double[layers[layers.length - 1].size];
        for (int i = 0; i < layers[layers.length - 1].size; i++) {
            errors[i] = targets[i] - layers[layers.length - 1].neurons[i];
        }
        for (int k = layers.length - 2; k >= 0; k--) {
            Layer l = layers[k];
            Layer l1 = layers[k + 1];
            double[] errorsNext = new double[l.size];
            double[] gradients = new double[l1.size];
            for (int i = 0; i < l1.size; i++) {
                gradients[i] = errors[i] * dActivation.apply(layers[k + 1].neurons[i]);
                gradients[i] *= learningRate;
            }
            double[][] deltas = new double[l1.size][l.size];
            for (int i = 0; i < l1.size; i++) {
                for (int j = 0; j < l.size; j++) {
                    deltas[i][j] = gradients[i] * l.neurons[j];
                }
            }
            for (int i = 0; i < l.size; i++) {
                errorsNext[i] = 0;
                for (int j = 0; j < l1.size; j++) {
                    errorsNext[i] += l.weights[i][j] * errors[j];
                }
            }
            errors = new double[l.size];
```

```
                System.arraycopy(errorsNext, 0, errors, 0, l.size);
                double[][] weightsNew = new double[l.weights.length][l.weights[0].length];
                for (int i = 0; i < l1.size; i++) {
                    for (int j = 0; j < l.size; j++) {
                        weightsNew[j][i] = l.weights[j][i] + deltas[i][j];
                    }
                }
                l.weights = weightsNew;
                for (int i = 0; i < l1.size; i++) {
                    l1.biases[i] += gradients[i];
                }
            }
        }

        private void initBiasesAndWeights(int... sizes) {
            for (int i = 0; i < sizes.length; i++) {
                int nextSize = 0;
                if (i < sizes.length - 1) {
                    nextSize = sizes[i + 1];
                }
                layers[i] = new Layer(sizes[i], nextSize);
                for (int j = 0; j < sizes[i]; j++) {
                    layers[i].biases[j] = Math.random() * 2.0 - 1.0;
                    for (int k = 0; k < nextSize; k++) {
                        layers[i].weights[j][k] = Math.random() * 2.0 - 1.0;
                    }
                }
            }
        }
    }
    package com.ml.lab2.utils;

public class Pair<T, U> {

    private T first;
    private U second;

    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public void setFirst(T first) {
        this.first = first;
    }

    public U getSecond() {
        return second;
    }

    public void setSecond(U second) {
        this.second = second;
    }
}
    package com.ml.lab2.network;

public class Layer {
    public int size;
    public double[] neurons;
    public double[] biases;
    public double[][] weights;

    public Layer(int size, int nextSize) {
        this.size = size;
        neurons = new double[size];
        biases = new double[size];
```

```
        weights = new double[size][nextSize];
    }
}
```