**Министерство науки и высшего образования Российской Федерации**
**Федеральное государственное бюджетное образовательное учреждение высшего образования**
**«Московский государственный технический университет имени Н.Э. Баумана**
**(национальный исследовательский университет)»**
**(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

# Лабораторная работа № 2

## по курсу «Теория искусственных нейронных сетей»

«Разработка многослойного персептрона на основе обратного распространения ошибки FFNN»

Студентка группы ИУ9-72Б Самохвалова П. С.

Преподаватель Каганов Ю. Т.

*Москва 2023*

# 1 Цель работы

Изучение многослойного персептрона, исследование его работы на основе использования различных методов оптимизации и целевых функций.

# 2 Задание

- Реализовать на языке высокого уровня многослойный персептрон и проверить его работоспособность на примере данных, выбранных из MNIST dataset.
- Исследовать работу персептрона на основе использования различных целевых функций. (среднеквадратичная ошибка, перекрестная энтропия, дивергенция Кульбака-Лейблера).
- Исследовать работу многослойного персептрона с использованием различных методов оптимизации (градиентный, Флетчера-Ривза (FR), Бройдена-Флетчера-Гольдфарба-Шенно (BFGS)).
- Подготовить отчет с распечаткой текста программы, графиками результатов исследования и анализом результатов.

# 3 Практическая реализация

Исходный код программы представлен в листинге 1.

Листинг 1: Многослойный персептрон

```
import random
import pickle
import gzip
import numpy as np
from matplotlib import pyplot as plt


def load_data():
    with gzip.open('../data/mnist.pkl.gz', 'rb') as f:
        (training_data, validation_data, test_data) = pickle.load(f,
    encoding='latin1')
```

```python
    return (training_data, validation_data, test_data)


def load_data_wrapper():
    tr_d, va_d, te_d = load_data()
    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
    training_results = [vectorized_result(y) for y in tr_d[1]]
    training_data = list(zip(training_inputs, training_results))
    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
    validation_data = list(zip(validation_inputs, va_d[1]))
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = list(zip(test_inputs, te_d[1]))
    return (training_data, validation_data, test_data)


def vectorized_result(j):
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e


def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def sigmoid_der(x):
    return sigmoid(x) * (1 - sigmoid(x))


def relu(x):
    x = x.flatten()
    return np.array([[max(c, 0)] for c in x])


def relu_der(x):
    x = x.flatten()
    return np.array([[1 if c > 0 else 0] for c in x])


def softmax(z):
    e_z = np.exp(z - np.max(z))
    return e_z / e_z.sum()


def softmax_der(z):
    s = softmax(z)
```

```
    return np.diag(s) - np.outer(s, s)


def mse(y_true, y_received):
    return np.linalg.norm(y_true - y_received)


def mse_der(y_true, y_received):
    return y_true - y_received


def categorical_cross_entropy(y0, y):
    return -(y0 * np.log(y) + (1 - y0) * np.log(1 - y))


def categorical_cross_entropy_der(y0, y):
    return -(y0 / y - (1 - y0) / (1 - y))


def kl_divergence(y0, y):
    if y0 == 0:
        return 0
    else:
        return y0 * np.log(y0 / y)


def kl_divergence_der(y0, y):
    return np.log(y0 / y) + 1


class MultilayerPerceptron(object):

    def __init__(self, sizes, optimization_method, activation_function, loss_function):
        self.num_layers = len(sizes)
        self.optimization_method = optimization_method
        self.activation_function = activation_function
        if activation_function == sigmoid:
            self.activation_function_der = sigmoid_der
        elif activation_function == relu:
            self.activation_function_der = relu_der
        elif activation_function == softmax:
            self.activation_function_der = softmax_der
        self.loss_function = loss_function
        if loss_function == mse:
            self.loss_function_der = mse_der
```

```
        elif loss_function == categorical_cross_entropy:
            self.loss_function_der = categorical_cross_entropy_der
        elif self.loss_function == kl_divergence:
            self.loss_function_der = kl_divergence_der
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]
        self.loss = []

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = self.activation_function(np.dot(w, a) + b)
        return a

    def train(self, training_data, epochs, mini_batch_size, eta, test_data):
        if self.optimization_method == "gradient_descent":
            self.SGD(training_data, epochs, mini_batch_size, eta, test_data)
        elif self.optimization_method == "fletcher_reeves":
            self.fletcher_reeves_optimization(training_data, epochs, mini_batch_size, eta, test_data)
        elif self.optimization_method == "bfgs":
            self.bfgs(training_data, epochs, mini_batch_size, test_data)

    def SGD(self, training_data, epochs, mini_batch_size, eta, test_data):
        n = len(training_data)
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [training_data[k : k+mini_batch_size] for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            self.test(test_data)

    def update_mini_batch(self, mini_batch, eta):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
```

```
        self.weights = [w - (eta / len(mini_batch)) * nw for w, nw in
    zip(self.weights, nabla_w)]
        self.biases = [b - (eta / len(mini_batch)) * nb for b, nb in zip
    (self.biases, nabla_b)]

    def backprop(self, x, y):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]

        activation = x
        activations = [x]
        zs = []
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation) + b
            zs.append(z)
            activation = self.activation_function(z)
            activations.append(activation)

        delta = self.loss_function_der(activations[-1], y) * self.
    activation_function_der(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())

        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = self.activation_function_der(z)
            delta = np.dot(self.weights[-l + 1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l - 1].transpose())

        return (nabla_b, nabla_w)

    def fletcher_reeves(self, g, old_g, d):
        beta = np.dot(g.T, g) / np.dot(old_g.T, old_g)
        new_d = -g + beta * d
        return new_d

    def fletcher_reeves_optimization(self, training_data, epochs,
    mini_batch_size, eta, test_data):
        n = len(training_data)
        old_grad = [np.zeros(w.shape) for w in self.weights]
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [training_data[k : k+mini_batch_size] for k
    in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
```

```
                old_grad = self.update_mini_batch_fletcher_reeves(mini_batch, eta, old_grad)
            self.test(test_data)

    def update_mini_batch_fletcher_reeves(self, mini_batch, eta, old_grad):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

        grad = nabla_w
        d = [-g for g in grad]
        if old_grad[0][0][0] != 0:
            d = self.fletcher_reeves(grad, old_grad, d)
        for i in range(len(self.weights)):
            self.weights[i] += (eta / len(mini_batch)) * d[i]
        return grad

    def bfgs(self, training_data, epochs, mini_batch_size, test_data):

        H = np.identity(sum(self.sizes[1:]))
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [training_data[k: k + mini_batch_size] for k in range(0, len(training_data), mini_batch_size)]
            for mini_batch in mini_batches:
                H, nabla_b, nabla_w = self.update_bfgs(H, mini_batch)
                self.weights -= np.dot(H, nabla_w)
                self.biases -= np.dot(H, nabla_b)
            self.test(test_data)

    def update_bfgs(self, H, mini_batch):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
```

```
        gradient = np.concatenate((np.array(nabla_b).ravel(), np.array(nabla_w).ravel()))
        y = np.concatenate((np.array(nabla_b).ravel(), np.array(nabla_w).ravel()))
        s = y - gradient
        yTB = np.dot(y, H)
        yTBy = np.dot(yTB, y)
        Bs = np.dot(H, s)
        H += np.outer(y, y) / yTBy - np.outer(Bs, Bs) / np.dot(s, Bs)
        return H, nabla_b, nabla_w

    def test(self, test_data):
        n_test = len(test_data)
        s = 0
        for i in range(n_test):
            y_receivied = self.feedforward(test_data[i][0])
            y_true = vectorized_result(test_data[i][1])
            s += self.loss_function(y_true, y_receivied)
        s /= n_test
        self.loss.append(s)

    def get_loss(self, i):
        return self.loss[i]

    def recognize(self, test_example):
        return np.argmax(self.feedforward(test_example))


epochs = 100
eta = 0.1


training_data, validation_data, test_data = load_data_wrapper()

perceptron = MultilayerPerceptron([784, 8, 10], "gradient_descent", sigmoid, mse)

perceptron.train(training_data, epochs, 10, eta, test_data)

print(perceptron.weights)

plt.plot([i for i in range(epochs)], [perceptron.get_loss(i) for i in range(epochs)], label='Gradient', color='blue')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss and Epochs')
```

```
plt.legend()
plt.show()

print()
for i in range(5):
    print("Expected:", test_data[i][1])
    print("Receivied:", perceptron.recognize(test_data[i][0]))
    print()


# training_data, validation_data, test_data = load_data_wrapper()
#
# perceptron = MultilayerPerceptron([784, 8, 10], "fletcher_reeves",
    sigmoid, mse)
#
# perceptron.train(training_data, epochs, 10, eta, test_data)
#
# print(perceptron.weights)
#
# plt.plot([i for i in range(epochs)], [perceptron.get_loss(i) for i in
    range(epochs)], label='Fletcher-Reeves optimization', color='green')
# plt.xlabel('Epochs')
# plt.ylabel('Loss')
# plt.title('Loss and Epochs')
# plt.legend()
# plt.show()
#
# print()
# for i in range(5):
#     print("Expected:", test_data[i][1])
#     print("Receivied:", perceptron.recognize(test_data[i][0]))
#     print()
```

# 4 Результаты

Результаты работы программы представлены на рисунках 1 – 4.

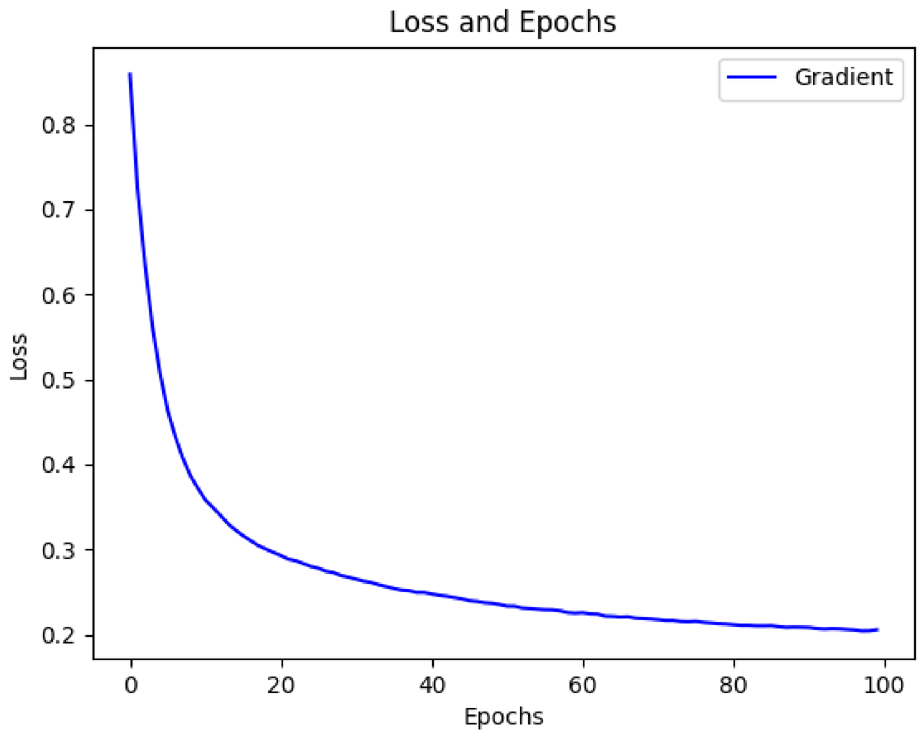


Рис. 1 — График зависимости функции потерь от числа эпох

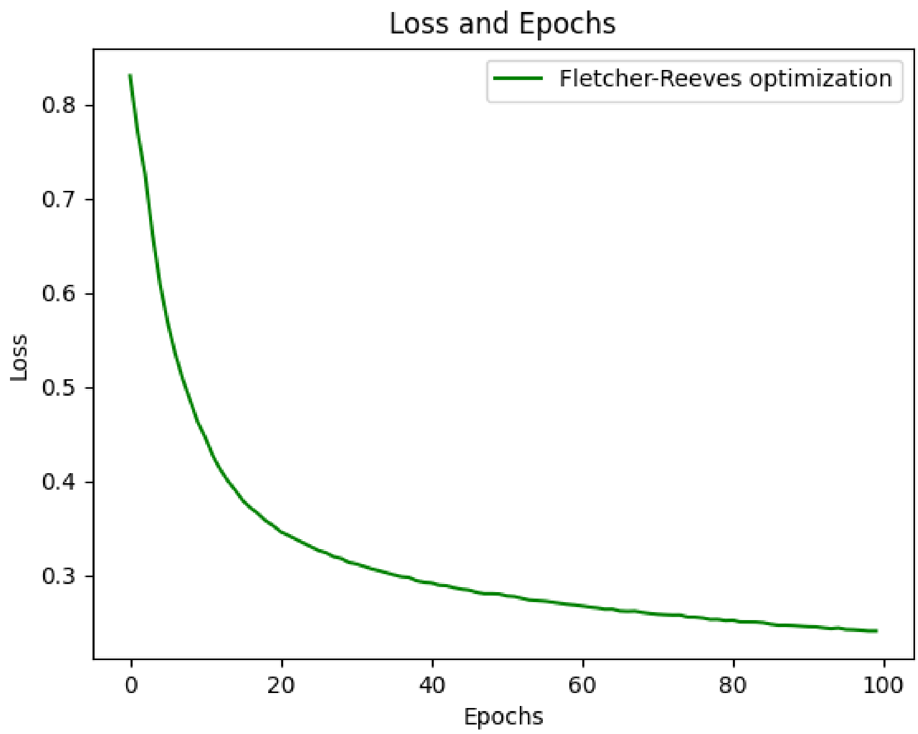


Рис. 2 — График зависимости функции потерь от числа эпох

```
[array([[-1.87237693, -0.65443321,  1.74505721, ...,  0.21916059,
        -0.34381263, -0.90158286],
       [-0.16731531,  0.53418319, -0.75857503, ...,  1.982786  ,
         0.17231751, -0.85635505],
       [-0.12715591, -0.28470413, -0.04934877, ..., -1.18526192,
        -0.29120575, -1.20825856],
       ...,
       [-1.88428879, -0.58762552,  0.27691955, ...,  0.97638738,
        -0.14732429, -1.43191046],
       [ 1.11330164,  1.63602621,  0.65914208, ..., -2.78345617,
        -1.37100326, -0.0151105 ],
       [-1.93877633,  1.5594095 , -1.15605335, ...,  0.57309025,
         0.3499533 , -0.04842958]]), array([[-0.0329875 , -3.86246728, -4.45963898,  5.2866286 , -6.67570569,
        -5.46785083, -2.05752451, -4.03071084],
       [-5.68074171,  2.65377146,  4.78774288, -1.21943296,  2.59713194,
        -4.57549858,  5.23985109,  2.36023706],
       [-2.12855988,  5.73462403, -3.25934159,  1.92007323, -4.63706666,
        -2.55723272, -4.10814598,  2.86601569],
       [-4.04510758, -3.43941021, -5.19829351,  3.42348007,  4.54389593,
        -2.11264921, -1.37669811,  5.42792948],
       [-2.22731175, -2.04446508, -0.0449339 , -6.30427681, -0.43719859,
         6.66399441, -4.01037598,  2.28917103],
       [-0.94633714,  0.42851234,  3.27641775, -0.71704505,  4.89626002,
        -3.06811846, -2.45641278, -5.93253553],
       [ 0.98220577, -4.51487224, -4.27724841, -6.4658062 , -5.05851261,
        -5.90075996, -1.65913966,  0.71258143],
       [-2.48049576,  0.66516471, -1.48114767,  1.96679317, -4.00865582,
         6.57239063,  6.27648945, -5.00501119],
       [-3.76350397, -5.35592019,  5.08037741,  2.65079541, -3.62330541,
```

Рис. 3 — Полученные веса

```
Expected: 7
Receivied: 7

Expected: 2
Receivied: 2

Expected: 1
Receivied: 1

Expected: 0
Receivied: 0

Expected: 4
Receivied: 4
```

Рис. 4 — Пример распознавания цифр

# 5 Выводы

В результате выполнения лабораторной работы был реализован многослойный персептрон, были реализованы различные функции активации, функции потерь и оптимизации.