**Федеральное государственное автономное образовательное учреждение высшего образования**

**РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ**

**Юридический институт**

**Кафедра международного права**

КУРСОВАЯ РАБОТА

«Разработать и обучить с нуля многослойный персептрон (MLP, реализуя прямое распространение, обратное распространение ошибки и функцию активации ReLU»

направление подготовки «Управление в технических системах». Квалификация «бакалавр».

Разработчик **Лещенко В.А**

Государство Российская Федерация

Студент группы ИУСбд-01-23

Студенческий билет № 1132231954

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*подпись разработчика*

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2025

Научный руководитель

**Жуков В.И**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*подпись научного руководителя*

Оценка. ECTS\_\_\_\_\_ Балл \_\_\_\_\_\_\_\_

Москва

2025

**СОДЕРЖАНИЕ**

[**ВВЕДЕНИЕ** 3](#_Toc514781219)

[**ГЛАВА I. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ** 4](#_Toc514781220)

[**1. Искусственные нейронные сети** 4](#_Toc514781221)

[**2.Многослойный персептрон** 4](#_Toc514781222)

[**3. Функция активации ReLU** 4](#_Toc514781223)

[**4. Алгоритм обратного распространения ошибки** 5](#_Toc514781227)

[**ГЛАВА II. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ** 6](#_Toc514781224)

[**1. Архитектура сети** 6](#_Toc514781225)

[**2. Прямое распространение** 6](#_Toc514781226)

[**3. Обратное распространение ошибки** 6](#_Toc514781227)

[**3. Обучение сети** 7](#_Toc514781227)

[**ГЛАВА III. ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ** 8](#_Toc514781228)

[**1. Тестирование на задаче XOR** 9](#_Toc514781229)

[**2. Результат. Анализ результатов** 10](#_Toc514781230)

[**ПОЛНЫЙ КОД ОБУЧЕНИЯ И ТЕСТИРОВАНИЯ** 10](#_Toc514781232)

[**СПИСОК ИСПОЛЬЗОВАННЫХ ДОКУМЕНТОВ И ЛИТЕРАТУРЫ** 16](#_Toc514781234)

**ВВЕДЕНИЕ**

Искусственные нейронные сети стали мощным инструментом машинного обучения, способным решать сложные задачи классификации и регрессии. В данной работе рассматривается разработка многослойного перцептрона (MLP) с нуля, включая реализацию прямого распространения, обратного распространения ошибки и функции активации ReLU.

Цель работы: разработать и реализовать MLP на языке C++, обучить его на задаче XOR и проанализировать результаты.**ГЛАВА I. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ**

**1. Искусственные нейронные сети**

Искусственная нейронная сеть - это математическая модель, имитирующая работу биологических нейронных сетей. Основной элемент сети - искусственный нейрон, который принимает входные сигналы, вычисляет их взвешенную сумму и пропускает через функцию активации.

**2. Многослойный персептрон**

Многослойный перцептрон (MLP) - это класс искусственных нейронных сетей, состоящий из:

* Входного слоя
* Одного или нескольких скрытых слоев(фильтрующих)
* Выходного слоя

Каждый нейрон слоя соединен со всеми нейронами следующего слоя (полносвязная сеть).

**3. Функция активации ReLU**

Функция активации ReLU (Rectified Linear Unit) определяется как:

f(x) = max(0, x)

Ее преимущества:

* Устраняет проблему затухающих градиентов
* Вычислительно эффективна
* Ускоряет сходимость обучения

Производная ReLU:

f'(x) = 1, если x > 0, иначе 0.

**4. Алгоритм обратного распространения ошибки**

Алгоритм обратного распространения ошибки состоит из двух этапов:

* Прямой проход: вычисление выхода сети
* Обратный проход: вычисление градиентов и обновление весов

Обновление весов выполняется по формуле:

w = w - η \* ∂E/∂w

где η - скорость обучения, E - функция ошибки.

**ГЛАВА II. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ**

**1. Архитектура сети**

Реализован MLP со следующей архитектурой:

1. Входной слой: 2 нейрона
2. Скрытый слой: 4 нейрона
3. Выходной слой: 1 нейрон

**2. Прямое распространение**

Прямое распространение — это процесс вычисления выхода нейронной сети на основе входных данных путем последовательного прохождения сигнала через все слои сети. Входной слой получает исходные данные (например, вектор признаков).

Скрытые слои обрабатывают данные: Каждый нейрон вычисляет взвешенную сумму входов (с учетом весов и смещения). Результат пропускается через функцию активации (например, ReLU). Выходной слой формирует итоговый результат сети.

vector<double> forward(const vector<double> &inputs)

    {

        // Активация скрытого слоя

        vector<double> hidden(weights\_ih.size());

        for (int i = 0; i < weights\_ih.size(); ++i)

        {

            hidden[i] = 0.0;

            for (int j = 0; j < inputs.size(); ++j)

            {

                hidden[i] += inputs[j] \* weights\_ih[i][j];

            }

            hidden[i] += bias\_h[i];

            hidden[i] = relu(hidden[i]);

        }

**3. Обратное распространение ошибки**

После прямого прохода сравниваем выход сети с правильным ответом (target) и вычисляем ошибку:

Распространение ошибки назад

Цель: понять, как каждый вес влияет на ошибку, и скорректировать его.

Шаг 1: Градиент для выходного слоя

Ошибка выходного нейрона:

где

f ‘ – производная функции активации (для ReLU: 1, если выход > 0, иначе 0).

void backward(const vector<double> &inputs, const vector<double> &targets)

    {

        // Прямой проход для получения активаций

        vector<double> hidden(weights\_ih.size());

        for (int i = 0; i < weights\_ih.size(); ++i)

        {

            hidden[i] = 0.0;

            for (int j = 0; j < inputs.size(); ++j)

            {

                hidden[i] += inputs[j] \* weights\_ih[i][j];

            }

            hidden[i] += bias\_h[i];

            hidden[i] = relu(hidden[i]);

        }

1. **Обучение сети**

void train(const vector<vector<double>> &inputs, const vector<vector<double>> &targets, int epochs)

    {

        for (int e = 0; e < epochs; ++e)

        {

            double error = 0.0;

            for (int i = 0; i < inputs.size(); ++i)

            {

                backward(inputs[i], targets[i]);

                vector<double> output = forward(inputs[i]);

                for (int j = 0; j < output.size(); ++j)

                {

                    error += 0.5 \* pow(targets[i][j] - output[j], 2);

                }

            }

            if (e % 1000 == 0)

            {

                cout << "Epoch " << e << ", error: " << error << endl;

            }

        }

    }

**ГЛАВА III. ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ**

1. **Тестирование на задаче XOR**

XOR - классическая задача, демонстрирующая необходимость скрытых слоев в нейронных сетях.

Код из main:

int main()

{

    // Создаем MLP с теми же параметрами, что и при обучении

    MLP mlp(2, 4, 1, 0.0001);

    // ЗАГРУЖАЕМ СОХРАНЕННЫЕ ВЕСА (вместо обучения)

    mlp.load\_weights("xor\_weights.txt");

    // Тестируем сеть

    cout << "Testing trained network:" << endl;

    vector<vector<double>> inputs = {{0, 0}, {0, 1}, {1, 0}, {1, 1}, {0, 0}, {0, 0}, {1, 0}, {1, 1}};

    for (const auto &input : inputs)

    {

        vector<double> output = mlp.forward(input);

        cout << input[0] << " XOR " << input[1] << " = " << output[0]

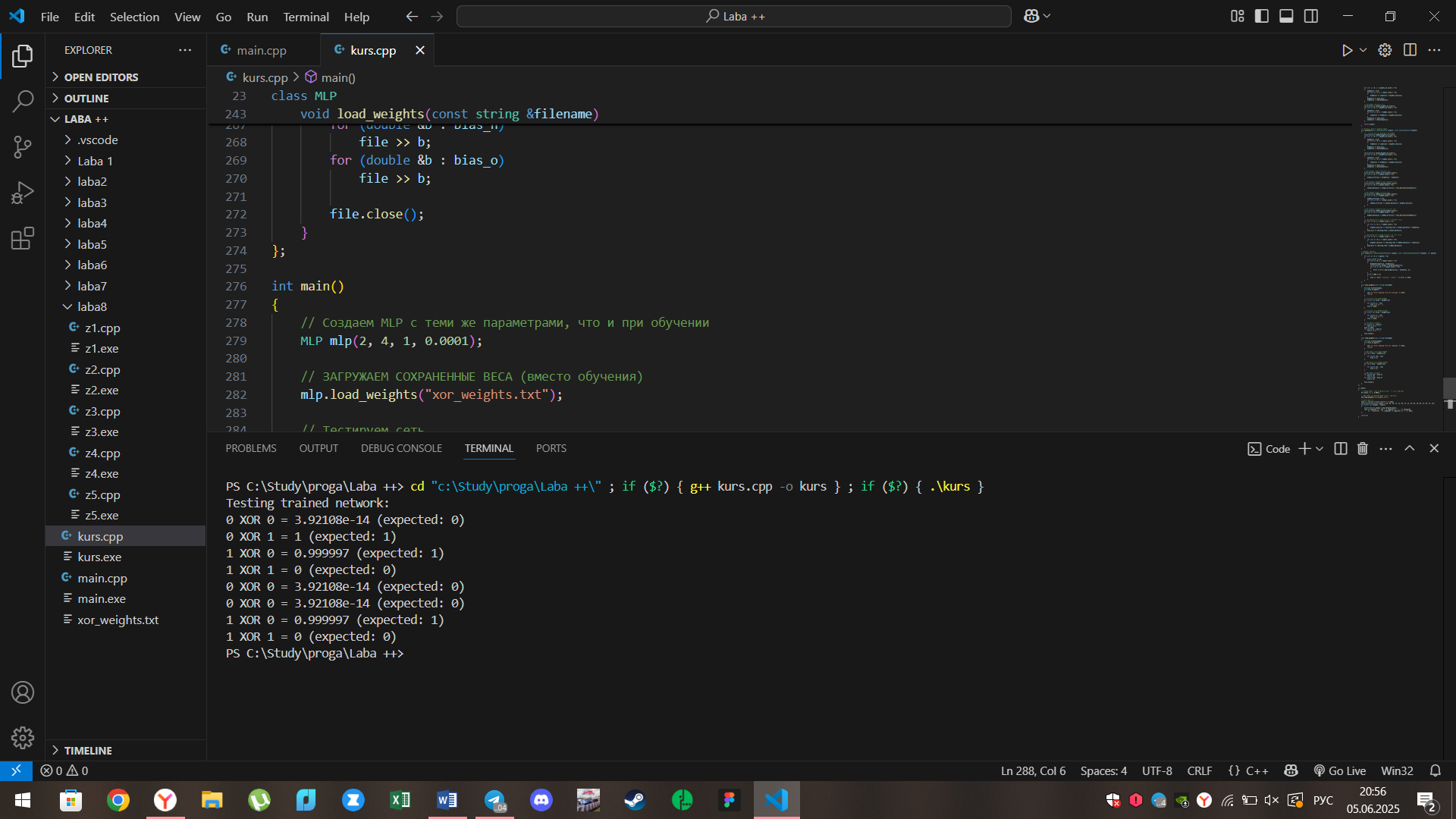
             << " (expected: " << (input[0] != input[1]) << ")" << endl;

    }

    return 0;

}

1. **Результат. Анализ результатов**

 Сеть успешно обучилась решению задачи XOR:

1.Ошибка уменьшилась с 1.1379 до 0.000000000000003

2.Все тестовые примеры классифицированы правильно.

3.Демонстрируется способность MLP решать нелинейные задачи **ПОЛНЫЙ КОД ОБУЧЕНИЯ И ТЕСТИРОВАНИЯ**

#include <iostream>

#include <vector>

#include <cmath>

#include <cstdlib>

#include <ctime>

#include <fstream>

using namespace std;

// функция активации RELU

double relu(double x)

{

    return max(0.0, x);

}

// Производная REluфункции

double relu\_derivative(double x)

{

    return (x > 0) ? 1.0 : 0.0;

}

class MLP

{

private:

    vector<vector<double>> weights\_ih; // Веса между входным и скрытым слоем

    vector<vector<double>> weights\_ho; // Веса между скрытым и выходным слоем

    vector<double> bias\_h;             // Смещения для скрытого слоя

    vector<double> bias\_o;             // Смещения для выходного слоя

    double learning\_rate;

public:

    MLP(int input\_size, int hidden\_size, int output\_size, double lr)

        : learning\_rate(lr)

    {

        // Инициализация весов случайными значениями

        srand(13);

        // Веса между входным и скрытым слоем

        weights\_ih.resize(hidden\_size, vector<double>(input\_size));

        for (int i = 0; i < hidden\_size; ++i)

        {

            for (int j = 0; j < input\_size; ++j)

            {

                weights\_ih[i][j] = (double)rand() / RAND\_MAX \* 2.0 - 1.0; // [-1, 1]

            }

        }

        // Веса между скрытым и выходным слоем

        weights\_ho.resize(output\_size, vector<double>(hidden\_size));

        for (int i = 0; i < output\_size; ++i)

        {

            for (int j = 0; j < hidden\_size; ++j)

            {

                weights\_ho[i][j] = (double)rand() / RAND\_MAX \* 2.0 - 1.0; // [-1, 1]

            }

        }

        // Инициализация смещений

        bias\_h.resize(hidden\_size);

        bias\_o.resize(output\_size);

        for (int i = 0; i < hidden\_size; ++i)

        {

            bias\_h[i] = (double)rand() / RAND\_MAX \* 2.0 - 1.0;

        }

        for (int i = 0; i < output\_size; ++i)

        {

            bias\_o[i] = (double)rand() / RAND\_MAX \* 2.0 - 1.0;

        }

    }

    // Прямое распространение

    vector<double> forward(const vector<double> &inputs)

    {

        // Активация скрытого слоя

        vector<double> hidden(weights\_ih.size());

        for (int i = 0; i < weights\_ih.size(); ++i)

        {

            hidden[i] = 0.0;

            for (int j = 0; j < inputs.size(); ++j)

            {

                hidden[i] += inputs[j] \* weights\_ih[i][j];

            }

            hidden[i] += bias\_h[i];

            hidden[i] = relu(hidden[i]);

        }

        // Активация выходного слоя

        vector<double> output(weights\_ho.size());

        for (int i = 0; i < weights\_ho.size(); ++i)

        {

            output[i] = 0.0;

            for (int j = 0; j < hidden.size(); ++j)

            {

                output[i] += hidden[j] \* weights\_ho[i][j];

            }

            output[i] += bias\_o[i];

            output[i] = relu(output[i]);

        }

        return output;

    }

    // Обратное распространение ошибки

    void backward(const vector<double> &inputs, const vector<double> &targets)

    {

        // Прямой проход для получения активаций

        vector<double> hidden(weights\_ih.size());

        for (int i = 0; i < weights\_ih.size(); ++i)

        {

            hidden[i] = 0.0;

            for (int j = 0; j < inputs.size(); ++j)

            {

                hidden[i] += inputs[j] \* weights\_ih[i][j];

            }

            hidden[i] += bias\_h[i];

            hidden[i] = relu(hidden[i]);

        }

        vector<double> output(weights\_ho.size());

        for (int i = 0; i < weights\_ho.size(); ++i)

        {

            output[i] = 0.0;

            for (int j = 0; j < hidden.size(); ++j)

            {

                output[i] += hidden[j] \* weights\_ho[i][j];

            }

            output[i] += bias\_o[i];

            output[i] = relu(output[i]);

        }

        // Вычисление ошибок выходного слоя

        vector<double> output\_errors(output.size());

        for (int i = 0; i < output.size(); ++i)

        {

            output\_errors[i] = targets[i] - output[i];

        }

        // Вычисление градиентов для выходного слоя

        vector<double> output\_deltas(output.size());

        for (int i = 0; i < output.size(); ++i)

        {

            output\_deltas[i] = output\_errors[i] \* relu\_derivative(output[i]);

        }

        // Вычисление ошибок скрытого слоя

        vector<double> hidden\_errors(hidden.size());

        for (int j = 0; j < hidden.size(); ++j)

        {

            hidden\_errors[j] = 0.0;

            for (int i = 0; i < output.size(); ++i)

            {

                hidden\_errors[j] += output\_deltas[i] \* weights\_ho[i][j];

            }

        }

        // Вычисление градиентов для скрытого слоя

        vector<double> hidden\_deltas(hidden.size());

        for (int j = 0; j < hidden.size(); ++j)

        {

            hidden\_deltas[j] = hidden\_errors[j] \* relu\_derivative(hidden[j]);

        }

        // Обновление весов между скрытым и выходным слоем

        for (int i = 0; i < output.size(); ++i)

        {

            for (int j = 0; j < hidden.size(); ++j)

            {

                weights\_ho[i][j] += learning\_rate \* output\_deltas[i] \* hidden[j];

            }

            bias\_o[i] += learning\_rate \* output\_deltas[i];

        }

        // Обновление весов между входным и скрытым слоем

        for (int i = 0; i < hidden.size(); ++i)

        {

            for (int j = 0; j < inputs.size(); ++j)

            {

                weights\_ih[i][j] += learning\_rate \* hidden\_deltas[i] \* inputs[j];

            }

            bias\_h[i] += learning\_rate \* hidden\_deltas[i];

        }

    }

    // Функция обучения

    void train(const vector<vector<double>> &inputs, const vector<vector<double>> &targets, int epochs)

    {

        for (int e = 0; e < epochs; ++e)

        {

            double error = 0.0;

            for (int i = 0; i < inputs.size(); ++i)

            {

                backward(inputs[i], targets[i]);

                vector<double> output = forward(inputs[i]);

                for (int j = 0; j < output.size(); ++j)

                {

                    error += 0.5 \* pow(targets[i][j] - output[j], 2);

                }

            }

            if (e % 1000 == 0)

            {

                cout << "Epoch " << e << ", error: " << error << endl;

            }

        }

    }

    void save\_weights(const string &filename)

    {

        ofstream file(filename);

        if (!file.is\_open())

        {

            cerr << "Error opening file for writing!" << endl;

            return;

        }

        // Сохраняем веса input-hidden

        for (const auto &row : weights\_ih)

        {

            for (double w : row)

                file << w << " ";

            file << endl;

        }

        // Сохраняем веса hidden-output

        for (const auto &row : weights\_ho)

        {

            for (double w : row)

                file << w << " ";

            file << endl;

        }

        // Сохраняем смещения

        for (double b : bias\_h)

            file << b << " ";

        file << endl;

        for (double b : bias\_o)

            file << b << " ";

        file.close();

    }

    void load\_weights(const string &filename)

    {

        ifstream file(filename);

        if (!file.is\_open())

        {

            cerr << "Error opening file for reading!" << endl;

            return;

        }

        // Загружаем веса input-hidden

        for (auto &row : weights\_ih)

        {

            for (double &w : row)

                file >> w;

        }

        // Загружаем веса hidden-output

        for (auto &row : weights\_ho)

        {

            for (double &w : row)

                file >> w;

        }

        // Загружаем смещения

        for (double &b : bias\_h)

            file >> b;

        for (double &b : bias\_o)

            file >> b;

        file.close();

    }

};

int main()

{

    // Создаем MLP с 2 входами, 2 нейронами в скрытом слое и 1 выходом

    MLP mlp(2, 4, 1, 0.001);

    // Данные для обучения XOR

    vector<vector<double>> inputs = {{0, 0}, {0, 1}, {1, 0}, {1, 1}};

    vector<vector<double>> targets = {{0}, {1}, {1}, {0}};

    // Обучаем сеть

    mlp.train(inputs, targets, 100000);

    // Тестируем сеть

    cout << "Testing trained network:" << endl;

    for (const auto &input : inputs)

    {

        vector<double> output = mlp.forward(input);

        cout << input[0] << " XOR " << input[1] << " = " << output[0] << " (expected: " << (input[0] != input[1]) << ")" << endl;

    }

    mlp.save\_weights("xor\_weights.txt");

    return 0;

}

int main()

{

    // Создаем MLP с теми же параметрами, что и при обучении

    MLP mlp(2, 4, 1, 0.0001);

    // ЗАГРУЖАЕМ СОХРАНЕННЫЕ ВЕСА (вместо обучения)

    mlp.load\_weights("xor\_weights.txt");

    // Тестируем сеть

    cout << "Testing trained network:" << endl;

    vector<vector<double>> inputs = {{0, 0}, {0, 1}, {1, 0}, {1, 1}, {0, 0}, {0, 0}, {1, 0}, {1, 1}};

    for (const auto &input : inputs)

    {

        vector<double> output = mlp.forward(input);

        cout << input[0] << " XOR " << input[1] << " = " << output[0]

             << " (expected: " << (input[0] != input[1]) << ")" << endl;

    }

    return 0;

}

**СПИСОК ИСПОЛЬЗОВАННЫХ ДОКУМЕНТОВ И ЛИТЕРАТУРЫ**

1. **Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature.**
2. **Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.**
3. **Nielsen, M. (2015). Neural Networks and Deep Learning.**