**Федеральное государственное автономное образовательное учреждение высшего образования**

**РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ**

**Юридический институт**

**Кафедра международного права**

КУРСОВАЯ РАБОТА

«Разработка и обучение многослойного персептрона»

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

Разработчик **Шуркин И. А.**

Государство Российская Федерация

Студент группы ИУСбд-02-23

Студенческий билет № 1132233614

Научный руководитель

**Жуков В. И.**

Москва

2025

**СОДЕРЖАНИЕ**

[**ВВЕДЕНИЕ** 4](#_Toc200047660)

[**ГЛАВА I. Теоретическая часть** 5](#_Toc200047661)

[**1. Терминология основных понятий** 5](#_Toc200047662)

[**2. Архитектура MLP и принципы работы** 6](#_Toc200047663)

[**3. Основные концепции MLP** 7](#_Toc200047664)

[**1. Код программы с обучением многослойного персептрона** 8](#_Toc200047665)

[**2. Скриншоты работы программы по обучению и решению тестовой выборки** 13](#_Toc200047666)

[**3. Код программы по тестированию обученного MLP** 13](#_Toc200047667)

[**ГЛАВА II. Ссылка на GitHub** 14](#_Toc200047668)

[**Заключение** 15](#_Toc200047668)

[**Список Литературы** 16](#_Toc200047668)

**ВВЕДЕНИЕ**

В современном мире машинное обучение играет ключевую роль во многих областях, таких как компьютерное зрение, обработка естественного языка и робототехника. Среди множества моделей машинного обучения, многослойный персептрон (MLP) занимает важное место благодаря своей простоте и способности заменять сложные нелинейные зависимости.

Данная курсовая работа посвящена разработке и обучению MLP на языке C++. Основной задачей работы является углубленное понимание принципов работы MLP, включая прямое распространение, обратное распространение ошибки и использование функции активации ReLU. Разработанная модель будет обучена на задаче XOR, а ее производительность будет оценена на тестовом наборе данных.

**Актуальность работы:** Углубленное понимание принципов работы MLP является важным для эффективного применения этой модели и разработки более сложных архитектур нейронных сетей. Реализация MLP с нуля позволяет лучше понять внутренние механизмы алгоритма и освоить ключевые концепции машинного обучения.

**Цель работы:** разработать и обучить с нуля многослойный персептрон (MLP) на языке C++, реализуя прямое распространение, обратное распространение ошибки и функцию активации ReLU.

**Задачи работы:**

Изучить теоретические основы многослойного персептрона (MLP).

Разработать архитектуру MLP.

Выбрать и подготовить данные для обучения и тестирования (XOR).

Обучить разработанный MLP на выбранном наборе данных.

Протестировать готовую модель персептрона.

**ГЛАВА I. Теоретическая часть**

Тема машинного обучения, интеллектуальных алгоритмов и искусственного интеллекта чрезвычайно популярна в наше время. Это легко можно увидеть, наблюдая за новостями на различных IT порталах. Подтверждает это и статистика.

**1. Терминология основных понятий**

Искусственный интеллект (ИИ) – это:

1. наука о создании интеллектуальных (умных) машин (чаще всего — компьютерных программ).

2. свойство интеллектуальной системы выполнять творческие функции, которые считаются прерогативой человека.

**Машинное обучение** — подраздел искусственного интеллекта, изучающий различные способы построения обучающихся алгоритмов.

**Искусственный нейрон** — упрощенная модель биологического нейрона.

**Искусственная нейронная сеть** — совокупность взаимодействующих между собой искусственных нейронов.

Персептрон (Perceptron) — простейший вид нейронных сетей. В основе лежит математическая модель восприятия информации мозгом, состоящая из сенсоров, ассоциативных и реагирующих элементов.

Многослойный персептрон по Розеблатту — персептрон, у которого имеется более 1 слоя скрытых элементов.

Многослойный персептрон по Румельхарту **—** многослойный персептрон по Розенблатту, у которого обучению подлежат еще и S-A связи, а также само обучение производится по методу обратного распространения ошибки.

**2. Архитектура MLP и принципы работы**

Типичная архитектура MLP состоит из трех основных типов слоев:

Входной слой (Input Layer): Входной слой принимает исходные данные, представляющие собой вектор признаков. Количество нейронов во входном слое соответствует количеству признаков в наборе данных. Нейроны входного слоя не выполняют никаких вычислений, а просто передают входные значения на следующий слой.

Скрытые слои (Hidden Layers): Скрытые слои являются ядром MLP, выполняющим сложные нелинейные преобразования входных данных. MLP может содержать один или несколько скрытых слоев. Каждый нейрон в скрытом слое получает взвешенную сумму выходов всех нейронов предыдущего слоя, добавляет к этой сумме смещение (bias) и пропускает полученное значение через функцию активации. Функция активации вводит нелинейность в модель, позволяя ей аппроксимировать сложные зависимости. Количество нейронов в каждом скрытом слое является гиперпараметром, который необходимо подбирать эмпирически, в зависимости от сложности задачи.

Выходной слой (Output Layer): Выходной слой формирует выходные данные модели, соответствующие прогнозам или классификациям. Количество нейронов в выходном слое зависит от типа задачи. Для задачи бинарной классификации выходной слой обычно содержит один нейрон, выдающий вероятность принадлежности к одному из классов. Для задачи многоклассовой классификации выходной слой содержит столько нейронов, сколько классов в наборе данных, и использует функцию активации softmax для получения вероятностей принадлежности к каждому классу.

Принцип работы MLP заключается в последовательном преобразовании входных данных от входного слоя к выходному слою посредством прямого распространения (forward propagation) и последующей корректировке весов сети на основе ошибки прогноза с использованием обратного распространения ошибки (backpropagation).

**3. Основные концепции MLP**

Прямое распространение (forward propagation) — процесс вычисления выхода сети на основе входных данных. Для каждого нейрона вычисляется взвешенная сумма входов, к которой применяется функция активации.

Формула для выхода нейрона:

Где f – функция активации, wi – веса, xi – входы, b – смещение.

Обратное распространение ошибки (backpropagation) — алгоритм обучения, который корректирует веса сети на основе градиента ошибки.

Шаги алгоритма:

* Вычисление ошибки на выходе сети.
* Распространение ошибки назад по слоям.
* Обновление весов с использованием градиентного спуска.

Формула обновления весов:

Где s – скорость обучения, bi – градиент ошибки нейрона.

Функция активации ReLU (Rectified Linear Unit) – это простая, но очень эффективная функция, которая широко используется в нейронных сетях, особенно в скрытых слоях.

ReLU определяется как: ReLU(x) = max(0, x). Это означает, что если вход x положительный, то выход равен x. Если x отрицательный или равен нулю, то выход равен 0. А ее производная ReLU`(x) = 1, если x > 0; = 0, если x <= 0.**ГЛАВА II. Основной код программы по созданию и обучению MLP**

**1. Код программы с обучением многослойного персептрона**

#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 функции ReLU

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;

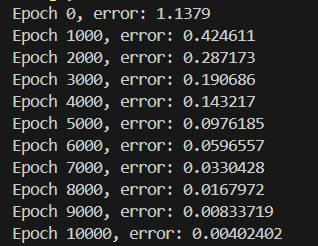
    }

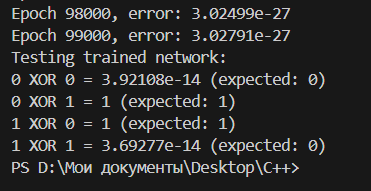
    mlp.save\_weights("xor\_weights.txt");

    return 0;

}

**2. Скриншоты работы программы по обучению и решению тестовой выборки**





**3. Код программы по тестированию обученного MLP**

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]

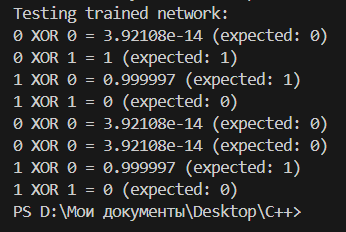
             << " (expected: " << (input[0] != input[1]) << ")" << endl;

    }

    return 0;

}

**Результат работы программы после тестирования:**

****

**ГЛАВА II. Ссылка на GitHub**

<https://github.com/ShurkinIlya>

**ЗАКЛЮЧЕНИЕ**

В ходе курсовой работы был реализован многослойный персептрон (MLP) с использованием алгоритмов прямого распространения, обратного распространения ошибки и функции активации ReLU. Модель успешно обучена решению задачи XOR, что демонстрирует ее способность к аппроксимации нелинейных функций.**СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ**

1. Онлайн учебник «Нейронные сети», https://neural.radkopeter.ru/book/
2. Нейросетевые технологии обработки данных: учеб. пособие / В. А. Головко, В. В. Краснопрошин. – Минск: БГУ, 2017. – 263 с.