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 *Робототехники и комплексной автоматизации*

КАФЕДРА *Системы автоматизированного проектирования (РК-6)*

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

по дисциплине: «Введение в искусственный интеллект»

Студент	Косенков Александр Александрович
Группа	РК6-12М
Тип задания	Лабораторная работа №3
Тема лабораторной работы	Программирование искусственного нейрона

Студент _____ **Косенков А.А.**
подпись, дата *фамилия, и.о.*

Преподаватель _____ **Федорук В.Г.**
подпись, дата *фамилия, и.о.*

Оценка _____

Москва, 2021 г.

Оглавление

Задание на лабораторную работу

Вариант 2-7

Разработать программу, моделирующую поведение искусственного трехвходового сигмоидального нейрона и обеспечивающую его обучение для решения задачи классификации. Рекомендуется, в тех ситуациях, когда это возможно, использовать режим обучения «оффлайн». Обучить разработанный нейрон на предложенном варианте двухмерных данных (рис. 1) и проверить его работу на ряде контрольных точек.

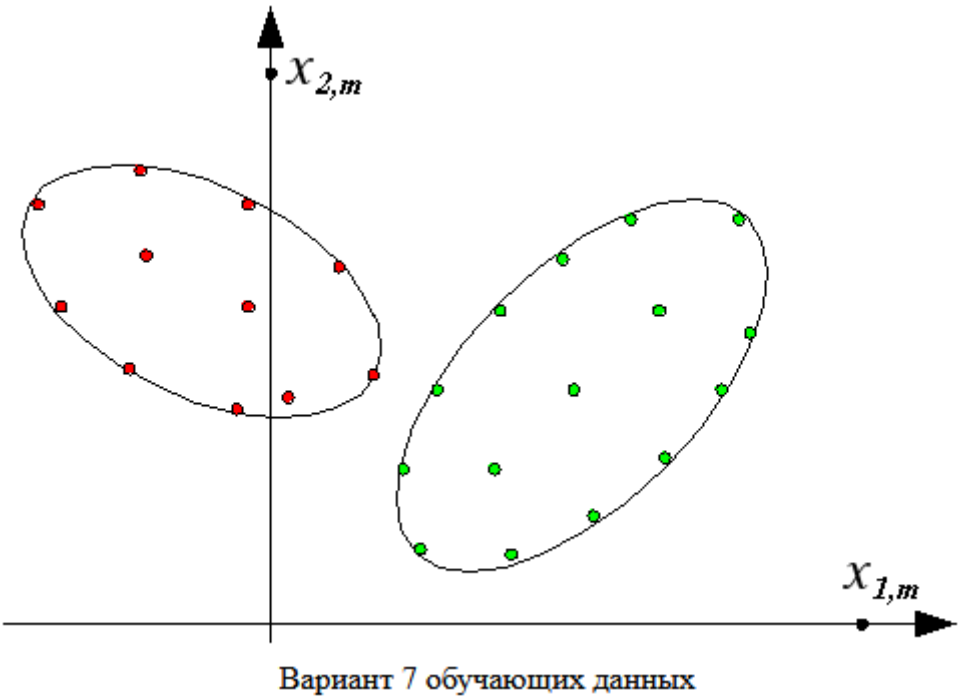

Рис. 1. Обучающая выборка данных.

Метод

Описание сигмоидального нейрона

Нейрон данного типа устраняет основной недостаток персептрона (модель искусственного нейрона, предложенная в 1943 г.) – разрывность функции активации $f(u_i)$. Структурная схема сигмоидального нейрона представлена ниже:

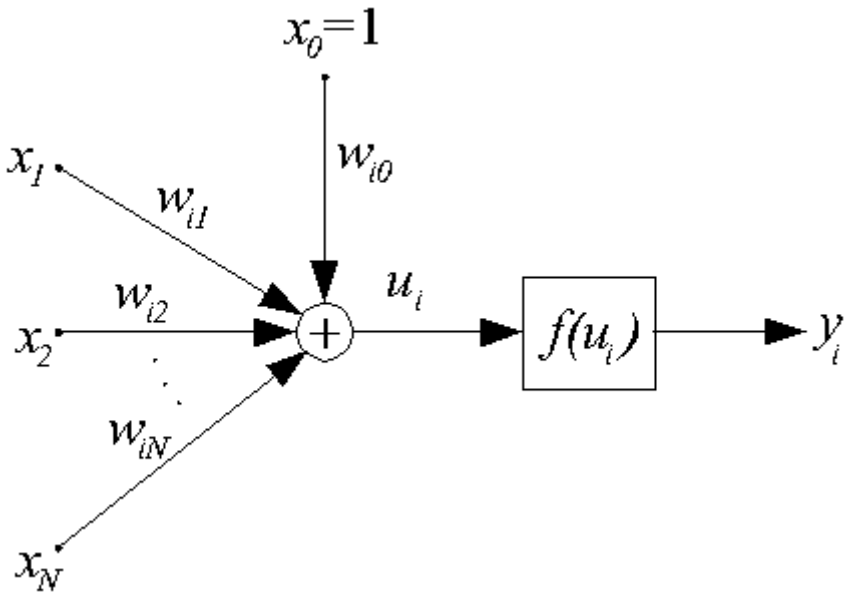

Рис. 2. Структурная схема сигмоидального нейрона.

В качестве функции активации $f(u_i)$ выступает сигмоидальная функция (рис. 3). На практике используются как униполярные, так и биполярные функции активации.

Пусть u_i – взвешенная сумма входных сигналов для i-го нейрона в сети (в рамках текущей задачи рассматривается модель только одного нейрона, поэтому индекс i опускается), которую можно записать как:

$$u = \sum_{j=0}^{N} (\omega_j x_j), \tag{1}$$

где N – количество входов нейрона, j – номер связи. Тогда униполярная функция активации сигмоидального нейрона записывается как:

$$f(u) = \frac{1}{1 + e^{-\sigma u}}, \tag{2}$$

где σ – коэффициент, определяющий «крутизну» функции (в рамках задачи полагается равным 1).

В свою очередь производная униполярной функции активации выражается как:

$$\frac{\partial f(u)}{\partial u} = \sigma f(u)\big(1 - f(u)\big). \tag{3}$$

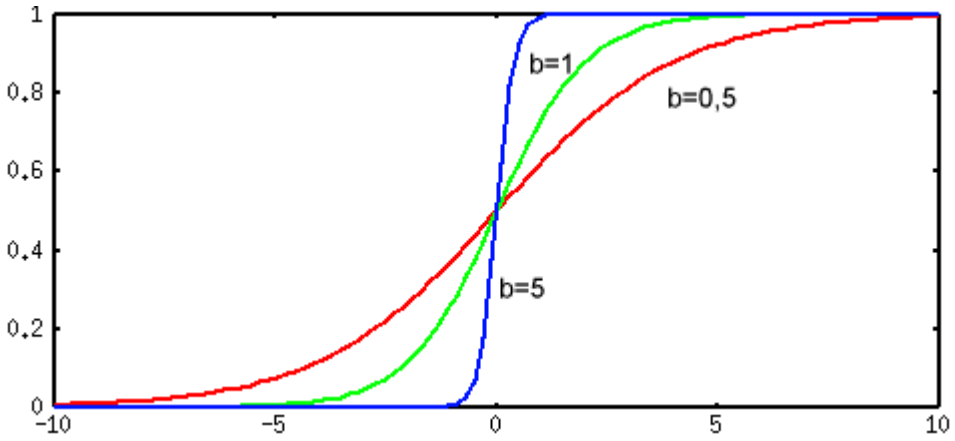

Рис. 3. Униполярная функция активации сигмоидального нейрона (b = σ).

Для обучения сигмоидального нейрона используется стратегия «с учителем», которая подразумевает поиск минимума целевой функции, которая в случае режима обучения «оффлайн» (учитываются все обучающие выборки сразу) имеет вид:

$$E(\bar{W}) = \frac{1}{2}\sum_{k=1}^{p}(y^k - d^k)^2, \tag{4}$$

где p – количество обучающих выборок, y^k – выходной сигнал нейрона для k-й выборки, d^k – ожидаемое значение выходного сигнала для k-й выборки.

В отличие от персептрона, для поиска минимума целевой функции (4) используются методы поисковой оптимизации первого порядка, в которых целенаправленное изменение весовых коэффициентов ω_j осуществляется в направлении отрицательного градиента $E(\bar{W})$.

Так, j-й компонент вектора градиента целевой функции имеет вид:

$$\frac{\partial E(\bar{W})}{\partial \omega_j} = \sum_{k=1}^{p}(y^k - d^k)\frac{\partial f(u^k)}{\partial u^k}x_j^p. \tag{5}$$

В таком случае обучение производится посредством коррекции весов нейрона методом градиента по формуле:

$$\omega_j(t+1) = \omega_j(t) - \mu\frac{\partial E(\bar{W})}{\partial \omega_j}, \tag{6}$$

где μ – коэффициент обучения из метода градиентного спуска.

Основными проблемами в реализации метода градиента являются:

- выбор стратегии определения коэффициента обучения μ;
- выбор критерия окончания вычислительного процесса.

В простейшем варианте метода градиента коэффициент μ выбирается постоянной величиной из диапазона $(0, 1]$, а процесс поиска минимума завершается при выполнении условия

$$\left|\frac{\partial E(t)}{\partial \bar{W}}\right| < \epsilon_E, \tag{7}$$

где ϵ_E – константа, определяющая точность отыскания минимума.

Однако, для эффективного обучения нейрона необходимы адаптивные стратегии подбора коэффициента обучения μ в ходе поиска экстремума.

Хорошие результаты демонстрирует следующая стратегия адаптации коэффициента μ:

1 В текущем t-ом цикле обучения вычисляется новое значение вектора входных весов нейрона $\bar{W}'(t+1)$ по стандартной формуле (6) и сравниваются значения целевой функции для $\bar{W}(t)$ и $\bar{W}'(t+1)$.

2 Если $E\left(\bar{W}'(t+1)\right) < E(\bar{W}(t))$, то $\bar{W}(t+1) = \bar{W}'(t+1)$ и счетчик числа подряд идущих удачных применений величины μ увеличивается на 1. Если этот счетчик превышает наперед заданную величину (например, 2), то значение μ удваивается, а счетчик сбрасывается в 0. Далее переход на следующую итерацию цикла $t+1$ (п. 1)

3 Если $E\left(\bar{W}'(t+1)\right) \geq E(\bar{W}(t))$, то значение коэффициента μ уменьшаешься в 2 раза, а счетчик цикла подряд идущих удачных применений величины μ сбрасывается в 0. Далее переход к п.1

4 Процесс обучения завершается при условии $\mu < \epsilon_\mu$, где ϵ_μ – наперед заданная константа (например, 1E-3).

В разработанной программной реализации сигмоидального нейрона применен описанный выше алгоритм адаптивного выбора коэффициента обучения μ.

Программная реализация

В рамках лабораторной работы был программно реализован трехвходовой сигмоидальный нейрон (один из входов является входом поляризации). Программный код представлен в Листингах 4 и 5 и представляет собой класс с публичными методами learn и test, производящими обучение нейрона в режиме оффлайн и тестирование нейрона соответственно.

Для настройки коэффициентов обучения был разработан механизм конфигурации с помощью текстового файла config.cfg (Листинг 7) без необходимости перекомпиляции исходного кода.

Так, значение ϵ_μ устанавливается параметром LEARNING_COEF_EPS, а точность обучения ϵ_E устанавливается параметром TARGET_FUNC_EPS. В качестве значения ϵ_μ было выбрано 1E-3.

Обучение нейрона

Для генерации выборок в соответствии с заданием (рис. 1) были разработаны классы EllipseParams (Листинг 2) и EllipseDataGenerator (Листинги 3 и 6), позволяющие генерировать точки в кластерах эллипсов, параметры которых также задаются в конфигурационном файле config.cfg.

Так, результат генерации точек по рис. 1 представлен на рис. 4.

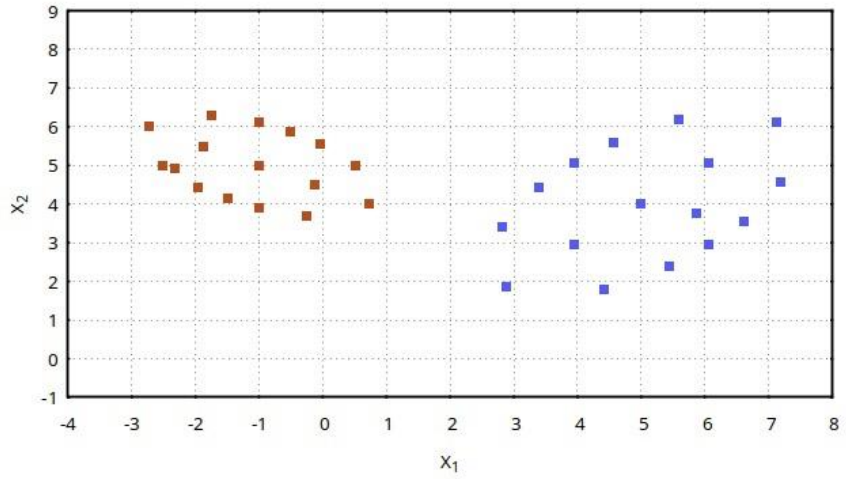

Рис. 4. Сгенерированные точки для обучающей выборки.

При этом начальные значения весов задаются случайным образом в интервале $(0, 1)$.

Результаты работы программы

Для обучающей выборки, продемонстрированной на рис. 4 было проведено обучение в режиме «оффлайн» и получено разбиение на 2 кластера за 3 итерации. Демонстрация обучения с линией весов представлена на рис. 5.

Помимо этого, также была сформирована контрольная выборка точек внутри эллипсов и проведено тестирование, результаты которого для определенного количества точек представлено на рис. 6

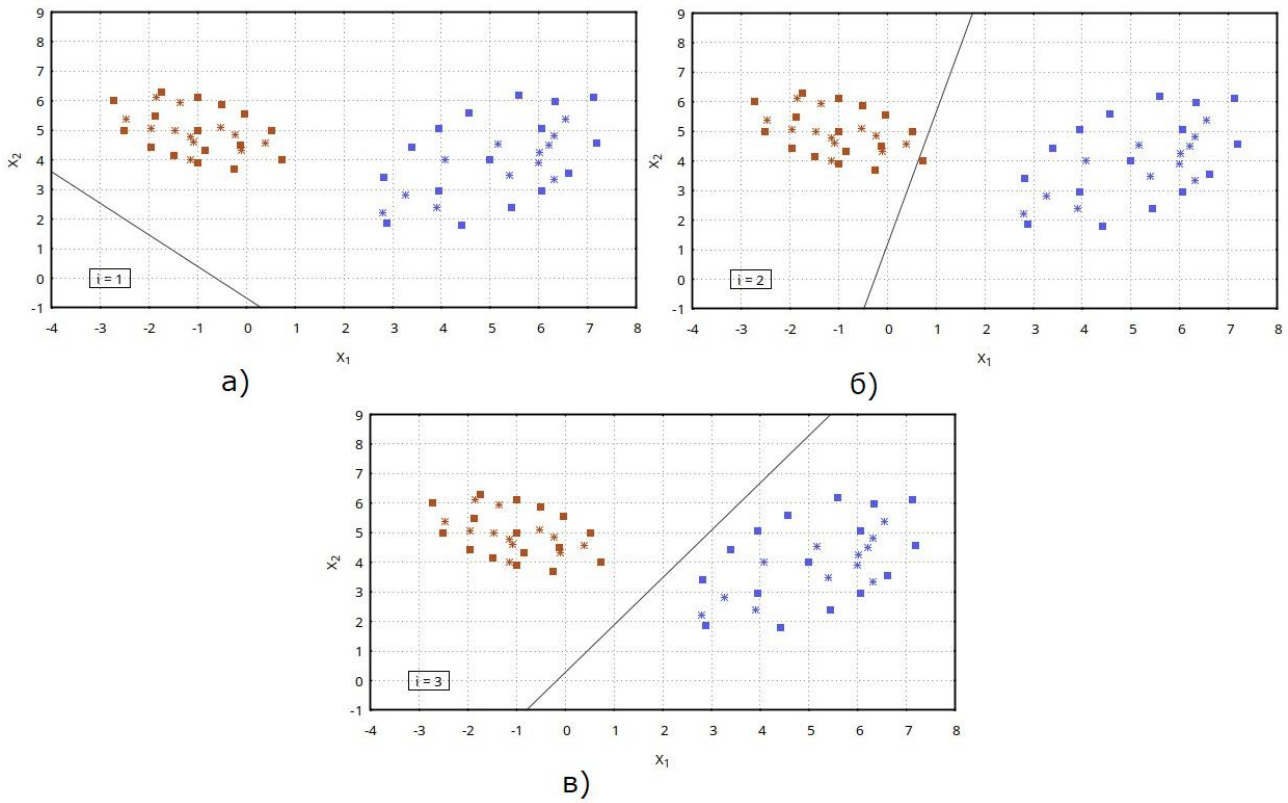

Рисунок 5. Процесс обучения нейрона на тестовой выборке. Черная линия – линия весов, демонстрирующая разделение данных на 2 кластера. а) 1-я итерация б) 2-я итерация в) 3-я, конечная итерация.

```
0.00268018 = 0? → true
0.0013134 = 0? → true
0.0102637 = 0? → true
0.0406051 = 0? → true
0.0234641 = 0? → true
0.0025677 = 0? → true
0.00816331 = 0? → true
0.0147819 = 0? → true
0.00480999 = 0? → true
0.0584647 = 0? → true
0.00109291 = 0? → true

Precision of classification: 100%
```

Рис. 6. Результаты тестирования обученного нейрона.

Сравнение значений весов для начального состояния нейрона и обученного представлено в таблице 1.

Таблица 1. Сравнение значений весов нейрона

$\bar{W}$	ω_0	ω_1	ω_2	i
Начальное значение весов	0.55099	0.875552	0.820256	1
Итоговое значение весов	0.213148	1.20636	-0.753643	3

Текст программы

Листинг 1. Файл main.cpp

```
#include <vector>
#include <cmath>
#include <iostream>

#include "config.hpp"
#include "SigmoidalNeuron.hpp"
#include "EllipseDataGenerator.hpp"
#include "EllipseParams.hpp"

// Метод для конкатенации двух векторов
template<typename T>
std::vector<T> operator+(const std::vector<T>& v1, const std::vector<T>& v2){
    std::vector<T> vr(std::begin(v1), std::end(v1));
    vr.insert(std::end(vr), std::begin(v2), std::end(v2));
    return vr;
}

int main(int argc, const char** argv) {
    ConfigurationSingleton& configuration =
ConfigurationSingleton::getInstance();
    EllipseParams ellipse1(configuration, "1");
    EllipseParams ellipse2(configuration, "2");

    // Генерация данных в эллипсах для обучения
    EllipseDataGenerator ellipse1Generator(ellipse1);
    EllipseDataGenerator ellipse2Generator(ellipse2);

    int ellipse1PointsLearn =
configuration.getVariableInt("ELLIPSE_1_LEARN_POINTS");
    int ellipse2PointsLearn =
configuration.getVariableInt("ELLIPSE_2_LEARN_POINTS");
    int ellipse1PointsTest =
configuration.getVariableInt("ELLIPSE_1_TEST_POINTS");
    int ellipse2PointsTest =
configuration.getVariableInt("ELLIPSE_2_TEST_POINTS");

    std::vector<std::vector<double>> XLearn1 =
ellipse1Generator.generateLearnData(ellipse1PointsLearn, "first_ellipse");
    std::vector<std::vector<double>> XLearn2 =
ellipse2Generator.generateLearnData(ellipse2PointsLearn, "second_ellipse");

    std::vector<double> DLearn1(XLearn1.size(), 1.0);
    std::vector<double> DLearn2(XLearn2.size(), 0.0);
```

```
    std::vector<std::vector<double>> XLearn = XLearn1 + XLearn2;
    std::vector<double> DLearn = DLearn1 + DLearn2;


EllipseDataGenerator::writeToFile(configuration.getVariable("LEARN_DATA_FILENAME
"), XLearn, DLearn);

    // Генерация данных в эллипсе для тестирования
    std::vector<std::vector<double>> XTest1 =
ellipse1Generator.generateRandomData(ellipse1PointsTest);
    std::vector<std::vector<double>> XTest2 =
ellipse2Generator.generateRandomData(ellipse2PointsTest);

    std::vector<double> DTest1(XTest1.size(), 1.0);
    std::vector<double> DTest2(XTest2.size(), 0.0);

    std::vector<std::vector<double>> XTest = XTest1 + XTest2;
    std::vector<double> DTest = DTest1 + DTest2;


EllipseDataGenerator::writeToFile(configuration.getVariable("TEST_DATA_FILENAME"
), XTest, DTest);

    // Создание сигмоидального нейрона, обучение, тестирование и переобучение по
необходимости
    SigmoidalNeuron neuron(3, configuration);
    neuron.learn(XLearn, DLearn);

    neuron.test(XTest, DTest);

    return 0;
}
```

Листинг 2. Файл EllipseParams.hpp

```
#ifndef ELLIPSEPARAMS_H
#define ELLIPSEPARAMS_H

#include <string>

#include "config.hpp"


struct EllipseParams {
    // Полуоси эллипса
    double a;
    double b;
    // Координаты точки центра
    double x0;
    double y0;
    // Угол поворота осей эллипса
    double rotate;

    EllipseParams() = default;
    explicit EllipseParams(ConfigurationSingleton& configuration, std::string
ellipseName) {
        this->a = configuration.getVariableDouble("ELLIPSE_" + ellipseName +
"_A");
        this->b = configuration.getVariableDouble("ELLIPSE_" + ellipseName +
"_B");
```

```
        this->x0 = configuration.getVariableDouble("ELLIPSE_" + ellipseName +
"_X0");
        this->y0 = configuration.getVariableDouble("ELLIPSE_" + ellipseName +
"_Y0");
        this->rotate = configuration.getVariableDouble("ELLIPSE_" + ellipseName
+ "_ROTATE");
    }
};

#endif // ELLIPSEPARAMS_H
```

Листинг 3. Файл EllipseDataGenerator.hpp

```
#ifndef GENERATOR_H
#define GENERATOR_H

#include <string>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <cmath>

#ifndef M_PI
    #define M_PI 3.14159265358979323846
#endif

#include "EllipseParams.hpp"


class EllipseDataGenerator {
private:
    EllipseParams ellipse;
    // Генерация точек внутри эллипса
    std::vector<std::vector<double>> generatePointsInsideEllipse(size_t
points_num) const;
    // Генерация точек по периметру эллипса
    std::vector<std::vector<double>> generatePointsOnEllipse(size_t points_num)
const;
    // Получение точки эллипса в абсолютных координатах (сдвиг + поворот осей)
    std::vector<double> pointToAbsoluteCoord(double xLocal, double yLocal)
const;
public:
    explicit EllipseDataGenerator(EllipseParams& _ellipse) : ellipse(_ellipse) {
        srand(time(nullptr));
        this->ellipse.rotate = this->ellipse.rotate * M_PI / 180.0;
    };

    // Сгенерерировать случайное число в интервале [l, r)
    static double generateRandNumInRange(double l, double r);

    static void writeToFile(const std::string& filename, const
std::vector<std::vector<double>>& res);
    static void writeToFile(const std::string& filename, const
std::vector<std::vector<double>>& res, const std::vector<double>& D_res);
    static void plotDots(
        std::vector<std::vector<double>>& data,
        std::vector<double>& DLearn,
        std::string dataFilename,
        std::string gnuplotFilename
    );

    // Сгенерировать точки в эллипсе для обучения нейрона
```

```
    std::vector<std::vector<double>> generateLearnData(size_t points_num,
std::string filename) const;
    std::vector<std::vector<double>> generateRandomData(size_t points_num)
const;

    ~EllipseDataGenerator() = default;
};

#endif // GENERATOR_H
```

Листинг 4. Файл SigmoidalNeuron.hpp

```
#ifndef NEURON_H
#define NEURON_H

#include <vector>
#include <fstream>

#include "config.hpp"


class SigmoidalNeuron {
private:
    // Количество входов нейрона
    size_t inputsNumber;
    double sigma;
    std::vector<double> weights;
    std::ofstream weightsFile;
    ConfigurationSingleton& configuration;

    double neuronOutput(std::vector<double>& X, std::vector<double>&
extWeights);
    double getUSum(std::vector<double>& X, std::vector<double>& extWeights);
    double getDerivative(std::vector<double>& X);
    double targetFunction(std::vector<double>& D, std::vector<double>& Y);
    void appendWeightsToFile();
public:
    explicit SigmoidalNeuron(int _inputs_num, ConfigurationSingleton&
_configuration);

    void learn(
        std::vector<std::vector<double>>& XLearn,
        std::vector<double>& DLearn
    );
    bool test(std::vector<std::vector<double>>& XTest, std::vector<double>&
DTest);

    std::vector<double> getWeights() { return this->weights; };
    ~SigmoidalNeuron() = default;
};

#endif // NEURON_H
```

Листинг 5. Файл SigmoidalNeuron.cpp

```
#include <string>
#include <iostream>
#include <chrono>
#include "SigmoidalNeuron.hpp"
#include "EllipseDataGenerator.hpp"
```

```
SigmoidalNeuron::SigmoidalNeuron(int _inputsNumber, ConfigurationSingleton&
_configuration)
        : inputsNumber(_inputsNumber), configuration(_configuration) {
    this->sigma = configuration.getVariableDouble("SIGMA");

    std::string weightsFilename = configuration.getVariable("WEIGHTS_FILENAME");

    this->weights.resize(inputsNumber);
    this->weightsFile.open(weightsFilename);
    if (!this->weightsFile.is_open()) {
        std::cerr << "Error: can't open file " << weightsFilename << std::endl;
        throw std::runtime_error(std::string("Error: can't open file ") +
weightsFilename);
    }

    for (size_t i = 0; i < inputsNumber; ++i) {
        weights.at(i) = EllipseDataGenerator::generateRandNumInRange(0, 1);
    }

    std::cout << "Initial weights:" << std::endl;
    this->appendWeightsToFile();
}

double SigmoidalNeuron::neuronOutput(std::vector<double>& X,
std::vector<double>& extWeights) {
    // Функция активации
    return 1 / ( 1 + std::exp(- this->sigma * this->getUSum(X, extWeights)) );
}

double SigmoidalNeuron::getUSum(std::vector<double>& X, std::vector<double>&
extWeights) {
    double u = 0;
    // X0 - вход поляризации = 1
    u += extWeights.at(0) * 1.0;

    for (size_t i = 1; i < this->inputsNumber; ++i) {
        u += X.at(i - 1) * extWeights.at(i);
    }

    return u;
}

double SigmoidalNeuron::getDerivative(std::vector<double>& X) {
    double output = this->neuronOutput(X, this->weights);
    return this->sigma * output * (1 - output);
}

double SigmoidalNeuron::targetFunction(std::vector<double>& D,
std::vector<double>& Y) {
    double E = 0.0;
    for (size_t k = 0; k < D.size(); ++k) {
        E += std::pow(Y.at(k) - D.at(k), 2.0);
    }

    return 0.5 * E;
}

void SigmoidalNeuron::learn(
    std::vector<std::vector<double>>& XLearn,
    std::vector<double>& DLearn
) {
    std::cout << "learning..." << std::endl;
    std::vector<double> correctedWeights(this->weights.size());
```

```
    double learningCoef = this-
>configuration.getVariableDouble("LEARNING_COEF_INIT");
    size_t successIterations = 0;
    double currentTargetFunc = 1.0;

    double learningCoefEPS = this-
>configuration.getVariableDouble("LEARNING_COEF_EPS");
    double targetFuncEPS = this-
>configuration.getVariableDouble("TARGET_FUNC_EPS");

    while (learningCoef > learningCoefEPS && currentTargetFunc > targetFuncEPS)
{
        for (size_t j = 0; j < this->inputsNumber; ++j) {
            double derivative = 0.0;

            for (size_t k = 0; k < XLearn.size(); ++k) {
                double neuronOut = this->neuronOutput(XLearn.at(k), this-
>weights);

                double xj = 1.0;
                if (j > 0) {
                    xj = XLearn.at(k).at(j - 1);
                }
                // Вычисление компоненты градиента
                derivative += (neuronOut - DLearn.at(k)) * this-
>getDerivative(XLearn.at(k)) * xj;
            }

            // коорекция веса
            correctedWeights.at(j) = this->weights.at(j) - learningCoef *
derivative;
        }

        std::vector<double> prevE(DLearn.size());
        std::vector<double> currE(DLearn.size());

        for (size_t k = 0; k < XLearn.size(); ++k) {
            prevE.at(k) = this->neuronOutput(XLearn.at(k), this->weights);
            currE.at(k) = this->neuronOutput(XLearn.at(k), correctedWeights);
        }

        double targetFuncCurr = this->targetFunction(DLearn, currE);
        double targetFuncPrev = this->targetFunction(DLearn, prevE);

        if (targetFuncCurr < targetFuncPrev) {
            std::cout << "Successed iteration" << std::endl;
            if (successIterations > 2) {
                learningCoef *= 2.0;
                successIterations = 0;
            } else {
                ++successIterations;
            }

            this->weights = correctedWeights;
            currentTargetFunc = targetFuncCurr;

            std::cout << "Corrected weights: " << std::endl;
            this->appendWeightsToFile();
            std::cout << "Current learning coefficient: " << learningCoef <<
std::endl;
            std::cout << "Current target function value: " << currentTargetFunc
<< std::endl;
```

```
        } else {
            std::cout << "Failed iteration" << std::endl;
            learningCoef /= 2.0;
            successIterations = 0;
        }
    }
}

void SigmoidalNeuron::appendWeightsToFile() {
    std::cout << "W = [ ";
    for (size_t i = 0; i < this->inputsNumber; ++i) {
        this->weightsFile << this->weights.at(i) << " ";
        std::cout << this->weights.at(i) << " ";
    }

    this->weightsFile << std::endl;
    std::cout << "]" << std::endl << std::endl;
}

bool SigmoidalNeuron::test(std::vector<std::vector<double>>& XTest,
std::vector<double>& DTest) {
    std::cout << "testing..." << std::endl;
    double testDiffEPS = this->configuration.getVariableDouble("TEST_DIFF_EPS");

    size_t matchesCount = 0;

    for (size_t i = 0; i < DTest.size(); ++i) {
        double result = this->neuronOutput(XTest.at(i), this->weights);
        std::cout << result << " = " << DTest.at(i) << "? -> ";

        if (std::fabs(result -  DTest.at(i)) < testDiffEPS) {
            std::cout << "true";
            ++matchesCount;
        } else {
            std::cout << "false";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;

    std::cout << "Precision of classification: " << (double) matchesCount /
XTest.size() * 100. << "%" << std::endl;

    if (matchesCount != XTest.size()) {
        return false;
    } else {
        return true;
    }
}
```

Листинг 6. Файл EllipseDataGenerator.cpp

```
#include <fstream>
#include <string>
#include "EllipseDataGenerator.hpp"
#include "EllipseParams.hpp"
#include <cmath>


// Сгенерерировать случайное число в интервале [l, r)
double EllipseDataGenerator::generateRandNumInRange(double l, double r) {
    double tmp = (double) std::rand() / (RAND_MAX / (r - l));
```

```
    return l + tmp;
}

std::vector<double> EllipseDataGenerator::pointToAbsoluteCoord(double xLocal,
double yLocal) const {
    EllipseParams ellipse = this->ellipse;

    double x = ellipse.x0 + (xLocal * std::cos(ellipse.rotate) - yLocal *
std::sin(ellipse.rotate));
    double y = ellipse.y0 + (xLocal * std::sin(ellipse.rotate) + yLocal *
std::cos(ellipse.rotate));

    return std::vector<double>{x, y};
}

// Сгенерировать точки внутри эллипса
std::vector<std::vector<double>>
EllipseDataGenerator::generatePointsInsideEllipse(size_t points_num) const {
    EllipseParams ellipse = this->ellipse;

    std::vector<std::vector<double>> res;

    for (size_t i = 0; i < points_num; ++i) {
        double rad = std::sqrt(generateRandNumInRange(0, 1));
        double phi = 2 * M_PI * generateRandNumInRange(0, 1);

        double x_e = rad * std::cos(phi) * ellipse.a;
        double y_e = rad * std::sin(phi) * ellipse.b;

        res.push_back(this->pointToAbsoluteCoord(x_e, y_e));
    }

    return res;
}

// Сгенерировать точки на эллипсе
std::vector<std::vector<double>>
EllipseDataGenerator::generatePointsOnEllipse(size_t points_num) const {
    EllipseParams ellipse = this->ellipse;

    std::vector<std::vector<double>> result;
    double stepAngle = 360.0 / points_num;

    for (size_t i = 0; i < points_num; ++i) {
        double phi = i * stepAngle * M_PI / 180.0;

        // Переход от угла эллипса к углу образующих окружностей
        double phiAbs = std::atan2((ellipse.a * std::sin(phi)), (ellipse.b *
std::cos(phi)));

        // Получение точки эллипса в локальных координатах
        double xLocal = ellipse.a * std::cos(phiAbs);
        double yLocal = ellipse.b * std::sin(phiAbs);

        result.push_back(this->pointToAbsoluteCoord(xLocal, yLocal));
    }

    return result;
}

// Генерация точек в области эллипса
std::vector<std::vector<double>> EllipseDataGenerator::generateLearnData(size_t
points_num, std::string filename) const {
```

```
    // Генерация точек по периметру эллипса
    std::vector<std::vector<double>> points =
generatePointsOnEllipse(points_num);

    // Генерация точек на главной оси эллипса (полуэллипс содержит главную ось)
    std::vector<double> aBegin = points.at(0); // Начало полуэллипса (начало
оси)
    std::vector<double> aEnd = points.at(points_num / 2); // Конец полуэллипса
(конец оси)

    // Центральная точка
    points.push_back({ (aBegin.at(0) + aEnd.at(0)) / 2, (aBegin.at(1) +
aEnd.at(1)) / 2});

    auto p3 = points.at(12);

    // Точки по бокам от центра
    double x_1 = aBegin.at(0) + p3.at(0);
    double y_1 = aBegin.at(1) + p3.at(1);
    double x_2 = p3.at(0) + aEnd.at(0);
    double y_2 = p3.at(1) + aEnd.at(1);

    points.push_back({x_1 / 2, y_1 / 2});
    points.push_back({x_2 / 2, y_2 / 2});

    // Добавить несколько точек внутри эллипса
    std::vector<std::vector<double>> pointsInside = this-
>generatePointsInsideEllipse(1);
    points.insert(points.end(), pointsInside.begin(), pointsInside.end());

    std::string resultFilename = std::string(filename + "_" +
std::to_string(this->ellipse.a) + "_" + std::to_string(this->ellipse.b) +
".dat");
    EllipseDataGenerator::writeToFile(resultFilename, points);

    return points;
}

std::vector<std::vector<double>> EllipseDataGenerator::generateRandomData(size_t
points_num) const {
    return this->generatePointsInsideEllipse(points_num);
}

void EllipseDataGenerator::writeToFile(const std::string& filename, const
std::vector<std::vector<double>>& res) {
    std::ofstream file(filename);
    for (auto & row : res) {
        for (auto & el : row) {
            file << el << " ";
        }
        file << std::endl;
    }
}

void EllipseDataGenerator::writeToFile(const std::string& filename,
                                       const std::vector<std::vector<double>>& res,
                                       const std::vector<double>& D_res) {
    std::ofstream file(filename);

    for (size_t i = 0; i < res.size(); i++) {
        for (auto & el : res.at(i)) {
            file << el << " ";
        }
```

```
        file << D_res.at(i) << std::endl;
    }
}
```

Листинг 7. Файл config.cfg

```
ELLIPSE_1_A = 3.0
ELLIPSE_1_B = 1.5
ELLIPSE_1_X0 = 5.0
ELLIPSE_1_Y0 = 4.0
ELLIPSE_1_ROTATE = 45.0

ELLIPSE_2_A = 2.0
ELLIPSE_2_B = 1.0
ELLIPSE_2_X0 = -1.0
ELLIPSE_2_Y0 = 5.0
ELLIPSE_2_ROTATE = 150.0

ELLIPSE_1_LEARN_POINTS = 12
ELLIPSE_2_LEARN_POINTS = 12

ELLIPSE_1_TEST_POINTS = 12
ELLIPSE_2_TEST_POINTS = 12

SIGMA = 1
LEARNING_COEF_INIT = 0.5
LEARNING_COEF_EPS = 1e-3
TARGET_FUNC_EPS = 1e-2

TEST_DIFF_EPS = 1e-1

WEIGHTS_FILENAME = weights.dat
LEARN_DATA_FILENAME = learn_data.dat
TEST_DATA_FILENAME = test_data.dat
```

Листинг 8. Файл gnuplot_window.gp

```
reset
set wxt

color_set1 = '#b54f1e'
color_set2 = '#565af5'

color_line = '#3e3c3d'

set border linewidth 1.5
set pointsize 1
set style line 1 lc rgb color_line pt 7

set palette defined (0 color_set1, 1 color_set2)

set grid

unset key

unset colorbox

set tics scale 0.1
set xtics 1
set ytics 1
set yrange[-1:9]
```

```
set xrange[-4:8]
set xlabel 'X_1'
set ylabel 'X_2'

N = system("wc -l weights.dat")

do for [i=1:N] {
    w_0 = system("sed -n " . i . "p weights.dat | awk '{print $1}'")
    w_1 = system("sed -n " . i . "p weights.dat | awk '{print $2}'")
    w_2 = system("sed -n " . i . "p weights.dat | awk '{print $3}'")

    LABEL = "i = " . i
    set obj 10 rect at graph 0.1,0.1 size char strlen(LABEL), char 1
    set obj 10 fillstyle empty border -1 front
    set label 10 at graph 0.1,0.1 LABEL front center

    f(x) = -1/(w_2) * (w_1 * x + w_0)

    plot f(x) w l ls 1, 'learn_data.dat' using 1:2:3 w p pt 5 lc palette z,
'test_data.dat' using 1:2:3 w p pt 3 lc palette z
}
```