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 №4
Тема лабораторной работы	Программирование искусственной нейронной сети

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 6

Разработать, используя язык C/C++, программу, моделирующую поведение многослойного персептрона с сигмоидальными функциями активации, обеспечивающую ее обучение для решения задач аппроксимации. Отладить модель нейронной сети и процедуру ее обучения на произвольных данных.

Многослойный персептрон

Сети подобного типа получили наибольшее распространение на практике в силу относительной простоты их математического описания. История искусственных нейронных сетей началась с однослойных сетей данного типа. Однако, возможности однослойных сетей крайне скромны, поскольку расположенные на одном уровне нейроны функционируют независимо друг от друга, и свойства всей сети ограничены свойствами отдельных нейронов. Многослойные сети получили свое развитие с конца 70-ых годов, когда были предложены эффективные алгоритмы их обучения.

Однонаправленные многослойные сети сети сигмоидального типа имеют устоявшееся название многослойный персептрон MLP (MultiLayer Perceptron). На рис. 1 представлена структурная схема двухслойного персептрона.

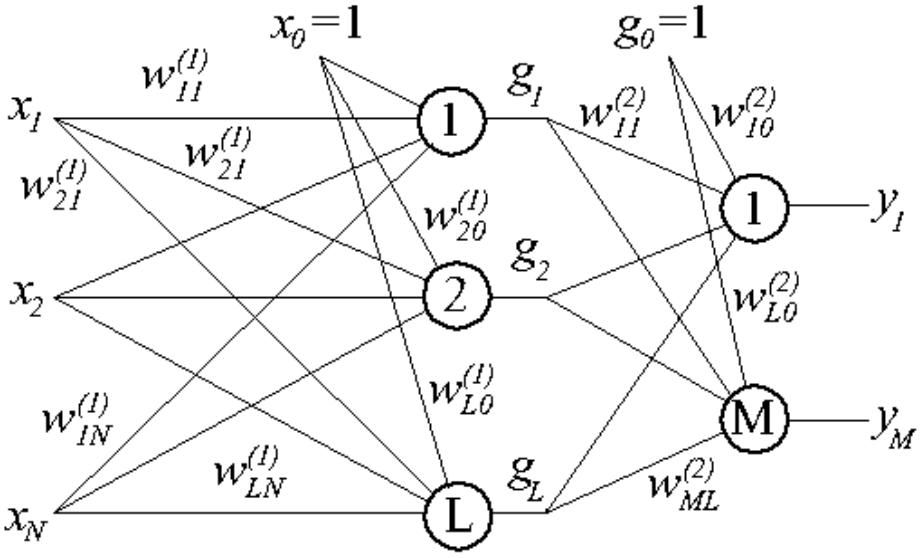

Рис. 1. – Структурная схема двухслойного персептрона в общем случае.

На рис. 1 N – количество входных сигналов, L – количество нейронов в первом слое, M – количество нейронов во втором слое, g_L – выходные сигналы

первого слоя нейронов, верхние индексы в скобках $(m), m = 1, 2$, означают номер слоя нейрона.

Выходные сигналы нейронных слоев рассчитываются по следующим формулам:

$$g_l = f\left(\sum_{j=0}^{N} \omega_{lj}^1 x_j\right), l = 1, \dots, L, \quad (1)$$

$$y_i = f\left(\sum_{l=0}^{L} \omega_{il}^2 g_l\right), i = 1, \dots, M. \quad (2)$$

При этом функции активации абсолютно всех нейронов сети абсолютно идентичны (в том числе имеют одинаковое значение параметра σ).

Цель обучения многослойного нейрона заключается в подборе таких значений всех весовых коэффициентов сети ω_{ij}^l, которые обеспечивают максимальное совпадение выходного вектора Y^k и эталонного вектора ожидаемых значений D^k при предъявлении входного вектора X^k.

В случае единичной обучающей выборки $< X, D >$ целевая функция задается в виде:

$$E(\mathrm{W}) = \frac{1}{2}\sum_{i=1}^{M}(y_i - d_i)^2, \quad (3)$$

где M – количество нейронов выходного слоя.

В случае нескольких обучающих пар $< X^k, D^k >, k = 1,2, \dots p$, целевая функция (3) превращается в сумму по всем парам:

$$E(\mathrm{W}) = \frac{1}{2}\sum_{k=1}^{p}\sum_{i=1}^{M}\left(y_i^k - d_i^k\right)^2, \quad (4)$$

Представленные ранее выражения для целевой функции (4) характеризуются, во-первых, аналитическим видом, во-вторых, дважды дифференцируемостью. Это дает возможность использовать для отыскания минимума $E(W)$ практически весь арсенал универсальных методов поисковой оптимизации. Кроме того, специально для целей обучения ИНС разработаны модификации «канонических» методов.

Так, в данной работе используется метод градиента, в котором очередное приближение к оптимальному решению определяется по формуле:

$$W^{k+1} = W^k - \mu * gradE(W^k), \quad (5)$$

где μ – коэффициент обучения, адаптивно подбираемый в ходе поиска так, чтобы $E^{k+1} < E^k$.

Метод обратного распространения ошибки

Данный метод является версией метода градиента (канонического метода параметрической оптимизации первого порядка), адаптированной для целей обучения многослойных ИНС. Рассмотрим его суть на примере двухслойного персептрона (рис. 1).

Далее приведены выражения для метода обучения «оффлайн» в общем случае (N слоев). В качестве целевой функции рассматривается выражение (4). Для каждого нейрона в каждом слое происходит так называемое накопление ошибки с предыдущих, считая от конца к началу, слоев. Накопление происходит путем суммирования взвешенных накопленных ошибок по всем нейронам предшествующего (от конца к началу) слоя. В случае, если ошибка рассчитывается для выходного слоя, происходит вычисление разницы между эталонным и рассчитанным значением. Так расчет накопленной ошибки на i-м нейроне слоя l для выборки k рассчитывается как:

$$err_i^{l,k} = \sum_{m=1}^{M_l} err_m^{l+1,k} * \omega_{mi}^{l+1},\ l = 1, \dots, N-1, \quad (6.1)$$

$$err_i^{N,k} = \sum_{m=1}^{M} (y_m - d_m) \frac{\partial f(u_m^N)}{\partial u_m^N}. \quad (6.2)$$

В свою очередь j-й компонент вектора градиента целевой функции для i-го нейрона слоя l записывается как:

$$\frac{\partial E(W)}{\partial \omega_{ij}^l} = \sum_{k=1}^{p} err_i^{l,k} \frac{\partial f(u_i^l)}{\partial u_i^l} g_j^{l-1,k}. \quad (7)$$

Рассмотренный перенос погрешности $(y_m - d_m)$ с выхода сети к ее предыдущим слоям и обусловил название метода «обратного распространения ошибки». Данное правило иллюстрирует рис. 2.

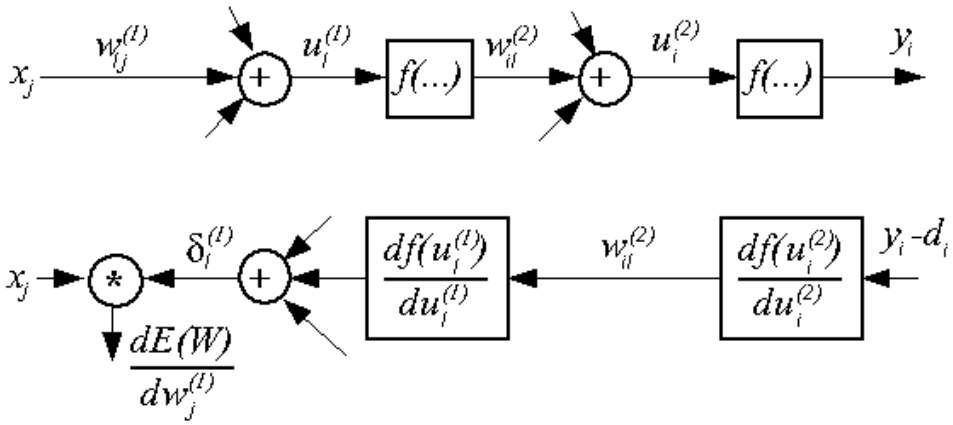

Рисунок 2. Иллюстрация правила обратного распространения ошибки.

Аппроксимация

В качестве практической задачи для решения общей задачи аппроксимации была выбрана задача приближения к математическим функциям. Так, для тестирования были выбраны функции $y = x^2$, $y = \sin(x)$, а также поверхность $z = \sin(x) + \cos(y)$.

Было установлено, что оптимальной конфигурацией сети для аппроксимации подавляющего количества графиков является сеть, состоящая из 2-х слоев – входного и выходного, где в качестве выходного слоя выступает один нейрон-сумматор, который в качестве своего выходного значения возвращает взвешенную сумму u. Разработанная сеть была настроена в соответствии с представленным описанием. Структура подобной сети представлена на рис. 3:

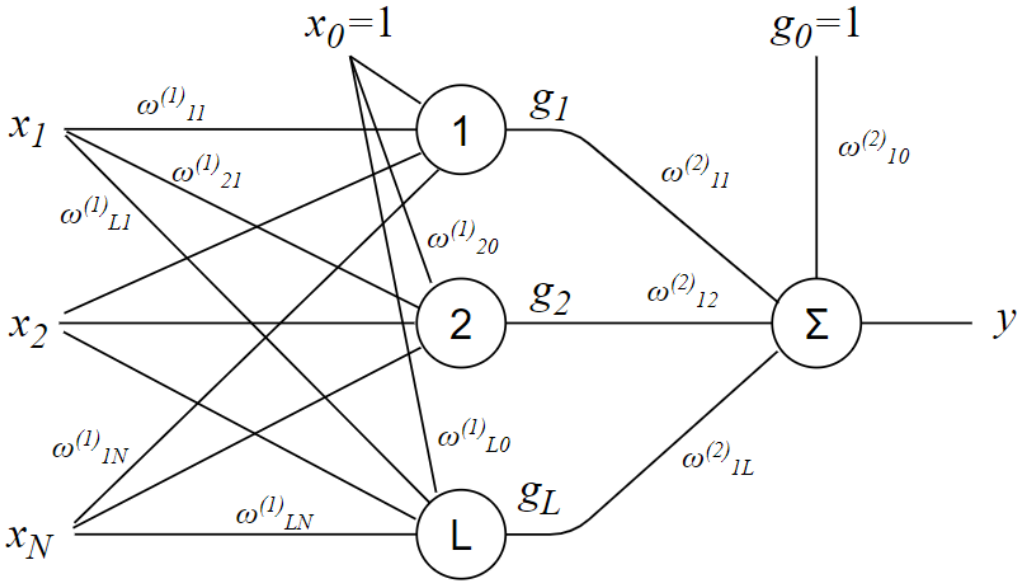

Рис. 3. Структура разработанной нейронной сети для решения задачи аппроксимации.

Результаты работы программы

Для тестирования работы разработанной программы была проведена аппроксимация к 3-м различным функциям, результаты которой представлены на рис. 4-9.

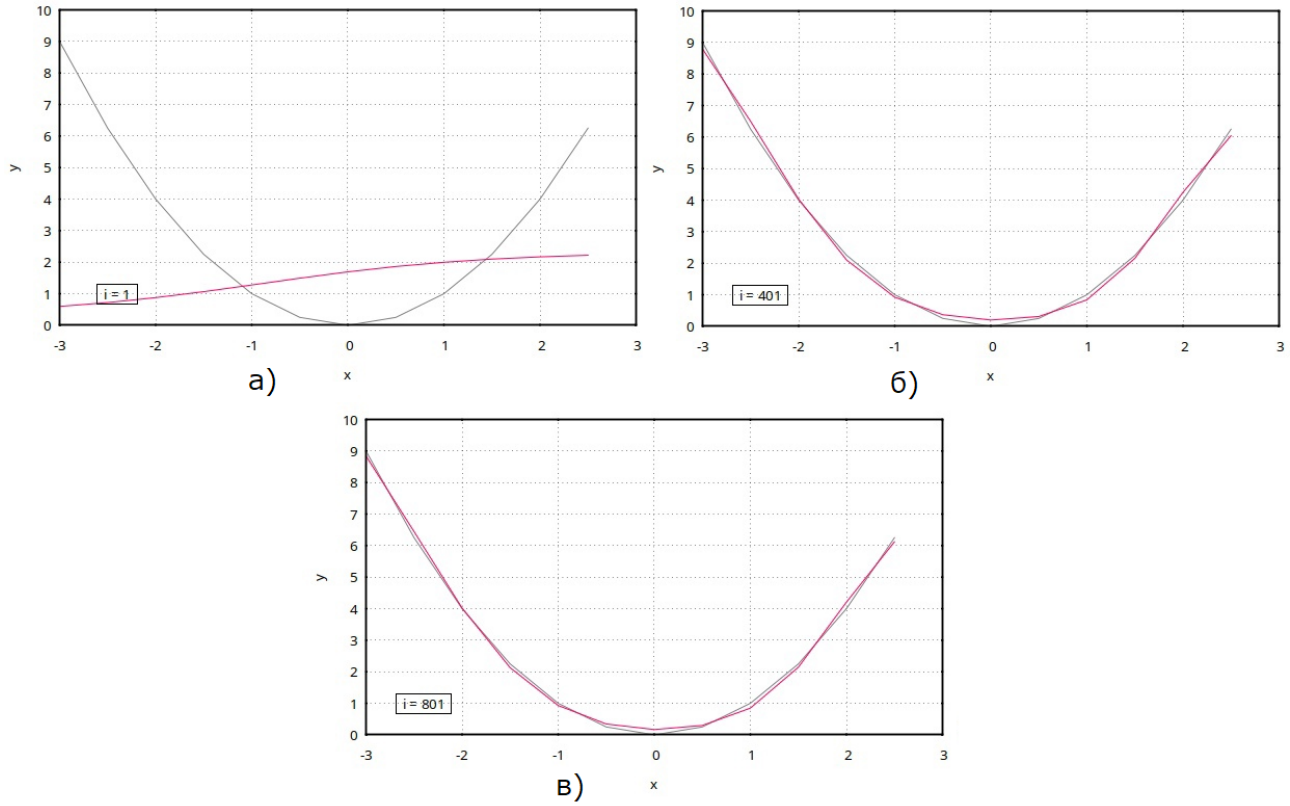

Рис. 4. Процесс аппроксимации функции $y = x^2$ на а) 1-й итерации, б) 401-й итерации, в) 801-й итерации. (4 нейрона, $E(W) = 0.103$)

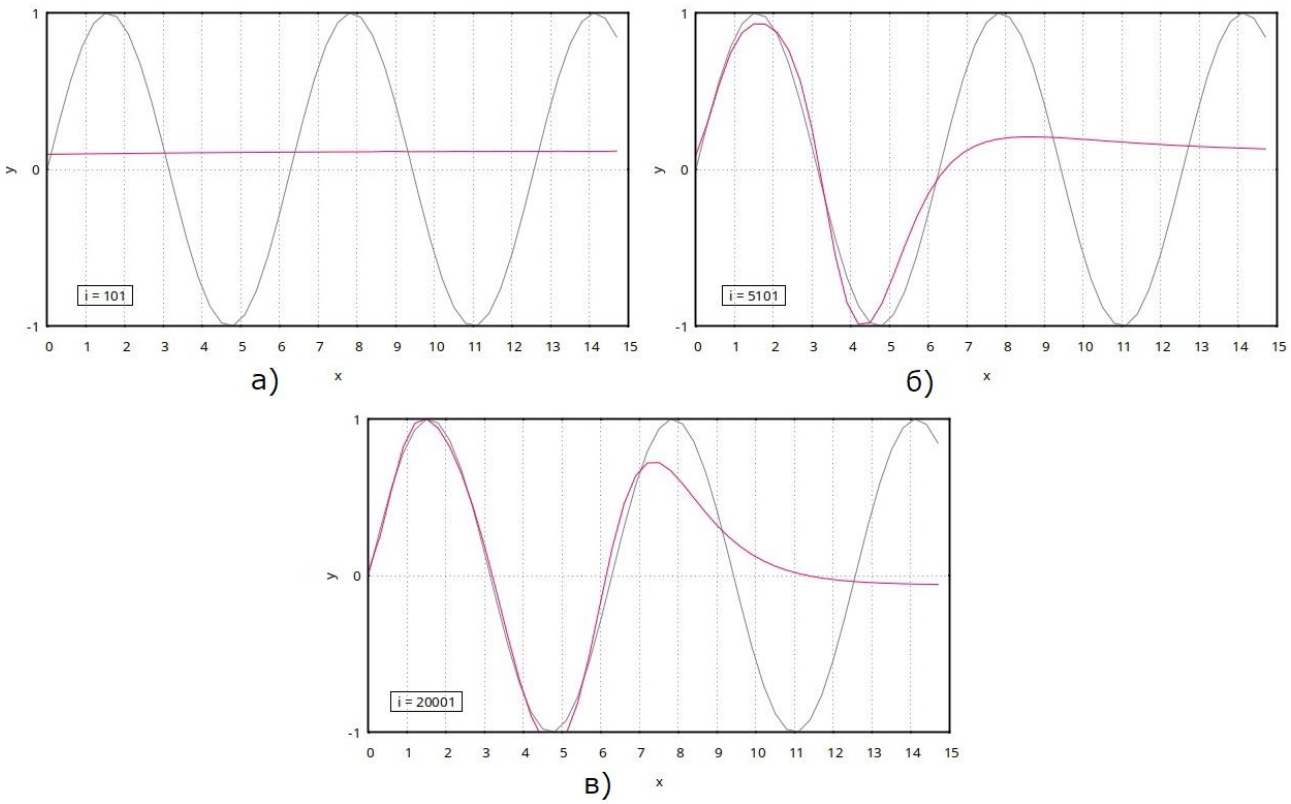

Рис. 5. Процесс аппроксимации функции $y = \sin(x)$ на а) 101-й итерации, б) 5101-й итерации, в) 20001-й итерации. (2 слоя по 4 нейрона, $E(W) = 5.792$)

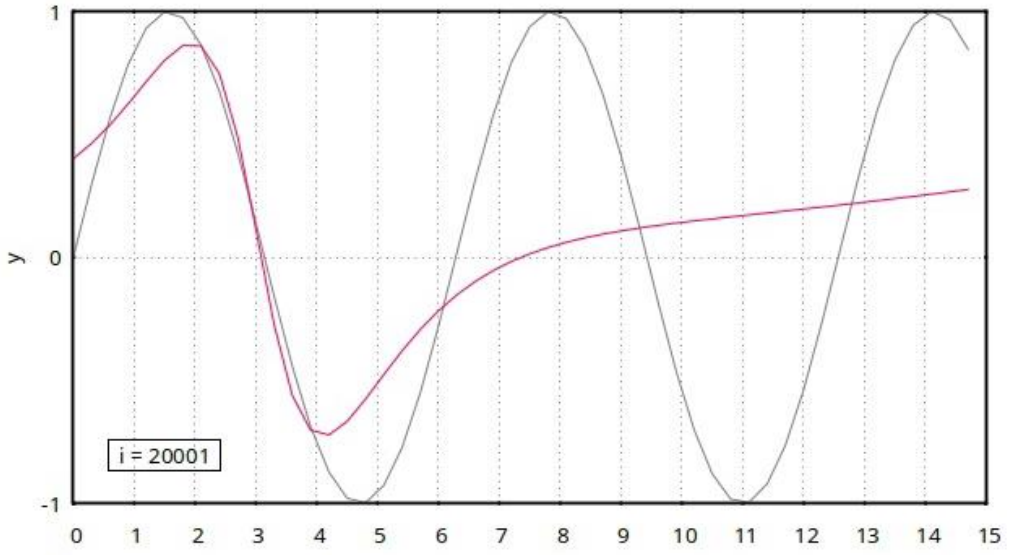

Рис. 6. Результат аппроксимации функции $y = \sin(x)$ на 20001-й итерации. (4 нейрона, $E(W) = 8.027$)

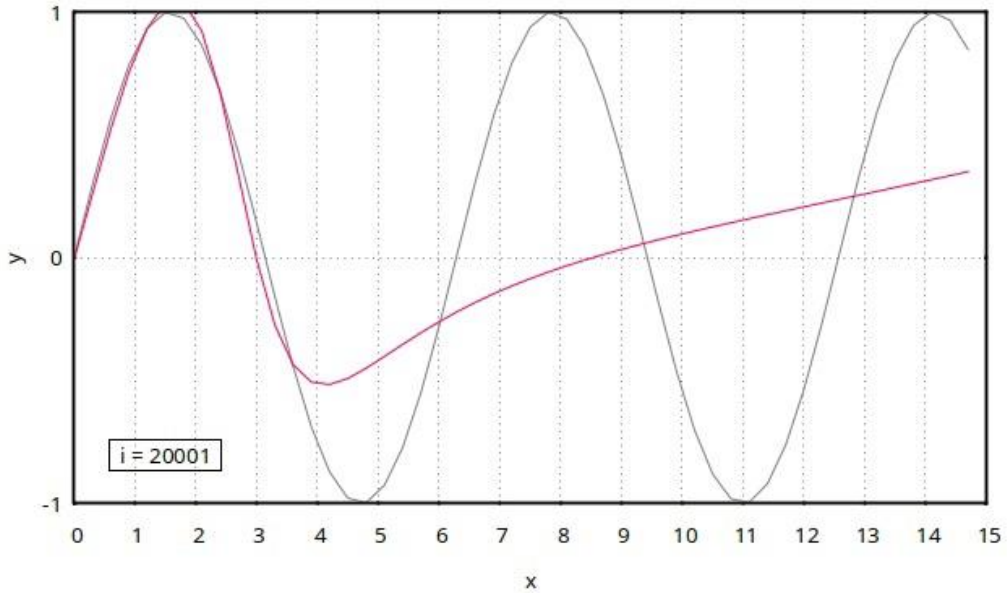

Рис. 7. Результат аппроксимации функции $y = \sin(x)$ на 20001-й итерации. (8 нейронов, $E(W) = 8.42$)

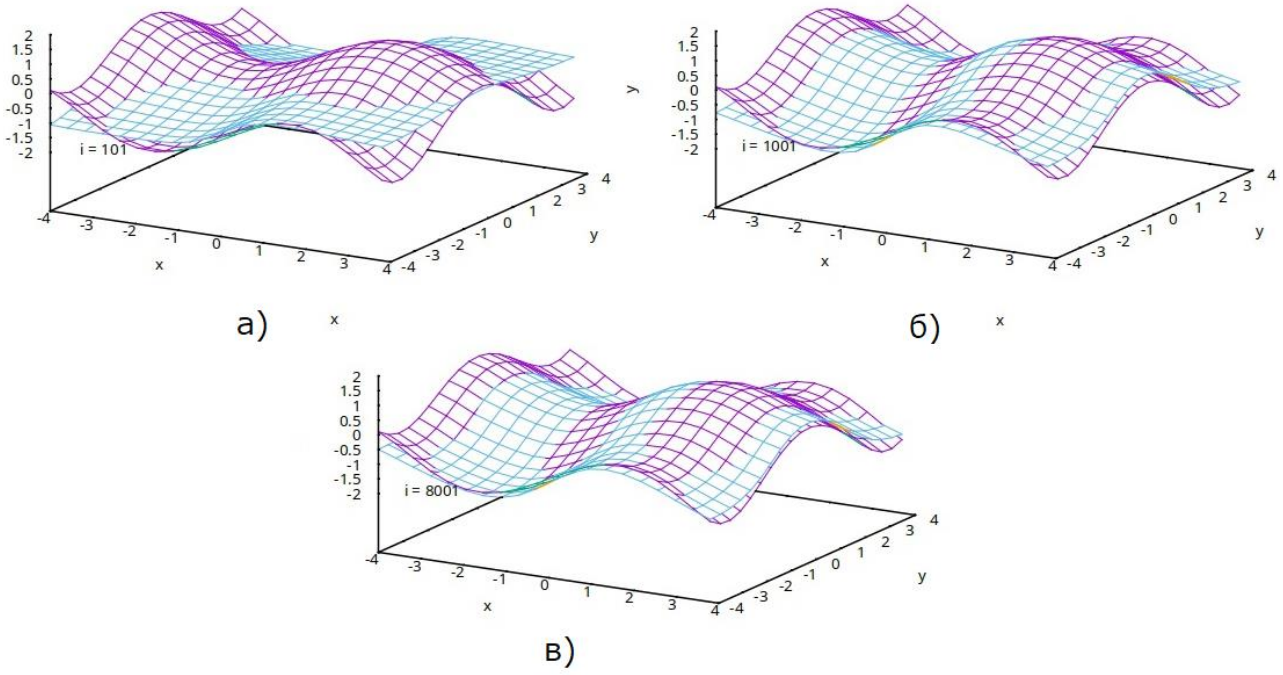

Рис. 8. Процесс аппроксимации функции $z = \sin(x) + \cos(y)$ на а) 101-й итерации, б) 1001-й итерации, в) 8001-й итерации. (4 нейрона, $E(W) = 21.1533$)

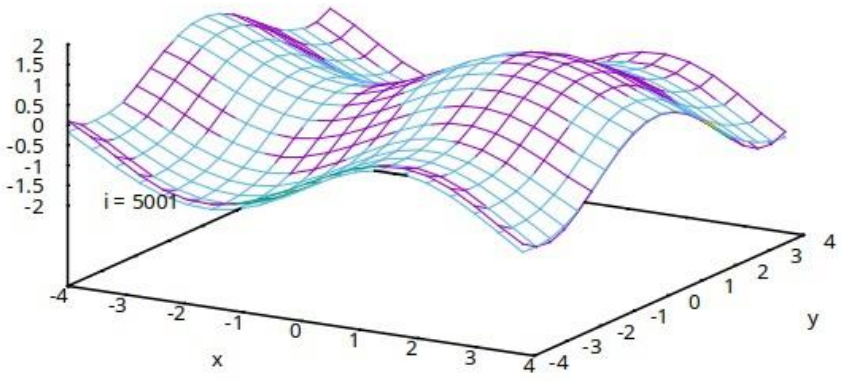

Рис. 9. Результат аппроксимации функции $z = \sin(x) + \cos(y)$ на 5001-й итерации. (8 нейронов, $E(W) = 4.85096$).

Анализ результатов демонстрирует корректность разработанной программы, а также тот факт, что для каждого случая существует своя оптимальная конфигурация сети, т.к. для параболы оказалось достаточно 4-х нейронов, для синусоиды наиболее оптимальные результаты показала двухслойная сеть по 4 нейрона, а для поверхности оптимальной конфигурацией является 8 нейронов. При этом, для аппроксимации периодических функций на больших интервалах (рис. 5-7) требуется больше времени обучения, т.к. даже на 20 тысячах итераций самая оптимальная конфигурация среди протестированных конфигураций сети не аппроксимировала график полностью, однако в ходе процесса наблюдается постепенная аппроксимация каждого «горба».

Текст программы

Листинг 1. Файл main.cpp

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

#include "config.hpp"
#include "SigmoidalNeuron.hpp"
#include "SigmoidalMLP.hpp"

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();

    std::vector<std::vector<double>> XLearn;
    std::vector<std::vector<double>> DLearn;


//    for (double x = 0; x < 15.0; x += 0.3) {
//        XLearn.emplace_back(std::vector<double>{x});
//        DLearn.emplace_back(std::vector<double>{std::sin(x)});
//    }


//    for (double x = -3; x < 3.0; x += 0.5) {
//        XLearn.emplace_back(std::vector<double>{x});
//        DLearn.emplace_back(std::vector<double>{x * x});
//    }

    for (double x = -4.0; x < 4.0; x += 0.2) {
        for (double y = -4.0; y < 4.0; y += 0.2) {
                XLearn.emplace_back(std::vector<double>{x, y});
                DLearn.emplace_back(std::vector<double>{std::sin(x) +
std::cos(y)});
        }
    }


    std::string initPlotFilename =
configuration.getVariable("INIT_PLOT_FILENAME");
    std::vector<std::vector<double>> initPlotData(XLearn.size());
    for (size_t k = 0; k < XLearn.size(); ++k) {
        initPlotData.at(k) = XLearn.at(k) + DLearn.at(k);
    }
    SigmoidalMLP::writePlotDataToFile(initPlotFilename, initPlotData);


    SigmoidalMLP mlpNetwork(2, 1, configuration);

    mlpNetwork.learn(XLearn, DLearn);


    return 0;
}
```

Листинг 2. Файл SigmoidalMLP.hpp

```
#ifndef SIGMOIDALMLP_H
#define SIGMOIDALMLP_H

#include <vector>
#include <fstream>
#include <cmath>

#include "SigmoidalNeuron.hpp"
#include "config.hpp"

std::ostream &operator<<(std::ostream &os, const std::vector<double> &input);
// Метод для конкатенации двух векторов
template<typename T>
std::vector<T> operator+(const std::vector<T>& v1, const std::vector<T>& v2);
```

```
class SigmoidalMLPLayer {
private:
    std::vector<SigmoidalNeuron> neurons;
    ConfigurationSingleton& configuration =
ConfigurationSingleton::getInstance();
public:
    SigmoidalMLPLayer() = default;
    SigmoidalMLPLayer& operator=(const SigmoidalMLPLayer& layer) {
        this->neurons = layer.neurons;
        this->configuration = layer.configuration;
        return *this;
    }
    SigmoidalMLPLayer(std::vector<SigmoidalNeuron> _neurons,
ConfigurationSingleton& _configuration)
        : neurons(_neurons), configuration(_configuration) {};

    explicit SigmoidalMLPLayer(size_t neuronsNumber, size_t prevLayerNeuronsNum,
ConfigurationSingleton& configuration);
    size_t getNumberOfNeurons() { return this->neurons.size(); }
    std::vector<SigmoidalNeuron>& getNeurons() { return this->neurons; };
};

class SigmoidalMLP {
private:
    std::vector<SigmoidalMLPLayer> layers;
    std::ofstream plotsFile;
    std::pair<double, double> outputScale;
    ConfigurationSingleton& configuration;

    double targetFunction(std::vector<std::vector<double>>& D,
std::vector<std::vector<double>>& Y);

    void appendPlotToFile(std::vector<std::vector<double>>& plotData);

    // Выходные значения каждого предыдущего слоя каждого нейрона после
последнего вызова getOutput
    std::vector<std::vector<double>> tmpOutputs;
public:
    explicit SigmoidalMLP(size_t inputSize, size_t outputSize,
ConfigurationSingleton& _configuration);

    void learn(std::vector<std::vector<double>>& XLearn,
std::vector<std::vector<double>>& DLearn);
    std::vector<double> getOutput(std::vector<double>& X);
    std::vector<double> getOutput(std::vector<double>& X,
std::vector<SigmoidalMLPLayer> extLayers);

    static void writePlotDataToFile(const std::string& filename,
std::vector<std::vector<double>>& plotData);

    ~SigmoidalMLP() = default;
};

#endif // SIGMOIDALMLP_H
```

Листинг 3. Файл Sigmoidal.cpp

```
#include <string>
#include <iostream>
#include <ctime>
#include <cmath>
#include "SigmoidalNeuron.hpp"
```

```
#include "SigmoidalMLP.hpp"

std::ostream &operator<<(std::ostream &os, const std::vector<double> &input) {
    os.precision(3);
    os << "{";
    for (auto const &i: input) {
        os << std::fixed << i << " ";
    }
    os << "}";
    return os;
}

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;
}

SigmoidalMLPLayer::SigmoidalMLPLayer(size_t neuronsNumber, size_t
prevLayerOutputsNum, ConfigurationSingleton& _configuration)
        : configuration(_configuration) {
    for (size_t i = 0; i < neuronsNumber; ++i) {
        this->neurons.emplace_back(SigmoidalNeuron(prevLayerOutputsNum + 1,
this->configuration));
    }
};

SigmoidalMLP::SigmoidalMLP(size_t inputSize, size_t outputSize,
ConfigurationSingleton& _configuration)
        : configuration(_configuration) {
    srand(time(nullptr));
    size_t layersNum = configuration.getVariableInt("MLP_LAYERS_NUMBER");

    for (size_t i = 0; i < layersNum; ++i) {
        size_t neuronsNum = configuration.getVariableInt("MLP_LAYER" +
std::to_string(i + 1) + "_NEURONS");
        // Если слой первый - то количество входов соотвтвует размерности данных
        size_t prevLayerOutputsNum = inputSize;
        if (i > 0) {
            // Иначе - количеству нейронов на прошлом слое
            prevLayerOutputsNum = configuration.getVariableInt("MLP_LAYER" +
std::to_string(i) + "_NEURONS");
        }
        this->layers.emplace_back(SigmoidalMLPLayer(neuronsNum,
prevLayerOutputsNum, this->configuration));
    }

    size_t lastLayerSize = this->layers.at(this->layers.size() -
1).getNumberOfNeurons();
    this->layers.emplace_back(SigmoidalMLPLayer(outputSize, lastLayerSize, this-
>configuration));

    std::string plotsFilename = configuration.getVariable("PLOTS_FILENAME");

    this->plotsFile.open(plotsFilename);
    if (!this->plotsFile.is_open()) {
        std::cerr << "Error: can't open file " << plotsFilename << std::endl;
        throw std::runtime_error(std::string("Error: can't open file ") +
plotsFilename);
    }
}
```

```
void SigmoidalMLP::appendPlotToFile(std::vector<std::vector<double>>& plotData)
{
    this->plotsFile << std::endl;
    for (size_t point = 0; point < plotData.size(); ++point) {
        for (size_t i = 0; i < plotData.at(point).size(); ++i) {
            this->plotsFile << plotData.at(point).at(i) << " ";
        }
        this->plotsFile << std::endl;
    }
}

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


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

    return 0.5 * E;
}

void SigmoidalMLP::learn(std::vector<std::vector<double>>& XLearn,
std::vector<std::vector<double>>& DLearn) {
    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");

    // Заполнить файл обучения первоначальным графиком
    std::vector<std::vector<double>> initialMlpPlot(XLearn.size());
    for (size_t k = 0; k < XLearn.size(); ++k) {
        initialMlpPlot.at(k) = XLearn.at(k) + this->getOutput(XLearn.at(k));
    }
    this->appendPlotToFile(initialMlpPlot);

    size_t iterations = 1;
    while (learningCoef > learningCoefEPS && currentTargetFunc > targetFuncEPS)
{
        // Выходные значения с прошлого слоя для каждой выборки каждого слоя
каждого нейрона
        std::vector<std::vector<std::vector<double>>> prevLayersOutputs;

        // Накопленная ошибка нейрона для каждой выборки каждого нейрона
        std::vector<std::vector<double>> currLayerErrors;
```

```
        std::vector<std::vector<double>> leftLayerErrors;

        // Инициализация ошибок на последнем (выходном) слое для каждой выборки
        for (size_t k = 0; k < XLearn.size(); ++k) {
            // Получение результата работы сети для k-й обучающей выборки
            std::vector<double> mlpOutput = this->getOutput(XLearn.at(k));
            prevLayersOutputs.push_back(this->tmpOutputs);
            std::vector<double> errors;

            // Заполнение ошибок (y_i - d_i) для каждого нейрона выходного слоя
            for (size_t i = 0; i < mlpOutput.size(); ++i) {
                errors.push_back(mlpOutput.at(i) - DLearn.at(k).at(i));
            }
            leftLayerErrors.push_back(errors);
        }

        // Массив скорректированных весов для каждого слоя для каждого нейрона
        std::vector<std::vector<std::vector<double>>> correctedWeights(this-
>layers.size());

        // Для каждого слоя
        for (int l = this->layers.size() - 1; l >= 0; --l) {
            std::vector<SigmoidalNeuron> layerNeurons = this-
>layers.at(l).getNeurons();
            currLayerErrors = leftLayerErrors;

            // Для каждого нейрона в слое
            for (size_t n = 0; n < layerNeurons.size(); ++n) {
                SigmoidalNeuron& neuron = layerNeurons.at(n);
                std::vector<double> weights = neuron.getWeights();
                correctedWeights.at(l).resize(layerNeurons.size());

                if (l != this->layers.size() - 1) {
                    // Умножение ошибок на производную
                    for (size_t k = 0; k < XLearn.size(); ++k) {
                        currLayerErrors.at(k).at(n) *=
neuron.getDerivative(prevLayersOutputs.at(k).at(l));
                    }
                }

                // Для каждого синапса (связи)
                for (size_t j = 0; j < neuron.getInputsNumber(); ++j) {

correctedWeights.at(l).at(n).resize(neuron.getInputsNumber());

                    double derivative = 0.0;

                    // Для каждой обучающей выборки
                    for (size_t k = 0; k < XLearn.size(); ++k) {
                        // xj - входное значение нейрона связи j
                        double xj = 1.0;
                        // Первая связь - поляризация
                        if (j > 0) {
                            // Выходные значения нейронов на предыдущем слое
являются входными для текущего
                            // (j - 1) т.к. j = 0 - поляризация и связи с
нейронами начинаются с j = 1
                            xj = prevLayersOutputs.at(k).at(l).at(j - 1);
                        }

                        derivative += currLayerErrors.at(k).at(n) * xj;
                    }
                    // Коррекция веса
```

```
                    correctedWeights.at(l).at(n).at(j) = weights.at(j) -
learningCoef * derivative;
                }
            }

            // Расчет ошибок для обратного распространения
            if (l > 0) {
                std::vector<SigmoidalNeuron> leftLayerNeurons = this-
>layers.at(l - 1).getNeurons();

                for (size_t k = 0; k < XLearn.size(); ++k) {
                    leftLayerErrors.at(k).resize(leftLayerNeurons.size());

                    // Для каждого нейрона из предыдущего слоя
                    for (size_t nLeft = 0; nLeft < leftLayerNeurons.size();
++nLeft) {
                        double error = 0.0;

                        for (size_t n = 0; n < layerNeurons.size(); ++n) {
                            SigmoidalNeuron& neuron = layerNeurons.at(n);
                            // Взвешиваем ошибку синапсом между левым и текущим
нейроном
                            error += currLayerErrors.at(k).at(n) *
neuron.getWeights().at(nLeft + 1);
                        }
                        leftLayerErrors.at(k).at(nLeft) = error;
                    }
                }
            }
        }

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

        std::vector<SigmoidalMLPLayer> correctedLayers;
        for (size_t l = 0; l < this->layers.size(); ++l) {
            size_t numberOfNeurons = this->layers.at(l).getNeurons().size();
            size_t numberOfInputs = this-
>layers.at(l).getNeurons().at(0).getInputsNumber() - 1;
            correctedLayers.emplace_back(SigmoidalMLPLayer(numberOfNeurons,
numberOfInputs, this->configuration));
        }

        for (size_t l = 0; l < correctedLayers.size(); ++l) {
            std::vector<SigmoidalNeuron>& neurons =
correctedLayers.at(l).getNeurons();
            for (size_t n = 0; n < neurons.size(); ++n) {
                SigmoidalNeuron& neuron = neurons.at(n);
                neuron.setWeights(correctedWeights.at(l).at(n));
                SigmoidalNeuron& neuronCompare = neurons.at(n);
            }
        }

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

        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->layers = correctedLayers;
            currentTargetFunc = targetFuncCurr;

            size_t iterationPrintStep = this-
>configuration.getVariableInt("PRINT_EVERY_ITERATION");

            if (iterations % iterationPrintStep == 0) {
                std::cout << "Current target function value: " <<
currentTargetFunc << std::endl;
                std::cout << "EPOCH: " << iterations << std::endl;
            }
            ++iterations;

            std::vector<std::vector<double>> plotData(currE.size());
            for (size_t k = 0; k < currE.size(); ++k) {
                plotData.at(k) = XLearn.at(k) + currE.at(k);
            }
            this->appendPlotToFile(plotData);
        } else {
//            std::cout << "Failed iteration" << std::endl;
            learningCoef /= 2.0;
            successIterations = 0;
        }
    }

    std::cout << "Learning completed in " << iterations << " epochs." <<
std::endl;
}

// Возвращает выходные значения для каждого нейрона в выходном слое
std::vector<double> SigmoidalMLP::getOutput(std::vector<double>& X) {
    std::vector<double> prevLayerOutputs = X;
    std::vector<double> currLayerOutputs;

    this->tmpOutputs.resize(this->layers.size());

    for (size_t l = 0; l < this->layers.size(); ++l) {
        std::vector<SigmoidalNeuron>& layerNeurons = this-
>layers.at(l).getNeurons();
        currLayerOutputs.resize(layerNeurons.size());
        this->tmpOutputs.at(l).resize(layerNeurons.size());

        if (l != this->layers.size() - 1) {
            for (size_t n = 0; n < layerNeurons.size(); ++n) {
                currLayerOutputs.at(n) =
layerNeurons.at(n).getOutput(prevLayerOutputs);
            }
        } else {
            for (size_t n = 0; n < layerNeurons.size(); ++n) {
                auto tmpWeights = layerNeurons.at(n).getWeights();
                currLayerOutputs.at(n) =
layerNeurons.at(n).getUSum(prevLayerOutputs, tmpWeights);
            }
        }

        this->tmpOutputs.at(l) = prevLayerOutputs;
```

```
        prevLayerOutputs = currLayerOutputs;
    }

    return prevLayerOutputs;
}

// Возвращает выходные значения для каждого нейрона в выходном слое для
указанной конфигурации сети
std::vector<double> SigmoidalMLP::getOutput(std::vector<double>& X,
std::vector<SigmoidalMLPLayer> extLayers) {
    std::vector<double> prevLayerOutputs = X;
    std::vector<double> currLayerOutputs;

    for (size_t l = 0; l < extLayers.size(); ++l) {
        std::vector<SigmoidalNeuron> layerNeurons =
extLayers.at(l).getNeurons();
        currLayerOutputs.resize(layerNeurons.size());

        if (l != this->layers.size() - 1) {
            for (size_t n = 0; n < layerNeurons.size(); ++n) {
                currLayerOutputs.at(n) =
layerNeurons.at(n).getOutput(prevLayerOutputs);
            }
        } else {
            for (size_t n = 0; n < layerNeurons.size(); ++n) {
                auto tmpWeights = layerNeurons.at(n).getWeights();
                currLayerOutputs.at(n) =
layerNeurons.at(n).getUSum(prevLayerOutputs, tmpWeights);
            }
        }

        prevLayerOutputs = currLayerOutputs;
    }

    return prevLayerOutputs;
}
```

Листинг 4. Файл config.cfg

```
SIGMA = 1
LEARNING_COEF_INIT = 0.5
LEARNING_COEF_EPS = 1e-6
TARGET_FUNC_EPS = 5e-4

MLP_LAYERS_NUMBER = 1
MLP_LAYER1_NEURONS = 4

PRINT_EVERY_ITERATION = 100

PLOTS_FILENAME = plots.dat
INIT_PLOT_FILENAME = init_plot.dat
```

Листинг 5. Файл gnuplot_surface_window.gp

```
reset
set term wxt

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

color_line = '#3e3c3d'
```

```
set border linewidth 1.5
set pointsize 1

# стиль для тестовых данных с классом 1 (класс 0 отображается дефолтным стилем)
set style line 1 lc rgb '#808080' pt 7   # circle gray

# стиль для весов
set style line 2 lc rgb '#fc0362' pt 9   # triangle red

set hidden3d
set dgrid3d 20,20 qnorm 2

unset key

unset colorbox

set tics scale 0.1
set xtics 1
set ytics 1
set yrange[-4:4]
set xrange[-4:4]
set zrange[-2:2]
set xlabel 'x'
set ylabel 'y'

DOTS = system("wc -l init_plot.dat")
N = system("wc -l plots.dat") / (DOTS + 1)

do for [i=0:N:1000] {
    LABEL = "i = " . (i+1)
    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

    splot 'init_plot.dat' w l ls 1, 'plots.dat' every :::i::i w l ls 2
}
pause mouse
```