Содержание

Лабораторная работа № 2

Задание (вариант 53)

Дано алгебраическое уравнение над целыми числами, в котором знаки операций (+, -, *, /) заменены буквами (например, *Z, Y, X, ...*). Написать универсальную программу, отыскивающую подстановку для букв, превращающую уравнение в тождество. Пример.

Уравнение: (3 *Z* 5) *Y* 12 = 27

Подстановка: *Z* = *, *Y* = +

Исходный код

```
% parse input string into list of symbols
token_parser([], [')']).


% skip whitespaces
token_parser([SymCode | InputSymbols], Tokens) :-
    char_type(SymCode, white),
    token_parser(InputSymbols, Tokens).


%parse upper letters (Z, Y, X, ...) as variables
token_parser([SymCode | InputSymbols], [_ | Tokens]) :-
    char_type(SymCode, upper),
    token_parser(InputSymbols, Tokens).


% parse digits
token_parser([SymCode | InputSymbols], [ParsedToken | Tokens]) :-
    char_type(SymCode, digit),
    ParsedToken is SymCode - 48,
    token_parser(InputSymbols, Tokens).


% parse parentheses and operators
token_parser([40 | InputSymbols], ['(' | Tokens]) :- token_parser(InputSymbols,
Tokens).
token_parser([41 | InputSymbols], [')' | Tokens]) :- token_parser(InputSymbols,
Tokens).
token_parser([42 | InputSymbols], ['*' | Tokens]) :- token_parser(InputSymbols,
Tokens).
token_parser([43 | InputSymbols], ['+' | Tokens]) :- token_parser(InputSymbols,
Tokens).
token_parser([45 | InputSymbols], ['-' | Tokens]) :- token_parser(InputSymbols,
Tokens).
token_parser([47 | InputSymbols], ['/' | Tokens]) :- token_parser(InputSymbols,
Tokens).
```

```
token_parser([61 | InputSymbols], ['-', '(' | Tokens]) :- token_parser(InputSymbols,
Tokens).


% parse list of symbols into list of lexems
lexem_parser([], []).
lexem_parser([Token], [Token]).

% parse variables or parentheses or operators
lexem_parser([Token1, Token2 | Tokens], [Token1 | Lexems]) :-
    (var(Token1); atom(Token1)),
    lexem_parser([Token2 | Tokens], Lexems).

% parse single-digit number
lexem_parser([Token1, Token2 | Tokens], [Token1 | Lexems]) :-
    integer(Token1),
    (var(Token2); atom(Token2)),
    lexem_parser([Token2 | Tokens], Lexems).

% parse number from digits
lexem_parser([Token1, Token2 | Tokens], Lexems) :-
    integer(Token1),
    integer(Token2),
    Number is Token1 * 10 + Token2,
    lexem_parser([Number | Tokens], Lexems).

% syntax Definite Clause Grammar parser
ex(E) --> eterm(E).
ex([S, E1, E2]) --> sterm(E1), sn(S), eterm(E2).

sterm(E) --> eterm(E).
sterm([S, E1, E2]) --> eterm(E1), sn(S), eterm(E2).
sterm([S2, [S1, E1, E2], E3]) --> eterm(E1), sn(S1), sterm(E2), sn(S2), eterm(E3).

eterm(E) --> fct(E).
eterm([S2, [S1, E1, E2], E3]) --> fct(E1), sn2(S1), eterm(E2), sn2(S2), fct(E3).
eterm([S, E1, E2]) --> fct(E1), sn2(S), fct(E2).

sn2(X) --> [X], {var(X)}.
sn2(*) --> [*].
sn2(/) --> [/].
```

```
fct(E) --> number(E).
fct(E) --> lb, ex(E), rb.
fct(E) --> sn(E), fct(E).

number(X) --> [X], {integer(X)}.

lb --> ['('].
rb --> [')'].

sg(X) --> [X], {var(X)}.
sg(+) --> [+].
sg(-) --> [-].
sn(E) --> sg(E).

syntax_parser(Lexems, Expression) :- ex(Expression, Lexems, []).

calc([S, A1, A2], Nr) :-
    calc(A1, N1),
    calc(A2, N2),
    calc1(S, N1, N2, Nr).

calc(A1, A1) :- A1 \= [_ | _].
calc1(*, N1, N2, Nr) :- Nr is N1 * N2.
calc1(/, N1, N2, Nr) :- Nr is N1 / N2.
calc1(+, N1, N2, Nr) :- Nr is N1 + N2.
calc1(-, N1, N2, Nr) :- Nr is N1 - N2.

lab2 :-
    read(Input),
    string_codes(Input, InputSymCodes),
    token_parser(InputSymCodes, Tokens),
    write("Tokens = "), write(Tokens), nl,
    lexem_parser(Tokens, Lexems),
    write("Lexems = "), write(Lexems), nl,
    syntax_parser(Lexems, Expression),
    write("Source Expression = "), write(Expression), nl, !,
    calc(Expression, Result),
    Result = 0,
    write("Result Expression = "), write(Expression), nl.
```

Лабораторная работа № 3

Задание (вариант 5-7)

Разработать, используя язык C/C++, программу, моделирующую поведение искусственного трехвходового нейрона указанного преподавателем типа и обеспечивающую его обучение для решения задачи классификации.

Краткое описание нейронов типа *WTA*

Нейроны типа *WTA* (*Winner Takes All* — победитель получает все) всегда используются группами, в которых конкурируют между собой. Структурная схема группы (слоя) нейронов типа *WTA* представлена на рис. 2.1.

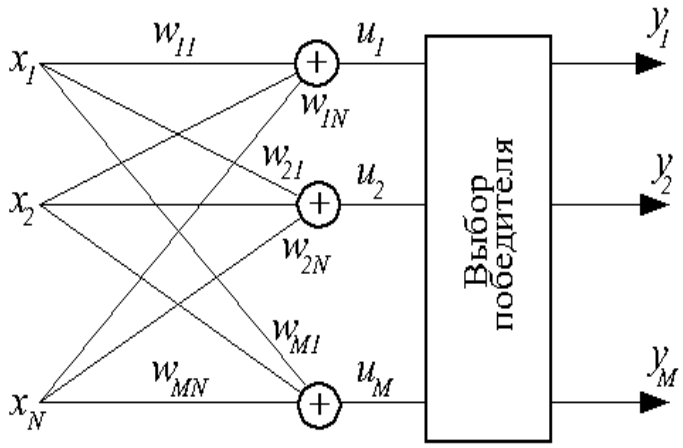

Рис. 2.1. Структурная схема слоя нейронов типа *WTA*

Нейроны типа *WTA* не имеют входов поляризации и функции активации на выходе, т. е. представляют собой обычные сумматоры. Вектора весов каждого нейрона и входные вектора должны быть всегда нормализованы. Нормализация векторов производится по формуле:

$$x_i^{normalized} = \frac{x_i}{\|X\|} = \frac{x_i}{\sqrt{\sum_{j=1}^{N} x_j^2}}$$

Обучение нейронов типа *WTA* осуществляется без учителя. Нейроны данного типа решают задачу кластеризации самостоятельно.

На вход каждого нейрона в рамках одной группы поступают одинаковые входные сигналы. После вычисления взвешенных сумм нейронов выполняется процедура определения нейрона-победителя. В результате выполнения этой процедуры выходной сигнал нейрона-победителя будет иметь значение 1, выходные сигналы остальных нейронов будут иметь значение 0. Для нейрона-победителя выполняется процедура корректировки весов; веса остальных нейронов остаются прежними.

Корректировка весов осуществляется по правилу Гроссберга с учетом того, что значение выходного сигнала нейрона-победителя равно 1:

$$w_{ij}(t+1) = w_{ij} + \eta \cdot (x_j^k - w_{ij}(t)) \quad ,$$

где η — коэффициент обучения, значение которого выбирается в диапазоне (0, 1); t — шаг корректировки, k — номер вектора из обучающей выборки.

В каждом цикле обучения побеждает тот нейрон, чей текущий вектор входных весов

W_i наиболее близок входному вектору X^k . При этом вектор W_i корректируется в сторону вектора X^k . Поэтому в ходе обучения каждая группа близких друг другу входных векторов (кластер) обслуживается отдельным нейроном.

Понятие «близости» двух векторов можно продемонстрировать на следующем примере. Пусть на очередной итерации обучения сети в режиме «онлайн» имеется входной вектор X^k . Тогда вычисление взвешенной суммы для *i*-го нейрона осуществляется по формуле:

$$u_i = W_i^T \cdot X^k = \sum_{j=1}^{N} w_{ij} \cdot x_j^k = \|W_i^T\| \cdot \|X^k\| \cdot \cos\left(\measuredangle W_i, X^k\right)$$

Поскольку вектора W_i и X^k нормализованы, то взвешенная сумма *i*-го нейрона равна косинусу угла между вектором весов и входным вектором. На рис. 2.2 представлена геометрическая интерпретация. Чем ближе весовой вектор ко входному, тем ближе косинус угла между векторами к 1.

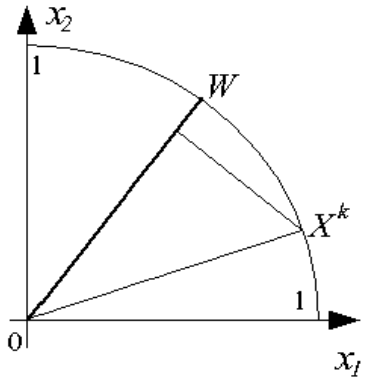

Рис. 2.2. Геометрическая
интерпретация весов нейронов

В режиме кластеризации при подаче на вход слоя нейронов типа *WTA* очередного вектора X^k определяется степень его близости к векторам W_i в виде косинусов углов между этими векторами, после чего определяется наиболее «близкий» вектор весов, отвечающий за тот или иной кластер.

Серьезная проблема в использовании нейронов типа WTA — возможность возникновения "мертвых" нейронов, т.е. нейронов, ни разу не победивших в конкурентной борьбе в ходе обучения и поэтому оставшихся в начальном состоянии. Для исключения "ложных" срабатываний в режиме классификации мертвые нейроны после окончания обучения должны быть удалены.

Для уменьшения количества мертвых нейронов (и, следовательно, повышения точности распознавания) используется модифицированное обучение, основанное на учете числа побед нейронов и штрафовании наиболее "зарвавшихся" среди них. Дисквалификация может быть реализована либо назначением порога числа побед, после которого слишком активный нейрон "засыпает" на заданное число циклов обучения, либо искусственным уменьшением величины взвешенной суммы нейрона пропорционально числу побед.

Реализация

Конфигурация слоя нейронов типа *WTA* сформирована в зависимости от вида обучающей выборки, а также следующих параметров: количество нейронов, количество эпох обучения, размер обучающей выборки, величина коэффициента обучения, величина штрафа при обучении. Описание отдельных параметров приведено ниже.

Множества точек, формирующие обучающую выборку, представлены на рис 2.3. Те же множества, но в нормализованном виде, представлены на рисунке рис. 2.4.

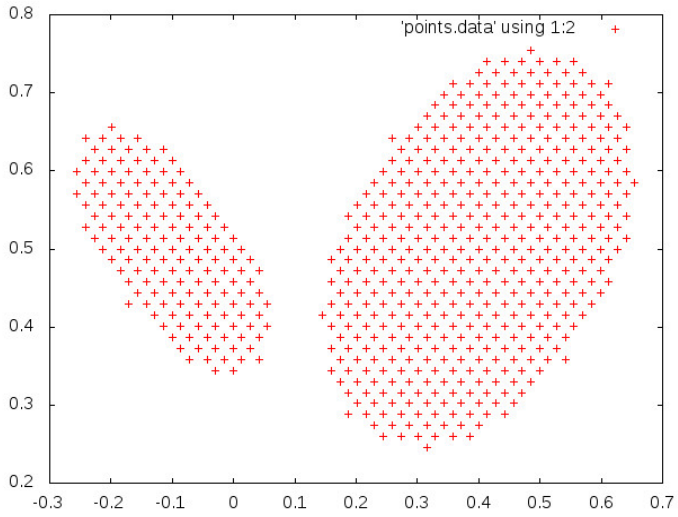

Рис. 2.3. Обучающая выборка

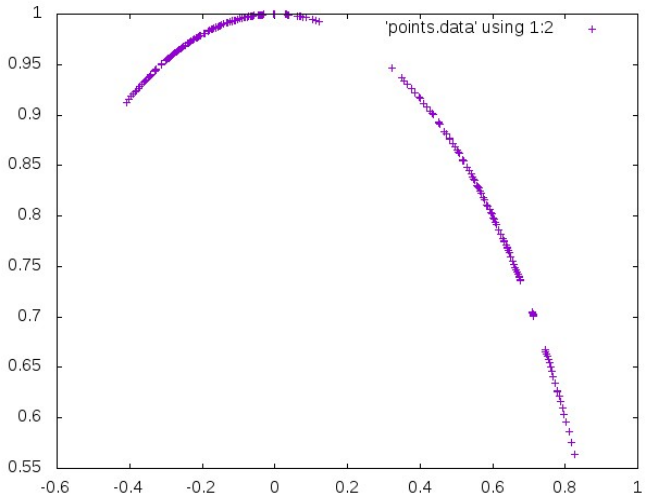

Рис. 2.4. Нормализованная обучающая выборка

Для обучения были выбраны следующие значения параметров:

1. Размеры обучающих множеств: 104 и 117 точек для левого и правого эллипсов из рис 2.3 соответственно. Важно, чтобы мощности обучающих множеств содержали приблизительно одинаковое количество элементов, тогда обучение будет проходить без «перевеса» в сторону того или иного множества.
2. Размерность входных векторов: 2.
3. Количество нейронов: 4.
4. Количество эпох обучения: 2.
5. Коэффициент обучения: 0,1.
6. Коэффициент штрафа при обучении: 0,01.
7. Начальные значения весов: для всех весов были заданы координаты (-1, -1).

Обучение слоя проводилось в режиме «онлайн». Это означает, что перерасчет весов для нейронов-победителей осуществлялся после подачи каждого входного вектора на вход слоя.

Обучающее множество было перемешано случайным образом.

Коэффициент штрафа при обучении необходим для решения проблемы «мертвых» нейронов. Решение данной проблемы заключается в следующем: для каждого нейрона в слое вводится история побед. История побед изменяется на каждой итерации обучения для нейронов-победителей. Эта история используется для коррекции взвешенных сумм нейронов перед процедурой определения нейрона-победителя. Взвешенная сумма каждого нейрона рассчитывается по формуле:

$$u_i = \left(\sum_{j=1}^{N} w_{ij} \cdot x_j\right) - p \cdot N_i^{wins} \quad ,$$

где p — коэффициент штрафа при обучении, N_i^{wins} - количество побед i-го нейрона на очередной итерации обучения. В итоге, для «зарвавшихся» нейронов взвешенные суммы будут уменьшаться по мере роста количества побед, что даст возможность остальным нейронам участвовать в процессе обучения.

Результаты обучения слоя нейронов при заданных параметрах представлены на рис. 2.4. Результирующие веса нейронов представлены в табл. 2.1.

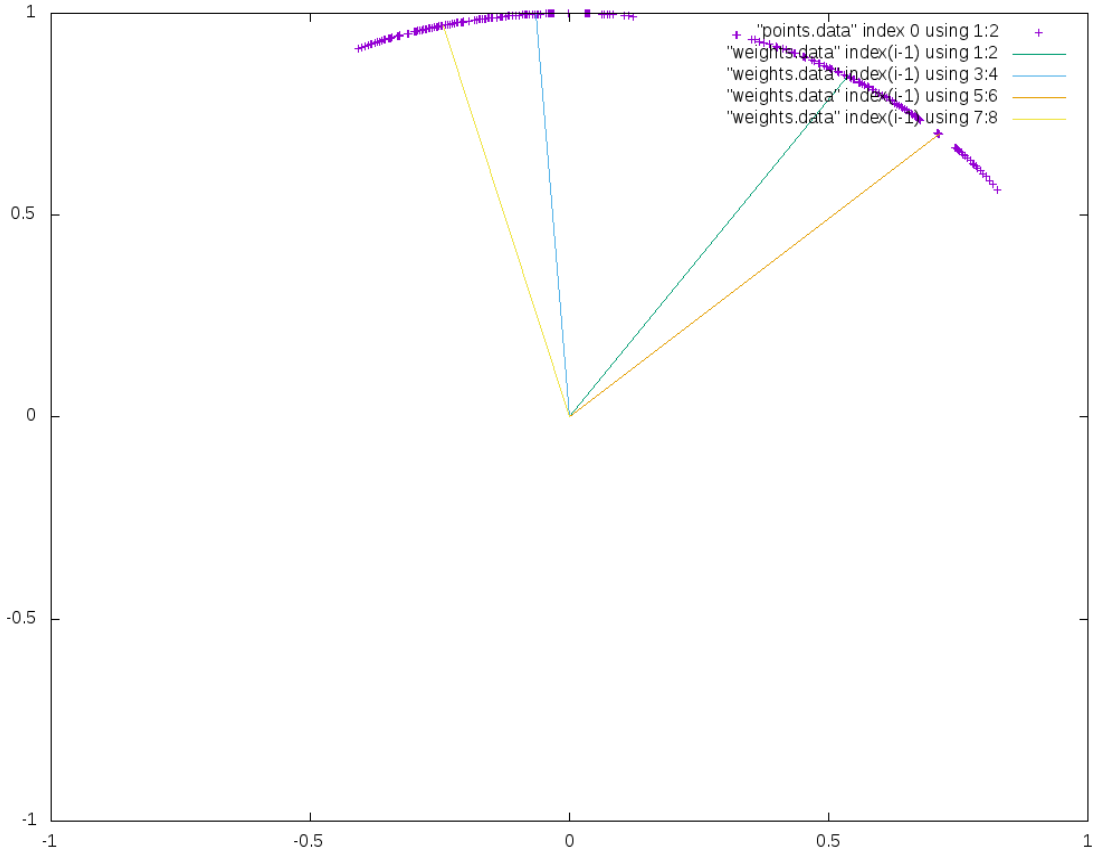

Рис. 2.4. Результаты обучения слоя нейронов типа *WTA*

Табл. 2.1. Результирующие веса нейронов после обучения слоя.

Номер нейрона / Индекс веса	1	2
1	0.5363105409204906	0.8440207365329189
2	-0.06510319629862805	0.9978785366123987
3	0.7136365788272219	0.7005161192719112
4	-0.2441630786117885	0.9697341857657765

Листинг программы

WTALayer.h

```
#ifndef LAB3_WTA_LAYER_H
#define LAB3_WTA_LAYER_H

#include <vector>
#include <iostream>
#include <fstream>
using size_t = std::size_t;

class WTALayer {
public:
    struct Config {
        size_t neurons;
        size_t inVecDim;
        int    trainEpochs;
        double trainCoeff;
        double trainPenalty;
    };

    void Init(const Config &conf, bool randomizeWeights = true);
    bool SetWeights(const std::vector<double> &initialWeights);
    size_t Test(const std::vector<double> &inVec);

    void Train(const std::vector<std::vector<double>> &trainSet,
        std::ostream &output);

    void DumpWeights(std::ostream &output, int precision,
        const std::string &title = "", bool gnuplot = false);

private:
    Config config;
    std::vector<double>   weights;
    std::vector<uint64_t> winHistory;

    void RandomizeWeights();
    std::vector<double> GetWeightedSums(const std::vector<double> &inVec);
    size_t DetectWinner(const std::vector<double> &weightedSums);
    void AdjustWeights(size_t iWinner, const std::vector<double> &inVec);
```

```
    void AdjustWinHistory(size_t iWinner) { winHistory[iWinner] += 1; }
};

#endif // LAB3_WTA_LAYER_H
```

WTALayer.cpp

```
#include "WTALayer.h"
#include <iomanip>
#include <random>
#include <algorithm>

void WTALayer::Init(const Config &conf, bool randomizeWeights) {
    config = conf;

    if (!randomizeWeights)
        return;

    weights.resize(config.neurons * config.inVecDim);
    winHistory.resize(config.neurons * config.inVecDim, 0);
    RandomizeWeights();
}

bool WTALayer::SetWeights(const std::vector<double> &initialWeights) {
    if (initialWeights.size() != config.neurons * config.inVecDim) {
        std::cerr << "invalid weights size: " << initialWeights.size() << std::endl;
        return false;
    }

    weights = initialWeights;
    winHistory.resize(config.neurons * config.inVecDim, 0);

    return true;
}

void WTALayer::RandomizeWeights() {
    const double WEIGHT_INF = -1.0;
    const double WEIGHT_SUP = 1.0;

    std::random_device randomizer;
```

```
    std::mt19937 randGen(randomizer());
    std::uniform_real_distribution<> dist(WEIGHT_INF, WEIGHT_SUP);

    double weightNorm;
    double weight = 0.0;

    // randomize and normalize all weights
    for (size_t iNeuron = 0; iNeuron < config.neurons; ++iNeuron) {
        weightNorm = 0.0;

        for (size_t iWeight = 0; iWeight < config.inVecDim; ++iWeight) {
            weight = dist(randGen);
            weights[iWeight + iNeuron * config.inVecDim] = weight;
            weightNorm += weight * weight;
        }

        weightNorm = sqrt(weightNorm);

        for (size_t iWeight = 0; iWeight < config.inVecDim; ++iWeight)
            weights[iWeight + iNeuron * config.inVecDim] /= weightNorm;
    }
}

std::vector<double> WTALayer::GetWeightedSums(const std::vector<double> &inVec) {
    std::vector<double> weightedSums(config.neurons, 0.0);

    for (size_t iNeuron = 0; iNeuron < config.neurons; ++iNeuron) {
        for (size_t iWeight = 0; iWeight < config.inVecDim; ++iWeight) {
            weightedSums[iNeuron] += weights[iWeight + iNeuron * config.inVecDim] *
inVec[iWeight];
            // TODO code refactoring
            weightedSums[iNeuron] -= config.trainPenalty * winHistory[iNeuron];
        }
    }

    return weightedSums;
}

size_t WTALayer::DetectWinner(const std::vector<double> &weightedSums) {
    return std::distance(weightedSums.begin(),
        std::max_element(weightedSums.begin(), weightedSums.end()));
```

```
}

size_t WTALayer::Test(const std::vector<double> &inVec) {
    if (inVec.size() != config.inVecDim)
        throw std::string("WTALayer::Test --> invalid input vector size!");

    return DetectWinner(GetWeightedSums(inVec));
}

void WTALayer::AdjustWeights(size_t iWinner, const std::vector<double> &inVec) {
    double prevWeight;
    double currWeight;
    double weightNorm = 0.0;

    for (size_t iWeight = 0; iWeight < config.inVecDim; ++iWeight) {
        prevWeight = weights[iWeight + iWinner * config.inVecDim];
        currWeight = prevWeight + config.trainCoeff * (inVec[iWeight] - prevWeight);
        weights[iWeight + iWinner * config.inVecDim] = currWeight;
        weightNorm += currWeight * currWeight;
    }

    weightNorm = sqrt(weightNorm);

    // normalize weights
    for (size_t iWeight = 0; iWeight < config.inVecDim; ++iWeight) {
        weights[iWeight + iWinner * config.inVecDim] /= weightNorm;
    }
}

void WTALayer::Train(const std::vector<std::vector<double>> &trainSet,
        std::ostream &output) {
    size_t iWinner;

    for (size_t iEpoch = 0; iEpoch < config.trainEpochs; ++iEpoch) {
        std::cout << "Train epoch: " << iEpoch << std::endl;

        for (size_t iVec = 0; iVec < trainSet.size(); ++iVec) {
            iWinner = Test(trainSet[iVec]);
            AdjustWinHistory(iWinner);
            AdjustWeights(iWinner, trainSet[iVec]);
```

```
            DumpWeights(output, 16, "", true);
        }
    }
}

void WTALayer::DumpWeights(std::ostream &output, int precision,
        const std::string &title, bool gnuplot) {
    if (!gnuplot)
        output << title << std::endl;

    if (gnuplot) {
        for (size_t iWeight = 0; iWeight < weights.size(); ++iWeight)
            output << 0.0 << "\t";

        output << std::endl;
    }

    for (size_t iWeight = 0; iWeight < weights.size(); ++iWeight)
        output << std::setprecision(precision) << weights[iWeight] << "\t";

    output << std::endl;

    if (gnuplot)
        output << std::endl << std::endl;
}
```

main.cpp

```
#include "WTALayer.h"
#include <algorithm>
#include <cmath>

void GenerateElipsis2DPointSet(
        std::vector<std::vector<double>> *points,
        double a, double b,
        double xOffset, double yOffset,
        double rotation, double step) {
    a = (a < 0) ? -a : a;
    b = (b < 0) ? -b : b;
    double norm;
```

```
    for (double x = -a; x <= a; x += step) {
        for (double y = -b; y <= b; y += step) {
            if (((x * x) / (a * a) + (y * y) / (b * b)) <= 1) {
                std::vector<double> v(2);
                v[0] = (x * cos(rotation) + y * sin(rotation)) + xOffset;
                v[1] = (-x * sin(rotation) + y * cos(rotation)) + yOffset;
                // normalize radius-vectors to specified points
                norm = sqrt(v[0] * v[0] + v[1] * v[1]);
                v[0] /= norm;
                v[1] /= norm;
                points->emplace_back(std::move(v));
            }
        }
    }
}

std::vector<std::vector<double>> Generate2DTrainSet() {
    std::vector<std::vector<double>> trainSet;

    GenerateElipsis2DPointSet(&trainSet, 0.3, 0.2, 0.4, 0.5, -M_PI_4, 0.042);
    GenerateElipsis2DPointSet(&trainSet, 0.2, 0.1, -0.1, 0.5, M_PI_4, 0.022);

    std::random_shuffle(trainSet.begin(), trainSet.end());
    return trainSet;
}

void DumpPoints(std::ostream &output, const std::vector<std::vector<double>> &points) {
    for (size_t iPoint = 0; iPoint < points.size(); ++iPoint) {
        for (size_t iCoord = 0; iCoord < points[iPoint].size(); ++iCoord)
            output << points[iPoint][iCoord] << "\t";

        output << std::endl;
    }
}

int main() {
    const std::string WEIGHTS_DUMP_FILE = "weights.data";
    const std::string TRAIN_SET_FILE    = "points.data";
```

```
    WTALayer nnet;
    WTALayer::Config nnetConfig;


    nnetConfig.neurons  = 4;
    nnetConfig.inVecDim = 2;
    nnetConfig.trainEpochs = 2;
    nnetConfig.trainCoeff  = 0.1;
    nnetConfig.trainPenalty = 0.01;
    const std::vector<double> INITIAL_WEIGHTS(nnetConfig.neurons * nnetConfig.inVecDim,
-1.0);


    nnet.Init(nnetConfig, false);


    if (!nnet.SetWeights(INITIAL_WEIGHTS))
        return 1;


    nnet.DumpWeights(std::cout, 6, "Initial weights:");


    std::ofstream trainSetDump(TRAIN_SET_FILE, std::ios::binary);
    std::ofstream weightsDump(WEIGHTS_DUMP_FILE, std::ios::binary);
    std::vector<std::vector<double>> trainSet(std::move(Generate2DTrainSet()));
    DumpPoints(trainSetDump, trainSet);
    nnet.Train(trainSet, weightsDump);
    nnet.DumpWeights(std::cout, 6, "Result weights:");
}
```

Лабораторная работа № 4

Задание (вариант 6)

Разработать, используя язык C/C++, программу, моделирующую поведение искусственной нейронной сети указанного преподавателем типа и обеспечивающую ее обучение для решения задач сжатия данных с потерями, классификации и аппроксимации.

Описание реализованной модели нейронной сети

Для решения задачи аппроксимации данных была реализована двухслойная нейронная сеть, в которой были задействованы персептроны с сигмоидальной функцией активации. Структурная схема сети представлена на рис. 3.1.

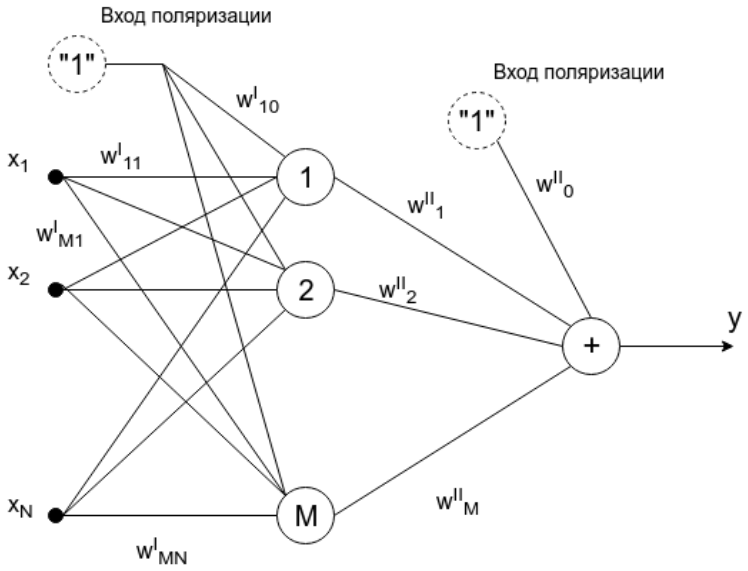

Рис. 3.1. Структурная схема нейронной сети.

В первом (скрытом) слое были использованы персептроны с сигмоидальной функцией активации:

$$f(u_i)=\frac{1}{1+e^{-\sigma\cdot u_i}} \quad ,$$

где u_i - взвешенная сумма *i*-го нейрона в текущем слое, σ - параметр, характеризующий «крутизну» сигмоиды (рис. 3.2). Производная от функции активации имеет вид:

$$f(u_i)=\sigma\cdot f(u_i)\cdot(1-f(u_i))$$

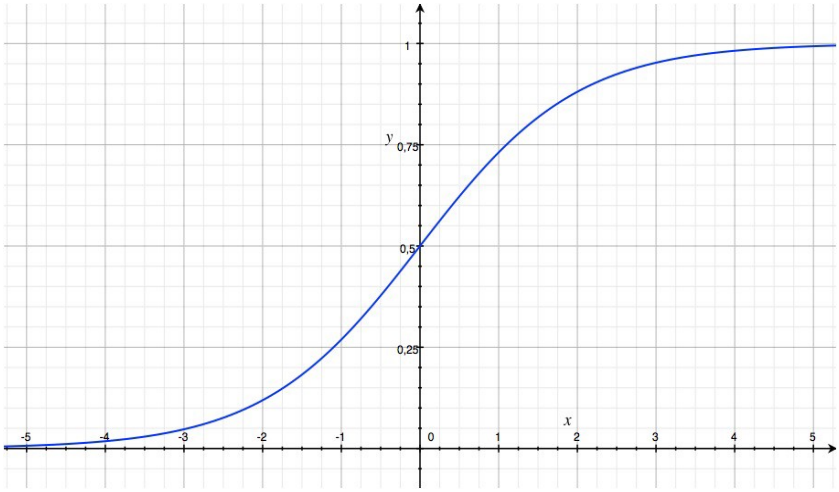

Рис. 3.2. Сигмоидальная функция активации.

В первом (скрытом) слое сети присутствуют персептроны с сигмоидальными функциями активации. Количество входов для каждого нейрона определяется размерностью аппроксимируемой функции. Каждый нейрон в скрытом слое имеет вход поляризации. Формально, этот вход является синапсом между данным нейроном скрытого слоя и, так называемым, «биасом» или нейроном смещения. Поляризатор отвечает за смещение выходного сигнала нейрона в том или ином направлении. Применительно к скрытому слою, поляризатор отвечает за смещение сигмоид (выходных сигналов нейронов) влево или вправо по оси абсцисс (для двумерной задачи аппроксимации). Важно заметить, что нейроны смещения могут либо присутствовать в каждом слое (по одному нейрону на слой), либо отсутствуют вовсе. Количество нейронов в скрытом слое влияет на качество аппроксимации.

Во втором (выходном) слое сети присутствует всего один выходной нейрон. Входами этого нейрона являются выходы нейронов с предыдущего слоя, а так же выход «биаса» в текущем слое. Нейрон представляет собой обычный сумматор, для него отсутствует функция активации. В данном случае, поляризатор отвечает за смещение аппроксимируемой функции вверх или вниз по оси ординат (для двумерной задачи аппроксимации).

Значения выходных сигналов для всех нейронов смещения равны 1, определяющим является вес синапса, корректируемый в процессе обучения вместе с остальными весами.

Обучение нейронной сети

Обучение сети проводилось в режиме «онлайн» с учителем. Для обучения сети был использовал метод обратного распространения ошибки. Суть метода заключается в том, что после расчета выходов нейронной сети в обратную сторону (т. е. от выходных слоев к скрытым) распространяется ошибка обучения, необходимая для корректировки весов. После корректировки весов выполняется новая итерация обучения, в которой также расчитываются выходы сети, а затем распространяется ошибка обучения от выходных слоев сети ко внутренним. Более подробное описание метода представлено ниже.

Целью обучения сети является минимизация функции ошибки, полученной по методу наименьших квадратов:

$$E(W)=\frac{1}{2}\cdot\sum_{i=1}^{L}(y_i^k-d_i^k)^2 \quad ,$$

где W — вектор, компонентами которого являются вектора весов нейронной сети; L — количество нейронов в выходном слое сети; k — номер входного вектора из обучающей выборки; y_i^k - значение выходного сигнала в i-ом нейроне выходного слоя сети; d_i^k - эталонное значение выходного сигнала в i-ом нейроне выходного слоя сети. Поскольку в выходном слое рассматриваемой сети присутствует всего лишь один нейрон, то целевая функция ошибки для заданного режима обучения имеет вид:

$$E(W)=\frac{1}{2}\cdot(y^k-d^k)^2=\frac{1}{2}\cdot([\sum_{i=0}^{M}w_i^{II}\cdot f(u_i^I)]-d^k)^2=\frac{1}{2}\cdot([\sum_{i=0}^{M}w_i^{II}\cdot f(\sum_{j=0}^{N}w_{ij}^I\cdot x_j^k)]-d^k)^2 \quad ,$$

где M — количество нейронов в скрытом слое сети; w_i^{II} - вес связи (синапса) между i-ым нейроном скрытого слоя и нейроном выходного слоя; u_i^I - взвешенная сумма входных сигналов i-го нейрона в скрытом слое; N — размерность входного вектора из обучающей выборки; w_{ij}^I - вес j-го входа i-го нейрона в скрытом слое; x_j^k - j-ая компонента k-го вектора из обучающей выборки. Нужно заметить, что $f(u_0^I)=1$, $x_0^k=1$.

Метод обратного распространения ошибки является адаптированной под обучение нейронных сетей версией метода градиента. Поэтому корректировка весов происходит по формуле:

$$W(t+1)=W(t)-\eta\cdot grad\,E(W(t)) \quad ,$$

где η - коэффициент обучения, обычно выбирается в диапазоне (0, 1); t — шаг корректировки. Частные производные, являющиеся компонентами вектора градиента, рассчитываются по следующим формулам:

$$\frac{\partial E(W)}{\partial w_i^{II}}=(y^k-d^k)\cdot\frac{\partial f(u^{II})}{\partial u^{II}}\cdot f(u_i^I)=(y^k-d^k)\cdot f(\sum_{j=0}^{N}w_{ij}^I\cdot x_j^k) \quad ,$$

$$\frac{\partial E(W)}{\partial w_{ij}^{I}} = (y^k - d^k) \cdot \frac{\partial f(u^{II})}{\partial u^{II}} \cdot w_i^{II} \cdot \frac{\partial f(u_i^I)}{\partial u_i^I} \cdot x_j^k = (y^k - d^k) \cdot w_i^{II} \cdot \frac{\partial f(u_i^I)}{\partial u_i^I} \cdot x_j^k \quad ,$$

где $\frac{\partial f(u^{II})}{\partial u^{II}} = 1$, $\frac{\partial f(u_i^I)}{\partial u_i^I}$ - производная функции активации по взвешенной сумме входных сигналов *i*-го нейрона скрытого слоя. Чтобы наглядно увидеть распространение ошибки обучения, введем следующие обозначения:

$$\delta^{II} = (y^k - d^k) \cdot \frac{\partial f(u^{II})}{\partial u^{II}}$$ - ошибка обучения в выходном слое;

$$\delta_i^I = (y^k - d^k) \cdot \frac{\partial f(u^{II})}{\partial u^{II}} \cdot \frac{\partial f(u_i^I)}{\partial u_i^I} = \delta^{II} \cdot \frac{\partial f(u_i^I)}{\partial u_i^I}$$ - ошибка обучения в скрытом слое.

Замечание: приведенные формулы справедливы для представленной выше структуры сети. В общем виде, формулы имеют несколько усложненный вид.

Расчет ошибок обучения происходит в следующем порядке: сначала рассчитываются ошибки обучения для всех нейронов в выходном слое, после чего ошибки с выходного слоя распространяются на скрытый слой. В скрытом слое для каждого нейрона осуществляется суммирование ошибок, пришедших с выходного слоя, после чего вычисляется ошибка обучения для нейрона в скрытом слое. Эта ошибка распространяется на следующий скрытый слой в обратном направлении по отношению к структуре сети. После вычисления всех ошибок выполняется процедура корректировки всех весов. Применительно к рассматриваемой сети, формулы корректировки выглядят следующим образом:

$$w_{ij}^I(t+1) = w_{ij}^I(t) - \eta \cdot \delta_i^I \cdot x_j \quad ,$$

$$w_i^{II}(t+1) = w_i^I(t) - \eta \cdot \delta^{II} \cdot g_i \quad ,$$

где $g_i = f(u_i^I), \forall i = 1,..,M ; g_0 = 1 ; x_0 = 1$.

Процесс обучение продолжается в течение заданного количества эпох обучения. Значения и описания параметров сети, задаваемых при обучении, представлены ниже.

Реализация

Обучающая выборка сгенерирована на основе функции от одной переменной $y = \sin(x), x = 0,..,4\pi$. Всего сгенерировано точек: 2512 (шаг по x = 0.005). Количество точек, используемых в обучающей выборке: 1256. Количество точек, используемых для тестирования сети после обучения: 1256. Обучающая выборка перемешана случайным образом.

Параметры сети, заданные для при обучении:

1. Количество нейронов в скрытом слое: 6.
2. Количество эпох обучения: 200.
3. Коэффициент обучения: 0.063.
4. Коэффициент крутизны сигмоиды: 1.
5. Диапазон значений для выбора начальных весов: [-0.5, 0.5]. В качестве распределения случайной величины было выбрано равномерное распределение.
6. Начальное значение весового коэффициента поляризатоа в выходном слое: 0.0.

Результат обучения сети и аппроксимации заданной функции представлен на рис. 3.3. На этом же рисунке представлены базисные функции — сигмоиды, за счет которых осуществлялась аппроксимация исходной функции.

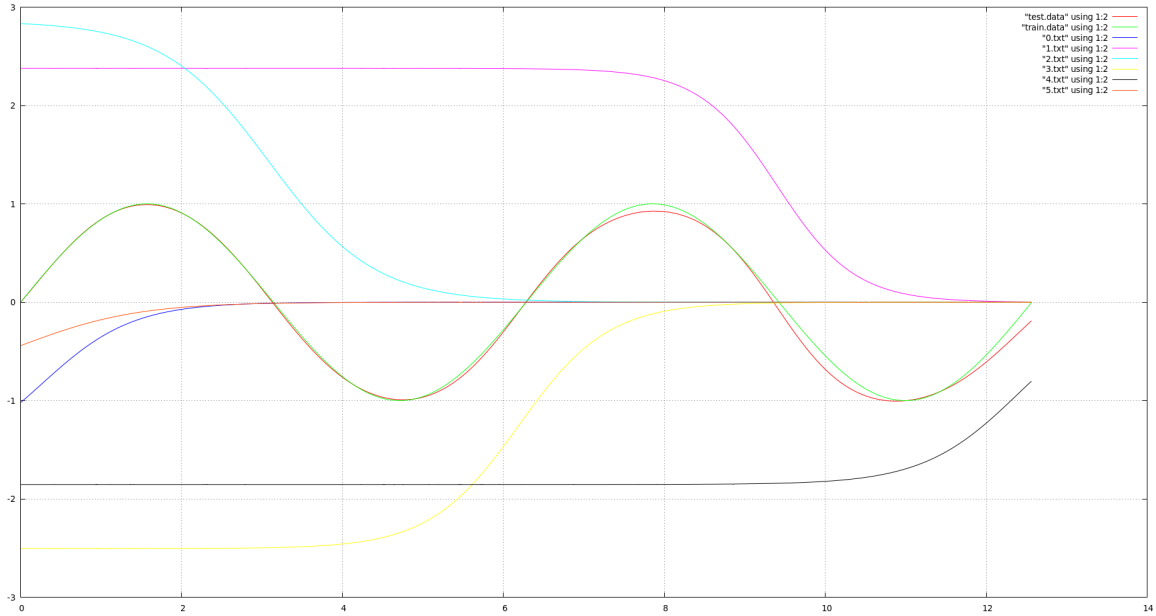

Рис. 3.3. Результат аппроксимации заданной функции базисными сигмоидами.

Максимальная абсолютная погрешность аппроксимации для тестовой выборки составила 0.137.

Стоит заметить, что результаты обучения сильно варьируются в зависимости от начальных параметров модели нейронной сети. Например, для приведенных выше параметров от запуска программы к запуску получались разные результаты обучения, с разной абсолютной погрешностью. В табл. 3.1 приведены сведения об изменениях абсолютной погрешности от запуска к запуску.

Табл. 3.1. Значения максимальных абсолютных погрешностей обучения.

№ запуска	1	2	3	4	5	6	7	8	9	10
Абс. погр.	0.181	0.261	0.826	0.335	0.396	0.826	0.249	0.184	0.137	0.426

Такой разлет значений можно обосновать выбором начального приближения для метода градиента. Метод градиента — метод локальной оптимизации, а значит, выбор начального приближения (начальные значения весов), количество шагов оптимизации (количество эпох обучения и размер обучающей выборки) и скорость градиентного спуска (коэффициент обучения) оказывают существенное влияние на сходимость к оптимальному решению.

Листинг программы

MLPApproxNet.h

```
#ifndef _LAB4_REFACTORED_MLP_APPROX_NET_H_
#define _LAB4_REFACTORED_MLP_APPROX_NET_H_

#include <vector>

typedef std::pair<std::vector<double>, double> TrainPair;
using size_t = std::size_t;

class MLPApproxNet {
public:
    struct Config {
        size_t inputLayerNeurons;
        size_t inputVecDim;
        size_t epochs;
        double sigma;
        double trainCoeff;
    };

    void   Init(const Config &conf);
    void   OnlineTrain(const std::vector<TrainPair> &trainSet);
    double Test(const std::vector<double> &inVec);
    void   DumpSigmoids(const std::vector<TrainPair> &vecSet);

private:
    Config config;
    size_t inputWeightDim;
    size_t outputWeightDim;
    std::vector<double> inputLayerWeights;
    std::vector<double> inputLayerOutputs;
    std::vector<double> outputLayerWeights;

    void   RandomizeWeights();
    double ActivationFunction(double weightedSum);
    double ActivationFunctionDerivative(double weightedSum);
    void   SetInputLayerOutputs(const std::vector<double> &inVec);
    double GetNeuralNetOutput();
    void   AdjustWeightsByBPROP(double trainError, const std::vector<double> &inVec);
```

```
};

#endif // _LAB4_REFACTORED_MLP_APPROX_NET_H_
```

MLPApproxNet.cpp

```
#include "MLPApproxNet.h"
#include <fstream>
#include <iomanip>
#include <string>
#include <cmath>
#include <random>
#include <algorithm>

void MLPApproxNet::Init(const Config &conf) {
    config = conf;
    inputWeightDim = config.inputVecDim + 1;
    outputWeightDim = config.inputLayerNeurons + 1;
    inputLayerWeights.resize(inputWeightDim * config.inputLayerNeurons);
    inputLayerOutputs.resize(config.inputLayerNeurons);
    outputLayerWeights.resize(outputWeightDim);

    RandomizeWeights();
}

void MLPApproxNet::RandomizeWeights() {
    const double WEIGHT_SUP = 0.5;
    const double WEIGHT_INF = -0.5;

    std::random_device randomizer;
    std::mt19937 randGen(randomizer());
    std::uniform_real_distribution<> dist(WEIGHT_INF, WEIGHT_SUP);

    size_t iNeuron = 0;
    size_t inputWeightInd;

    for (; iNeuron < config.inputLayerNeurons; ++iNeuron) {
        // input layer weights initialization
        for (size_t iWeight = 0; iWeight < inputWeightDim; ++iWeight) {
            inputWeightInd = iWeight + iNeuron * inputWeightDim;
            inputLayerWeights[inputWeightInd] = dist(randGen);
        }
```

```
        // output layer weights initialization
        outputLayerWeights[iNeuron] = dist(randGen);
    }

    //output layer polarizator initializing
    outputLayerWeights[iNeuron] = 0.0;
}

double MLPApproxNet::ActivationFunction(double weightedSum) {
    return 1.0 / (1.0 + exp(-config.sigma * weightedSum));
}

double MLPApproxNet::ActivationFunctionDerivative(double activationFunction) {
    return config.sigma * activationFunction * (1.0 - activationFunction);
}

void MLPApproxNet::SetInputLayerOutputs(const std::vector<double> &inVec) {
    size_t iWeight;
    size_t weightInd;

    // calculating weighted sums for input layer
    for (size_t iNeuron = 0; iNeuron < config.inputLayerNeurons; ++iNeuron) {
        for (iWeight = 0; iWeight < config.inputVecDim; ++iWeight) {
            weightInd = iWeight + iNeuron * inputWeightDim;
            inputLayerOutputs[iNeuron] += inputLayerWeights[weightInd] *
inVec[iWeight];
        }

        // calculate weighted sum for input layer polarizator
        weightInd = iWeight + iNeuron * inputWeightDim;
        inputLayerOutputs[iNeuron] += inputLayerWeights[weightInd];
    }

    // calculating outputs for input layers
    for (size_t iNeuron = 0; iNeuron < config.inputLayerNeurons; ++iNeuron)
        inputLayerOutputs[iNeuron] = ActivationFunction(inputLayerOutputs[iNeuron]);
}

double MLPApproxNet::GetNeuralNetOutput() {
    size_t iNeuron = 0;
```

```
    double output = 0.0;

    for (; iNeuron < config.inputLayerNeurons; ++iNeuron)
        output += inputLayerOutputs[iNeuron] * outputLayerWeights[iNeuron];

    // sum output layer polarizator contribution
    output += outputLayerWeights[iNeuron];
    return output;
}

double MLPApproxNet::Test(const std::vector<double> &inVec) {
    std::fill(inputLayerOutputs.begin(), inputLayerOutputs.end(), 0.0);
    SetInputLayerOutputs(inVec);
    return GetNeuralNetOutput();
}

void MLPApproxNet::AdjustWeightsByBPROP(double trainError,
        const std::vector<double> &inVec) {
    size_t iNeuron;
    size_t iWeight;
    size_t weightInd;
    double weightDerivative;
    double backPropError;

    for (iNeuron = 0; iNeuron < config.inputLayerNeurons; ++iNeuron) {
        backPropError = trainError * outputLayerWeights[iNeuron];
        backPropError *= ActivationFunctionDerivative(inputLayerOutputs[iNeuron]);

        for (iWeight = 0; iWeight < config.inputVecDim; ++iWeight) {
            weightInd = iWeight + iNeuron * inputWeightDim;
            weightDerivative = backPropError * inVec[iWeight];
            inputLayerWeights[weightInd] -= config.trainCoeff * weightDerivative;
        }

        // adjusting input layer polarizator weight
        weightInd = iWeight + iNeuron * inputWeightDim;
        inputLayerWeights[weightInd] -= config.trainCoeff * backPropError;

        // adjusting output layer weights
        weightDerivative = trainError * inputLayerOutputs[iNeuron];
```

```
        outputLayerWeights[iNeuron] -= config.trainCoeff * weightDerivative;
    }

    // adjusting output layer polarizator weight
    outputLayerWeights[iNeuron] -= config.trainCoeff * trainError;
}

void MLPApproxNet::DumpSigmoids(const std::vector<TrainPair> &vecSet) {
    double s;
    std::vector<std::vector<std::pair<double, double>>>
sigmoids(config.inputLayerNeurons);

    for (size_t iVec = 0; iVec < vecSet.size(); ++iVec) {
        const std::vector<double> &trainVec = vecSet[iVec].first;
        Test(trainVec);

        for (size_t iNeuron = 0; iNeuron < config.inputLayerNeurons; ++iNeuron) {
            s = inputLayerOutputs[iNeuron] * outputLayerWeights[iNeuron];
            sigmoids[iNeuron].emplace_back(std::make_pair(trainVec[0], s));
        }
    }

    std::string filename;

    for (size_t iNeuron = 0; iNeuron < config.inputLayerNeurons; ++iNeuron) {
        filename = std::to_string(iNeuron) + ".txt";
        std::ofstream out(filename, std::ios::binary);

        for (size_t iPoint = 0; iPoint < sigmoids[iNeuron].size(); ++iPoint) {
             out << std::setprecision(15) << sigmoids[iNeuron][iPoint].first
                << "\t" << sigmoids[iNeuron][iPoint].second << std::endl;
        }
    }
}

void MLPApproxNet::OnlineTrain(const std::vector<TrainPair> &trainSet) {
    double netOutput;
    double trainOutput;
    double trainError;

    for (size_t iEpoch = 0; iEpoch < config.epochs; ++iEpoch) {
```

```
        for (size_t iVec = 0; iVec < trainSet.size(); ++iVec) {
            const std::vector<double> &trainVec = trainSet[iVec].first;
            trainOutput = trainSet[iVec].second;
            netOutput = Test(trainVec);
            trainError = netOutput - trainOutput;
            AdjustWeightsByBPROP(trainError, trainVec);
        }
    }
}
```

main.cpp

```
#include "MLPApproxNet.h"
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
#include <cmath>
#include <algorithm>

std::vector<TrainPair> GenerateTrainSet() {
    const double STEP_X = 0.005;
    const double INF_X = 0.0;
    const double SUP_X = 12.56;

    std::vector<TrainPair> trainSet;

    for (double x = INF_X; x <= SUP_X; x += STEP_X) {
        double y = x;
        double z = sin(y);
        std::cout << y << " " << z << " " << std::endl;
        trainSet.emplace_back(std::make_pair(std::vector<double>{y}, z));
    }

    std::random_shuffle(trainSet.begin(), trainSet.end());

    return trainSet;
}

void DumpPoints(const std::vector<TrainPair> &points, std::ostream &output) {
    for (size_t iPoint = 0; iPoint < points.size(); ++iPoint) {
        output << std::setprecision(15);
```

```
        output << points[iPoint].first[0] /*<< "\t" <<  points[iPoint].first[1]*/
            << "\t" << points[iPoint].second << std::endl;
    }
}

bool comparator(const TrainPair &t1, const TrainPair &t2) {
    return t1.first[0] > t2.first[0];
}

int main() {
    const std::string TRAIN_SET_FILE = "train.data";
    const std::string TEST_SET_FILE  = "test.data";

    MLPApproxNet::Config nnetConfig;
    nnetConfig.inputLayerNeurons = 6;
    nnetConfig.inputVecDim = 1;
    nnetConfig.epochs = 200;
    nnetConfig.sigma = 1.0;
    nnetConfig.trainCoeff = 0.063; // should be [0; 1]

    MLPApproxNet nnet;
    std::vector<TrainPair> pointSet(std::move(GenerateTrainSet()));
    const size_t half = pointSet.size() / 2;
    std::vector<TrainPair> trainSet(pointSet.begin(), pointSet.begin() + half);
    std::vector<TrainPair> testSet(pointSet.begin() + half, pointSet.end());

    std::ofstream trainSetDump(TRAIN_SET_FILE, std::ios::binary);
    std::ofstream testSetDump(TEST_SET_FILE, std::ios::binary);

    nnet.Init(nnetConfig);
    nnet.OnlineTrain(trainSet);

    std::sort(trainSet.begin(), trainSet.end());
    DumpPoints(trainSet, trainSetDump);

    double testVal;
    double maxTestError = 0.0;
    double testError;

    for (size_t iPoint = 0; iPoint < testSet.size(); ++iPoint) {
```

```
        testVal = nnet.Test(testSet[iPoint].first);
        testError = fabs(testVal - testSet[iPoint].second);

        if (testError > maxTestError)
            maxTestError = testError;

        testSet[iPoint].second = testVal; // for vizualization
    }

    std::cout << "Max approximation error: " << maxTestError << std::endl;
    std::sort(testSet.begin(), testSet.end(), comparator);
    DumpPoints(testSet, testSetDump);
    nnet.DumpSigmoids(testSet);

    return 0;
}
```