**БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

Международный Институт Дистанционного Образования

Кафедра “Информационные системы и технологии”

Контрольная работа по дисциплине «Методы и алгоритмы компьютерной графики»

Выполнил:   
студент 3 курса, гр. 41703120  
Реут Владислав Леонидович

Проверил: Анцыпов Н.А.

Минск 2023

**ОГЛАВЛЕНИЕ**

ВВЕДЕНИЕ 3

ЗАДАНИЕ 4

**Описание программы 4**

Испытание программы 14

[Выводы](file:///D:\BNTU\BNTU\OOP\Course_Project_OOP.docx#_Toc125733112) 16

ЛИТЕРАТУРА 17

**Листинг программы 18**

**ВВЕДЕНИЕ**

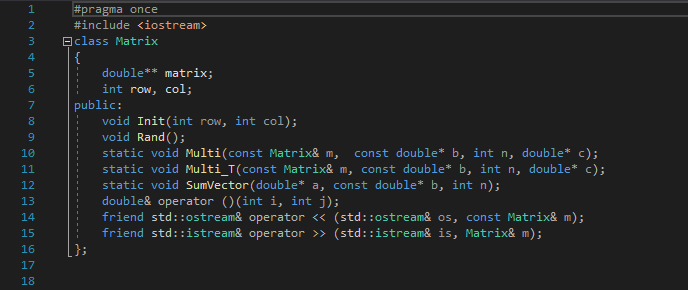
Нейронные сети (или искусственные нейронные сети) - это математическая модель, а также ее программное или аппаратное воплощение, построенная по принципу организации и функционирования биологических нейронных сетей - сетей нервных клеток живого организма. Они представляют собой систему соединенных и взаимодействующих между собой простых процессоров (искусственных нейронов), каждый из которых имеет дело только с сигналами, которые он периодически получает, и сигналами, которые он отправляет другим процессорам. Нейронные сети используются в искусственном интеллекте для обработки данных таким же способом, как и человеческий мозг. Они могут использоваться для решения различных задач, таких как прогнозирование, распознавание образов, управление и другие. Обучение нейронных сетей происходит путем первичной обработки больших наборов размеченных или неразмеченных данных.

**Задание**

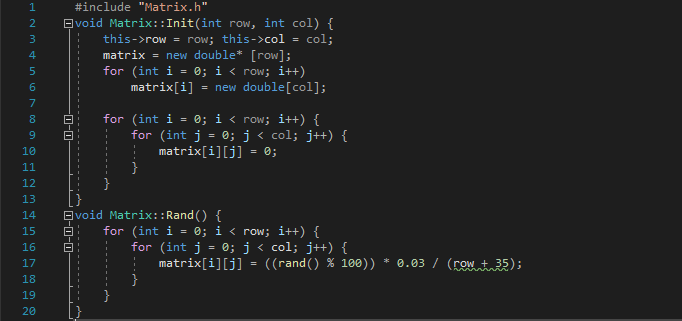
Написать программу способную распознать рукописный ввод на примере цифр.

**Описание программы**

Для решения данной задачи была написана примитивная нейронная сеть с применением языка программирования С++. Обучена сеть при помощи доступных в сети данных (MNIST). Так же для взаимодействия с пользователем, с использованием фреймворка Qt, был разработан интерфейс для ввода символов.

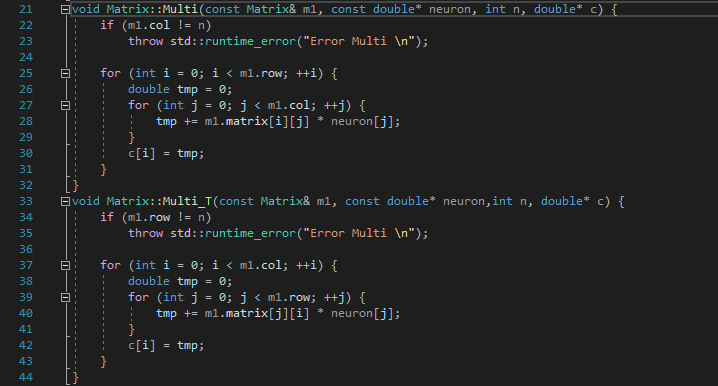


*Рис.1* ***Matrix.h.***

****

*Рис.2* ***Matrix.cpp*** *функции* ***Init*** *и* ***Rand****.*

На Рис.2 находятся функции инициализации и заполнения матриц.



*Рис.3* ***Matrix.cpp*** *функции* ***Multi*** *и* ***Multi\_t****.*

Рис.3 содержит функции перемножения матриц (обычной и транспонированной).

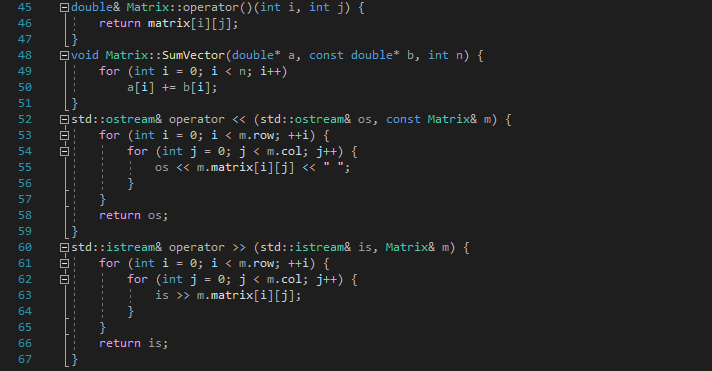
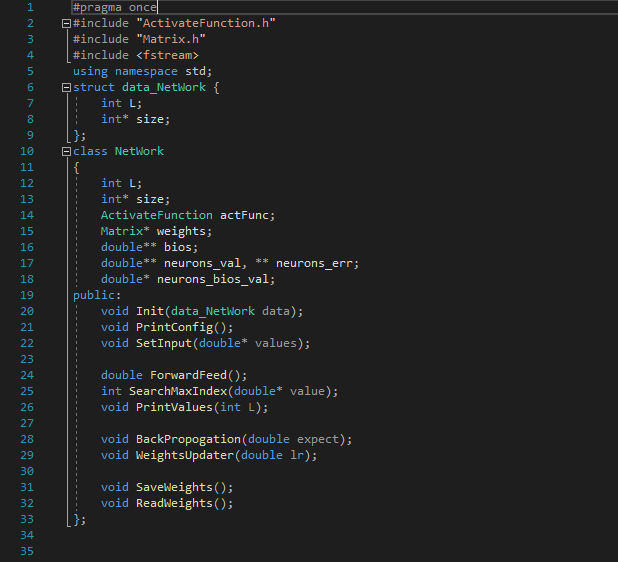
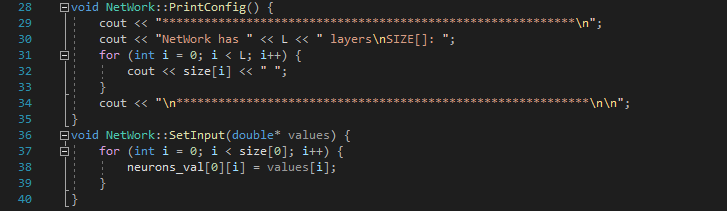


Рис.4 ***Matrix.cpp*** *функция* ***SumVector*** *и перегруженные операторы.*

На Рис.4 можно увидеть функцию **SumVector**, которая предназначена для сложения векторов.

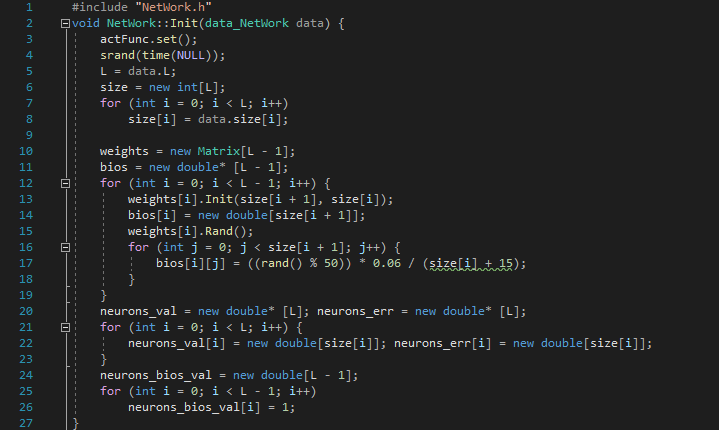


*Рис.5* ***NetWork.h****.*

**

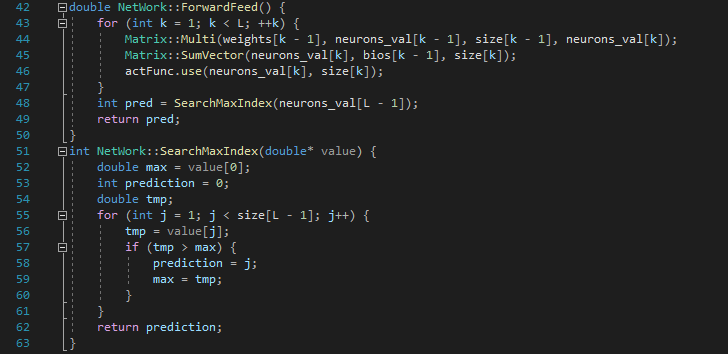
*Рис.6* ***NetWork.cpp****. функции* ***PrintConfig*** *и* ***SetInput****.*

На Рис.6 изображены функция вывода в консоль конфигурации создаваемой сети и функция задания входных значений.

**

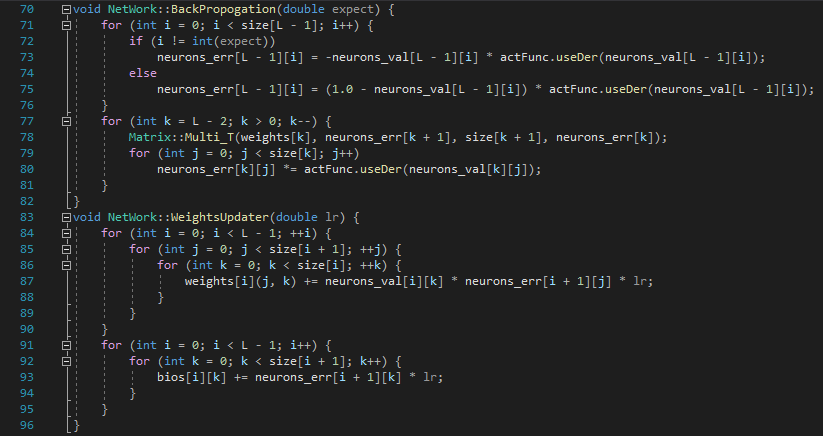
*Рис.7* ***NetWork.cpp****. функция* ***Init****.*

Рис.7 демонстрирует функцию **Init**, это функция инициализации сети. В ней происходит выбор функции активации (об этом ниже) и заполнение всех необходимых матриц.

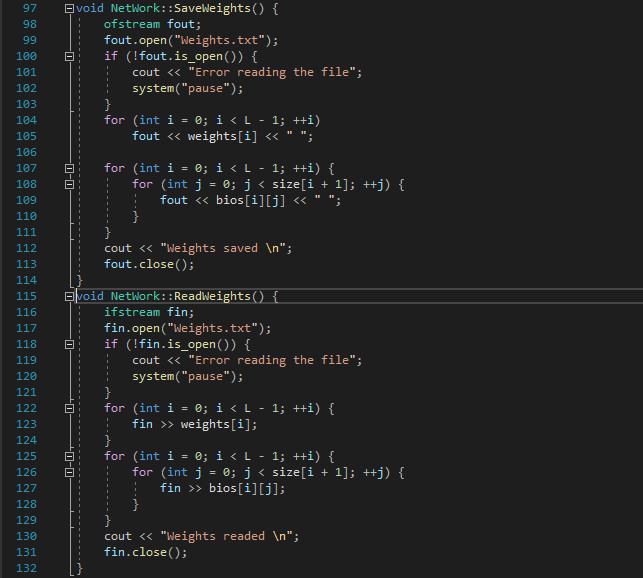


*Рис.8* ***NetWork.cpp****. функции* ***ForwardFeed*** *и* ***SearchMaxIndex****.*

Рис.8 демонстрирует функцию **ForwardFeed** – функция прямого распространения сети (двигаемся от входных нейронов через скрытые к выходным). Там же находится и функция **SearchMaxIndex** которая возвращает индекс элемента с максимальным значением.



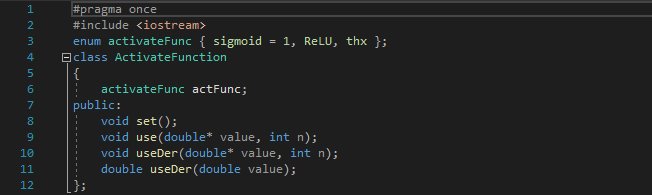
*Рис.9* ***NetWork.cpp****. функции* ***BackPropogation*** *и* ***WeightsUpdater****.*



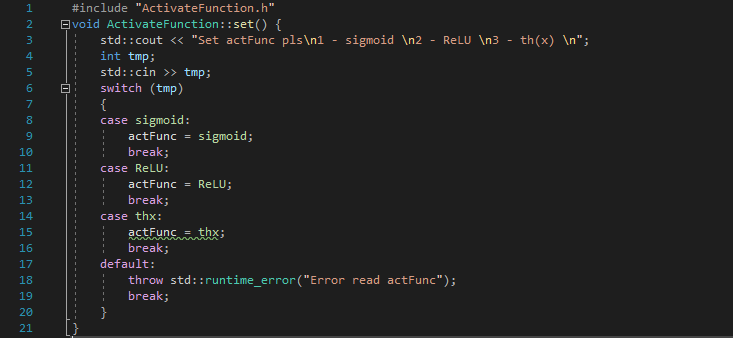
*Рис.10* ***NetWork.cpp****. функции* ***SaveWeights*** *и* ***ReadWeights****.*

На Рис.9 представлены функции **BackPropogation** и **WeightsUpdater**. Первая получает на вход правильное значение и высчитывает ошибку необходимую для обучения сети. Вторая отвечает за обновление весов.

На Рис.10 изображены функции сохранения весов в файл, а так же чтение их же из файла.

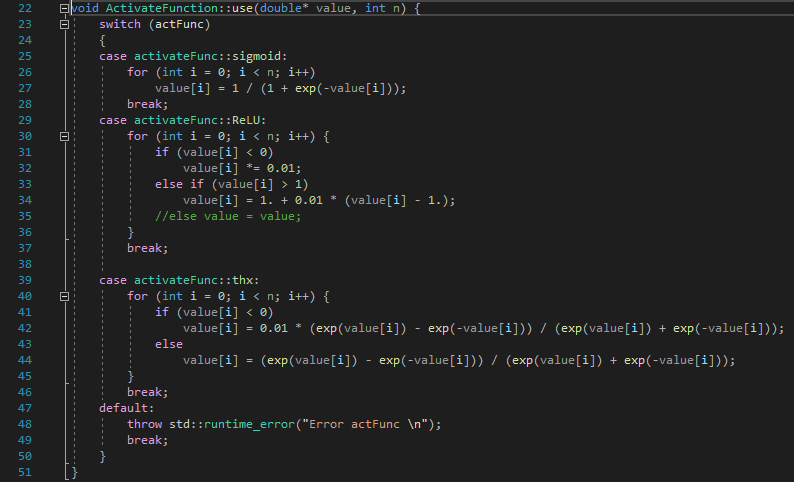


*Рис.11* ***ActivateFunction.h****.*



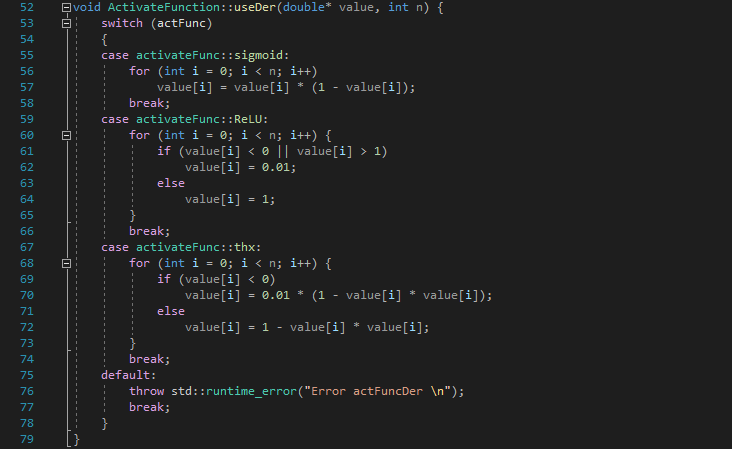
*Рис.12* ***ActivateFunction.cpp*** *функция* ***Set****.*

Рис.12 содержит функцию выбора функции активации. В этой работе их всего три.

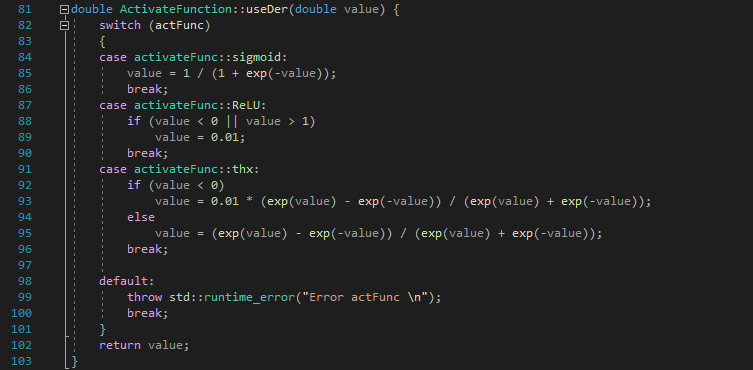


*Рис.13* ***ActivateFunction.cpp*** *функция* ***Use****.*

Функция (Рис.13) содержит в себе реализации самих функций активации.

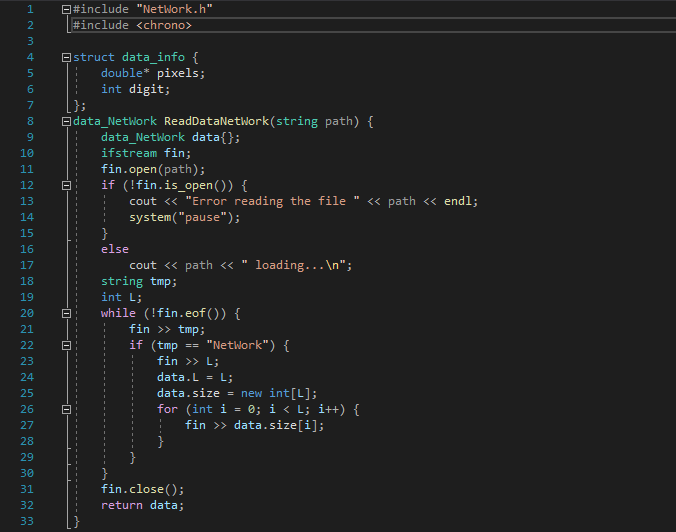


*Рис.14* ***ActivateFunction.cpp*** *функция* ***useDer****.*



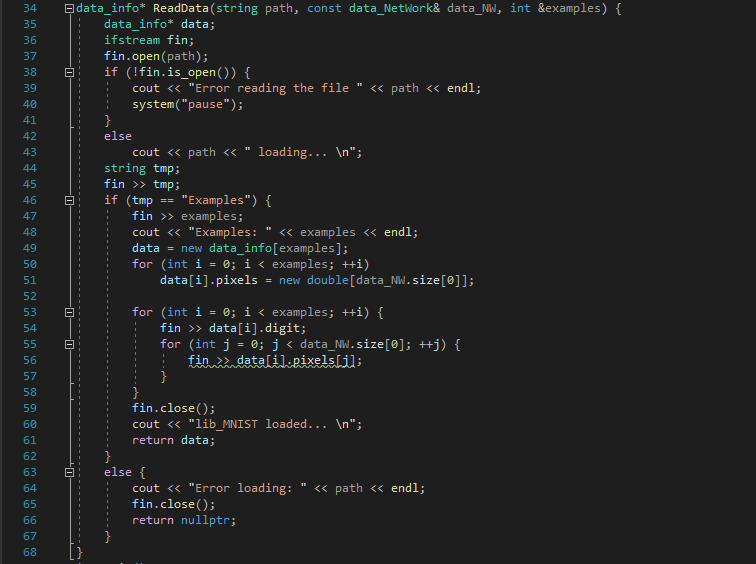
*Рис.15* ***ActivateFunction.cpp*** *перегрузка**функции* ***useDer***

Рис.14 содержит функцию **useDer** – производные функций активаций. А на Рис.15 изображена ее перегрузка.



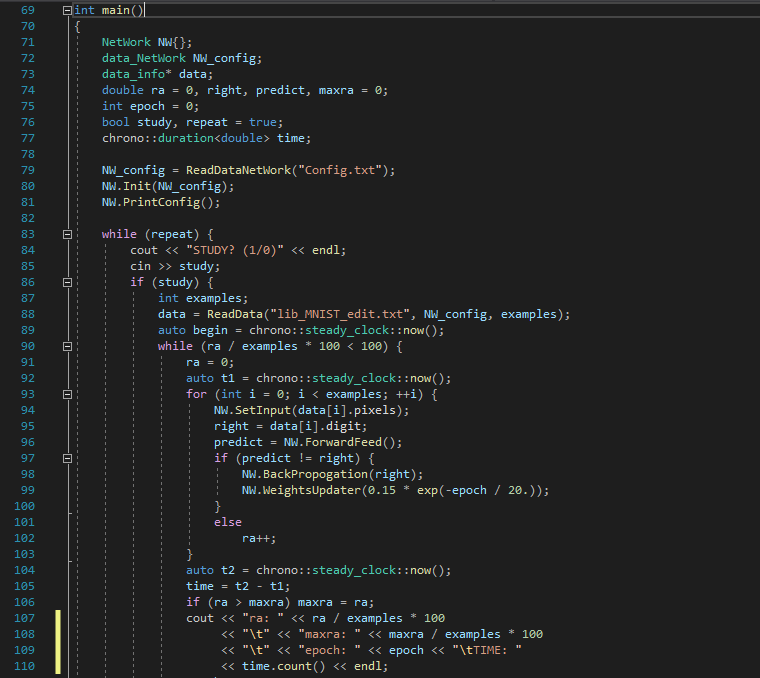
*Рис.16* ***Source.cpp*** *структура* ***data\_info*** *и функция* ***ReadDataNetWork****.*

**data\_info** (Рис.16) содержит поля для цифр и пикселей. **ReadDataNetWork** считывает с файла данные о количестве слоев и нейронов в ней. В данном случае это 3 слоя 784 – 264 – 10 нейронов.

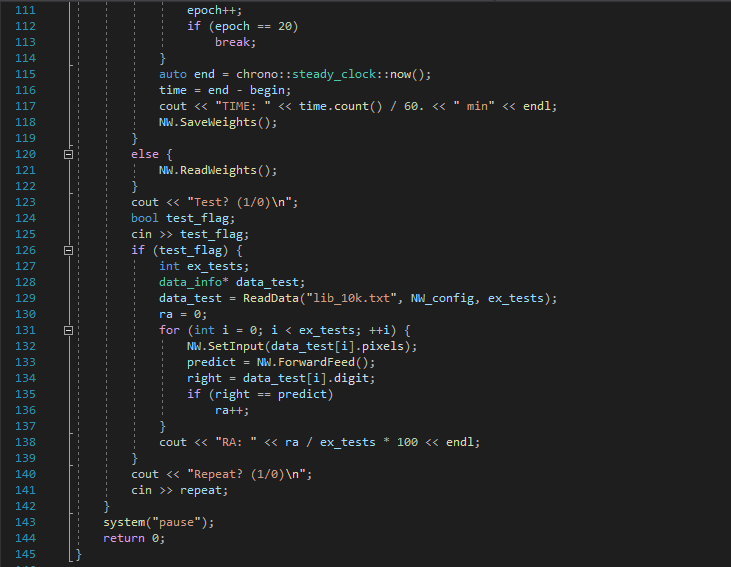
****

*Рис.17* ***Source.cpp*** *функция* ***ReadData.***

**ReadData** (Рис.17) – считываем данные с датасета.



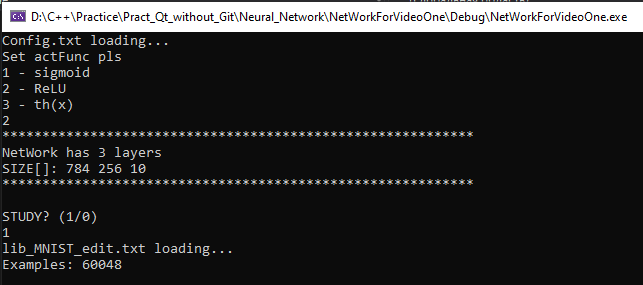
*Рис.18* ***Source.cpp*** *функция* ***main*** *(начало).*



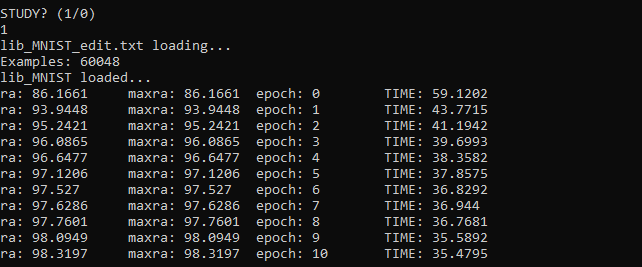
*Рис.19* ***Source.cpp*** *функция* ***main*** *(окончание).*

На Рис.18-19 находится функция **main**, которая инициализирует все необходимые объекты а так же запускает основной цикл обучения.

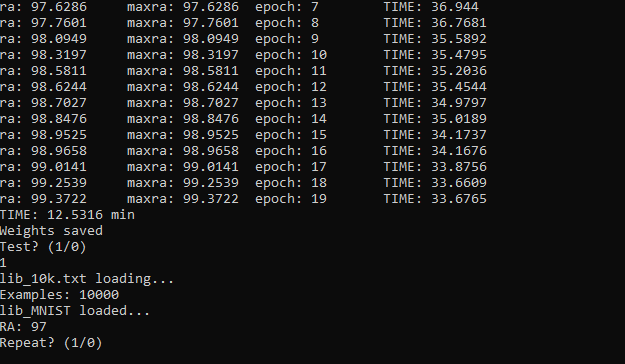
**Тестирование программы**

****

*Рис.20 Процесс обучения сети (начало)*

****

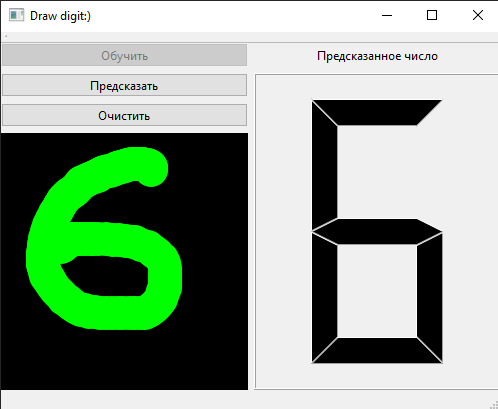
*Рис.21 Процесс обучения сети (продолжение)*

****

*Рис.21 Процесс обучения сети (окончание)*

ra – правильный ответ, maxra – максимально правильный ответ. Время обучения заняло 12 минут. Сеть обучилась распознавать значения в 97% случаев.

Теперь протестируем рукописный ввод:



**Вывод**

В результате разработки приложения были изучены принципы работы нейронных сетей, а так же оценена сложность и перспективность данной области.

**Литература**

1. Создаем нейронную сеть. Тарик Рашид.

**Листинг программы**

**Matrix.h:**

#pragma once

#include <iostream>

class Matrix

{

double\*\* matrix;

int row, col;

public:

void Init(int row, int col);

void Rand();

static void Multi(const Matrix& m, const double\* b, int n, double\* c);

static void Multi\_T(const Matrix& m, const double\* b, int n, double\* c);

static void SumVector(double\* a, const double\* b, int n);

double& operator ()(int i, int j);

friend std::ostream& operator << (std::ostream& os, const Matrix& m);

friend std::istream& operator >> (std::istream& is, Matrix& m);

};

**Matrix.cpp:**

#include "Matrix.h"

void Matrix::Init(int row, int col) {

this->row = row; this->col = col;

matrix = new double\* [row];

for (int i = 0; i < row; i++)

matrix[i] = new double[col];

for (int i = 0; i < row; i++) {

for (int j = 0; j < col; j++) {

matrix[i][j] = 0;

}

}

}

void Matrix::Rand() {

for (int i = 0; i < row; i++) {

for (int j = 0; j < col; j++) {

matrix[i][j] = ((rand() % 100)) \* 0.03 / (row + 35);

}

}

}

void Matrix::Multi(const Matrix& m1, const double\* neuron, int n, double\* c) {

if (m1.col != n)

throw std::runtime\_error("Error Multi \n");

for (int i = 0; i < m1.row; ++i) {

double tmp = 0;

for (int j = 0; j < m1.col; ++j) {

tmp += m1.matrix[i][j] \* neuron[j];

}

c[i] = tmp;

}

}

void Matrix::Multi\_T(const Matrix& m1, const double\* neuron,int n, double\* c) {

if (m1.row != n)

throw std::runtime\_error("Error Multi \n");

for (int i = 0; i < m1.col; ++i) {

double tmp = 0;

for (int j = 0; j < m1.row; ++j) {

tmp += m1.matrix[j][i] \* neuron[j];

}

c[i] = tmp;

}

}

double& Matrix::operator()(int i, int j) {

return matrix[i][j];

}

void Matrix::SumVector(double\* a, const double\* b, int n) {

for (int i = 0; i < n; i++)

a[i] += b[i];

}

std::ostream& operator << (std::ostream& os, const Matrix& m) {

for (int i = 0; i < m.row; ++i) {

for (int j = 0; j < m.col; j++) {

os << m.matrix[i][j] << " ";

}

}

return os;

}

std::istream& operator >> (std::istream& is, Matrix& m) {

for (int i = 0; i < m.row; ++i) {

for (int j = 0; j < m.col; j++) {

is >> m.matrix[i][j];

}

}

return is;

}

**ActivateFunction.h:**

#pragma once

#include <iostream>

enum activateFunc { sigmoid = 1, ReLU, thx };

class ActivateFunction

{

activateFunc actFunc;

public:

void set();

void use(double\* value, int n);

void useDer(double\* value, int n);

double useDer(double value);

};

**ActivateFunction.cpp:**

#include "ActivateFunction.h"

void ActivateFunction::set() {

std::cout << "Set actFunc pls\n1 - sigmoid \n2 - ReLU \n3 - th(x) \n";

int tmp;

std::cin >> tmp;

switch (tmp)

{

case sigmoid:

actFunc = sigmoid;

break;

case ReLU:

actFunc = ReLU;

break;

case thx:

actFunc = thx;

break;

default:

throw std::runtime\_error("Error read actFunc");

break;

}

}

void ActivateFunction::use(double\* value, int n) {

switch (actFunc)

{

case activateFunc::sigmoid:

for (int i = 0; i < n; i++)

value[i] = 1 / (1 + exp(-value[i]));

break;

case activateFunc::ReLU:

for (int i = 0; i < n; i++) {

if (value[i] < 0)

value[i] \*= 0.01;

else if (value[i] > 1)

value[i] = 1. + 0.01 \* (value[i] - 1.);

//else value = value;

}

break;

case activateFunc::thx:

for (int i = 0; i < n; i++) {

if (value[i] < 0)

value[i] = 0.01 \* (exp(value[i]) - exp(-value[i])) / (exp(value[i]) + exp(-value[i]));

else

value[i] = (exp(value[i]) - exp(-value[i])) / (exp(value[i]) + exp(-value[i]));

}

break;

default:

throw std::runtime\_error("Error actFunc \n");

break;

}

}

void ActivateFunction::useDer(double\* value, int n) {

switch (actFunc)

{

case activateFunc::sigmoid:

for (int i = 0; i < n; i++)

value[i] = value[i] \* (1 - value[i]);

break;

case activateFunc::ReLU:

for (int i = 0; i < n; i++) {

if (value[i] < 0 || value[i] > 1)

value[i] = 0.01;

else

value[i] = 1;

}

break;

case activateFunc::thx:

for (int i = 0; i < n; i++) {

if (value[i] < 0)

value[i] = 0.01 \* (1 - value[i] \* value[i]);

else

value[i] = 1 - value[i] \* value[i];

}

break;

default:

throw std::runtime\_error("Error actFuncDer \n");

break;

}

}

double ActivateFunction::useDer(double value) {

switch (actFunc)

{

case activateFunc::sigmoid:

value = 1 / (1 + exp(-value));

break;

case activateFunc::ReLU:

if (value < 0 || value > 1)

value = 0.01;

break;

case activateFunc::thx:

if (value < 0)

value = 0.01 \* (exp(value) - exp(-value)) / (exp(value) + exp(-value));

else

value = (exp(value) - exp(-value)) / (exp(value) + exp(-value));

break;

default:

throw std::runtime\_error("Error actFunc \n");

break;

}

return value;

}

**NetWork.h:**

#pragma once

#include "ActivateFunction.h"

#include "Matrix.h"

#include <fstream>

using namespace std;

struct data\_NetWork {

int L;

int\* size;

};

class NetWork

{

int L;

int\* size;

ActivateFunction actFunc;

Matrix\* weights;

double\*\* bios;

double\*\* neurons\_val, \*\* neurons\_err;

double\* neurons\_bios\_val;

public:

void Init(data\_NetWork data);

void PrintConfig();

void SetInput(double\* values);

double ForwardFeed();

int SearchMaxIndex(double\* value);

void PrintValues(int L);

void BackPropogation(double expect);

void WeightsUpdater(double lr);

void SaveWeights();

void ReadWeights();

};

**NetWork.cpp:**

#include "NetWork.h"

void NetWork::Init(data\_NetWork data) {

actFunc.set();

srand(time(NULL));

L = data.L;

size = new int[L];

for (int i = 0; i < L; i++)

size[i] = data.size[i];

weights = new Matrix[L - 1];

bios = new double\* [L - 1];

for (int i = 0; i < L - 1; i++) {

weights[i].Init(size[i + 1], size[i]);

bios[i] = new double[size[i + 1]];

weights[i].Rand();

for (int j = 0; j < size[i + 1]; j++) {

bios[i][j] = ((rand() % 50)) \* 0.06 / (size[i] + 15);

}

}

neurons\_val = new double\* [L]; neurons\_err = new double\* [L];

for (int i = 0; i < L; i++) {

neurons\_val[i] = new double[size[i]]; neurons\_err[i] = new double[size[i]];

}

neurons\_bios\_val = new double[L - 1];

for (int i = 0; i < L - 1; i++)

neurons\_bios\_val[i] = 1;

}

void NetWork::PrintConfig() {

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout << "NetWork has " << L << " layers\nSIZE[]: ";

for (int i = 0; i < L; i++) {

cout << size[i] << " ";

}

cout << "\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

}

void NetWork::SetInput(double\* values) {

for (int i = 0; i < size[0]; i++) {

neurons\_val[0][i] = values[i];

}

}

double NetWork::ForwardFeed() {

for (int k = 1; k < L; ++k) {

Matrix::Multi(weights[k - 1], neurons\_val[k - 1], size[k - 1], neurons\_val[k]);

Matrix::SumVector(neurons\_val[k], bios[k - 1], size[k]);

actFunc.use(neurons\_val[k], size[k]);

}

int pred = SearchMaxIndex(neurons\_val[L - 1]);

return pred;

}

int NetWork::SearchMaxIndex(double\* value) {

double max = value[0];

int prediction = 0;

double tmp;

for (int j = 1; j < size[L - 1]; j++) {

tmp = value[j];

if (tmp > max) {

prediction = j;

max = tmp;

}

}

return prediction;

}

void NetWork::PrintValues(int L) {

for (int j = 0; j < size[L]; j++) {

cout << j << " " << neurons\_val[L][j] << endl;

}

}

void NetWork::BackPropogation(double expect) {

for (int i = 0; i < size[L - 1]; i++) {

if (i != int(expect))

neurons\_err[L - 1][i] = -neurons\_val[L - 1][i] \* actFunc.useDer(neurons\_val[L - 1][i]);

else

neurons\_err[L - 1][i] = (1.0 - neurons\_val[L - 1][i]) \* actFunc.useDer(neurons\_val[L - 1][i]);

}

for (int k = L - 2; k > 0; k--) {

Matrix::Multi\_T(weights[k], neurons\_err[k + 1], size[k + 1], neurons\_err[k]);

for (int j = 0; j < size[k]; j++)

neurons\_err[k][j] \*= actFunc.useDer(neurons\_val[k][j]);

}

}

void NetWork::WeightsUpdater(double lr) {

for (int i = 0; i < L - 1; ++i) {

for (int j = 0; j < size[i + 1]; ++j) {

for (int k = 0; k < size[i]; ++k) {

weights[i](j, k) += neurons\_val[i][k] \* neurons\_err[i + 1][j] \* lr;

}

}

}

for (int i = 0; i < L - 1; i++) {

for (int k = 0; k < size[i + 1]; k++) {

bios[i][k] += neurons\_err[i + 1][k] \* lr;

}

}

}

void NetWork::SaveWeights() {

ofstream fout;

fout.open("Weights.txt");

if (!fout.is\_open()) {

cout << "Error reading the file";

system("pause");

}

for (int i = 0; i < L - 1; ++i)

fout << weights[i] << " ";

for (int i = 0; i < L - 1; ++i) {

for (int j = 0; j < size[i + 1]; ++j) {

fout << bios[i][j] << " ";

}

}

cout << "Weights saved \n";

fout.close();

}

void NetWork::ReadWeights() {

ifstream fin;

fin.open("Weights.txt");

if (!fin.is\_open()) {

cout << "Error reading the file";

system("pause");

}

for (int i = 0; i < L - 1; ++i) {

fin >> weights[i];

}

for (int i = 0; i < L - 1; ++i) {

for (int j = 0; j < size[i + 1]; ++j) {

fin >> bios[i][j];

}

}

cout << "Weights readed \n";

fin.close();

}

**Source.cpp:**

#include "NetWork.h"

#include <chrono>

struct data\_info {

double\* pixels;

int digit;

};

data\_NetWork ReadDataNetWork(string path) {

data\_NetWork data{};

ifstream fin;

fin.open(path);

if (!fin.is\_open()) {

cout << "Error reading the file " << path << endl;

system("pause");

}

else

cout << path << " loading...\n";

string tmp;

int L;

while (!fin.eof()) {

fin >> tmp;

if (tmp == "NetWork") {

fin >> L;

data.L = L;

data.size = new int[L];

for (int i = 0; i < L; i++) {

fin >> data.size[i];

}

}

}

fin.close();

return data;

}

data\_info\* ReadData(string path, const data\_NetWork& data\_NW, int &examples) {

data\_info\* data;

ifstream fin;

fin.open(path);

if (!fin.is\_open()) {

cout << "Error reading the file " << path << endl;

system("pause");

}

else

cout << path << " loading... \n";

string tmp;

fin >> tmp;

if (tmp == "Examples") {

fin >> examples;

cout << "Examples: " << examples << endl;

data = new data\_info[examples];

for (int i = 0; i < examples; ++i)

data[i].pixels = new double[data\_NW.size[0]];

for (int i = 0; i < examples; ++i) {

fin >> data[i].digit;

for (int j = 0; j < data\_NW.size[0]; ++j) {

fin >> data[i].pixels[j];

}

}

fin.close();

cout << "lib\_MNIST loaded... \n";

return data;

}

else {

cout << "Error loading: " << path << endl;

fin.close();

return nullptr;

}

}

int main()

{

NetWork NW{};

data\_NetWork NW\_config;

data\_info\* data;

double ra = 0, right, predict, maxra = 0;

int epoch = 0;

bool study, repeat = true;

chrono::duration<double> time;

NW\_config = ReadDataNetWork("Config.txt");

NW.Init(NW\_config);

NW.PrintConfig();

while (repeat) {

cout << "STUDY? (1/0)" << endl;

cin >> study;

if (study) {

int examples;

data = ReadData("lib\_MNIST\_edit.txt", NW\_config, examples);

auto begin = chrono::steady\_clock::now();

while (ra / examples \* 100 < 100) {

ra = 0;

auto t1 = chrono::steady\_clock::now();

for (int i = 0; i < examples; ++i) {

NW.SetInput(data[i].pixels);

right = data[i].digit;

predict = NW.ForwardFeed();

if (predict != right) {

NW.BackPropogation(right);

NW.WeightsUpdater(0.15 \* exp(-epoch / 20.));

}

else

ra++;

}

auto t2 = chrono::steady\_clock::now();

time = t2 - t1;

if (ra > maxra) maxra = ra;

cout << "ra: " << ra / examples \* 100

<< "\t" << "maxra: " << maxra / examples \* 100

<< "\t" << "epoch: " << epoch << "\tTIME: "

<< time.count() << endl;

epoch++;

if (epoch == 20)

break;

}

auto end = chrono::steady\_clock::now();

time = end - begin;

cout << "TIME: " << time.count() / 60. << " min" << endl;

NW.SaveWeights();

}

else {

NW.ReadWeights();

}

cout << "Test? (1/0)\n";

bool test\_flag;

cin >> test\_flag;

if (test\_flag) {

int ex\_tests;

data\_info\* data\_test;

data\_test = ReadData("lib\_10k.txt", NW\_config, ex\_tests);

ra = 0;

for (int i = 0; i < ex\_tests; ++i) {

NW.SetInput(data\_test[i].pixels);

predict = NW.ForwardFeed();

right = data\_test[i].digit;

if (right == predict)

ra++;

}

cout << "RA: " << ra / ex\_tests \* 100 << endl;

}

cout << "Repeat? (1/0)\n";

cin >> repeat;

}

system("pause");

return 0;

}