|  |  |
| --- | --- |
| **Scenariusz 3** | |
| **Temat ćwiczenia:** | Budowa i działanie sieci wielowarstwowej |
| **Wykonał:** | Mateusz Mazur, gr 2 |

1. **Cel ćwiczenia**

Celem ćwiczenia jest poznanie budowy i działanie wielowarstwowych sieci neuronowych poprzez uczenie z użyciem algorytmu wstecznej propagacji błędu rozpoznawania konkretnych liter alfabetu.

1. **Budowa sieci i wykorzystany algorytm uczenia**

Celem budowanej sieci jest rozpoznawanie liter w tym celu stworzony został zestaw liter (20 dużych), które są reprezentowane w postaci dwuwymiarowej tablicy 4x5 pikseli dla jednej litery. Użyte litery są przedstawione w poniższej tabeli.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0110  1001  1111  1001  1001 | D:\Pobrane\Alfabet (1)\Alfabet\A.png | 1110  1001  1110  1001  1110 |
| D:\Pobrane\Alfabet (1)\Alfabet\G.png | 0110  1001  1000  1001  0110 | D:\Pobrane\Alfabet (1)\Alfabet\D.png | 1110  1001  1001  1001  1110 |
| D:\Pobrane\Alfabet (1)\Alfabet\D.png | 1111  1000  1111  1000  1111 | D:\Pobrane\Alfabet (1)\Alfabet\F.png | 1111  1000  1110  1000  1000 |
| D:\Pobrane\Alfabet (1)\Alfabet\F.png | 1111  1000  1011  1001  1111 | D:\Pobrane\Alfabet (1)\Alfabet\H.png | 1001  1001  1111  1001  1001 |
| D:\Pobrane\Alfabet (1)\Alfabet\J.png | 1111  0110  0110  0100  1111 | D:\Pobrane\Alfabet (1)\Alfabet\J.png | 1111  0001  0011  0101  1111 |
| D:\Pobrane\Alfabet (1)\Alfabet\L.png | 1001  1010  1100  1010  1001 | D:\Pobrane\Alfabet (1)\Alfabet\L.png | 1000  1000  1000  1000  1111 |
| D:\Pobrane\Alfabet (1)\Alfabet\B.png | 1001  1001  1101  1011  1001 | D:\Pobrane\Alfabet (1)\Alfabet\B.png | 0110  1001  1001  1001  0110 |
| D:\Pobrane\Alfabet (1)\Alfabet\P.png | 1111  1001  1111  1000  1000 | D:\Pobrane\Alfabet (1)\Alfabet\P.png | 1111  1001  1111  1010  1001 |
| D:\Pobrane\Alfabet (1)\Alfabet\H.png | 1111  1111  0110  0110  0110 | D:\Pobrane\Alfabet (1)\Alfabet\E.png | 1001  1001  1001  1001  1111 |
| D:\Pobrane\Alfabet (1)\Alfabet\E.png | 1001  1001  1111  0110  0110 | D:\Pobrane\Alfabet (1)\Alfabet\G.png | 1111  0001  0110  1000  1111 |

Tab. 1 Przygotowany alfabet

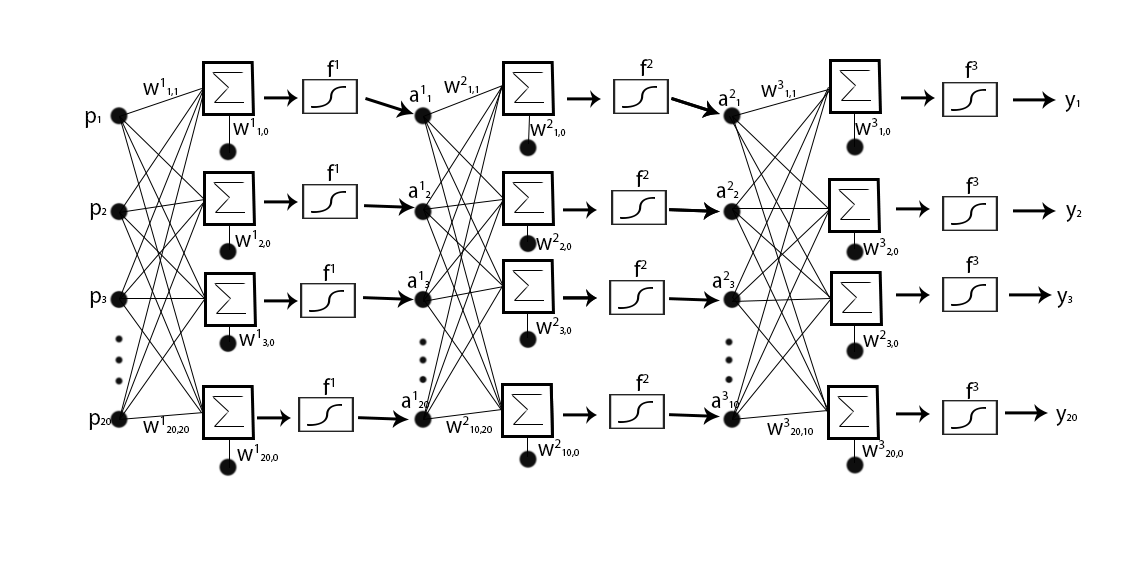
A więc ciąg uczący składa się z 20 wektorów uczących, a także mamy 20 wektorów wyjściowych.

|  |  |
| --- | --- |
| **Wektor wejścia** | **Wektor wyjścia** |
| A | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| B | 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| C | 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| D | 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| E | 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| F | 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| G | 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| H | 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 |
| I | 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 |
| J | 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 |
| K | 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 |
| L | 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 |
| N | 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 |
| O | 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 |
| P | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 |
| R | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 |
| T | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 |
| U | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 |
| Y | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 |
| Z | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 |

Każdy wektor wejściowy składa się z 20 wartości tak samo jak wektor wyjściowy, dlatego też postanowiono stworzyć sieć trójwarstwową składającą się z następującej liczby neuronów w poszczególnych warstwach:

* Warstwa wejściowa: 20 neuronów
* Warstwa ukryta: 10 neuronów
* Warstwa wyjściowa: 20

Schemat całej sieci został zamieszczony na poniższym rysunku.



Rys. 1 Schemat sieci neuronowej

Sieć wykorzystuje unipolarną, logistyczną(sigmoidalną) funkcję aktywacji, natomiast proces uczenia wykorzystuje algorytm wstecznej propagacji błędu, która odbywa się w kierunku przeciwnym niż przepływ sygnałów w sieci.

Kolejną zmianą jest także nadanie procesowi pewnej bezwładności, która ma zapobiegać gwałtownym zmianom wartości wagowych, które są niekorzystne w procesie uczenia. W jej wyniku zmiana wag w bieżącym cyklu zależy również od zmiany jaka nastąpiła w cyklu poprzednim i pośrednio w cyklach jeszcze wcześniejszych. Bezwładność tę nadaje współczynnik *momentum*. Jak łatwo zauważyć, reguluje on wpływ zmiany wag na proces uczenia. Kiedy stosowane jest momentum, korekta wag neuronu zależy nie tylko od sygnału wejściowego i błędu, jaki neuron popełnił, ale również od tego, jak duża była korekta wag w poprzednim kroku uczenia. W ten sposób szybkość uczenia (wielkość korekty wag) automatycznie maleje w miarę zbliżania się do właściwego rozwiązania, a proces uczenia staje się płynniejszy.

A więc uczenie przebiega według następującego schematu w danej epoce uczenia:  
1. Wybór > 0 oraz Emax > 0  
2. Wybór początkowych wartości wag, jako niewielkich liczb losowych  
3. Ustawienie sumy błędu kwadratowego na zero (E = 0)  
4. Dla danego wektora uczącego obliczmy odpowiedź sieci (warstwa po warstwie). A więc dla pojedynczego neuronu obliczana jest suma ilorazów sygnałów wejściowych i wag - uki. Następnie obliczamy wartość wyjściową neuronu aki wykorzystując tą wartość za pomocą unipolarnej, sigmoidalnej funkcji aktywacji według wzoru:

gdzie wybrano z zakresu (0, 2>

A więc elementy wektora wyjściowego w naszym przypadku możemy zapisać jako:

5. Dla każdego neuronu wyliczany jest jego błąd, który można określić jako:

gdzie to kolejno wagi oraz błędu neuronów, do których analizowany neuron wysyła swój wygnał.

6. Każdy neuron modyfikuje wagi na podstawie wartości błędu z wykorzystaniem następujących zależności:

gdzie:  
-α to współczynnik bezwładności, którego wartość optymalna jest testowana  
-η to współczynnik uczenia wybierany z zakresu od 0 do 1  
-pochodną funkcji wyliczamy ze wzoru:

7. Obliczenie łącznego błędu epoki:

8. Uczenie uważamy za zakończone, jeżeli E < Emax. W przeciwnym razie rozpoczynamy nową epokę uczenia i wracamy do punktu nr 3.

1. **Otrzymane wyniki**

Wielokrotne uruchamianie procesu uczenia przygotowanej sieci umożliwiło sprawdzenia jak wielu epok potrzebuje sieć, aby osiągnąć założoną wartość błędu średniokwadratowego, która w tym przypadku została ustawiona na 1,0. Czynnikiem różnicującym poszczególne procesy uczenia było użycie różnych wartości współczynnika uczenia, a także różnych wartości współczynnika bezwładności. Proces testowania zaimplementowanej sieci neuronowej polegał na wczytaniu do programu wcześniej przygotowanych danych testujących składających się z 20 dużych liter alfabetu i sprawdzenia czy sieć jest w stanie bezbłędnie rozpoznać wczytaną literę. Testy przeprowadzono z osobna dla każdej z kombinacji przyjętych wartości współczynnika uczenia jak i współczynnika bezwładności.

W poniższej tabeli zebrano dane z pracy programu, zawierające ilość epok jakich potrzebowała sieć przy przyjętych wartościach współczynnika bezwładności oraz współczynnika uczenia, aby uzyskany przez nią błąd średniokwadratowy był mniejszy od wartości 1,0.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Wartość współczynnika bezwładności | Wartość współczynnika uczenia | | | | |
| 0,2 | 0,4 | 0,6 | 0,8 | 1,4 |
| Liczba epok potrzebna do uzyskania akceptowalnej poprawności | | | | |
| 0,1 | 3173 | 2176 | 1393 | 1253 | 453 |
| 0,3 | 1203 | 563 | 327 | 263 | 262 |
| 0,6 | 661 | 270 | 169 | 165 | 135 |
| 0,9 | 404 | 193 | 140 | 120 | 112 |
| 1,4 | 221 | 131 | 242 | 92 | 163 |

Tab. 2 Zestawienie ilości epok potrzebnych do uzyskania założonych wartości błędu średniokwadratowego z zależności od wartości współczynnika uczenia i współczynnika bezwładności

Na poniższym wykresie pokazana jak liczba epok potrzebnych do uzyskania przez sieć wysokiego współczynnika poprawnych odpowiedzi jest uzależniona od współczynnika uczenia oraz współczynnika bezwładności.

Wykres. 1 Zestawienie ilości epok potrzebnych do uzyskania założonych wartości błędu średniokwadratowego z zależności od wartości współczynnika uczenia i współczynnika bezwładności

Na poniższym wykresie pokazany jest wpływ współczynnika bezwładności na zmiany średniego błędu kwadratowego z epoki na epokę. Widzimy, iż przy wyższym współczynniku bezwładności zmiany te są zdecydowanie bardziej łagodne i brakuje tutaj gwałtownych i nieprzewidywanych zmian wartości błędu średniokwadratowego.

Wykres. 2 Porównanie wartości błędu średniokwadratowego w zależności od wartości współczynnika bezwładności

1. **Analiza wyników**

Na podstawie danych z Tabeli nr. 2 jak i Wykresu nr. 1 można stwierdzić, iż wartość współczynnika uczenia jak i wartość współczynnika bezwładności w znaczący sposób wpływają na ilość epok potrzebnych do uzyskania przez naszą sieć założonej wartości błędu średniokwadratowego wynoszącej w moim przypadku 1,0. Istnieją jednak sytuację, kiedy liczba epok potrafi wzrosnąć jak to ma miejsce np. w przypadku kombinacji współczynnika bezwładności i współczynnika uczenia wynosi kolejno: 1,4 – 0,6 jak i 1,4 – 1,4. Na tej podstawie możemy stwierdzić, iż wartości współczynników muszą być dobrane mądrze, wynikać z wielokrotnych testów, tak aby stworzona przez na sieć cechowała się jak największą skutecznością, procentem poprawnych odpowiedzi w zastosowaniach do których to będzie wykorzystywana. W przypadku zbyt dużych lub po prostu nieodpowiednych współczynników obserwujemy proces nagłego zwiększania się wartości błędu średniokwadratowego w procesie uczenia pomimo tego, iż początkowo wartości te sukcesywnie ulegały zmniejszaniu. Skoki te możemy zaobserwować chociażby na poniższym wykresie oznaczonym numerem 3.

Wykres nr. 3 Wahania wartości błędu średniokwadratowego przy nieodpowiednio dobranych wartościach współczynnika bezwładności oraz uczenia

Na powyższym wykresie widzimy jak dużego rzędu potrafią być to skoki oraz to ile epok potrafi trwać względne unormowanie się wartości błędu. Na podstawie tego wykresu widzimy jak nieodpowiednie wartości dobranych współczynników mogą wpłynąć na zdolność do nauki naszej sieci, a także na wydajność, czas i złożoność obliczeniową naszego algorytmu.

Podczas uczenia się naszej sieci założono, iż będzie się ona uczyć tak długo, dopóki błąd średniokwadratowy nie osiągnie wartości mniejszej od 1,0. Po przeprowadzeniu wielokrotnych testów na wcześniej przygotowanych danych testujących można stwierdzić, iż tak nauczona sieć cechuje się 100% skutecznością w rozpoznawaniu nauczonych przez nią liter alfabetu.

Wykres nr. 4 Procent poprawnych i błędnych odpowiedzi podczas procesu uczenia sieci neuronowej

Na końcu postanowiono sprawdzić skuteczność sieci podczas rozpoznawania liter, które to nie znajdowały się w danych uczących. Wyniki uzyskane przez sieć okazały się być całkiem zadowalające, jednak nie można mieć tutaj 100% pewności, że odpowiedź zwrócona przez sieć będzie odpowiedzą poprawną.

Wykres nr. 5 Procent poprawnych i błędnych odpowiedzi podczas procesu testowania sieci neuronowej dla liter nie znajdujących się w danych uczących

Jak przedstawiono na powyższym wykresie procent poprawnych odpowiedzi na poziome 91% pozwala stwierdzić, iż sieć jest w stanie w pewnym stopniu rozpoznawać nawet te litery, które to nie znalazły się w zestawie danych uczących.

1. **Wnioski**

* Efektywność procesu uczenia dla sieci neuronowej jest silnie powiązana z wartością współczynnika uczenia sieci jak i wartością współczynnika bezwładności.

Należy na podstawie wielokrotnych testów dobrać takie wartości współczynników, aby uniknąć bardzo powolnego procesu uczenia, ale też, aby nasza sieć nie borykała się z problemem nagłego wzrostu błędu średniokwadratowego co w znaczący sposób wydłuża proces nauki do akceptowalnego przez nas poziomu.

* Projektując sieć neuronową musimy dogłębnie przeanalizować problem który ma ona rozwiązać, ponieważ jej skuteczność w dużej mierze zależy od ilości warstw jak i ilości neuronów w każdej z warstw.
* Zbudowana przeze mnie sieć składający się z trzech warstw, w tym jednej ukrytej bardzo dobrze radzi sobie z rozpoznawaniem nauczonych liter alfabetu cechując się współczynnikiem poprawnych odpowiedzi na poziomie 100%, a w przypadku liter których to nie podano do danych uczących na poziomie 91%, co pozwala stwierdzić, iż wielowarstwowe sieci neuronowe zbudowane z wielu neuronów są stanie w sposób akceptowalny poradzić sobie z problemami do których nie zostały początkowo zaprojektowane.

1. **Kod programu**

**Neuron.h:**

#pragma once

#include <iostream>

#include <vector>

using namespace std;

class Neuron

{

public:

void createInputs(int amountOfDendrites, int amountOfSynapses);

void createDendrite() { \_dendrites.push\_back(0); }

void createSynapse(int index) { \_synapses.push\_back(0); }

int getDendritesAmount() { return \_dendrites.size(); }

int getSynapsesAmount() { return \_synapses.size(); }

double getInput(int index) { return \_dendrites[index]; }

void setInput(int index, double value) { \_dendrites[index] = value; }

double getSynapse(int index) { return \_synapses[index]; }

void setSynapse(int index, double value) { \_synapses[index] = value; }

double getSumOfAllInputs() { return \_sumOfAllInputs; }

double getOutputValue() { return \_outputValue; }

double processSingleInput(int index) { return \_dendrites[index] \* \_synapses[index]; }

void sumOfInputs();

void processOutput(bool testing);

void countNewWeights();

void setErrorSignal(double errorSignal) { \_errorSignal = errorSignal; }

double getErrorSignal() { return \_errorSignal; }

Neuron();

Neuron(int amountOfDendrites, int amountOfOutputs, double learningCoefficient, double inertiaCoefficient);

private:

double \_countFirstWeight();

vector<double> \_dendrites; //inputs

vector<double> \_synapses; //weights

vector<double> \_previousWeights;

double \_sumOfAllInputs;

double \_outputValue;

double \_learningCoefficient;

double \_inertiaCoefficient;

double \_errorSignal;

};

**Neuron.cpp:**

#include "Neuron.h"

#include <time.h>

#include <math.h>

Neuron::Neuron()

{

\_dendrites.resize(0);

\_synapses.resize(0);

\_previousWeights.resize(0);

\_sumOfAllInputs = 0;

\_outputValue = 0;

\_learningCoefficient = 0;

\_inertiaCoefficient = 0;

}

Neuron::Neuron(int amountOfDendrites, int amountOfOutputs, double learningCoefficient, double inertiaCoefficient)

{

createInputs(amountOfDendrites, amountOfOutputs);

\_learningCoefficient = learningCoefficient;

\_inertiaCoefficient = inertiaCoefficient;

\_sumOfAllInputs = 0;

\_outputValue = 0;

}

void Neuron::createInputs(int amountOfDendrites, int amountOfOutputs)

{

double weight = \_countFirstWeight();

for (int j = 0; j < amountOfDendrites; j++)

{

\_dendrites.push\_back(0);

\_previousWeights.push\_back(0);

\_synapses.push\_back(\_countFirstWeight());

}

}

void Neuron::sumOfInputs()

{

\_sumOfAllInputs = 0.0;

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

\_sumOfAllInputs += processSingleInput(i);

}

void Neuron::processOutput(bool testing)

{

if (testing == true)

{

if (\_sumOfAllInputs > 0)

\_outputValue = 1;

else

\_outputValue = 0;

}

else

{

double beta = 2.0;

\_outputValue = (1.0 / (1.0 + (exp(-beta \* \_sumOfAllInputs))));

}

}

void Neuron::countNewWeights()

{

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

\_synapses[i] += \_inertiaCoefficient \* \_learningCoefficient \* \_errorSignal \* (\_outputValue \* (1 - \_outputValue) \* \_dendrites[i]);

}

//private methods

double Neuron::\_countFirstWeight()

{

double max = 0.5;

double min = -0.5;

double weight = ((double(rand()) / double(RAND\_MAX)) \* (max - min)) + min;

return weight;

}

**Layer.h:**

#pragma once

#include<vector>

#include"Neuron.h"

using namespace std;

class Layer

{

public:

Layer(int numberOfNeurons, int amountOfDendrites, int amountOfOutputs, double learningCoefficient, double inertiaCoefficient);

vector<Neuron> neurons;

int getNumberOfNeurons(){ return \_numberOfNeurons; }

void gatherAnswers();

vector<double> getAnswers() { return \_answers; }

double getAnswer(int index) { return \_answers[index]; }

private:

int \_numberOfNeurons;

double \_learningCoefficient;

vector<double> \_answers;

};

**Layer.cpp:**

#include"Layer.h"

Layer::Layer(int numberOfNeurons, int amountOfDendrites, int amountOfOutputs, double learningCoefficient, double inertiaCoefficient)

{

\_numberOfNeurons = numberOfNeurons;

\_learningCoefficient = learningCoefficient;

neurons.resize(numberOfNeurons);

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

neurons[i].Neuron::Neuron(amountOfDendrites, amountOfOutputs, \_learningCoefficient, inertiaCoefficient);

}

void Layer::gatherAnswers()

{

\_answers.clear();

for (int i = 0; i < \_numberOfNeurons; i++)

\_answers.push\_back(neurons[i].getOutputValue());

}

**main.cpp:**

#include <iostream>

#include <vector>

#include "Layer.h"

#include <time.h>

#include <fstream>

using namespace std;

void setInputValue(Neuron& neuron, vector< vector<int> > inputData, vector< vector<double> > neuronAnswers, int numberOfOutputs, int numberOfEntrances, int numberOfLayer, int inputDataRow);

void learn(vector<Layer>& layers, vector< vector<int> > inputData);

void backpropagationFunction(vector<Layer>& layers);

void test(vector<Layer>& layers, vector< vector<int> > inputData);

void readLearningData(vector< vector<int> > &learningInputData, int numberOfEntrances, int amountOfOutputs);

void readTestingData(vector< vector<int> > &testingInputData, int numberOfEntrances, int amountOfOutputs);

fstream OUTPUT\_LEARNING\_FILE;

fstream OUTPUT\_TESTING\_FILE;

fstream LEARNING\_DATA;

fstream TESTING\_DATA;

double E;

double E\_MAX = 0.5;

double E\_TESTING;

int BAD\_ANSWERS\_AMOUNT;

int TOTAL\_BAD\_ANSWERS\_AMOUNT;

int RIGHT\_ANSWERS\_AMOUNT;

int TOTAL\_RIGHT\_ANSWERS\_AMOUNT;

int NUMBER\_OF\_CORRECT\_ANSWERS;

int \*CORRECT\_ANSWERS;

int main()

{

srand(time(NULL));

vector<Layer> layers;

vector< vector<int> > learningInputData;

vector< vector<int> > testingInputData;

vector< vector<double> > neuronAnswers;

int numberOfNeuronsInFirstLayer = 20;

int numberOfNeuronsInSecondLayer = 10;

int numberOfNeuronsInThirdLayer = 20;

int numberOfEntrancesInFirstLayer = 21;

int numberOfEntrancesInSecondLayer = 21;

int numberOfEntrancesInThirdLayer = 11;

int numberOfOutputs = 1;

double learningCoefficient = 0.8;

double inertiaCoefficient = 1.4;

layers.push\_back(Layer::Layer(numberOfNeuronsInFirstLayer, numberOfEntrancesInFirstLayer, numberOfOutputs, learningCoefficient, inertiaCoefficient));

layers.push\_back(Layer::Layer(numberOfNeuronsInSecondLayer, numberOfEntrancesInSecondLayer, numberOfOutputs, learningCoefficient, inertiaCoefficient));

layers.push\_back(Layer::Layer(numberOfNeuronsInThirdLayer, numberOfEntrancesInThirdLayer, numberOfOutputs, learningCoefficient, inertiaCoefficient));

readLearningData(learningInputData, numberOfEntrancesInFirstLayer, numberOfOutputs);

readTestingData(testingInputData, numberOfEntrancesInFirstLayer, numberOfOutputs);

CORRECT\_ANSWERS = new int[NUMBER\_OF\_CORRECT\_ANSWERS];

do

{

cout << "1. Learn" << endl;

cout << "2. Test" << endl;

cout << "3. Exit" << endl;

int choice;

cin >> choice;

switch (choice)

{

case 1:

OUTPUT\_LEARNING\_FILE.open("output\_learning\_data.txt", ios::out);

for (int epochNumber = 1; ;)

{

learn(layers, learningInputData);

OUTPUT\_LEARNING\_FILE << "EPOCH NUMBER: " << epochNumber << " E: " << E << endl;

if (E < E\_MAX)

break;

epochNumber++;

E = 0;

}

break;

case 2:

OUTPUT\_TESTING\_FILE.open("output\_testing\_data.txt", ios::out);

BAD\_ANSWERS\_AMOUNT = 0;

RIGHT\_ANSWERS\_AMOUNT = 0;

E\_TESTING = 0;

test(layers, testingInputData);

OUTPUT\_TESTING\_FILE << "Bad answers amount: " << BAD\_ANSWERS\_AMOUNT << endl;

OUTPUT\_TESTING\_FILE << "Right answers amount: " << RIGHT\_ANSWERS\_AMOUNT << endl;

OUTPUT\_TESTING\_FILE << "E: " << E\_TESTING << endl;

break;

case 3:

OUTPUT\_LEARNING\_FILE.close();

OUTPUT\_TESTING\_FILE.close();

return 0;

}

} while (true);

return 0;

}

void setInputValue(Neuron& neuron, vector< vector<int> > inputData, vector<double> neuronAnswers, int numberOfOutputs, int numberOfEntrances, int numberOfLayer, int row)

{

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

{

if (numberOfLayer == 0)

neuron.setInput(i, inputData[row][i]);

else

{

if (i == 0)

neuron.setInput(i, 1);

else

neuron.setInput(i, neuronAnswers[i - 1]);

}

}

if (numberOfLayer == 2)

{

for (int i = 0, j = (inputData[0].size() - numberOfOutputs); i < NUMBER\_OF\_CORRECT\_ANSWERS; i++, j++)

CORRECT\_ANSWERS[i] = inputData[row][j];

}

}

void backpropagationFunction(vector<Layer>& layers)

{

for (int layerIndex = (layers.size() - 1); layerIndex >= 0; layerIndex--)

{

for (int i = 0; i < layers[layerIndex].getNumberOfNeurons(); i++)

{

if (layerIndex == (layers.size() - 1))

layers[layerIndex].neurons[i].setErrorSignal(CORRECT\_ANSWERS[i] - layers[layerIndex].neurons[i].getOutputValue());

else

{

double errorSignal = 0.0;

for (int j = 0; j < layers[layerIndex + 1].getNumberOfNeurons(); j++)

errorSignal += layers[layerIndex + 1].neurons[j].getErrorSignal() \* layers[layerIndex + 1].neurons[j].getSynapse(i + 1);

layers[layerIndex].neurons[i].setErrorSignal(errorSignal);

}

}

}

}

void learn(vector<Layer>& layers, vector< vector<int> > inputData)

{

for (int inputDataRow = 0; inputDataRow < inputData.size(); inputDataRow++)

{

for (int layerIndex = 0; layerIndex < layers.size(); layerIndex++)

{

for (int i = 0; i < layers[layerIndex].getNumberOfNeurons(); i++)

{

setInputValue(layers[layerIndex].neurons[i], inputData, layers[(layerIndex > 0) ? (layerIndex - 1) : 0].getAnswers(), layers[layerIndex].getNumberOfNeurons(), layers[layerIndex].neurons[i].getDendritesAmount(), layerIndex, inputDataRow);

layers[layerIndex].neurons[i].sumOfInputs();

layers[layerIndex].neurons[i].processOutput(false);

}

layers[layerIndex].gatherAnswers();

if (layerIndex == (layers.size() - 1))

for (int i = 0; i < layers[layers.size() - 1].getNumberOfNeurons(); i++)

E += 0.5 \* pow(((double)CORRECT\_ANSWERS[i] - layers[layers.size() - 1].getAnswer(i)), 2);

}

backpropagationFunction(layers);

for (int layerIndex = 0; layerIndex < layers.size(); layerIndex++)

for (int i = 0; i < layers[layerIndex].getNumberOfNeurons(); i++)

layers[layerIndex].neurons[i].countNewWeights();

}

}

void test(vector<Layer>& layers, vector< vector<int> > inputData)

{

for (int inputDataRow = 0; inputDataRow < inputData.size(); inputDataRow++)

{

for (int layerIndex = 0; layerIndex < layers.size(); layerIndex++)

{

for (int i = 0; i < layers[layerIndex].getNumberOfNeurons(); i++)

{

setInputValue(layers[layerIndex].neurons[i], inputData, layers[(layerIndex > 0) ? (layerIndex - 1) : 0].getAnswers(), layers[layerIndex].getNumberOfNeurons(), layers[layerIndex].neurons[i].getDendritesAmount(), layerIndex, inputDataRow);

layers[layerIndex].neurons[i].sumOfInputs();

if (layerIndex != (layers.size() - 1))

layers[layerIndex].neurons[i].processOutput(false);

else

layers[layerIndex].neurons[i].processOutput(true);

}

layers[layerIndex].gatherAnswers();

}

for (int i = 0; i < layers[layers.size() - 1].getNumberOfNeurons(); i++)

{

if (layers[layers.size() - 1].getAnswer(i) == CORRECT\_ANSWERS[i])

{

RIGHT\_ANSWERS\_AMOUNT++;

E\_TESTING += 0.5 \* pow(((double)CORRECT\_ANSWERS[i] - layers[layers.size() - 1].getAnswer(i)), 2);

OUTPUT\_TESTING\_FILE << layers[layers.size() - 1].neurons[i].getOutputValue() << " + ";

}

else

{

BAD\_ANSWERS\_AMOUNT++;

E\_TESTING += 0.5 \* pow(((double)CORRECT\_ANSWERS[i] - layers[layers.size() - 1].getAnswer(i)), 2);

OUTPUT\_TESTING\_FILE << layers[layers.size() - 1].neurons[i].getOutputValue() << " - ";

}

}

OUTPUT\_TESTING\_FILE << endl;

}

}

void readLearningData(vector< vector<int> > &learningInputData, int numberOfEntrances, int amountOfOutputs)

{

LEARNING\_DATA.open("learning\_data.txt", ios::in);

vector<int> row;

do

{

row.clear();

for (int i = 0, inputTmp = 0; i < numberOfEntrances; i++)

{

LEARNING\_DATA >> inputTmp;

row.push\_back(inputTmp);

}

for (int i = 0, answerTmp; i < (numberOfEntrances - 1); i++)

{

LEARNING\_DATA >> answerTmp;

row.push\_back(answerTmp);

}

learningInputData.push\_back(row);

}while (!LEARNING\_DATA.eof());

NUMBER\_OF\_CORRECT\_ANSWERS = row.size() / 2;

LEARNING\_DATA.close();

}

void readTestingData(vector< vector<int> > &testingInputData, int numberOfEntrances, int amountOfOutputs)

{

TESTING\_DATA.open("testing\_data.txt", ios::in);

vector<int> row;

while (!TESTING\_DATA.eof())

{

row.clear();

for (int i = 0, inputTmp; i < numberOfEntrances; i++)

{

TESTING\_DATA >> inputTmp;

row.push\_back(inputTmp);

}

for (int i = 0, answerTmp; i < (numberOfEntrances - 1); i++)

{

TESTING\_DATA >> answerTmp;

row.push\_back(answerTmp);

}

testingInputData.push\_back(row);

}

TESTING\_DATA.close();

}