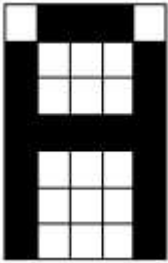
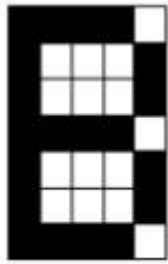
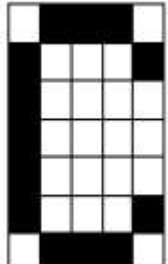
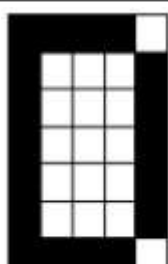


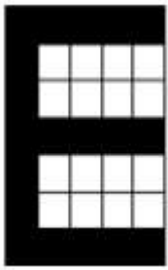
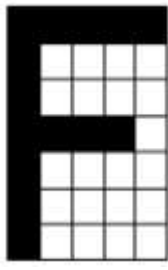
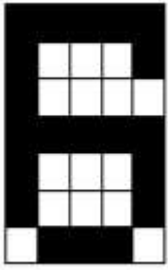
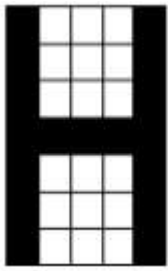
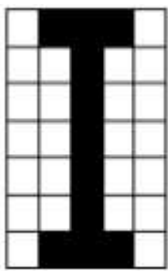
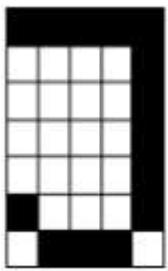
12.02.2018r.	Izabela Musztyfaga	Podstawy Sztucznej Inteligencji
Problem 6 - Budowa i działanie sieci Kohonena dla WTM		

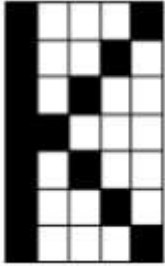
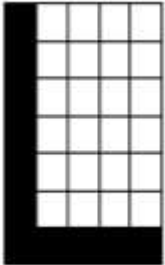
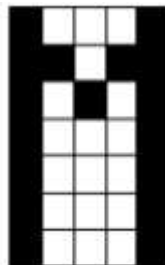
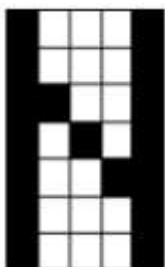
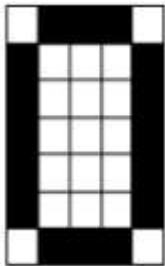
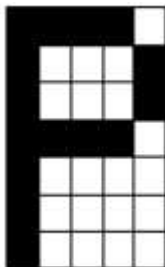
Cel ćwiczenia

Celem ćwiczenia jest poznanie budowy i działania sieci Kohonena przy wykorzystaniu reguły WTM do odwzorowywania istotnych cech liter alfabetu.

Dane uczące sieć oraz dane testujące sieć

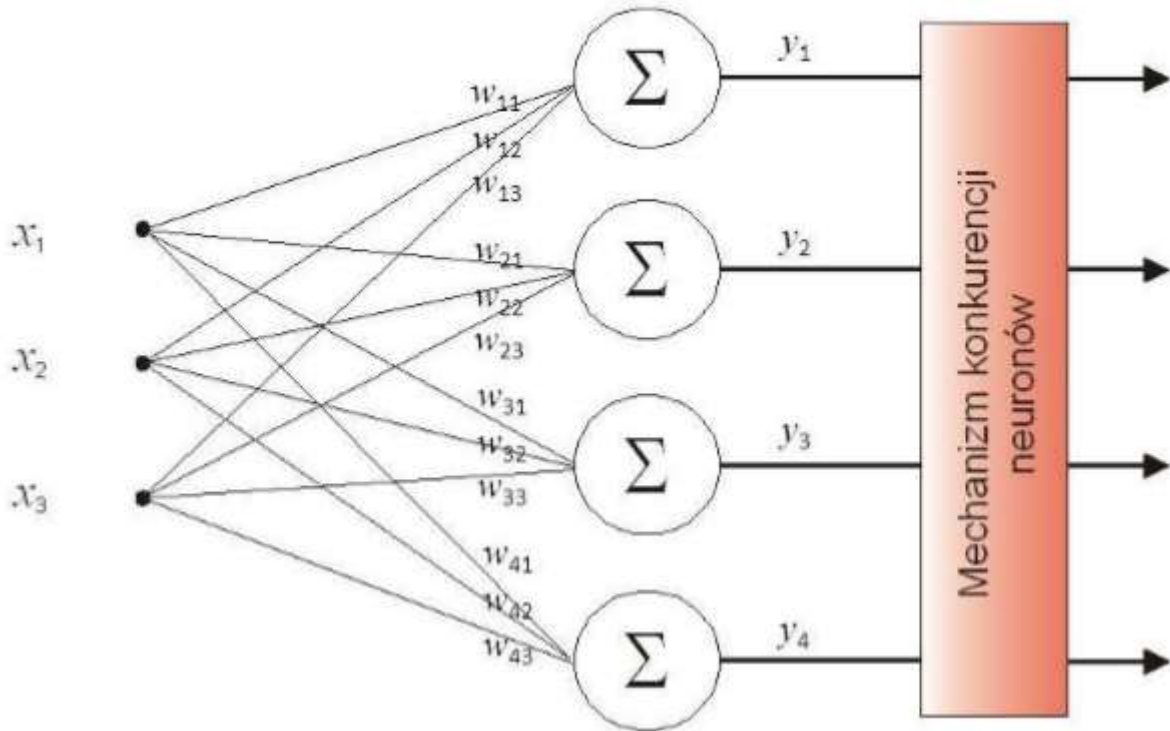
Lp.	Litera przedstawiona graficznie	Litera po konwersji do 0 i 1	Wektor
1		0 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1	0 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1
2		1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0	1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0
3		0 1 1 1 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 1 1 1 0	0 1 1 1 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 1 1 1 0
4		1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0	1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0

5		1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1	1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1
6		1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0	1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0
7		1 1 1 1 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 0 1 1 1 0	1 1 1 1 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 0 1 1 1 0
8		1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1	1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1
9		0 1 1 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 1 0	0 1 1 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 1 0
10		1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 1 1 1 0	1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 1 1 1 0

11		10001 10010 10100 11000 10100 10010 10001	10001100101010011000101001001010001
12		10000 10000 10000 10000 10000 10000 10000 11111	1000010000100001000010000100001000011111
13		10001 11011 10101 10001 10001 10001 10001 10001	10001110111010110001100011000110001
14		10001 10001 11001 10101 10011 10001 10001 10001	10001100011100110101100111000110001
15		01110 10001 10001 10001 10001 10001 10001 01110	0111000011000110001100011000101110
16		11110 10001 10001 11110 10000 10000 10000	11110100011000111110100001000010000

17		1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0 1 0 1 0 0 1 0 0 1 0 1 0 0 0 1	1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0 1 0 1 0 0 1 0 0 1 0 1 0 0 0 1
18		0 1 1 1 0 1 0 0 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 1 0 1 1 1 0	0 1 1 1 0 1 0 0 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 1 0 1 1 1 0
19		1 1 1 1 1 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0	1 1 1 1 1 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0
20		0 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 1 0 1 1 0 0 1 0 0 1 1 0 1	0 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 1 0 1 1 0 0 1 0 0 1 1 0 1

Syntetyczny opis sieci



Nauka sieci odbywa się za pomocą uczenia rywalizującego (metoda uczenia sieci samoorganizujących). Podczas procesu uczenia neurony są nauczane rozpoznawania danych i zbliżają się do obszarów zajmowanych przez te dane. Po wejściu każdego wektora uczącego wybierany jest tylko jeden neuron (neuron będący najbliższemu prezentowanemu wzorcowi). Wszystkie neurony rywalizują między sobą, gdzie zwycięża ten neuron, którego wartość jest największa. Zwycięski neuron przyjmuje na wyjściu wartość 1, pozostałe 0.

Sieć dzieli dane na grupy, biorąc pod uwagę to, aby elementy były do siebie jak najbardziej podobne, a jednocześnie zupełnie inne dla różnych grup.

Metoda WTM zakłada, że uczenie sieci wykorzystuje promień po to, aby aktualizować wagi również tych neuronów, które nie są zwycięzcami. Przy każdym kolejnym razie promień ten się zmniejsza. Celem tego jest to, że na końcu wagi aktualizuje tylko neuron zwycięski.

Algorytm uczenia sieci

1. Znormalizować wszystkie dane
2. Wybrać współczynnik uczenia η z przedziału $(0; 1)$
3. Wybrać początkowe wartości wag z przedziału $(0; 1)$
4. Dla danego zbioru uczącego (dla każdego pojedynczego neuronu) obliczyć odpowiedź sieci
5. Wybierać jest neuron, którego odległość euklidesowa - tylko dla niego aktualizuje się wagi, według następującego wzoru:

$$w_{i,j}(t+1) = w_{i,j}(t) + \eta * \theta(t) * (x_i * w_{i,j}(t))$$

przy czym $\theta(t)$ obliczamy ze wzoru:

$$\theta(t) = e^{\frac{-d^2}{2 \cdot R^2}}$$

$$d(i, w) = \sqrt{\sum_{i=1}^n (i_i - w_i)^2}$$

$$R(t) = R_0 * e^{-\frac{t}{\lambda}}$$

$$\lambda = \frac{x}{R_0}$$

6. Znormalizować wartości nowego wektora wag
7. 1 – odpowiedź zwycięskiego neuronu; 0 – odpowiedź reszty neuronów
8. Wczytać kolejny wektor uczący

Zestawienie otrzymanych wyników

Zbiorem danych testowych są litery (12), których kolejne wartości w wektorze są zmienione.
Ilość epok - 50

Promień = 2					
Współczynnik uczenia się = 0.7		Współczynnik uczenia się = 0.5		Współczynnik uczenia się = 0.3	
Grupa 1	C,D,G,J	Grupa 1	E,F,H,K,L	Grupa 1	A,H
Grupa 2	A,B,H,K,L	Grupa 2	A	Grupa 2	B,E,F,K,L
Grupa 3	E,F,I	Grupa 3	B,C,D,G,I,J	Grupa 3	C,I,J
Grupa 4	-	Grupa 4	-	Grupa 4	D,G
Grupa 5	-	Grupa 5	-	Grupa 5	-
Grupa 6	-	Grupa 6	-	Grupa 6	-
Grupa 7	-	Grupa 7	-	Grupa 7	-
Grupa 8	-	Grupa 8	-	Grupa 8	-
Grupa 9	-	Grupa 9	-	Grupa 9	-
Grupa 10	-	Grupa 10	-	Grupa 10	-

Promień = 5					
Współczynnik uczenia się = 0.7		Współczynnik uczenia się = 0.5		Współczynnik uczenia się = 0.3	
Grupa 1	A,B,F	Grupa 1	A,F	Grupa 1	K,L
Grupa 2	E,L	Grupa 2	H	Grupa 2	A,H
Grupa 3	H,K	Grupa 3	K,L	Grupa 3	B,E,F
Grupa 4	I	Grupa 4	E,I	Grupa 4	J
Grupa 5	J	Grupa 5	B,C,D,G,J	Grupa 5	C,D,G,I
Grupa 6	D	Grupa 6	-	Grupa 6	-
Grupa 7	C,G	Grupa 7	-	Grupa 7	-
Grupa 8	-	Grupa 8	-	Grupa 8	-
Grupa 9	-	Grupa 9	-	Grupa 9	-
Grupa 10	-	Grupa 10	-	Grupa 10	-

Promień = 10					
Współczynnik uczenia się = 0.7		Współczynnik uczenia się = 0.5		Współczynnik uczenia się = 0.3	
Grupa 1	F	Grupa 1	G,J	Grupa 1	C
Grupa 2	A,H	Grupa 2	C,I	Grupa 2	I
Grupa 3	B,G	Grupa 3	D,E	Grupa 3	B,D,G,I
Grupa 4	J	Grupa 4	L	Grupa 4	E
Grupa 5	I	Grupa 5	F,K	Grupa 5	L
Grupa 6	E,L	Grupa 6	A,B,H	Grupa 6	K
Grupa 7	C	Grupa 7	-	Grupa 7	F,G
Grupa 8	-	Grupa 8	-	Grupa 8	A
Grupa 9	-	Grupa 9	-	Grupa 9	-
Grupa 10	-	Grupa 10	-	Grupa 10	-

Wnioski

Na pierwszy rzut oka można wywnioskować, że im większy założony promień, tym więcej grup się tworzy. Współczynnik uczenia ma na to mniejszy wpływ.

Sieć – ponieważ posiada cechy sieci samoorganizującej się – sama potrafi podzielić dane na grupy (obszary), ze względu na podobieństwo cech. Z pewnością sama sieć wykorzystuje większe zasoby pamięci urządzenia, ponieważ nie zwraca uwagi tylko na neurony zwycięskie, w przeciwieństwie do 'metody' WTA – Winners Take All.

Listing programu wraz z komentarzami
(kod nie pisany, znaleziony w internecie)

main.cpp

```
#include <iostream>
#include <vector>
#include <ctime>
#include <fstream>

#include "Network.h"

using namespace std;

//wczytanie do tablic danych wejsciowych
void setInputData(Neuron& neuron, vector<vector<double>> inputData, int numberOfInputs, int row);
//uczenie sieci
void learn(Network& layer, vector<vector<double>> inputData);
//testowanie sieci
void test(Network& layer, vector<vector<double>> inputData);
//wczytanie danych uczacych
void loadTrainingData(vector<vector<double>>&learningInputData, int numberOfInputs);
//wczytanie danych testowych
void loadTestingData(vector<vector<double>>&testingInputData, int numberOfInputs);

//strumienie do plikow sluzace do wczytania danych uczacych oraz zapisu wynikow
fstream OUTPUT_FILE_LEARNING, OUTPUT_FILE_TESTING_DATA, OUTPUT_FILE_TESTING_NEURON;
fstream TRAINING_DATA, TESTING_DATA;

int main() {
    srand(time(NULL));

    //wektory z danymi uczacyimi oraz testujacymi
    vector<vector<double>> trainData;
    vector<vector<double>> testData;
    int numberOfNeurons = 20;
    int numberOfInputs = 35;
    double learningRate = 0.3;
    int epoch = 50;
    //stworzenie sieci Kohonena
    Network kohonenNetwork(numberOfNeurons, numberOfInputs, learningRate, epoch);
    //wczytanie danych uczacych
    loadTrainingData(trainData, numberOfInputs);
    //wczytanie danych testowych
    loadTestingData(testData, numberOfInputs);

    //"menu" programu
    do {
        cout << "1. Learn" << endl;
        cout << "2. Test" << endl;
        cout << "3. Exit" << endl;

        int choice;
        cin >> choice;
        switch (choice) {
            case 1:
                OUTPUT_FILE_LEARNING.open("output_learning_data.txt", ios::out);

                for (int epochNumber = 1, i = 0; i < epoch; i++, epochNumber++) {
                    //uczenie
                    learn(kohonenNetwork, trainData);
                    OUTPUT_FILE_LEARNING << "Epoch: " << epochNumber << endl;
                    cout << "Epoch: " << epochNumber << endl;
                }
            case 2:
                test(kohonenNetwork, testData);
                OUTPUT_FILE_TESTING_DATA.open("output_testing_data.txt", ios::out);
                for (int i = 0; i < testData.size(); i++) {
                    test(kohonenNetwork, testData[i]);
                    OUTPUT_FILE_TESTING_DATA << "Epoch: " << i << endl;
                    cout << "Epoch: " << i << endl;
                }
            case 3:
                break;
        }
    } while (choice != 3);

    OUTPUT_FILE_TESTING_NEURON.open("output_testing_neuron.txt", ios::out);
    for (int i = 0; i < kohonenNetwork.getNeurons().size(); i++) {
        test(kohonenNetwork.getNeurons()[i]);
        OUTPUT_FILE_TESTING_NEURON << "Epoch: " << i << endl;
        cout << "Epoch: " << i << endl;
    }
}
```



```

        }
        OUTPUT_FILE_LEARNING.close();
        break;
    case 2:
        OUTPUT_FILE_TESTING_DATA.open("output_testing_data.txt", ios::out);
        OUTPUT_FILE_TESTING_NEURON.open("output_testing_neuron.txt", ios::out);
        //testowanie
        test(kohonenNetwork, testData);
        break;
    case 3:
        OUTPUT_FILE_LEARNING.close();
        OUTPUT_FILE_TESTING_DATA.close();
        return 0;
    default:
        cout << "BAD BAD BAD" << endl;
    }
} while (true);

return 0;
}

//wczytanie do tablic danych wejsciowych
void setInputData(Neuron& neuron, vector<vector<double>> inputData, int numberOfInputs, int
row)
{
    for (int i = 0; i < numberOfInputs; i++)
        neuron.inputs[i] = inputData[row][i];
}

//uczenie
void learn(Network& layer, vector<vector<double>> inputData)
{
    static int currentIteration = 0;
    for (int rowOfData = 0; rowOfData < inputData.size(); rowOfData++) {
        for (int i = 0; i < layer.numberOfNeurons; i++) {
            //wczytanie danych do tablic
            setInputData(layer.neurons[i], inputData,
layer.neurons[i].getInputsSize(), rowOfData);
            //wyliczenie odleglosci euklidesowych
            layer.neurons[i].calculateScalarProduct();
        }
        //zmiana wag
        layer.changeWeights(currentIteration, true);

        OUTPUT_FILE_LEARNING << layer.winnerIndex << endl;
        cout << "Winner: " << layer.winnerIndex << endl;
        currentIteration++;
    }
}

//testowanie
void test(Network& layer, vector<vector<double>> inputData) {
    for (int rowOfData = 0; rowOfData < inputData.size(); rowOfData++) {
        for (int i = 0; i < layer.numberOfNeurons; i++) {
            //wczytanie danych do tablic
            setInputData(layer.neurons[i], inputData,
layer.neurons[i].getInputsSize(), rowOfData);
            //wyliczenie odleglosci euklidesowych
            layer.neurons[i].calculateScalarProduct();
        }
        char letter = 'A';
        layer.changeWeights(0, false);
        OUTPUT_FILE_TESTING_DATA << layer.neurons[layer.winnerIndex].getInputsSize()
<< endl;
        OUTPUT_FILE_TESTING_NEURON << (char)(letter + rowOfData) << " " <<

```

```

layer.winnerIndex << endl;
    cout << (char)(letter + rowOfData) << " " << layer.winnerIndex << endl;
}
}

//wczytanie danych uczacych z pliku
void loadTrainingData(vector<vector<double>> &inputData, int numberOfInputs) {
    TRAINING_DATA.open("data.txt", ios::in);
    vector<double> row;

    do {
        row.clear();

        for (int i = 0; i < numberOfInputs; i++) {
            double inputTmp = 0.0;
            TRAINING_DATA >> inputTmp;
            row.push_back(inputTmp);
        }

        //znormalizowanie danych uczacych
        double length = 0.0;

        for (int i = 0; i < numberOfInputs; i++)
            length += pow(row[i], 2);

        length = sqrt(length);

        for (int i = 0; i < numberOfInputs; i++)
            row[i] /= length;

        inputData.push_back(row);
    } while (!TRAINING_DATA.eof());

    TRAINING_DATA.close();
}

//wczytanie danych testujacych z pliku
void loadTestingData(vector<vector<double>> &testData, int numberOfInputs) {
    TESTING_DATA.open("datatest.txt", ios::in);
    vector<double> row;

    while (!TESTING_DATA.eof()) {
        row.clear();

        for (int i = 0; i < numberOfInputs; i++) {
            double inputTmp = 0.0;
            TESTING_DATA >> inputTmp;
            row.push_back(inputTmp);
        }

        //znormalizowanie danych uczacych
        double length = 0.0;

        for (int i = 0; i < numberOfInputs; i++)
            length += pow(row[i], 2);

        length = sqrt(length);

        for (int i = 0; i < numberOfInputs; i++)
            row[i] /= length;

        testData.push_back(row);
    }
}

```

```

    TESTING_DATA.close();
}

```

Neuron.h

```

#include <iostream>
#include <vector>

using namespace std;

class Neuron {
public:
    vector<double> inputs; //wejścia
    vector<double> weights; //wagi
    double sumOfAllInputs; //obliczenie odległości skalarnych
    double outputValue; //wartość wyjściowa
    double learningRate; //współczynnik uczenia
    double valueOfNeighborhoodFunction; //wartość funkcji sąsiedztwa (Gaussian
neighborhood function)
    double length; //odległość euklidesowa
    double neighbors; //odległość do sąsiadów

    double calculateFirstWeights(); //wylosowanie początkowych wag z zakresu <0;1)
    void calculateNeighbors(); //oblicza wartość funkcji sąsiedztwa (Gaussian
neighborhood function)
    void normalizeWeights(); //znormalizowanie zaktualizowanych wag

    //stworzenie początkowych wejść (ustawienie
wejs na 0, wykorzystanie metody calculateFirstWeights())
    void createInputs(int numberOfInputs);
    void activationFunction(); //funkcja sigmoidalna obliczająca wyjście
    void calculateNewWeights(); //obliczenie nowych wag
    double calculateScalarProduct(); //obliczenie odległości skalarnych
    void designateNeighbors(double radius, double currentIteration, double time);
//wyznaczenie odległości do sąsiadów

    int getInputsSize() { //zwraca rozmiar wejść
        return inputs.size();
    }

    int getWeightsSize() { //zwraca rozmiar wejści
        return weights.size();
    }

    Neuron(); //konstruktor
    Neuron(int numberOfInputs, double learningRate); //konstruktor
};

```

Neuron.cpp

```

#include "Neuron.h"
#include <ctime>
#include <cmath>

//konstruktor
Neuron::Neuron() {
    this->inputs.resize(0);
    this->weights.resize(0);
    this->sumOfAllInputs = 0.0;
    this->outputValue = 0.0;
    this->learningRate = 0.0;
}

```

```

//konstruktor
Neuron::Neuron(int numberOfInputs, double learningRate) {
    createInputs(numberOfInputs);
    normalizeWeights();
    this->learningRate = learningRate;
    this->sumOfAllInputs = 0.0;
    this->outputValue = 0.0;
}

//stworzenie początkowych wejść (ustawienie wejść na 0, wykorzystanie metody
calculateFirstWeights())
void Neuron::createInputs(int numberOfInputs) {
    for (int i = 0; i < numberOfInputs; i++) {
        inputs.push_back(0);
        weights.push_back(calculateFirstWeights());
    }
}

//obliczenie odległości skalarnych
double Neuron::calculateScalarProduct() {
    sumOfAllInputs = 0.0;
    for (int i = 0; i < getInputsSize(); i++)
        sumOfAllInputs += pow(inputs[i] - weights[i], 2);
    sumOfAllInputs = sqrt(sumOfAllInputs);

    return sumOfAllInputs;
}

//funkcja sigmoidalna obliczająca wyjście
void Neuron::activationFunction() {
    double beta = 1.0;
    this->outputValue = (1.0 / (1.0 + (exp(-beta * sumOfAllInputs))));
}

//obliczenie nowych wag
void Neuron::calculateNewWeights() {
    for (int i = 0; i < getWeightsSize(); i++)
        this->weights[i] += this->learningRate * this->valueOfNeighborhoodFunction * (this->inputs[i] - this->weights[i]);
    normalizeWeights();
}

//wyznaczenie odległości do sąsiadów
void Neuron::designateNeighbors(double radius, double currentIteraton, double timeConstant)
{
    this->neighbors = radius * exp(-currentIteraton / timeConstant);
}

//oblicza wartość funkcji sąsiedztwa (Gaussian neighborhood function)
void Neuron::calculateNeighbors() {
    //e^(-x^2) / 2 * y^2
    this->valueOfNeighborhoodFunction = exp(-pow(this->length, 2) / (2 * pow(this->neighbors, 2)));
}

//ustalenie początkowych wag dla wszystkich wejść - zakres <0;1)
double Neuron::calculateFirstWeights() {
    double max = 1.0;
    double min = 0.0;
    double weight = ((double(rand()) / double(RAND_MAX)) * (max - min)) + min;
    return weight;
}

//znormalizowanie zaktualizowanych wag
void Neuron::normalizeWeights() {

```

```

double vectorLength = 0.0;

for (int i = 0; i < getWeightsSize(); i++)
    vectorLength += pow(weights[i], 2);

vectorLength = sqrt(vectorLength);

for (int i = 0; i < getWeightsSize(); i++)
    weights[i] /= vectorLength;
}

```

Network.h

```

#include <vector>
#include "Neuron.h"
using namespace std;

class Network {
public:

    int numberOfNeurons; //liczba neuronow
    vector<Neuron> neurons; //wektor neuronow
    vector<double> scalarProducts; //wektor odleglosci euklidesowych
    int winnerIndex; //indeks zwyciezcy
    double radius; //promien wyznaczajacy obszar od zwycieskiego neuronu
    double time; //czas

    void changeWeights(double obecnaIteracja, bool testing); //zmiana wag dla aktualnej
iteracji (czasu)
    void findMinimum(); //szuka najmniejszej odleglosci euklidesowej
    void getScalarProducts(); //zwraca odleglosci euklidesowe

                                                                    //konstruktor
    Network(int numberOfNeurons, int numberOfInputs, double learningRate, double
iterationsNumber);

};

```

Network.cpp

```

#include "Network.h"

//konstruktor
Network::Network(int numberOfNeurons, int numberOfInputs, double learningRate, double
iterationsNumber) {
    this->numberOfNeurons = numberOfNeurons;
    neurons.resize(numberOfNeurons);
    //this->radius = (double)numberOfNeurons;
    this->radius = 10.0;
    this->time = iterationsNumber / this->radius;

    for (int i = 0; i < numberOfNeurons; i++)
        neurons[i].Neuron(numberOfInputs, learningRate);
}

//zmiana wag
void Network::changeWeights(double currentIteration, bool learning) {
    getScalarProducts();
    findMinimum();
    neurons[winnerIndex].activationFunction();

    if (learning) {

```

```

//wyznaczenie sasiadow zwycieskiego neuronu w zaleznosci od promienia i kroku
czasowego
neurons[winnerIndex].designateNeighbors(radius, currentIteration, time);
int radius = neurons[winnerIndex].neighbors;
int leftBorderNeuronIndex = 0;
int rightBorderNeuronIndex = 0;

//sprawdzenie czy dany neuron miesci sie w siatce
if (winnerIndex - radius < 0)
    leftBorderNeuronIndex = 0;
else
    leftBorderNeuronIndex = winnerIndex - radius;

if (winnerIndex + radius >= numberOfNeurons)
    rightBorderNeuronIndex = numberOfNeurons - 1;
else
    rightBorderNeuronIndex = winnerIndex + radius;

radius = (radius <= 0) ? 0 : --radius;

for (int i = leftBorderNeuronIndex; i < rightBorderNeuronIndex; i++) {
    //zmiana wag neuronow, ktore mieszczą się w promieniu
    neurons[i].length = (i < winnerIndex) ? (winnerIndex - i) : (i -
winnerIndex);

    neurons[i].neighbors = neurons[winnerIndex].neighbors;
    neurons[i].calculateNewWeights();
}
}

//zwraca odleglosci euklidesowe
void Network::getScalarProducts() {
    scalarProducts.clear();
    for (int i = 0; i < numberOfNeurons; i++)
        scalarProducts.push_back(neurons[i].calculateScalarProduct());
}

//szuka najmniejszej odleglosci euklidesowej
void Network::findMinimum() {
    double tmp = scalarProducts[0];
    this->winnerIndex = 0;
    for (int i = 1; i < scalarProducts.size(); i++) {
        if (tmp < scalarProducts[i]) {
            this->winnerIndex = i;
            tmp = scalarProducts[i];
        }
    }
}
}

```