

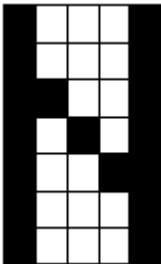
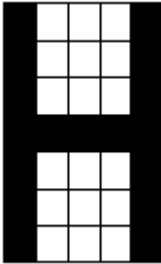
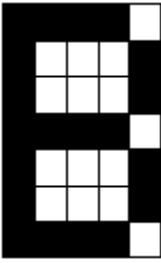
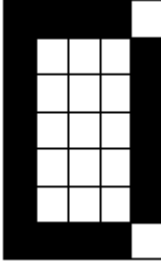
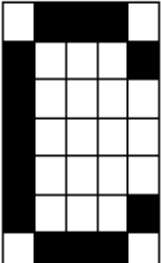
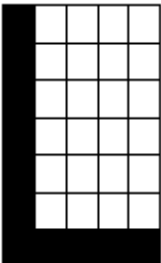
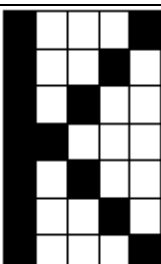
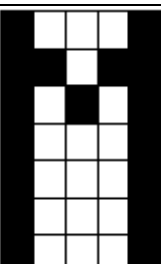
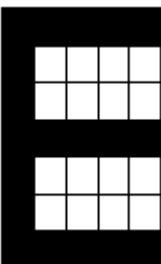
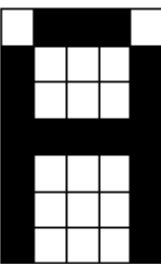
Paweł Nowak

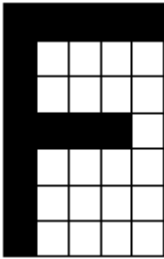
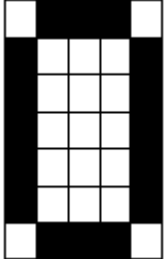
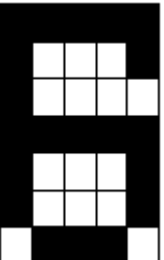
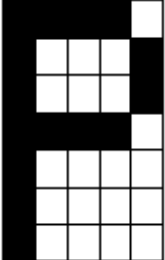
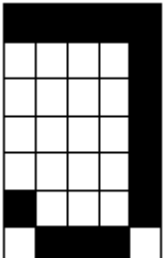
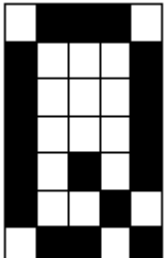
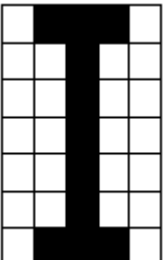
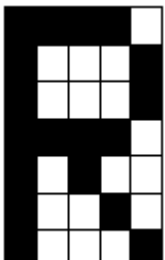
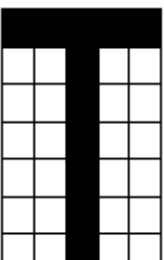
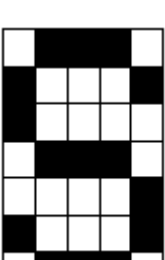
Scenariusz 6

Temat : 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 polskiego alfabetu.

1.Syntetyczny opis budowy i wykorzystania sieci i algorytmu uczenia.

	10001 10001 11001 10101 10011 10001 10001		10001 10001 10001 11111 10001 10001 10001
	11110 10001 10001 11110 10001 10001 11110		11110 10001 10001 10001 10001 10001 11110
	01110 10001 10000 10000 10000 10001 01110		10000 10000 10000 10000 10000 10000 11111
	10001 10010 10100 11000 10100 10010 10001		10001 11011 10101 10001 10001 10001 10001
	11111 10000 10000 11110 10000 10000 11111		01110 10001 10001 11111 10001 10001 10001

	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		0 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 0 1 1 1 0
	1 1 1 1 1 1 0 0 0 1 1 0 0 0 0 1 0 1 1 1 1 0 0 0 1 1 0 0 0 1 0 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 0 1 0 0 0 0 1 0 0 0 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		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 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		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 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 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

Nauka sieci polega na podziale danych na grupy i przyporządkowanie każdej danego elementu wyjścia. Dane należące do jednej grupy są do siebie podobne, zaś występują różnice między danymi należącymi do różnych grup.

OPIS SIECI :

W przypadku podanej sieci występuje metoda uczenia sieci samoorganizującej, którym jest uczenie rywalizujące. Neurony w danej sieci uczą się rozpoznawania danych, na których bazują a następnie zbliża się do terenu, w którym dane te są najmocniej osadzone. Najważniejszą zasadą w podanej sieci

jest fakt, że z pośród wszystkich neuronów wybierany jest ten, który znajduje się najbliżej centrum terenu. Neuron, którego wartość jest największa zostaje zwycięzcą, dzięki czemu na wyjściu jego wartość wynosi 1. Natomiast w przypadku pozostałych neuronów ich wartość na wyjściu wynosi 0. Reguła ta jest podobna do reguły WTA jednak występują pewne różnice. W przypadku reguły WTM występuje promień, który pozwala na aktualizację wag. Aktualizacja dotyczy neuronów których wyjście wynosi 0, czyli te które nie zwyciężyły. Wartość promienia maleje przy każdej iteracji konsekwencją czego coraz mniej neuronów ma możliwość zmiany wagi. Na koniec zmiana wagi dotyczy tylko jednego neuronu, neuronu zwycięskiego.

Schemat uczenia sieci :

Krok 1: Na początku wszystkie dane podlegają normalizacji.

Krok 2: Wybór współczynnika uczenia η ($0 < \eta < 1$)

Krok 3: Losowanie początkowych wartości wag z zakresu od 0 do 1

Krok 4: Dla każdego neuronu liczona jest suma ilorazów wag oraz sygnałów wejściowych

Krok 5: Dla neuronu z najwyższym wynikiem aktualizacja wag za pomocą wzoru :

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

gdzie:

$\theta(t)$ – funkcja sąsiedztwa (wg Gaussa), obliczana ze wzoru:

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

gdzie:

R – promień sąsiedztwa

d – jest to odległość pomiędzy zwycięskim neuronem oraz każdym dowolnym innym neuronem

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

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

gdzie:

i – wektor wejściowy

w – waga neuronu

t – obecna iteracja

λ – stała czasowa

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

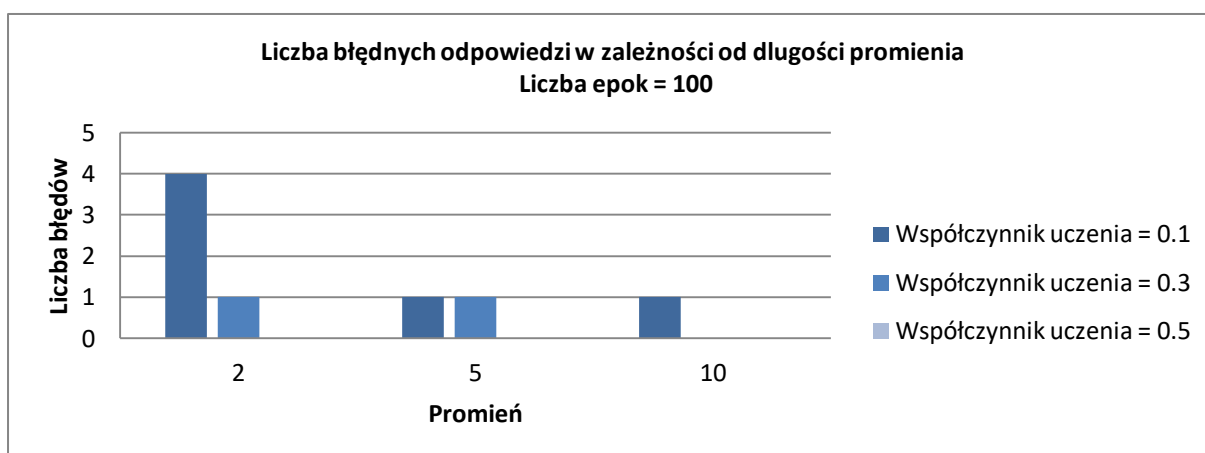
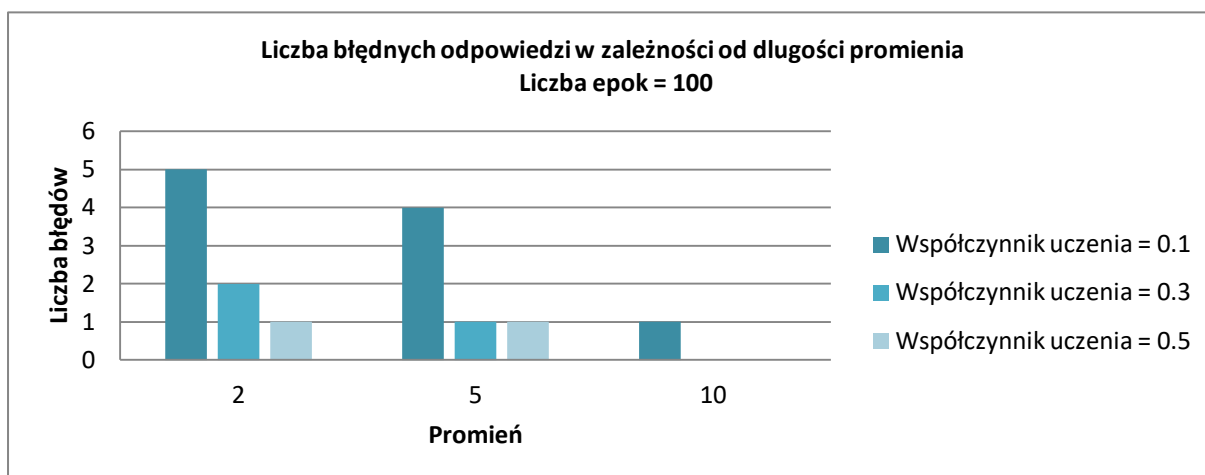
gdzie:

x – liczba iteracji

Krok 6: Następnie normalizacja wartości nowego wektora wag

Krok 7: Ustawienie wartości wyjściowej dla zwycięskiego wektora na 1 (reszta 0) i pobranie kolejnego wektora uczącego.

2.Zestawienie wyników i analiza programu :



Powyższe dwa wykresy zależności liczby błędów od długości promienia informują nas, że najmniejsza liczba błędów przypada na współczynnik 0.5, natomiast dla współczynnika 0.1 liczba błędów jest największa. Dla współczynnika wynoszącego 0.3 występuje niewielka liczba błędów. Warto także zauważyć, że wraz ze wzrostem promienia zmniejsza się liczba błędów.

TEST DLA 12 LITER :

Współczynnik uczenia = 0.5 (promień 2)		Współczynnik uczenia = 0.3 (promień 2)		Współczynnik uczenia = 0.1 (promień 2)	
Grupa 1	A	Grupa 1	A, H	Grupa 1	A, B, D, H
Grupa 2	B	Grupa 2	B	Grupa 2	C, E, F
Grupa 3	C	Grupa 3	C, D, L	Grupa 3	I
Grupa 4	D, G	Grupa 4	E, F	Grupa 4	G, J, K
Grupa 5	E, F	Grupa 5	G	Grupa 5	L
Grupa 6	I	Grupa 6	I, J		
Grupa 7	J	Grupa 7	K		
Grupa 8	K				
Grupa 9	L, H				

Współczynnik uczenia = 0.5 (promień 5)		Współczynnik uczenia = 0.3 (promień 5)		Współczynnik uczenia = 0.1 (promień 5)	
Grupa 1	A	Grupa 1	A	Grupa 1	A
Grupa 2	B	Grupa 2	B	Grupa 2	B, L
Grupa 3	C	Grupa 3	C, D	Grupa 3	C, D, E
Grupa 4	D	Grupa 4	E, F	Grupa 4	F
Grupa 5	E, F	Grupa 5	G	Grupa 5	G
Grupa 6	G	Grupa 6	H	Grupa 6	H
Grupa 7	H	Grupa 7	I, J	Grupa 7	I, J
Grupa 8	I, J	Grupa 8	K	Grupa 8	K
Grupa 9	K	Grupa 9	L		
Grupa 10	L				

Współczynnik uczenia = 0.5 (promień 10)		Współczynnik uczenia = 0.3 (promień 10)		Współczynnik uczenia = 0.1 (promień 10)	
Grupa 1	A	Grupa 1	A	Grupa 1	A
Grupa 2	B	Grupa 2	B	Grupa 2	B
Grupa 3	C	Grupa 3	C	Grupa 3	C
Grupa 4	D	Grupa 4	D	Grupa 4	D
Grupa 5	E	Grupa 5	E	Grupa 5	E
Grupa 6	F	Grupa 6	F	Grupa 6	F
Grupa 7	G	Grupa 7	G	Grupa 7	G
Grupa 8	H	Grupa 8	H	Grupa 8	H
Grupa 9	I	Grupa 9	I	Grupa 9	I
Grupa 10	J	Grupa 10	J	Grupa 10	J
Grupa 11	K	Grupa 11	K	Grupa 11	K
Grupa 12	L	Grupa 12	L	Grupa 12	L

Powyższe tabele przedstawiają grupowanie wektorów uczących. Dla promienia wynoszącego 10 sieć jest w stanie nie zależnie od współczynnika uczenia pogrupować dane prawidłowo. Natomiast im mniejsza jest wartość promienia, tym większą rolę zaczyna odgrywać współczynnik uczenia. Dla promienia wynoszącego 5, wektory wraz ze wzrostem współczynnika uczenia, dane są lepiej grupowane, do większej ilości grup. W przypadku promienia wynoszącego 2 i przy współczynniku 0.1 sieć nie uczy się prawidłowo i grupuje duże ilości liter to jednej grupy.

3.Wnioski:

Sieć Kohonena to sieć samoorganizująca się. Neurony grupują się i każda z grup ma różne wartości dla poszczególnych cech. Dzięki temu sieć nie potrzebuje nauczyciela więc może uczyć się sama. Jak przedstawiają wykresy powyżej, sieć uczy się różnie w zależności od współczynnika uczenia. Wraz ze wzrostem wartości współczynnika sieć uczy się szybciej jednak im wyższy jest ten współczynnik tym skuteczność nauki jest słabsza. Kiedy porównamy reguły WTA z WTM zauważymy, że zaletą na korzyść WTM jest fakt, iż jest lepiej uporządkowana ze względu na większą zbieżność algorytmu. Natomiast zaletą WTA w odróżnieniu od WTM jest to, że zajmuje mniej miejsca w pamięci komputera oraz krótszy czas działania. Spowodowane to jest tym, że w przypadku WTM aktualizacja wag dotyczy nie tylko zwycięskiego neuronu ale i pozostałych tylko w otoczeniu zwycięskiego neuronu.

4.Listing programu:

Layer.h

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

class Layer {
public:
    vector<Neuron> neuron;
    vector<double> scalarProducts; //odleglosci euklidesowych
    int liczba_neuronow;
    double promien; //promien wyznaczajacy obszar od zwycieskiego neuronu
    double czas;
    int zwycieski_neuron; //indeks
    void zmiana_wag(double obecnaIteracja, bool testing); //przy aktualnej iteracji
    void minimum_odleglosc_euklidesowa();
    void getOdleglosc_euklidesowa(); //zwraca odleglosci euklidesowe

    Layer(int liczba_neuronow, int numberOfInputs, double wspolczynnik_uczenia,
double iterationsNumber);
};
```

Layer.cpp

```
#include "Layer.h"

Layer::Layer(int liczba_neuronow, int liczba_wejsc, double wspolczynnik_uczenia,
double iterationsNumber) {
    this->liczba_neuronow = liczba_neuronow;
    neuron.resize(liczba_neuronow);
    this->czas = iterationsNumber / this->promien;
    this->promien = 5;
    for (int i = 0; i < liczba_neuronow; i++)
        neuron[i].Neuron(liczba_wejsc, wspolczynnik_uczenia);
}

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

```

    }
}

void Layer::zmiana_wag(double obecna_iteracja, bool uczing) {
    minimum_odleglosc_euklidesowa();
    getOdleglosc_euklidesowa();
    neuron[z zwycieski_neuron].funkcja_aktywacji();

    if (uczing) {
        neuron[z zwycieski_neuron].oblicz_odleglosc_od_zwycieskiego(promien,
obecna_iteracja, czas); // szukanie neuronow w otoczeniu wygranego neuronu
        int promien = neuron[z zwycieski_neuron].odleglosc_od_zwycieskiego;
        int leftBorderNeuronIndex = 0;
        int rightBorderNeuronIndex = 0;

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

        if (zwycieski_neuron + promien >= liczba_neuronow)
            rightBorderNeuronIndex = liczba_neuronow - 1;
        else
            rightBorderNeuronIndex = zwycieski_neuron + promien;

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

        for (int i = leftBorderNeuronIndex; i < rightBorderNeuronIndex; i++) {
            neuron[i].odl_euklides = (i < zwycieski_neuron) ?
(zwycieski_neuron - i) : (i - zwycieski_neuron); //zmiana wag dla neuronow z otoczenia
            neuron[i].odleglosc_od_zwycieskiego =
neuron[z zwycieski_neuron].odleglosc_od_zwycieskiego;
            neuron[i].nowa_waga();
        }
    }
}

void Layer::getOdleglosc_euklidesowa() {
    scalarProducts.clear();
    for (int i = 0; i < liczba_neuronow; i++)
        scalarProducts.push_back(neuron[i].oblicz_odleglosc_skalar());
}

```

Neuron.h

```

class Neuron {
public:
    vector<double> wejscia;
    vector<double> wagi;
    double wartosc_wyjscie;
    double odl_euklides;
    double odleglosc_od_zwycieskiego;
    double wspolczynnik_uczenia;
    double wartosc_sasiedztwa; //wartosc funkcji sasiedztwa (Gaussian neighborhood
function)
    double sumowanie_wejsc;

    void normalizacja_wag_zaktualizowanych();
    double losowanie_Wag();
}

```

```

        void oblicz_odleglosc_od_zwycieskiego(); //oblicza wartosc funkcji sasiedztwa
        (Gaussian neighborhood function)
        void nowa_waga();
        void stworz_wejscie(int liczba_wejsc); //ustawienie wejsc na 0 i skorzystanie z
        metody : losowanie_Wag()
        void funkcja_aktywacji(); //funkcja sigmoidalna
        double oblicz_odleglosc_skalar();
        void oblicz_odleglosc_od_zwycieskiego(double promien, double obecna_iteracja,
        double czas);

        Neuron(); // konstruktory
        Neuron(int liczba_wejsc, double wspolczynnik_uczenia);

        int getRozmiar_wejsc() { return wejscia.size(); } //zwraca rozmiar wejscia
        int getRozmiar_wag() { return wagi.size(); } // podaje wage
};

```

Neuron.cpp

```

Neuron::Neuron() {
    this->wejscia.resize(0);
    this->wagi.resize(0);
    this->sumowanie_wejsc = 0.0;
    this->wartosc_wyjście = 0.0;
    this->wspolczynnik_uczenia = 0.0;
}

Neuron::Neuron(int liczba_wejsc, double wspolczynnik_uczenia) {
    stworz_wejscie(liczba_wejsc);
    normalizacja_wag_zaktualizowanych();
    this->wspolczynnik_uczenia = wspolczynnik_uczenia;
    this->sumowanie_wejsc = 0.0;
    this->wartosc_wyjście = 0.0;
}

void Neuron::stworz_wejscie(int liczba_wejsc) { //stworzenie poczatkowych
wejsc(ustawienie wejsc na 0, wykorzystanie metody losowanie_Wag())
    for (int i = 0; i < liczba_wejsc; i++) {
        wejscia.push_back(0);
        wagi.push_back(losowanie_Wag());
    }
}

double Neuron::oblicz_odleglosc_skalar() {
    sumowanie_wejsc = 0.0;
    for (int i = 0; i < getRozmiar_wejsc(); i++)
        sumowanie_wejsc += pow(wejscia[i] - wagi[i], 2);
    sumowanie_wejsc = sqrt(sumowanie_wejsc);

    return sumowanie_wejsc;
}

void Neuron::funkcja_aktywacji() {
    double beta = 1.0;
    this->wartosc_wyjście = (1.0 / (1.0 + (exp(-beta * sumowanie_wejsc))));
}

void Neuron::nowa_waga() {
    for (int i = 0; i < getRozmiar_wag(); i++)
        this->wagi[i] += this->wspolczynnik_uczenia*this-
>wartosc_sasiedztwa*(this->wejscia[i] - this->wagi[i]);
}

```



```

        normalizacja_wag_zaktualizowanych();
    }

void Neuron::normalizacja_wag_zaktualizowanych() {
    double vectorodl_euklides = 0.0;

    for (int i = 0; i < getRozmiar_wag(); i++)
        vectorodl_euklides += pow(wagi[i], 2);

    vectorodl_euklides = sqrt(vectorodl_euklides);

    for (int i = 0; i < getRozmiar_wag(); i++)
        wagi[i] /= vectorodl_euklides;
}

void Neuron::oblicz_odleglosc_od_zwycieskiego(double promien, double currentIteraton,
double czasConstant) {
    this->odleglosc_od_zwycieskiego = promien * exp(-currentIteraton /
czasConstant);
}

void Neuron::oblicz_odleglosc_od_zwycieskiego() {
    this->wartosc_sasiedztwa = exp(-pow(this->odl_euklides, 2) / (2 * pow(this-
>odleglosc_od_zwycieskiego, 2)));
}

double Neuron::losowanie_Wag() {
    double max = 1.0;
    double min = 0.0;
    double weight = ((double(rand()) / double(RAND_MAX))*(max - min)) + min;
    return weight;
}
Source.cpp

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

    vector<vector<double>> dane_testujace;
    vector<vector<double>> dane_uczace;
    int liczba_neuronow = 20;
    int liczba_wejsc = 35;
    double wspolczynnik_uczenia = 0.05;
    int epoka = 50;

    Layer siec_Kohonena(liczba_neuronow, liczba_wejsc, wspolczynnik_uczenia, epoka);
    wczytaj_dane_uczace(dane_uczace, liczba_wejsc);
    wczytaj_dane_testujace(dane_testujace, liczba_wejsc);

    OUTPUT_FILE_uczING.open("output_uczing_data.txt", ios::out);

    for (int epokaNumber = 1, i = 0; i < epoka; i++, epokaNumber++) {
        ucz(siec_Kohonena, dane_uczace); // rozpoczynanie procesu
uczenia

        OUTPUT_FILE_uczING << "epoka: " << epokaNumber << endl;
        cout << "epoka: " << epokaNumber << endl;
    }
    OUTPUT_FILE_uczING.close();

    OUTPUT_FILE_strumien_danych_testujacych.open("output_strumien_danych_testujacych
.txt", ios::out);

```

```

        OUTPUT_FILE_TESTING_NEURON.open("output_testing_neuron.txt",
ios::out);

        test(siec_Kohonena, dane_testujace);

        OUTPUT_FILE_uczING.close();
        OUTPUT_FILE_strumien_danych_testujacych.close();
        system("pause");

        return 0;
    }

    void dane_wejscowe(Neuron& neuron, vector<vector<double>> inputData, int
liczba_wejsc, int row)
    {
        for (int i = 0; i < liczba_wejsc; i++)
            neuron.wejscia[i] = inputData[row][i];
    }

    void ucz(Layer& layer, vector<vector<double>> inputData)
    {
        static int obecna_iteracja = 0;
        for (int rowOfData = 0; rowOfData < inputData.size(); rowOfData++) {
            for (int i = 0; i < layer.liczba_neuronow; i++) {
                dane_wejscowe(layer.neuron[i], inputData,
layer.neuron[i].getRozmiar_wejsc(), rowOfData);
                layer.neuron[i].oblicz_odleglosc_skalar();
            }

            layer.zmiana_wag(obecna_iteracja, true);

            OUTPUT_FILE_uczING << layer.zwycieski_neuron << endl;
            cout << "Winner: " << layer.zwycieski_neuron << endl;
            obecna_iteracja++;
        }
    }

    void test(Layer& layer, vector<vector<double>> inputData) {
        for (int rowOfData = 0; rowOfData < inputData.size(); rowOfData++) {
            for (int i = 0; i < layer.liczba_neuronow; i++) {
                dane_wejscowe(layer.neuron[i], inputData,
layer.neuron[i].getRozmiar_wejsc(), rowOfData);
                layer.neuron[i].oblicz_odleglosc_skalar();
            }
            char letter = 'A';
            layer.zmiana_wag(0, false);
            OUTPUT_FILE_strumien_danych_testujacych <<
layer.neuron[layer.zwycieski_neuron].getRozmiar_wejsc() << endl;
            OUTPUT_FILE_TESTING_NEURON << (char)(letter + rowOfData) << " " <<
layer.zwycieski_neuron << endl;
            cout << (char)(letter + rowOfData) << " " << layer.zwycieski_neuron <<
endl;
        }
    }

    void wczytaj_dane_testujace(vector<vector<double>> &dane_testujace, int liczba_wejsc)
    {
        strumien_danych_testujacych.open("datatest.txt", ios::in);
        vector<double> row;
        double odl_euklides = 0;

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

```

```

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

        for (int i = 0; i < liczba_wejsc; i++) //znormalizowanie danych uczacych
            odl_euklides += pow(row[i], 2);
        odl_euklides = sqrt(odl_euklides);

        for (int i = 0; i < liczba_wejsc; i++)
            row[i] /= odl_euklides;
        dane_testujace.push_back(row);
    }
    strumien_danych_testujacych.close();
}

void wczytaj_dane_uczace(vector<vector<double>> &inputData, int liczba_wejsc) {
    strumie_danych_uczacych.open("data.txt", ios::in);
    double odl_euklides = 0;
    vector<double> row;
    do {
        row.clear();

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

        for (int i = 0; i < liczba_wejsc; i++)//znormalizowanie danych uczacych
            odl_euklides += pow(row[i], 2);

        odl_euklides = sqrt(odl_euklides);

        for (int i = 0; i < liczba_wejsc; i++)
            row[i] /= odl_euklides;

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

    strumie_danych_uczacych.close();
}

```