

## Scenariusz 6

Białek Tomasz, gr. 1

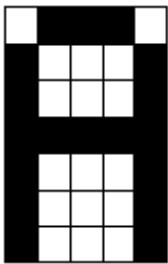
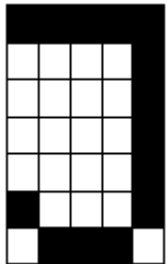
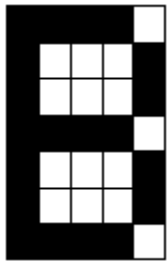
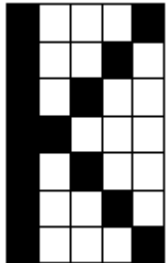
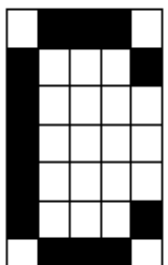
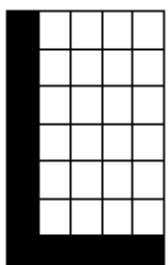
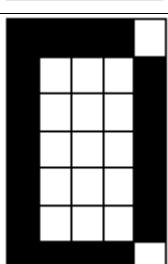
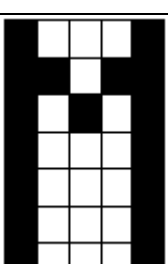
**Temat ćwiczenia: Budowa i działanie sieci Kohonena dla WTM.**

### 1. Cel ćwiczenia

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

### 2. Opis budowy sieci i algorytmów uczenia.

Celem zbudowanej sieci jest podział danych uczących na określone grupy i przyporządkowanie danej grupie danego elementu wyjściowego. Podział na grupy polega na tym, żeby elementy w danej grupie były jak najbardziej podobne do siebie jednocześnie będąc zupełnie inne od elementów z innych grup.

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

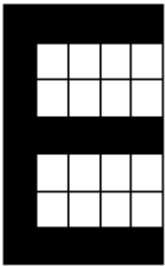
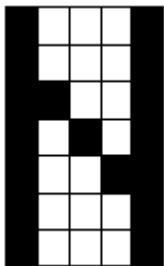
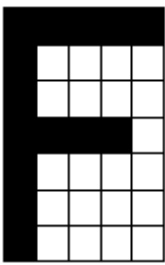
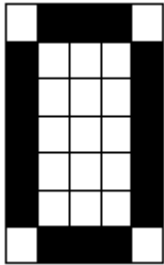
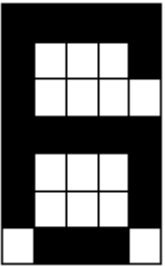
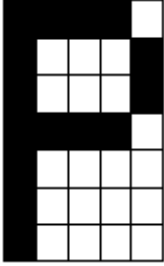
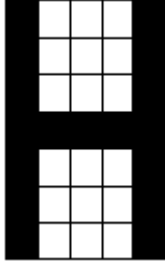
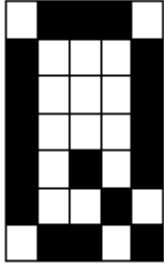
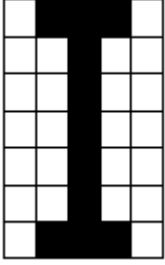
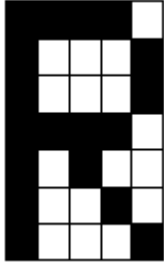
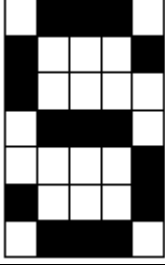
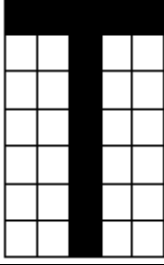
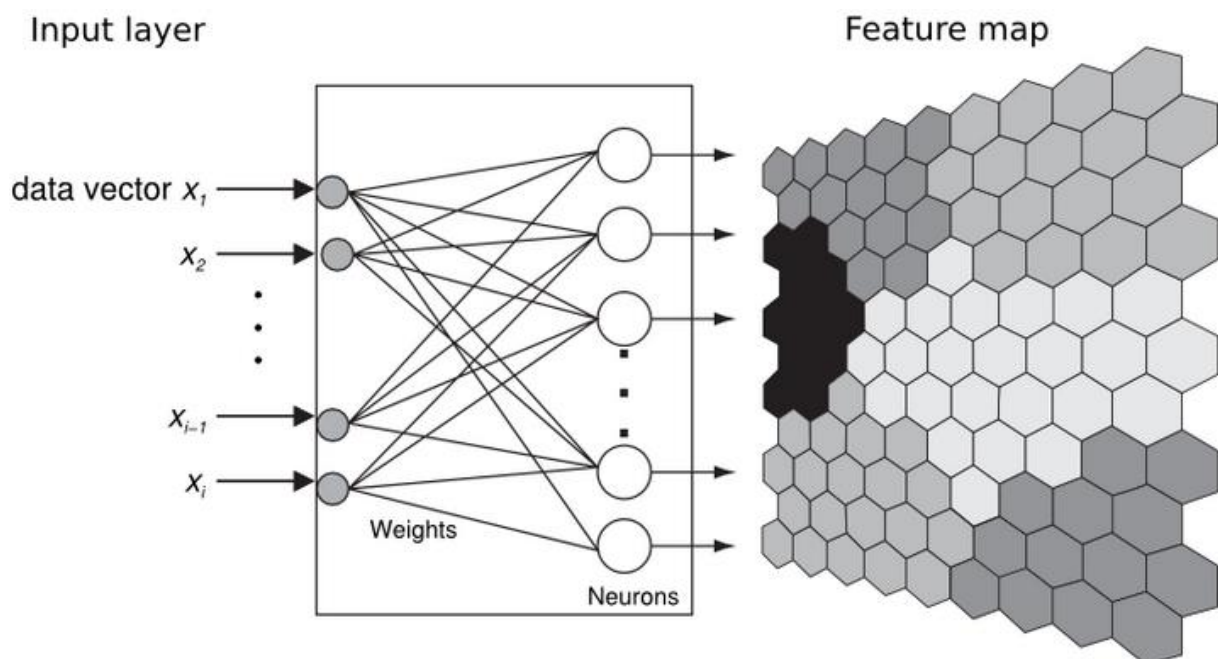
	1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1		1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 0 1 0 1 1 0 0 1 1 1 0 0 0 1 1 0 0 0 1
	1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0		0 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 0 1 1 1 0
	1 1 1 1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

Tabela 1. Litery i ich reprezentaja

### 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. Różnica pomiędzy WTM, a WTA polega na tym, że podczas procesu uczenia wykorzystywany jest promień, który pozwala na zaktualizowanie wag neuronów, które nie zwyciężyły. Jednakże z każdą iteracją (krokiem czasowym) promień ten zmniejsza się, żeby na samym końcu mógł zaktualizować swoje wartości tylko i wyłącznie zwycięski neuron.



Rys. 1 Sieć samoorganizująca

Proces uczenia przebiega według następującego schematu:

1. Normalizacja wszystkich danych
2. Wybór współczynnika uczenia  $\eta$  z przedziału  $(0; 1)$
3. Wybór początkowych wartości wag z przedziału  $(0; 1)$
4. Dla danego zbioru uczącego obliczamy odpowiedź sieci – dla każdego pojedynczego neuronu obliczana jest suma ilorazów sygnałów wejściowych oraz wag
5. Wybierany jest neuron, którego odległość euklidesowa. Tylko dla tego neuronu następuje aktualizacja wag. Wzór na zaktualizowanie wagi jest następujący:

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

gdzie:

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

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

gdzie:

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

R – promień sąsiedztwa

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

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

gdzie:

i – wektor wejściowy

w – waga neuronu

t – obecna iteracja

$\lambda$  – stała czasowa

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

gdzie:

x – liczba iteracji

$R_0$  – początkowy promień sąsiedztwa

6. Znormalizowanie wartości nowego wektora wag

7. Zwycięski neuron daje odpowiedź na swoim wyjściu równą 1, a pozostałe 0.

8. Wczytanie kolejnego wektora uczącego.

### 3. Zestawienie otrzymanych wyników

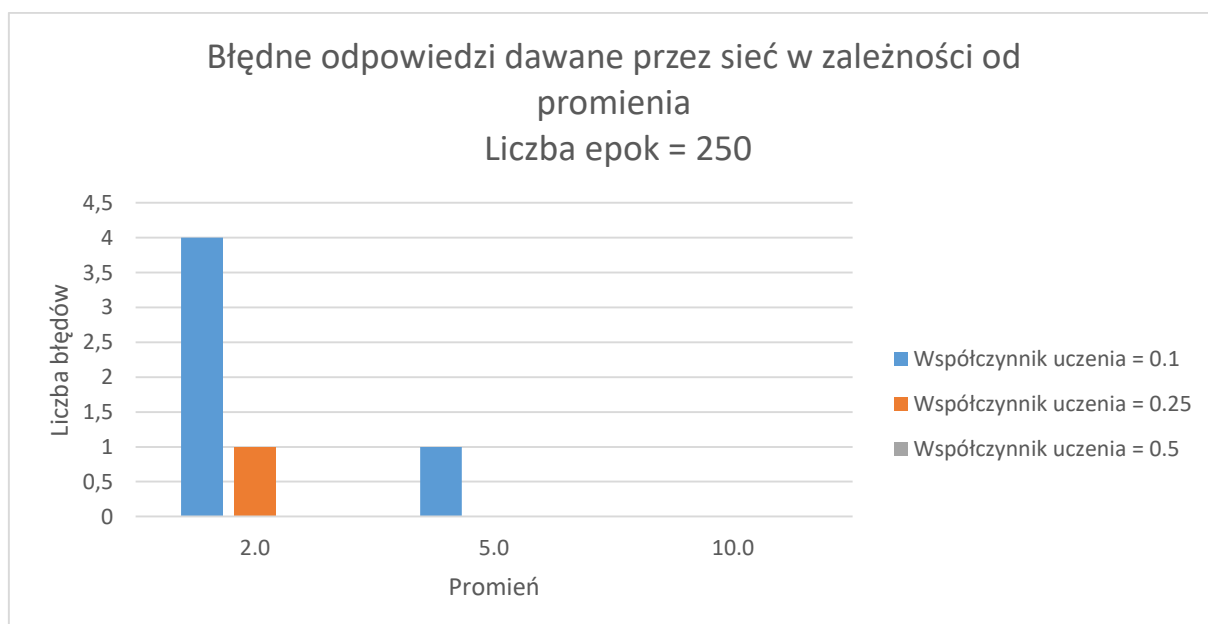
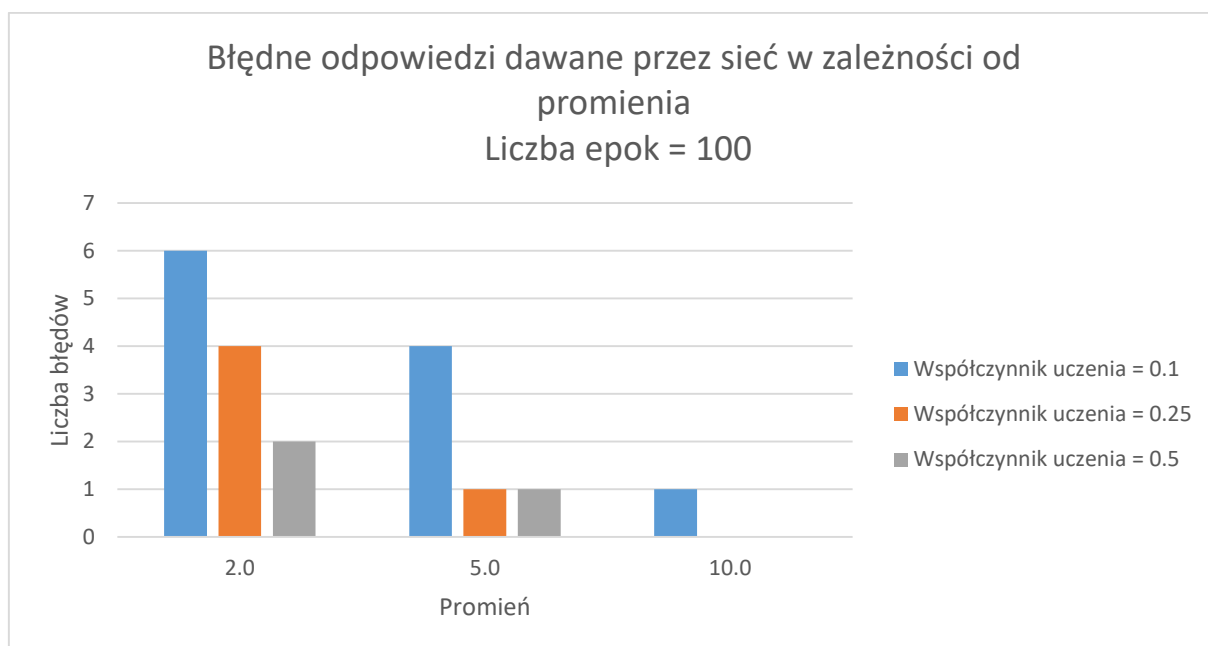
Do testowania użyto 12 zmodyfikowanych w niewielkim stopniu liter. Sieć była uczona aż do osiągnięcia 100 epoki.

Współczynnik uczenia = 0.5 (promień 2)			
Grupa 1	A		
Grupa 2	B	Współczynnik uczenia = 0.25 (promień 2)	
Grupa 3	C	Grupa 1	A, H
Grupa 4	D, G	Grupa 2	B
Grupa 5	E, F	Grupa 3	C, D, L
Grupa 6	I	Grupa 4	E, F
Grupa 7	J	Grupa 5	G
Grupa 8	K	Grupa 6	I
Grupa 9	L	Grupa 7	J
Grupa 10	H	Grupa 8	K
Współczynnik uczenia = 0.1 (promień 2)			
		Grupa 1	A, B, D, H
		Grupa 2	C, E, F
		Grupa 3	I
		Grupa 4	G, J
		Grupa 5	K
		Grupa 6	L

Współczynnik uczenia = 0.5 (promień 5)		Współczynnik uczenia = 0.25 (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	K	Grupa 7	I	Grupa 7	I, J
Grupa 8	I	Grupa 8	J	Grupa 8	K
Grupa 9	J	Grupa 9	K		
Grupa 10	H	Grupa 10	L		
Grupa 11	L				

Współczynnik uczenia = 0.5 (promień 10)		Współczynnik uczenia = 0.25 (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, F
Grupa 6	F	Grupa 6	F	Grupa 6	G
Grupa 7	G	Grupa 7	G	Grupa 7	H
Grupa 8	H	Grupa 8	H	Grupa 8	I
Grupa 9	I	Grupa 9	I	Grupa 9	J
Grupa 10	J	Grupa 10	J	Grupa 10	K
Grupa 11	K	Grupa 11	K	Grupa 11	L
Grupa 12	L	Grupa 12	L		

Z powyższego zestawienia można odczytać, że dla sieci Kohonena ważna jest wartość współczynnika uczenia oraz promienia. Dla małych współczynników uczenia sieć nie zdążyła się nauczyć (pogrupować) wektorów uczących. Dopiero dla w miarę dużego współczynnika uczenia = 0.05 sieć była w stanie dobrze pogrupować dane. Oprócz współczynnika uczenia ważną rolę pełni promień sąsiedztwa. Im jest on większy, tym dane są przydzielane do większej ilości oddzielnych grup. Spowodowane to może być sytuacją, że, w przeciwieństwie do WTA, szansę na aktualizację wag mają nie tylko neurony zwycięskie, lecz także te będące w ich bezpośrednim sąsiedztwie.



Z powyższych wykresów można odczytać, że największą ilością błędów cechuje się sieć ze współczynnikiem uczenia równym 0.01. Jest to za mało, żeby sieć zdążyła pogrupować wektory uczące w osobne zbiory, co skutkuje przypisywaniem do jednej grupy więcej niż jednej litery. Wraz ze wzrostem współczynnika uczenia powstaje więcej niezależnych grup. Taka sama sytuacja tyczy się promienia. Im jest on większy, tym liczba powstałych błędów jest mniejsza.

#### 4. Podsumowanie

Sieć Kohonena cechuje umiejętność podziału danych, które posiadają różne wartości dla poszczególnych cech, ponieważ jest to sieć samoorganizująca. Powoduje to, że odpowiedni podział na grupy może być wykonywane bez podawania wartości oczekiwanych (uczenie bez nauczyciela). Cały proces uczenia (jego efektywność) zależy od współczynnika uczenia. Im większy jest ten współczynnik, tym sieć uczy się szybciej. Jednakże wzrost efektywności nie jest wprost

proporcjonalny do współczynnika uczenia. Dla wzrostu (efektywności) przy małych współczynnikach uczenia następuje większa różnica w ilości potrzebnych epok aniżeli dla wzrostu przy dużych współczynnikach uczenia – występuje stabilizacja uczenia. W odróżnieniu od metody WTA, metoda WTM pozwala na wiele sposobów implementacji. Można stosować w niej różne metryki (np. euklidesową lub miejską). Ponadto, wykorzystując metodę WTM otrzymamy lepsze rezultaty, ponieważ sieć jest bardziej uporządkowana (większa zbieżność algorytmu). Wadą WTM w porównaniu do WTA jest większy narzut – potrzeba więcej czasu i miejsca w pamięci, ponieważ aktualizowane są nie tylko zwycięskie neurony, ale także te będące w ich sąsiedztwie. Sieć dla odpowiednich współczynników uczenia pozwala uzyskać 100% poprawnych odpowiedzi, co powoduje, że sieć Kohonena sprawdza się przy dużej ilości różniących się pomiędzy sobą danych.

## 5. Kod programu

„Source.cpp”

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

#include "Layer.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(Layer& layer, vector<vector<double>> inputData);
//testowanie sieci
void test(Layer& 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 = 0.05;
    int epoch = 50;
    //stworzenie sieci Kohonena
    Layer 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;
    }
    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(Layer& 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(Layer& 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();
}

```

„Layer.h”

```

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

class Layer {
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
    Layer(int numberOfNeurons, int numberOfInputs, double learningRate, double
    iterationsNumber);
};

```

„Layer.cpp”

```

#include "Layer.h"

//konstruktor
Layer::Layer(int numberOfNeurons, int numberOfInputs, double learningRate, double
iterationsNumber) {
    this->numberOfNeurons = numberOfNeurons;
    neurons.resize(numberOfNeurons);
    //this->radius = (double)numberOfNeurons;
    this->radius = 5;
    this->time = iterationsNumber / this->radius;

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

```

```

        neurons[i].Neuron::Neuron(numberOfInputs, learningRate);
    }

    //zmiana wag
    void Layer::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 sie w promieniu
                neurons[i].length = (i < winnerIndex) ? (winnerIndex - i) : (i -
winnerIndex);
                neurons[i].neighbors = neurons[winnerIndex].neighbors;
                neurons[i].calculateNewWeights();
            }
        }
    }

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

    //szuka najmniejszej odleglosci euklidesowej
    void Layer::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];
            }
        }
    }
}

```

„Neuron.h”

```

#pragma once
#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 wejść 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 poczatkowych wejsc(ustawienie wejsc 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 odleglosci 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 obliczajaca wyjscie
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 odleglosci do sasiadow
void Neuron::designateNeighbors(double radius, double currentIteraton, double
timeConstant) {
    this->neighbors = radius * exp(-currentIteraton / timeConstant);
}

//oblicza wartosc funkcji sasiedztwa (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 poczatkowych wag dla wszystkich wejsc - 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;  
}
```