

## Scenariusz 6

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

**Wykonała:** Burnat Magdalena, IS, gr. 3, 270652

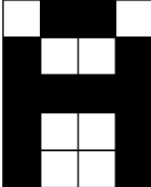
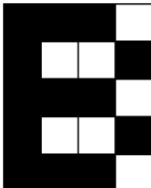
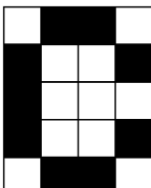
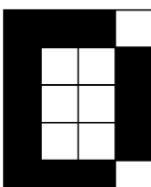
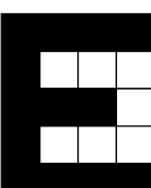
### 1. Cel ćwiczenia

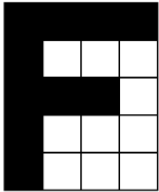
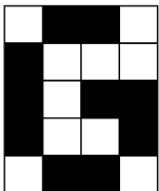
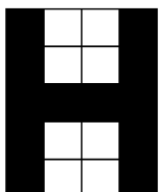
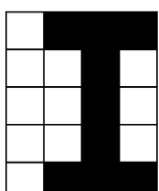
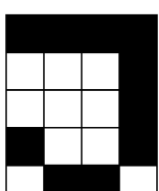
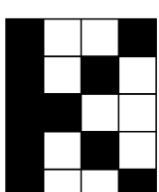
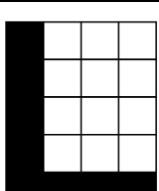
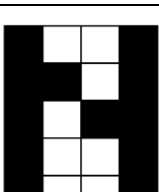
Celem ćwiczenia jest poznanie budowy i działania sieci Kohonena przy wykorzystaniu reguły WTM do odwzorowywania istotnych cech liter alfabetu.

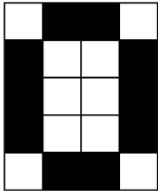
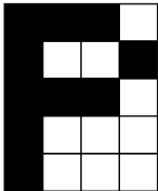
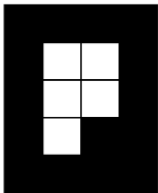
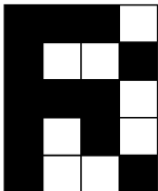
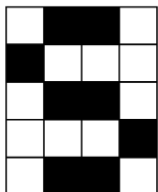
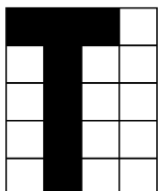
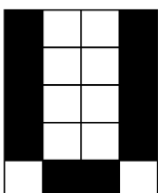
### 2. Opis budowy sieci i algorytmów uczenia.

Celem budowanej sieci jest podział wzorców uczących na klasy obrazów zbliżonych do siebie i przyporządkowanie każdej klasie osobnego elementu wyjściowego. Podział na grupy odbywa się w ten sposób, by elementy w tej samej grupie były do siebie podobne a jednocześnie jak najbardziej odmienne od elementów z pozostałych grup.

Tab. 1. Wykorzystane litery.

	Wektor wejściowy: 0 1 1 0 1 0 0 1 1 1 1 1 1 0 0 1 1 0 0 1
	Wektor wejściowy: 1 1 1 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1 1 0
	Wektor wejściowy: 0 1 1 0 1 0 0 1 1 0 0 0 1 0 0 1 0 1 1 0
	Wektor wejściowy: 1 1 1 0 1 0 0 1 1 0 0 1 1 0 0 1 1 1 1 0
	Wektor wejściowy: 1 1 1 1 1 0 0 0 1 1 1 0 1 0 0 0 1 1 1 1

	<p>Wektor wejściowy:</p> <p>1 1 1 1 1 0 0 0 1 1 1 0 1 0 0 0 1 0 0 0</p>
	<p>Wektor wejściowy:</p> <p>0 1 1 0 1 0 0 0 1 0 1 1 1 0 0 1 0 1 1 0</p>
	<p>Wektor wejściowy:</p> <p>1 0 0 1 1 0 0 1 1 1 1 1 1 0 0 1 1 0 0 1</p>
	<p>Wektor wejściowy:</p> <p>0 1 1 1 0 0 1 0 0 0 1 0 0 0 1 0 0 1 1 1</p>
	<p>Wektor wejściowy:</p> <p>1 1 1 1 0 0 0 1 0 0 0 1 1 0 0 1 0 1 1 0</p>
	<p>Wektor wejściowy:</p> <p>1 0 0 1 1 0 1 0 1 1 0 0 1 0 1 0 1 0 0 1</p>
	<p>Wektor wejściowy:</p> <p>1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1</p>
	<p>Wektor wejściowy:</p> <p>1 0 0 1 1 1 0 1 1 0 1 1 1 0 0 1 1 0 0 1</p>

	Wektor wejściowy: 0 1 1 0 1 0 0 1 1 0 0 1 1 0 0 1 0 1 1 0
	Wektor wejściowy: 1 1 1 0 1 0 0 1 1 1 1 0 1 0 0 0 1 0 0 0
	Wektor wejściowy: 1 1 1 1 1 0 0 1 1 0 0 1 1 0 1 1 1 1 1 1
	Wektor wejściowy: 1 1 1 0 1 0 0 1 1 1 1 0 1 0 1 0 1 0 0 1
	Wektor wejściowy: 0 1 1 0 1 0 0 0 0 1 1 0 0 0 0 1 0 1 1 0
	Wektor wejściowy: 1 1 1 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0
	Wektor wejściowy: 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 0 1 1 0

Aby możliwe była klasyfikacja liter na poszczególne grupy zostały odliczone odległości euklidesowe między poszczególnymi literami według wzoru:

$$d(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

Tab. 2. Otrzymane wartości odległości euklidesowej między poszczególnymi literami.

	A	B	C	D	E	F	G	H	I	J	K	L	N	O	P	Q	R	S	T	U
A	x	2,24	2,65	2,45	2,45	2,45	2,45	2,00	3,61	3,16	3,16	3,16	2,45	2,45	2,00	2,65	2,00	2,83	3,61	3,16
B	2,24	x	2,00	1,73	2,00	2,24	2,24	2,65	3,46	2,65	3,32	2,65	3,00	2,24	1,73	2,45	2,00	2,24	3,16	2,65
C	2,65	2,00	x	1,73	2,83	3,00	1,73	3,32	3,16	2,24	3,61	2,65	3,32	1,00	2,65	2,45	3,00	2,24	3,16	2,24
D	2,45	1,73	1,73	x	2,65	2,83	2,00	2,83	3,61	2,00	3,46	2,45	2,83	1,41	2,45	1,73	2,83	2,83	3,32	2,00
E	2,45	2,00	2,83	2,65	x	1,73	2,65	2,65	2,83	3,00	2,65	2,24	3,00	3,00	2,24	2,45	2,24	2,65	3,16	3,00
F	2,45	2,24	3,00	2,83	1,73	x	2,83	2,45	3,32	3,16	2,45	2,83	2,83	3,16	1,41	3,00	2,00	2,83	3,00	3,16
G	2,45	2,24	1,73	2,00	2,65	2,83	x	3,16	3,00	2,45	3,74	2,83	3,16	1,41	2,83	2,65	3,16	2,00	3,32	2,45
H	2,00	2,65	3,32	2,83	2,65	2,45	3,16	x	3,87	3,16	2,45	2,83	1,41	3,16	2,45	2,65	2,65	3,46	3,87	2,45
I	3,61	3,46	3,16	3,61	2,83	3,32	3,00	3,87	x	3,00	3,32	3,32	3,87	3,32	3,61	3,16	3,32	2,65	3,16	3,61
J	3,16	2,65	2,24	2,00	3,00	3,16	2,45	3,16	3,00	x	3,74	3,16	3,16	2,00	3,16	2,24	3,46	2,83	3,00	2,00
K	3,16	3,32	3,61	3,46	2,65	2,45	3,74	2,45	3,32	3,74	x	2,45	2,83	3,74	2,83	3,00	2,45	3,74	3,61	3,16
L	3,16	2,65	2,65	2,45	2,24	2,83	2,83	2,83	3,32	3,16	2,45	x	2,83	2,83	2,83	2,65	2,83	3,16	3,32	2,45
N	2,45	3,00	3,32	2,83	3,00	2,83	3,16	1,41	3,87	3,16	2,83	2,83	x	3,16	2,83	2,65	2,83	3,74	3,87	2,45
O	2,45	2,24	1,00	1,41	3,00	3,16	1,41	3,16	3,32	2,00	3,74	2,83	3,16	x	2,83	2,24	3,16	2,45	3,32	2,00
P	2,00	1,73	2,65	2,45	2,24	1,41	2,83	2,45	3,61	3,16	2,83	2,83	2,83	2,83	x	3,00	1,41	2,83	3,00	3,16
Q	2,65	2,45	2,45	1,73	2,45	3,00	2,65	2,65	3,16	2,24	3,00	2,65	2,65	2,24	3	x	2,65	3,32	3,87	2,24
R	2,00	2,00	3,00	2,83	2,24	2,00	3,16	2,65	3,32	3,46	2,45	2,83	2,83	3,16	1,41	2,65	x	3,16	3,32	3,46
S	2,83	2,24	2,24	2,83	2,65	2,83	2,00	3,46	2,65	2,83	3,74	3,16	3,74	2,45	2,83	3,32	3,16	x	2,65	3,16
T	3,61	3,16	3,16	3,32	3,16	3,00	3,32	3,87	3,16	3,00	3,61	3,32	3,87	3,32	3,00	3,87	3,32	2,65	x	3,61
U	3,16	2,65	2,24	2,00	3,00	3,16	2,45	2,45	3,61	2,00	3,16	2,45	2,45	2,00	3,16	2,24	3,46	3,16	3,61	x

Proces uczenia odbywa się przy pomocy uczenia rywalizującego, które jest metodą uczenia sieci samoorganizujących. Podczas uczenia neurony uczą się rozpoznawać dane i zbliżają się odpowiednio do obszarów zajmowanych przez te dane. Po wejściu każdego wektora uczącego wybierany jest tylko jeden neuron, najbliższy prezentowanemu wzorcowi. Neurony rywalizują

między sobą i zwycięża ten neuron, którego wartość jest największa. Zwycięski neuron przyjmuje na wyjściu wartość 1, a pozostałe neurony 0. Jest to uczenie bez nauczyciela.

Neuron zwycięski ma prawo uaktualnić swoje wagi wg jednej z dwóch zasad: WTA(Winner Takes All), WTM(Winner Takes Most). W tym projekcie została wykorzystana pierwsza zasada.

Stworzona sieć to sieć jednowarstwowa ilość neuronów została wybrana na podstawie testów i sieć składa się z 20 neuronów.

Tab. 3 Samodzielny podział na grupy na podstawie odległości euklidesowej

Lp. grupy	Litery należące do grupy
1	A
2	B C D O
3	E F P R
4	G S
5	H N
6	I
7	J U
8	K
9	Q
10	T

Proces uczenia przebiega według następującego schematu w danej epoce uczenia:

1. Wybór  $\eta$  za zakresu 0 do 1
2. Normalizacja wektorów danych wejściowych
3. Wybór początkowych wartości wag, jako niewielkich liczb losowych z zakresu -1 do 1
4. Dla pojedynczego neuronu obliczana jest odległość euklidesowa sygnałów wejściowych wektora uczącego i wag
5. Wybierany jest neuron, dla którego obliczona odległość euklidesowa jest najmniejsza i zaktualizowanie jego wag według wzoru:

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

gdzie:

$\eta$  to współczynnik uczenia wybierany z zakresu od 0 do 1

$\theta(t)$  to funkcja sąsiedztwa obliczana według wzoru

$$\theta(t) = e^{\left(\frac{-d^2}{2\sigma^2(t)}\right)}$$

gdzie:

d – dystans od zwycięskiego neuronu obliczany ze wzoru:

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

$i$  – wektor wejściowy

$w$  – waga neuronu

$\sigma(t)$  – promień sąsiedztwa obliczany ze wzoru

$$\sigma(t) = \sigma_0 e^{\left(\frac{-t}{\lambda}\right)}$$

$t$  – obecna iteracja

$\lambda$  – stała czasowa obliczana ze wzoru

$$\lambda = \frac{\text{liczba iteracji}}{\text{promień mapy sieci}}$$

$\sigma_0$  – promień mapy sieci

6. Znormalizowanie wartości nowego wektora wag

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

8. Wczytanie kolejnego wektora uczącego

### 3. Otrzymane wyniki

Zestaw liter podzielono na 16 liter do uczenia sieci oraz 6 do testowania.

Tab. 4. Podział danych

Litery do uczenia	A C D F G H I J K L Q R T U
Litery do testowania	B E N O P S

Testowanie sieci odbywało się dla pięciu różnych współczynników uczenia, dla każdego z nich wykonano dwie próby procesu uczenia i późniejszego testowania. Liczba epok była ustalona ogólnie na wartość 200. Wynik testów zostały przedstawione w poniższych tabelach.

Tab. 5. Wyniki dla współczynnika uczenia równego 0,001\*

Współczynnik uczenia 0,001		
	Próba 1	Próba 2
Grupa 0	D	B C I S
Grupa 1	H J R	D F K P
Grupa 2	O	L N
Grupa 3	N U	J Q
Grupa 4	F P	T
Grupa 5	G I S	G O

Tab. 6. Wyniki dla współczynnika uczenia równego 0,01\*

Współczynnik uczenia 0,01		
	Próba 1	Próba 2
Grupa 0	K	K
Grupa 1	F P R	A H N P R
Grupa 2	E	B
Grupa 3	B	L
Grupa 4	A	U
Grupa 5	C	D O

Grupa 6	Q K	A <b>E</b>
Grupa 7	A C	H R
Grupa 8	<b>B</b> <b>E</b> L	U
Grupa 9	T	

Grupa 6	I	C G Q
Grupa 7	H <b>N</b>	J
Grupa 8	D J <b>O</b> U	T
Grupa 9	G	<b>E</b>
Grupa 10	L Q	F
Grupa 11	<b>S</b> T	I
Grupa 12		<b>S</b>

Tab. 7. Wyniki dla współczynnika uczenia  
równego 0,1\*

Współczynnik uczenia 0,1		
	Próba 1	Próba 2
Grupa 0	<b>B</b> G <b>S</b> T	<b>B</b> C G I <b>O</b> <b>S</b>
Grupa 1	C D J <b>O</b>	D J Q
Grupa 2	Q	U
Grupa 3	L	L
Grupa 4	U	H <b>N</b>
Grupa 5	H <b>N</b>	A
Grupa 6	A	R
Grupa 7	<b>E</b>	<b>P</b>
Grupa 8	<b>P</b> R	T
Grupa 9	F	<b>E</b> F
Grupa 10	K	K
Grupa 11	I	

Tab. 8. Wyniki dla współczynnika uczenia  
równego 0,5\*

Współczynnik uczenia 0,5		
	Próba 1	Próba 2
Grupa 0	C D G J <b>O</b> Q U	L
Grupa 1	L	C <b>O</b>
Grupa 2	K	K
Grupa 3	H <b>N</b>	<b>S</b>
Grupa 4	A <b>S</b>	G
Grupa 5	R	<b>B</b> D
Grupa 6	<b>E</b>	<b>P</b> R
Grupa 7	F	A <b>E</b> F H <b>N</b>
Grupa 8	<b>B</b> <b>P</b>	Q
Grupa 9	T	U
Grupa 10	I	J
Grupa 11		T
Grupa 12		I

Tab. 9. Wyniki dla współczynnika uczenia równego 1,0\*

Współczynnik uczenia 1,0		
	Próba 1	Próba 2
Grupa 0	I	L
Grupa 1	T	T
Grupa 2	<b>S</b>	J Q U
Grupa 3	<b>B</b> C D G <b>O</b>	A <b>B</b> <b>P</b> R
Grupa 4	U	<b>E</b> F
Grupa 5	J	<b>N</b>
Grupa 6	<b>E</b> F	H
Grupa 7	<b>P</b> R	<b>S</b>
Grupa 8	A H K L <b>N</b> Q	C D G <b>O</b>
Grupa 9		I
Grupa 10		K

\* litery oznaczone kolorem czerwonym nie brały udziału w procesie uczenia

#### 4. Analiza wyników

Analizując wyniki przedstawione w tabeli nr 5, dla której wartość współczynnika uczenia była najmniejsza widzimy, że sieć nie do końca dzieli litery zbiór na poprawne grupy. Dzieje się tak, dlatego, że sieć nie zdążyła się nauczyć odpowiedniego działania w ciągu 200 epok przy tak małym współczynniku. Widoczne jest, że pojedyncze litery są przypisane poprawnie do wspólnych gruch np. F i P, z czego P nie brała udziału w uczeniu, a także litery, które wykazują się dużą indywidualnością są przyporządkowane do osobnych klastrow. Porównując również tą tabelę z pozostałymi zauważalne jest, że zbiór został podzielony na mniejszą ogólną liczbę klastrow, dlatego też niektóre litery, które nie powinny są w tych samych grupach.

Kolejne trzy tabele o numerach 6, 7 i 8 pokazują już w miarę poprawnie działającą sieć. Występuje w nich dużo większa liczba poprawnych pogrupowań na klastry. Większość zgadza się z odległościami euklidesowymi wyznaczonymi wcześniej i przedstawionymi w tabeli nr 2. Widzimy, że litery, które nie brały udziału w procesie uczenia w fazie testowania są przydzielane do odpowiednich grup np. litera N do grupy z literą H, czy litera O do grupy z literą C. Także liczba wszystkich grup jest odpowiednio zwiększona. Można więc wnioskować, że wzrost wartości współczynnika korzystnie działa na proces uczenia.

Ostatnia tabela z wynikami nr 9 odnosi się do współczynnika uczenia na poziomie 1,0. Ponownie zauważalne jest zmniejszenie się liczby klastrow i grupowanie poszczególnych liter jest mniej poprawne. Sieć przyporządkowuje dużą ilość liter do dwóch klastrow a pozostałe rozбивa na jedno elementowe. Można zauważyć, że występują poprawnie pogrupowane elementy, a więc sieć w jakimś stopniu spełnia swoją funkcję, jednak przy niższej wartości współczynnika uczenia wyniki były bardziej zadowalające.

Podsumowując można stwierdzić, że sieci, które, najlepiej grupują na klastry występują przy współczynniku uczenia na poziomie 0,1 oraz 0,5. A więc można wnioskować, że współczynnik ten powinien właśnie plasować się w granicach 0,1 do 0,5, aby proces uczenia przebiegał jak najbardziej sprawnie.

#### 5. Wnioski

- Sieć Kohonena jest siecią samoorganizującą pozwalającą podzielić dane posiadające różne wartości poszczególnych cech na odpowiednie klastry nie wymagając przy tym wartości oczekiwanych podczas procesu uczenia (nauczanie bez nauczyciela). A więc ten rodzaj sieci sprawdziłby się doskonale przy podziale i wykrywaniu wspólnych cech przy bardzo dużej ilości różnych obiektów, o których wiedza jest bardzo ograniczona.
- Większość wyznaczonych grup przez sieć Kohonena pokrywa się z grupami wydzielonymi subiektywnie na podstawie obliczonych odległości euklidesowych. Sensownym rozwiązaniem jest obliczenie również innego rodzaju odległości np. odległość max czy odległość miejską (Manhattan) i na tej podstawie ponownie wyznaczyć podział na klastry i porównać z tymi otrzymanymi za pomocą sieci. Możliwe, że byłoby jeszcze więcej podobieństw.



- Odpowiedzi sieci wskazują, że rozwiązanie przedstawionego problemu, a więc podziału zbioru na klastry można zrealizować w różnorodny sposób i każdy taki podział oznacza się poprawnością, nie ma tylko jednego, słusznego jego rozwiązania.
- Procesu uczenia zależy od współczynnika uczenia. Wraz z jego wzrostem proces uczenia jest szybszy. Jest to wytłumaczalne z tego względu, że im ta wartość jest większa tym przyrost wag, które na samym początku są niewielkie jest szybszy, a więc i proces uczenia przebiega szybciej. Jednak po pewnej jego wartości następuje stabilizacja i dalsze zwiększanie współczynnika może tylko pogorszyć proces uczenia i nie przynieść żadnych wymiernych korzyści. Natomiast jeżeli współczynnik ten jest bardzo mały sieć potrzebuje dużo większej ilości epok do odpowiedniego nauczania, bardzo wolno się uczy, co z jednej strony skutkuje dużym narzutem czasowym, jednak nie otrzymamy takiej sytuacji, że sieć się nigdy nie nauczy, jak może się zdarzyć przy większych współczynnikach.
- Porównując z wcześniej wykonanym projektem dotyczącym sieci Kohonena wykorzystującej podejście WTA otrzymujemy lepsze rezultaty, ponieważ uporządkowanie sieci jest lepsze, a także zbieżność algorytmu jest wyższa. Minusem natomiast jest to, że proces pojedynczego modyfikowanie wag trwa dłużej, bo modyfikowany jest nie tylko dla neuronu zwycięzcy, ale i sąsiadów, których wyznaczenie odbywa się z wykorzystaniem odpowiedniej funkcji, która dodatkowo potrzebuje czasu procesora.

## 6. Kod programu

main.cpp

```
#include <iostream>
#include <vector>
#include "Warstwa.h"
#include <time.h>
#include <fstream>
using namespace std;
```

```
void ustawDaneWejscowe(Neuron& neuron, vector< vector<double> > inputData, int numberOfEntrances, int inputDataRow);
void ucz(Warstwa& layer, vector< vector<double> > inputData);
void test(Warstwa& layer, vector< vector<double> > inputData);
void wczytajDaneUczace(vector< vector<double> > &learningInputData, int numberOfEntrances, int amountOfOutputs);
void wczytajDaneTestujace(vector< vector<double> > &testingInputData, int numberOfEntrances, int amountOfOutputs);
```

```
fstream PLIK_WYJSCIOWY_UCZENIE;
fstream PLIK_WYJSCIOWY_TESTOWANIE;
fstream TESTOWANIE_NEURONU;
fstream DANE_UCZACE;
fstream DANE_TESTUJACE;
```

```

int main()
{
    srand(time(NULL));
    vector< vector<double> > daneUczace;
    vector< vector<double> > daneTestujace;
    int liczbaNeuronow = 20;
    int liczbaWejsc = 20;
    int liczbaWyjsc = 1;
    double wspolczynnikUczenia = 1.0;
    int liczbaEpok = 200;
    Warstwa kohonenNetwork(liczbaNeuronow, liczbaWejsc, liczbaWyjsc, wspolczynnikUczenia,
(double)liczbaEpok);
    wczytajDaneUczace(daneUczace, liczbaWejsc, liczbaWyjsc);
    wczytajDaneTestujace(daneTestujace, liczbaWejsc, liczbaWyjsc);

    do
    {
        cout << "1. Ucz" << endl;
        cout << "2. Testuj" << endl;
        cout << "3. Wyjdz" << endl;
        int wybor;
        cin >> wybor;
        switch (wybor)
        {
            case 1:
                PLIK_WYJSCIOWY_UCZENIE.open("output_learning_data.txt", ios::out);

                for (int epochNumber = 1, i = 0; i < liczbaEpok; i++, epochNumber++)
                {
                    ucz(kohonenNetwork, daneUczace);
                    PLIK_WYJSCIOWY_UCZENIE << "EPOCH NUMBER: " << epochNumber <<
endl;
                }

                break;

            case 2:
                PLIK_WYJSCIOWY_TESTOWANIE.open("output_testing_data.txt", ios::out);
                TESTOWANIE_NEURONU.open("output_testing_neuron.txt", ios::out);
                test(kohonenNetwork, daneTestujace);
                break;

            case 3:
                PLIK_WYJSCIOWY_UCZENIE.close();
                PLIK_WYJSCIOWY_TESTOWANIE.close();
                return 0;
        }

    } while (true);

    return 0;
}

```

```

void ustawDaneWejscowe(Neuron& neuron, vector< vector<double> > daneWejscowe, int liczbaWejsc, int
wiersz)
{
    for (int i = 0; i < liczbaWejsc; i++)
        neuron.ustawWejscie(i, daneWejscowe[wiersz][i]);
}

void ucz(Warstwa& warstwa, vector< vector<double> > inputData)
{
    static int obecnaIteracja = 1;
    for (int wierszDanych = 0; wierszDanych < inputData.size(); wierszDanych++)
    {
        for (int i = 0; i < warstwa.wezLiczbeNeuronow(); i++)
        {
            ustawDaneWejscowe(warstwa.neurony[i], inputData,
warstwa.neurony[i].wezIloscWejsc(), wierszDanych);
            warstwa.neurony[i].liczIloczynSkalarny();
        }
        warstwa.zmienWagi(obecnaIteracja, false);
        PLIK_WYJSCIOWY_UCZENIE << warstwa.wezIndeksZwyciezcy() << endl;
    }
    obecnaIteracja++;
}

void test(Warstwa& warstwa, vector< vector<double> > daneWejscowe)
{
    for (int wierszDanych = 0, znak = 0; wierszDanych < daneWejscowe.size(); wierszDanych++)
    {
        for (int i = 0; i < warstwa.wezLiczbeNeuronow(); i++)
        {
            ustawDaneWejscowe(warstwa.neurony[i], daneWejscowe,
warstwa.neurony[i].wezIloscWejsc(), wierszDanych);
            warstwa.neurony[i].liczIloczynSkalarny();
        }

        if (wierszDanych == 12 )
            znak++;
        char letter = 'A';
        warstwa.zmienWagi(0, true);
        PLIK_WYJSCIOWY_TESTOWANIE <<
warstwa.neurony[warstwa.wezIndeksZwyciezcy()].wezSumeWejsc() << endl;
        TESTOWANIE_NEURONU << static_cast<char>(letter + wierszDanych + znak) << " " <<
warstwa.wezIndeksZwyciezcy() << endl;
    }
}

void wczytajDaneUczace(vector< vector<double> > &daneUczace, int liczbaWejsc, int liczbaWyjsc)
{
    DANE_UCZACE.open("learning_data.txt", ios::in);
    vector<double> wiersz;

    do
    {

```

```

        wiersz.clear();

        for (int i = 0; i < liczbaWejsc; i++)
        {
            double inputTmp = 0.0;
            DANE_UCZACE >> (double)inputTmp;
            wiersz.push_back(inputTmp);
        }

        double vectorLength = 0.0;

        for (int i = 0; i < liczbaWejsc; i++)
            vectorLength += pow(wiersz[i], 2);
        vectorLength = sqrt(vectorLength);

        for (int i = 0; i < liczbaWejsc; i++)
            wiersz[i] /= vectorLength;

        daneUczace.push_back(wiersz);

    } while (!DANE_UCZACE.eof());
    DANE_UCZACE.close();
}

void wczytajDaneTestujace(vector< vector<double> > &daneTestujace, int liczbaWejsc, int liczbaWyjsc)
{
    DANE_TESTUJACE.open("testing_data.txt", ios::in);
    vector<double> wiersz;

    while (!DANE_TESTUJACE.eof())
    {
        wiersz.clear();

        for (int i = 0; i < liczbaWejsc; i++)
        {
            double inputTmp = 0.0;
            DANE_TESTUJACE >> (double)inputTmp;
            wiersz.push_back(inputTmp);
        }

        double vectorLength = 0.0;

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

        vectorLength = sqrt(vectorLength);

        for (int i = 0; i < liczbaWejsc; i++)
            wiersz[i] /= vectorLength;

        daneTestujace.push_back(wiersz);
    }
}

```

```
        DANE_TESTUJACE.close();  
    }
```

## Neuron.h

```
#pragma once  
#include <iostream>  
#include <vector>  
using namespace std;  
  
class Neuron  
{  
public:  
    void stworzWejscia(int liczbaDendrytow, int liczbaSynaps);  
    void stworzWejscie() { _wejscia.push_back(0); }  
    void stworzWage(int index) { _wagi.push_back(0); }  
    int wezIloscWejsc() { return _wejscia.size(); }  
    int wezIloscWag() { return _wagi.size(); }  
    double wezWejscie(int index) { return _wejscia[index]; }  
    void ustawWejscie(int index, double value) { _wejscia[index] = value; }  
    double wezSynapse(int index) { return _wagi[index]; }  
    void ustawSynapse(int index, double value) { _wagi[index] = value; }  
    double wezSumeWejsc() { return _sumaWejsc; }  
    double wezWyjscie() { return _wartoscWyjscia; }  
    double procesWejscia(int index) { return _wejscia[index] * _wagi[index]; }  
    void procesWyjscia();  
    void liczNoweWagi();  
    double liczIloczynSkalarny();  
    void wyznaczSasiadow(double mapRadius, double obecnaIteracja, double czas);  
    double wezSasiadow() { return _sasiedzi; }  
    void ustawSasiadow(double neighbourhoodRadius) { _sasiedzi = neighbourhoodRadius; }  
    void ustawDystans(double odleglosc) { _odleglosc = odleglosc; }  
    Neuron();  
    Neuron(int liczbaDendrytow, int liczbaWyjsc, double wspolczynnikUczenia);  
  
private:  
    double liczPierwszeWagi();  
    void liczSasiadow();  
    void normalizujWagi();  
    vector<double> _wejscia;  
    vector<double> _wagi;  
    double _sumaWejsc;  
    double _wartoscWyjscia;  
    double _wspolczynnikUczenia;  
    double _wartoscFunkcjiSasiadow;  
    double _odleglosc;  
    double _sasiedzi;  
};
```

## Neuron.cpp

```
#include "Neuron.h"  
#include <time.h>
```

```
#include <math.h>
```

```
Neuron::Neuron()
```

```
{  
    _wejscia.resize(0);  
    _wagi.resize(0);  
    _sumaWejsc = 0.0;  
    _wartoscWyjscia = 0.0;  
    _wspolczynnikUczenia = 0.0;  
}
```

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

```
{  
    stworzWejscia(amountOfDendrites, amountOfOutputs);  
    normalizujWagi();  
    _wspolczynnikUczenia = learningCoefficient;  
    _sumaWejsc = 0.0;  
    _wartoscWyjscia = 0.0;  
}
```

```
void Neuron::stworzWejscia(int amountOfDendrites, int amountOfOutputs)
```

```
{  
    for (int j = 0; j < amountOfDendrites; j++)  
    {  
        _wejscia.push_back(0);  
        _wagi.push_back(liczPierwszeWagi());  
    }  
}
```

```
double Neuron::liczIloczynSkalarny()
```

```
{  
    _sumaWejsc = 0.0;  
  
    for (int i = 0; i < wezIloscWejsc(); i++)  
        _sumaWejsc += pow(_wejscia[i] - _wagi[i], 2);  
  
    _sumaWejsc = sqrt(_sumaWejsc);  
    return _sumaWejsc;  
}
```

```
void Neuron::procesWyjscia()
```

```
{  
    double beta = 1.0;  
    _wartoscWyjscia = (1.0 / (1.0 + (exp(-beta * _sumaWejsc))));  
}
```

```
void Neuron::liczNoweWagi()
```

```
{  
    liczSasiadow();  
    for (int i = 0; i < wezIloscWag(); i++)  
        _wagi[i] = _wagi[i] + (_wspolczynnikUczenia * _wartoscFunkcjiSasiadow * (_wejscia[i] -  
        _wagi[i]));  
}
```

```

        normalizujWagi();
    }

void Neuron::wyznaczSasiadow(double mapRadius, double currentIteration, double timeConstant)
{
    _sasiedzi = mapRadius * exp(-currentIteration / timeConstant);
}

//private methods

void Neuron::liczSasiadow()
{
    _wartoscFunkcjiSasiadow = exp(pow(-_odleglosc,2) / (2 * pow(_sasiedzi,2)));
}

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

void Neuron::normalizujWagi()
{
    double vectorLength = 0.0;

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

    vectorLength = sqrt(vectorLength);

    for (int i = 0; i < wezIloscWag(); i++)
        _wagi[i] /= vectorLength;
}

```

## Warstwa.h

```

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

class Warstwa
{
public:
    Warstwa(int liczbaNeuronow, int liczbaDendrytow, int liczbaWyjsc, double wspolczynnikUczenia,
double liczbalteracji);
    vector<Neuron> neurony;
    int wezLiczbeNeuronow() { return _liczbaNeuronow; }
    void zmienWagi(double obecnaalteracja, bool testing);
    double wezIloczynSkalarny(int index) { return _iloczynSkalarny[index]; }
    void zbierzIloczynySkalarne();
    int wezIndeksZwyciezcy() { return _indexzwyciezcy; }
}

```

```
private:
    void znajdzMax();
    int _liczbaNeuronow;
    vector<double> _iloczynSkalarny;
    int _indexzwyciezcy;
    double _mapaRadius;
    double _czas;
};
```

## Warstwa.cpp

```
#include "Warstwa.h"
```

```
Warstwa::Warstwa(int numberOfNeurons, int amountOfDendrites, int amountOfOutputs, double
learningCoefficient, double numberOfIterations)
{
    _liczbaNeuronow = numberOfNeurons;
    neurony.resize(numberOfNeurons);
    _mapaRadius = (double)numberOfNeurons;
    _czas = numberOfIterations / _mapaRadius;

    for (int i = 0; i < numberOfNeurons; i++)
        neurony[i].Neuron::Neuron(amountOfDendrites, amountOfOutputs, learningCoefficient);
}
```

```
void Warstwa::zmienWagi(double obecnaAlteracja, bool testing)
{
    zbierzIloczynnySkalarne();
    znajdzMax();
    neurony[_indexzwyciezcy].procesWyjscia();
    if (testing == false)
    {
        neurony[_indexzwyciezcy].wyznaczSasiadow(_mapaRadius, obecnaAlteracja, _czas);
        int radius = neurony[_indexzwyciezcy].wezSasiadow();
        int leftBorderNeuronIndex = 0;
        int rightBorderNeuronIndex = 0;

        if (_indexzwyciezcy - radius < 0)
            leftBorderNeuronIndex = 0;

        else
            leftBorderNeuronIndex = _indexzwyciezcy - radius;

        if (_indexzwyciezcy + radius >= _liczbaNeuronow)
            rightBorderNeuronIndex = _liczbaNeuronow - 1;

        else
            rightBorderNeuronIndex = _indexzwyciezcy + radius;

        for (int i = leftBorderNeuronIndex; i < rightBorderNeuronIndex; i++)
        {
```



```

        neurony[i].ustawDystans((i < _indexzwyciezcy) ? (_indexzwyciezcy - i) : (i -
_indexzwyciezcy));

        neurony[i].ustawSasiadow(neurony[_indexzwyciezcy].wezSasiadow());
        neurony[i].liczNoweWagi();
    }
}

void Warstwa::zbierzIloczynnySkalarne()
{
    _iloczynSkalarny.clear();

    for (int i = 0; i < _liczbaNeurnow; i++)
        _iloczynSkalarny.push_back(neurony[i].liczIloczynSkalarny());
}

void Warstwa::znajdzMax()
{
    double tmp = _iloczynSkalarny[0];
    _indexzwyciezcy = 0;

    for (int i = 1; i < _iloczynSkalarny.size(); i++)
    {
        if (tmp > _iloczynSkalarny[i])
        {
            _indexzwyciezcy = i;
            tmp = _iloczynSkalarny[i];
        }
    }
}

```