

**Podstawy Sztucznej Inteligencji**

Scenariusz 6: Budowa i działanie sieci Kohonena dla WTM

Wykonała:

**Magdalena Migas**

**Wydział Inżynierii Metali i Informatyki Przemysłowej**

**Informatyka Stosowana**

1. **Cel ćwiczenia**

Wykonane ćwiczenie miało na celu poznanie budowy i działania sieci Kohonena przy wykorzystaniu reguły Winner Takes Most do odwzorowywania istotnych cech liter alfabetu.

1. **Sieć Kohonena i reguła Winner Takes Most**

Sieci Kohonena realizują pewne **odwzorowania,** stąd ich powszechnie używany symbol: SOM(*Self Organizing Maps*).

Cechy charakterystyczne:

• sieć uczy się bez nauczyciela;

• uporządkowane neurony wyjściowe;

• jest konkurencja i wyłaniany jest neuron „zwycięski”;

• ważną rolę odgrywa „sąsiedztwo”;

• w wyniku uczenia powstaje mapa topologiczna;

• aprioryczna interpretacja wartości wyjściowych jest niemożliwa;

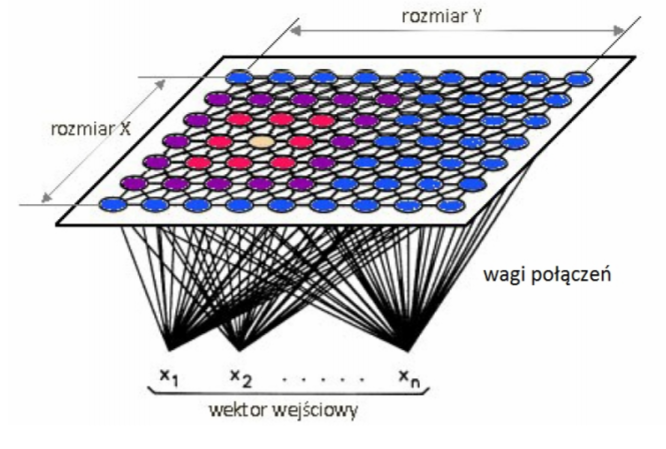
• po uczeniu można ustalić, jakie znaczenie mają poszczególne rejony mapy topologicznej ale wyłącznie na podstawie analizy konkretnych przykładów danych wejściowych.

Zasada Winner Takes Most polega na tym, że neuron zwycięski oraz neurony z nim sąsiadujące czyli należące do sąsiedztwa Nc(k), aktualizują swoje wagi według zasady:

gdzie .

Funkcja G oznacza tutaj wpływ sąsiedztwa. Istnieje kilka różnych formuł do pisu tej funkcji. Oryginalnie Kohonen zaproponował dwa rozwiązania:

1. Sąsiedztwo prostokątne:
2. Sąsiedztwo gaussowskie



Rysunek 1: Struktura mapy Kohonena

**Algorytm SOM dla WTM**

1. Przypisz wagom sieci o M neuronach warstwy wyjściowej i N wejściach niewielkie liczby losowe. Ustal liczbę neuronów należących do początkowego otoczenia neuronu.

2. Dołącz nowy wektor uczący x=[x1 , x2 , ..., xN ] do wejścia.

3. Oblicz odległości

4. Znajdź neuron zwycięski. Mając próbkę x ze zbioru uczącego X w danym kroku k fazy uczenia znajdujemy element mapy najbardziej zbliżony do wektora prezentowanego c(x):

**c(x) = arg min{||x-mi||}**

5. Wyznacz nowe wartości wag dla neuronu zwycięzcy i jego sąsiedztwa korzystając ze strategii:

6. Zmień odpowiednio wartości współczynnika uczenia i sąsiedztwa

7. Powtórz 2-6 dla następnych wzorców wejściowych aż do chwili ustalenia się odpowiedzi sieci

1. **Zbór uczący**

Przykładowe dane uczące znajdują się w pliku zbior\_uczacy1.txt. W pliku tym zawarto 20 liter polskiego alfabetu, które z matryc 35-elementowych rozwinięto na odpowiadające im wektory, dla przykładu litera A:

**-1 1 1 1 -1  
 1 -1 -1 -1 1  
 1 -1 -1 -1 1  
 1 1 1 1 1   
 1 -1 -1 -1 1   
 1 -1 -1 -1 1  
 1 -1 -1 -1 1**

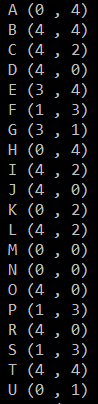
🡪 **-1 1 1 1 -1 1 -1 -1 -1 1 1 -1 -1 -1 1 1 1 1 1 1 1 -1 -1 -1 1 1 -1 -1 -1 1 1 -1 -1 -1 1**

1. **Uzyskane wyniki**

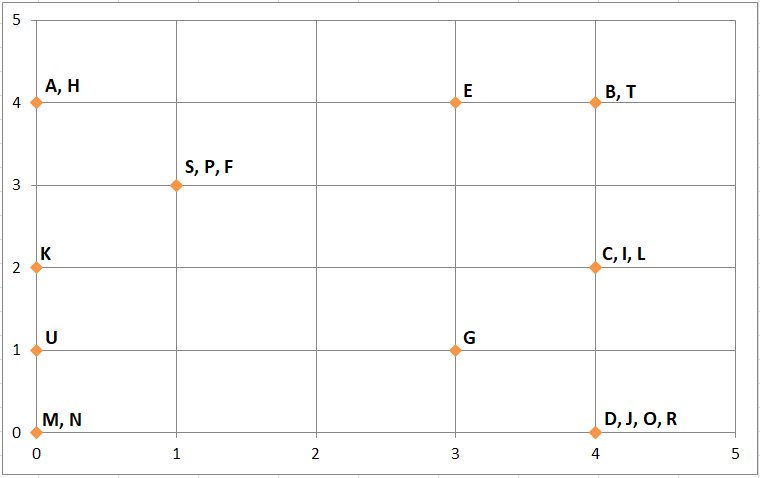
Stworzona sieć Kohonena uczy się aż osiągnie określoną przez użytkownika wartość błędu. Ponadto grupuje ona podane litery alfabetu na matrycy o rozmiarze również zdefiniowanym przez użytkownika. W związku z powyższym w trakcie testów wykonano kilka wariantów, które różnią się od siebie przyjętą wielkością błędu oraz wielkością matrycy wyjściowej.

**WARIANT I**

* Założona wartość błędu: 0.0000001;
* Rozmiar matrycy: 5x5



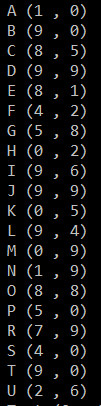
Rysunek 2: Wydruk programu dla wariantu I



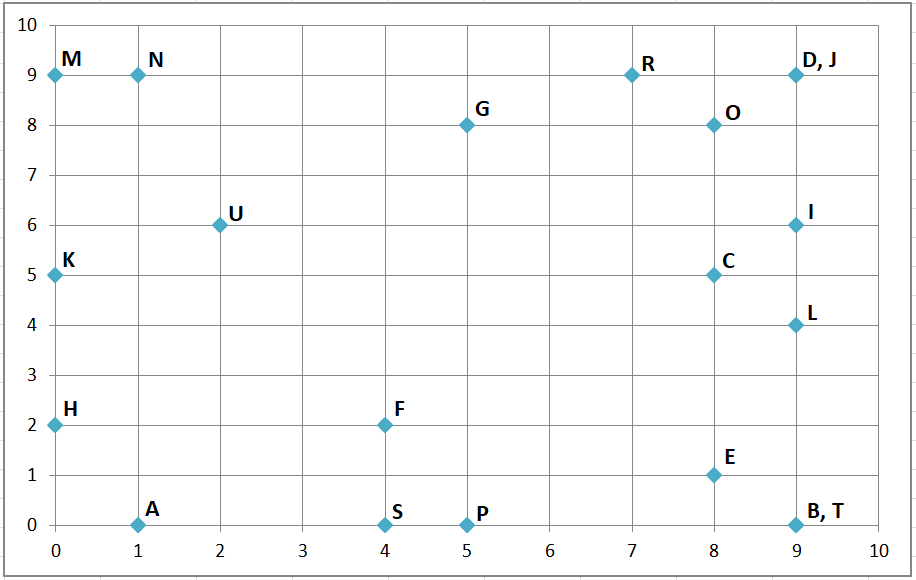
Rysunek 3: Pogrupowane litery dla wariantu I

**WARIANT II**

* Założona wartość błędu: 0.0000001;
* Rozmiar matrycy: 10x10



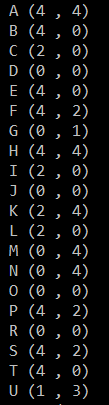
Rysunek 4: Wydruk programu dla wariantu II



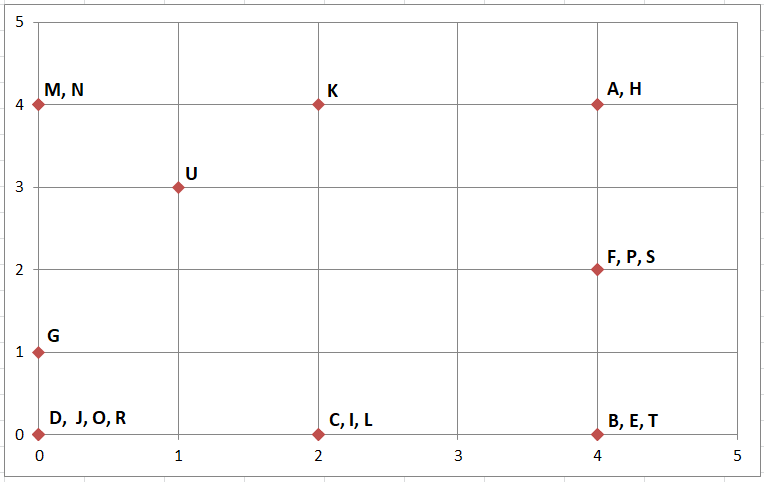
Rysunek 5: Pogrupowane litery dla wariantu II

**WARIANT III**

* Założona wartość błędu: 0.0001;
* Rozmiar matrycy: 5x5



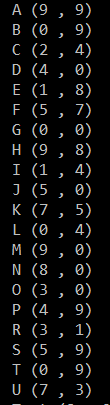
Rysunek 6: Wydruk programu dla wariantu III



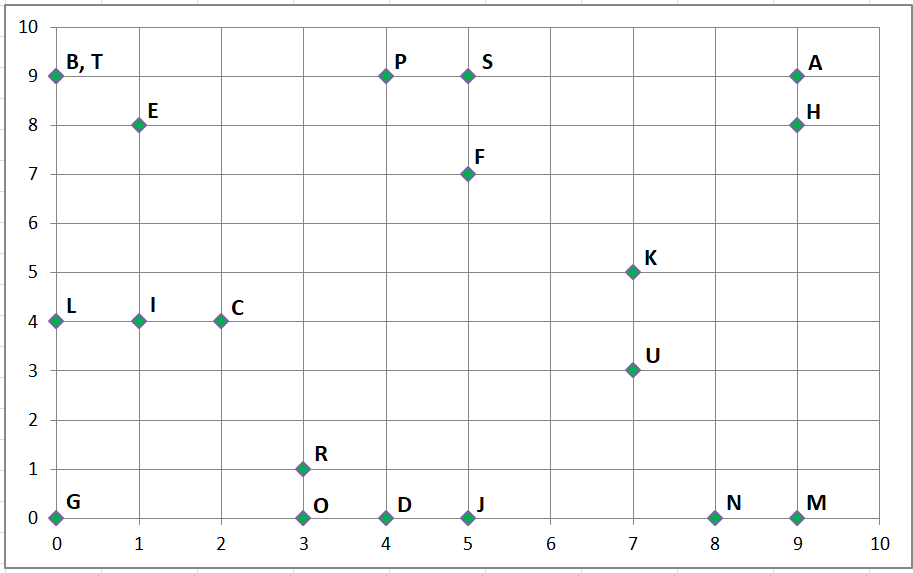
Rysunek 7: Pogrupowane litery dla wariantu III

**WARIANT IV**

* Założona wartość błędu: 0.0001;
* Rozmiar matrycy: 10x10



Rysunek 8: Wydruk programu dla wariantu IV



Rysunek 9: Pogrupowane litery dla wariantu IV

1. **Wnioski**

* W opracowanym zagadnieniu celem było grupowanie liter polskiego alfabetu z wykorzystaniem sieci Kohonena i algorytmu Winner Takes Most. W tym celu zaimplementowano program w języku C++, który realizuje to zadanie.
* W czasie etapu testów przeprowadzono 4 warianty, w których zastosowano zróżnicowane wielkości matryc wyjściowych oraz różne przyjęte wartości błędu.
* Ogólnie możemy przyjąć, że w każdym z wariantów program działa poprawnie gdyż litery zostają w pewien sposób pogrupowane. Co jasne, zmniejszenie tolerancji błędu sprawia, że grupowanie liter staje się dokładniejsze.
* O poprawnym grupowaniu liter przez stworzoną sieć może świadczyć fakt, że na otrzymanej mapie blisko siebie lub nawet na tym samym miejscu znajdują się litery obiektywnie do siebie podobne takie jak na przykład pary: I oraz J, M oraz N czy też para liter A oraz H.

1. **Literatura**

* <http://zsi.tech.us.edu.pl/~nowak/wi/som.pdf>
* <http://galaxy.agh.edu.pl/~vlsi/AI/koho_t/>

1. **Listing kodu programu**

#include<iostream>

#include<cstdlib>

#include<cmath>

#include<fstream>

#include<list>

#include<time.h>

using namespace std;

class Neuron

{

public:

double\* Weights;

int size;

int X;

int Y;

int length;

double nf;

Neuron(int x, int y, int length)

{

size = 35;

Weights = new double[size];

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

Weights[i] = 0;

X = x;

Y = y;

this->length = length;

nf = 1000 / log(length);

}

double Gauss(Neuron win, int it)

{

double A = pow((win.X - X), 2);

double B = pow(win.Y - Y, 2);

double distance =sqrt(A+B);

return exp(-pow(distance, 2) / (pow(Strength(it), 2)));

}

double LearningRate(int it)

{

return exp(-it / 1000) \* 0.1;

}

double Strength(int it)

{

return exp(-it / nf) \* length;

}

double UpdateWeights(double\* pattern, Neuron winner, int it)

{

double sum = 0;

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

{

double d = pattern[i];

double e = Weights[i];

double f = d - e;

double delta = LearningRate(it) \* Gauss(winner, it) \* (f);

Weights[i] += delta;

sum += delta;

}

return sum / size;

}

Neuron() {

size = 35;

Weights = new double[size];

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

Weights[i] = 0;

X = 0;

Y = 0;

length = 1;

nf = 1000 / log(length);

};

};

void setNeuron(Neuron neuron, int x, int y, int length)

{

neuron.size = 35;

neuron.Weights = new double[neuron.size];

for (int i = 0;i < neuron.size; i++)

neuron.Weights[i] = 0;

neuron.X = x;

neuron.Y = y;

neuron.length = length;

neuron.nf = 1000 / log(length);

}

class Map

{

public:

Neuron\*\* outputs;

int iteration;

int length;

int dimensions;

double\*\* patterns;

Map(int dimensions, int length)

{

patterns = new double\*[20];

for (int i = 0; i < 20;i++)

{

patterns[i] = new double[35];

}

for (int i = 0; i < 20;i++)

{

for (int j = 0; j <35; j++)

{

patterns[i][j] = 0;

}

}

this->length = length;

this->dimensions = dimensions;

Initialise();

LoadData();

NormalisePatterns();

Train(0.00001);

DumpCoordinates();

}

void Initialise()

{

outputs = new Neuron\*[length];

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

{

outputs[i] = new Neuron[length];

}

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

{

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

{

setNeuron(outputs[i][j],i, j, length);

outputs[i][j].Weights = new double[dimensions];

for (int k = 0; k < dimensions; k++)

{

outputs[i][j].Weights[k] = rand()%10+1;

}

}

}

}

void LoadData()

{

fstream plik;

plik.open("zbior\_uczacy1.txt");

if (plik.good())

{

int inputSize = 35;

int patternSize = 20;

for (int k = 0; k < patternSize; k++)

{

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

{

plik >> patterns[k][i];

}

}

}

else

{

cout << "blad otwarcia pliku!";

}

}

void NormalisePatterns()

{

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

{

double sum = 0;

for (int i = 0; i< 20; i++)

{

sum += patterns[i][j];

}

double average = sum / 20;

for (int i = 0; i< 20; i++)

{

patterns[i][j] = patterns[i][j] / average;

}

}

}

void normalizeData()

{

for (int i = 0; i <20; i++)

{

double sum = 0;

for (int j = 0; j< 35; j++)

{

sum += pow(patterns[i][j],2);

}

double a = sqrt(sum);

for (int j = 0; j< 20; j++)

{

patterns[i][j] = patterns[i][j] / a;

}

}

for (int k = 0; k < 20; k++)

{

for (int i = 0; i < 35; i++)

{

cout << patterns[k][i] << " ";

}

cout << endl;

}

}

void Train(double maxError)

{

double currentError = 1000000000;

while (currentError > maxError)

{

currentError = 0;

for(int i=0; i<20; i++)

{

currentError += TrainPattern(patterns[i]);

}

}

}

double TrainPattern(double\* pattern)

{

double error = 0;

Neuron winner = Winner(pattern);

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

{

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

{

error += outputs[i][j].UpdateWeights(pattern, winner, iteration);

}

}

iteration++;

return abs(error / (length \* length));

}

void DumpCoordinates()

{

for (int i=0; i<20; i++)

{

Neuron n = Winner(patterns[i]);

cout << n.X << " " << n.Y <<endl;

}

}

Neuron Winner(double\* pattern)

{

Neuron winner;

int size = 35;

double min = 5000000;

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

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

{

double d = Distance(pattern, outputs[i][j].Weights,size);

if (d < min)

{

min = d;

winner = outputs[i][j];

}

}

return winner;

}

double Distance(double\* vector1, double\* vector2, int size1)

{

double value = 0;

for (int i = 0; i < size1; i++)

{

double D = vector1[i];

double E = vector2[i];

double F = D - E;

value += pow(F, 2);

}

return sqrt(value);

}

};