**Podstawy Sztucznej Inteligencji**

Scenariusz 5: Budowa ​ i ​ ​działanie ​ ​sieci ​ ​Kohonena ​ ​dla ​ ​WTA

Wszystkie projekty znajdują się w jednym programie na repozytorium

**Wykonał:**

**Bartłomiej Leja**

**Wydział Inżynierii Metali i Informatyki Przemysłowej**

**Informatyka Stosowana**

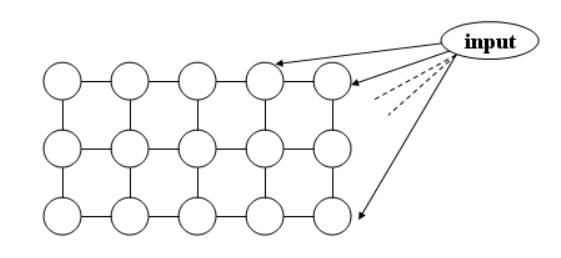
1. **Cel ćwiczenia**

Celem ćwiczenia było zapoznanie się z siecią Kohonena oraz z algorytmem uczenia rywalizującego WTA (Winner Takes All).

1. **Sieć Kohonena**

Sieć Kohonena nazywana również samoorganizującą siecią Kohonena lub mapą Kohonena,  sieć neuronowa uczona w trybie bez nauczyciela w celu wytworzenia niskowymiarowej (przeważnie dwuwymiarowej) zdyskretyzowanej reprezentacji przestrzeni wejściowej tak zwanej mapy. Sieć Kohonena wyróżnia się tym od innych sieci, że zachowuje odwzorowanie sąsiedztwa przestrzeni wejściowej. Wynikiem działania sieci jest klasyfikacja przestrzeni w sposób grupujący zarówno przypadki ze zbioru uczącego, jak i wszystkie inne wprowadzenia po procesie uczenia.

1. **Zasada działania sieci Kohonena**



Rys.1 Wygląd mapy Kohonena

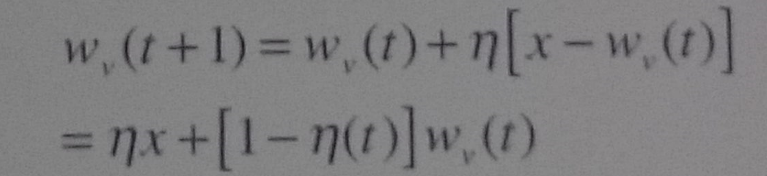
**Zasada działania:**

* Wejścia (tyle, iloma parametrami opisano obiekty) połączone są ze wszystkimi węzłami sieci
* Każdy węzeł przechowuje wektor wag o wymiarze identycznym z wektorami wejściowymi
* Każdy węzeł oblicza swój poziom aktywacji jako iloczyn skalarny wektora wag i wektora wejściowego (podobnie jak w zwykłym neuronie)
* Ten węzeł, który dla danego wektora wejściowego ma najwyższy poziom aktywacji, zostaje zwycięzcą i jest uaktywniony (metoda WTA)
* Wzmacniamy podobieństwo węzła-zwycięzcy do aktualnych danych wejściowych poprzez dodanie do wektora wag wektora wejściowego (z pewnym współczynnikiem uczenia)
* Każdy węzeł może być stowarzyszony z pewnymi innymi, sąsiednimi węzłami - wówczas te węzły również zostają zmodyfikowane, jednak w mniejszym stopniu (metoda WTM).

Inicjalizacja wag sieci Kohonena jest losowa. Wektory wejściowe stanowią próbę uczącą, podobnie jak w przypadku zwykłych sieci rozpatrywaną w pętli podczas budowy mapy. Wykorzystanie utworzonej w ten sposób mapy polega na tym, że zbiór obiektów umieszczamy na wejściu sieci i obserwujemy, które węzły sieci się uaktywniają. Obiekty podobne powinny trafiać w podobne miejsca mapy.

1. **Algorytm WTA**

Algorytm Winner Takes All w sieci Kohonena jest metodą uczenia rywalizacyjnego. W efekcie współzawodnictwa następuje samoorganizacja procesu uczenia. Neurony dopasowują swoje wagi w ten sposób, że przy reprezentacji grup wektorów wejściowych zbliżonych do siebie zwycięża zawsze ten sam neuron. Jego wagi będą odpowiadać uśrednionym wartością wektorów wejściowych, dla których dany neuron był zwycięzcą. Neuron, poprzez zwycięstwo we współzawodnictwie rozpoznaje swoją grupę (klasę). Neurony nie wygrywające nie zmieniają swoich wag – pozostają martwe.



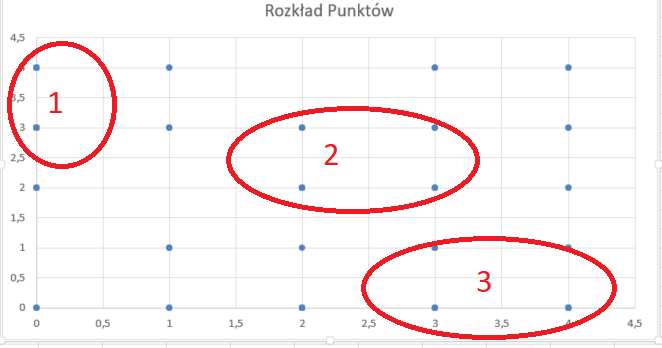
Rys.2 Metoda obliczania wag w WTA

1. **Zestaw danych uczących**

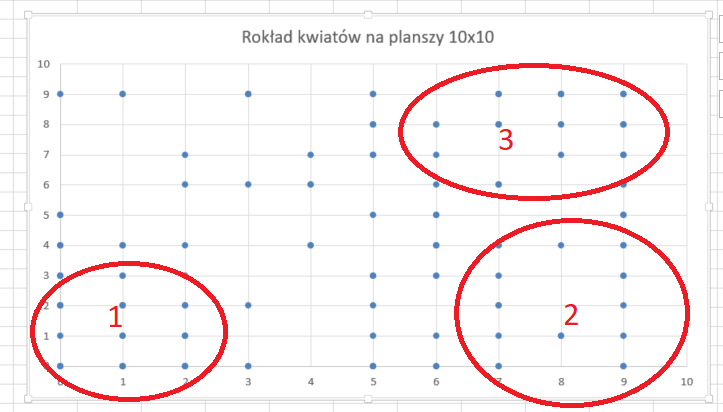
Dane uczące zostały zawarte w pliku KohonenWTAData.txt, który zawiera 150 zestawów cech, 3 rodzajów irysów w postaci 4 elementowych wektorów. Dla przykładu irys pierwszy to wektor: **1 5,1 3,5 1,4 0,2**  gdzie jeden oznacza iż jest to grupa pierwsza a pozostałe dane to 4 cechy tego irysa.

1. **Uzyskane wyniki**

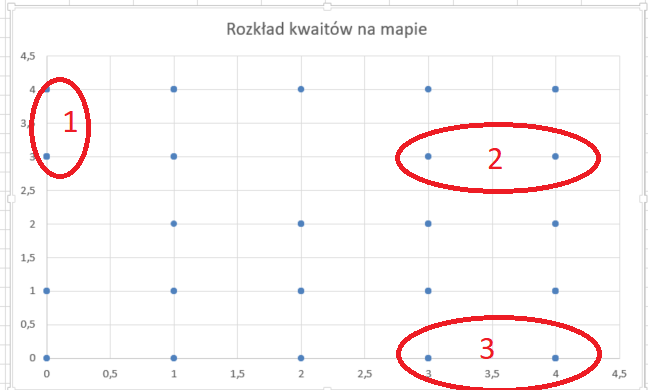
Algorytm Kohonena przedstawionej w sprawozdaniu implementacji uczy się aż osiągnie zadanego błąd. W związku z tym zostaną zobrazowane mapy dla błędu 0.0000001 0.0001 oraz 0.01. Dodatkowym parametrem którym można manipulować jest wielkość mapy, która została ustawiona na wielkość 5x5 oraz 10x10.



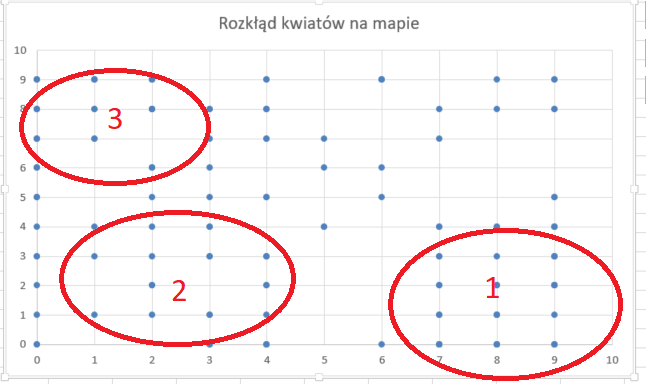
Mapa 1 Uszeregowanie różnych grup kwiatów dla mapy 5x5 z błędem 0.0000001 .



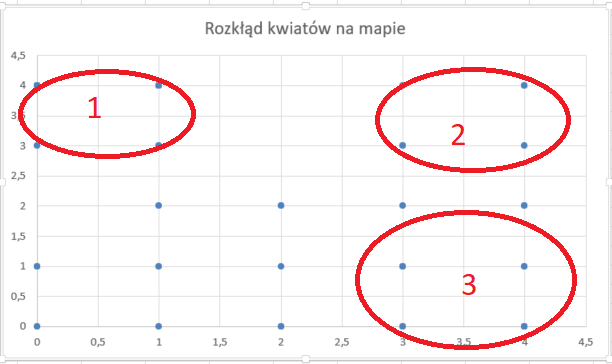
Mapa 2 Uszeregowanie różnych grup kwiatów dla mapy 10x10 z błędem 0.0000001 .



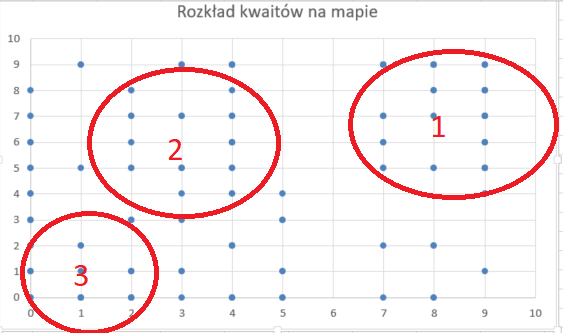
Mapa 3 Uszeregowanie różnych grup kwiatów dla mapy 5x5 z błędem 0.0001.



Mapa 4 Uszeregowanie różnych grup kwiatów dla mapy 10x10 z błędem 0.0001.



Mapa 5 Uszeregowanie różnych grup kwiatów dla mapy 5x5 z błędem 0.01.



Mapa 6 Uszeregowanie różnych grup kwiatów dla mapy 10x10 z błędem 0.01.

1. **Wnioski**

Dzięki przeprowadzonym doświadczeniom można stwierdzić, iż dla 150 rekordów uczących uszeregowanie na mapie 5 na 5 jest niewystarczające. Efektywność mapy Kohonena możemy zobaczyć na mapach które zostały utworzone na bazie większej siatki (10 na 10). Uszeregowanie kwiatków nie jest bezbłędne nawet dla przykładu z najmniejszym błędem (najwyższym współczynnikiem uczenia). Na mapie 2 możemy rozróżnić 3 grupy, skupione w lewym dolnym rogu, prawym górnym oraz prawym dolnym. Niestety dodatkowo występują punkty w lewym górnym rogu, które wydają się być nie przyporządkowane do żadnej grupy. Wraz ze wzrostem błędu można zauważyć wzrost ilości punktów, które trudno uporządkować do jakiejkolwiek grupy irysów.

1. **Literatura**

-wykłady

- <http://galaxy.agh.edu.pl/~vlsi/AI/koho_t/>

- <http://edu.pjwstk.edu.pl/wyklady/nai/scb/wyklad4/w4.htm>

**9. Listing kodu**

using PerceptronLerning.KohonenWTA;

using System;

using System.Collections.Generic;

namespace PerceptronLerning.KohonenMaps

{

class KohonenMap

{

private KohonenNeuron[,] outputs; // Collection of weights.

private int iteration; // Current iteration.

private int length; // Side length of output grid.

private int dimensions; // Number of input dimensions.

private Random rnd = new Random();

private List<string> labels = new List<string>();

private List<double[]> patterns = new List<double[]>();

private KohonenFlowersDataFileReader kohonenDataReader = new KohonenFlowersDataFileReader();

public KohonenMap(int dimensions, int length, string file)

{

this.length = length;

this.dimensions = dimensions;

Initialise();

kohonenDataReader.ConverTextForTestDataArray(file, patterns, labels);

NormalisePatterns();

Train(0.01);

DumpCoordinates();

}

private void Initialise()

{

outputs = new KohonenNeuron[length, length];

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

{

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

{

outputs[i, j] = new KohonenNeuron(i, j, length);

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

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

{

outputs[i, j].Weights[k] = rnd.NextDouble();

}

}

}

}

private void NormalisePatterns()

{

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

{

double sum = 0;

for (int i = 0; i < patterns.Count; i++)

{

sum += patterns[i][j];

}

double average = sum / patterns.Count;

for (int i = 0; i < patterns.Count; i++)

{

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

}

}

}

private void Train(double maxError)

{

double currentError = double.MaxValue;

while (currentError > maxError)

{

currentError = 0;

List<double[]> TrainingSet = new List<double[]>();

foreach (double[] pattern in patterns)

{

TrainingSet.Add(pattern);

}

for (int i = 0; i < patterns.Count; i++)

{

double[] pattern = TrainingSet[rnd.Next(patterns.Count - i)];

currentError += TrainPattern(pattern);

TrainingSet.Remove(pattern);

}

Console.WriteLine(currentError.ToString("0.0000000"));

}

}

private double TrainPattern(double[] pattern)

{

double error = 0;

KohonenNeuron 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 Math.Abs(error / (length \* length));

}

private void DumpCoordinates()

{

for (int i = 0; i < patterns.Count; i++)

{

KohonenNeuron n = Winner(patterns[i]);

Console.WriteLine($"{ labels[i]}, { n.X}, { n.Y}");

}

}

private KohonenNeuron Winner(double[] pattern)

{

KohonenNeuron winner = null;

double min = double.MaxValue;

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

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

{

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

if (d < min)

{

min = d;

winner = outputs[i, j];

}

}

return winner;

}

private double Distance(double[] vector1, double[] vector2)

{

double value = 0;

for (int i = 0; i < vector1.Length; i++)

{

value += Math.Pow((vector1[i] - vector2[i]), 2);

}

return Math.Sqrt(value);

}

}

}

using System;

namespace PerceptronLerning.KohonenMaps

{

class KohonenNeuron

{

public double[] Weights;

public int X;

public int Y;

private int length;

private double nf;

public KohonenNeuron(int x, int y, int length)

{

X = x;

Y = y;

this.length = length;

nf = 1000 / Math.Log(length);

}

private double Gauss(KohonenNeuron win, int it)

{

double distance = Math.Sqrt(Math.Pow(win.X - X, 2) + Math.Pow(win.Y - Y, 2));

return Math.Exp(-Math.Pow(distance, 2) / (Math.Pow(Strength(it), 2)));

}

private double LearningRate(int it)

{

return Math.Exp(-it / 1000) \* 0.1;

}

private double Strength(int it)

{

return Math.Exp(-it / nf) \* length;

}

public double UpdateWeights(double[] pattern, KohonenNeuron winner, int it)

{

double sum = 0;

for (int i = 0; i < Weights.Length; i++)

{

double delta = LearningRate(it) \* Gauss(winner, it) \* (pattern[i] - Weights[i]);

Weights[i] += delta;

sum += delta;

}

return sum / Weights.Length;

}

}

}