**Podstawy Sztucznej Inteligencji**

Scenariusz 6: Budowa ​ i ​ ​działanie ​ ​sieci ​ ​Kohonena ​ ​dla ​ ​WTM

Wszystkie projekty znajdują się w jednym programie na repozytorium

**Wykonał:**

**Bartłomiej Leja**

**Wydział Inżynierii Metali i Informatyki Przemysłowej**

**Informatyka Stosowana**

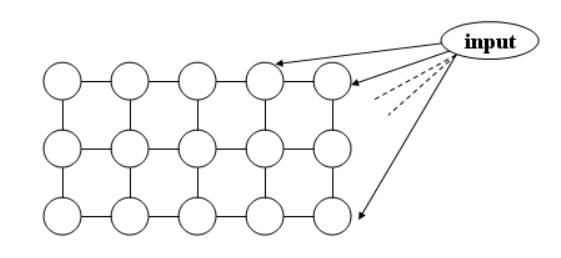
1. **Cel ćwiczenia**

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

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

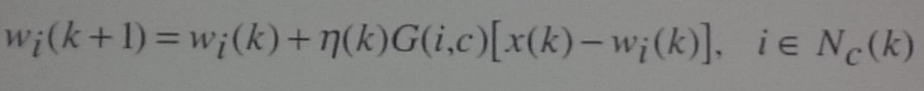
**Zasada działania:**

* 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).

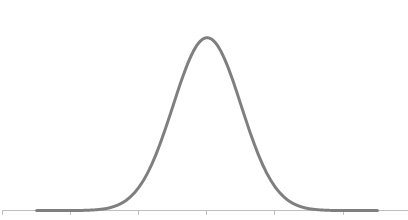
Inicjalizacja wag sieci Kohonena jest losowa. Wektory wejściowe stanowią próbę uczącą, podobnie jak w przypadku zwykłych sieci rozpatrywaną w pętli podczas budowy mapy. Wykorzystanie utworzonej w ten sposób mapy polega na tym, że zbiór obiektów umieszczamy na wejściu sieci i obserwujemy, które węzły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

1. **Algorytm WTM**

Zasada WTM (Winer Takes Most) neuron zwycięski oraz neurony sąsiadujące z neuronem zwycięskim, czyli należące do sąsiedztwa Nc(k), aktualizują swoje wagi według zasady:



Funkcja Gaussa oznaczona G jest odpowiedzialna za wpływ sąsiedztwa.



Rys. 1 Kształt funkcji Gaussa używanej do aktualizacji wag neuronów z otoczenia neuronu zwycięskiego

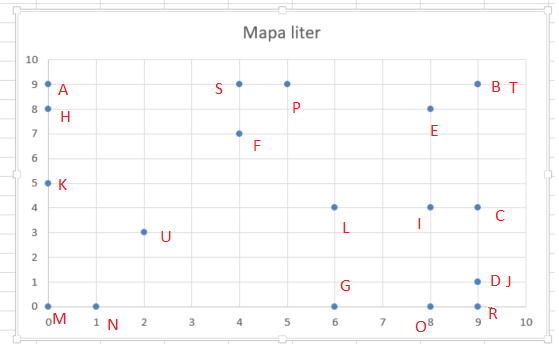
1. **Zestaw danych uczących**

Dane uczące zostały zawarte w pliku kohonenMapLearningData.txt, który zawiera 20 liter polskiego alfabetu w postaci 35 elementowych wektorów utworzonych na bazie matryc 7x5. Dla przykładu litera A to:

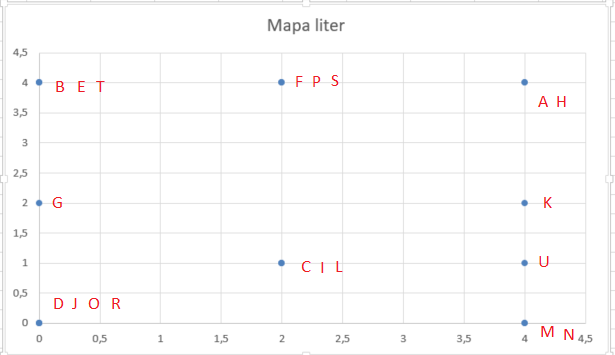
**-1 1 1 1 -1  
1 -1 -1 -1 1  
1 -1 -1 -1 1  
1 1 1 1 1** 🡪 **-1 1 1 1-1 1-1-1-1 1 1-1-1-1 1 1 1 1 1 1 1-1-1-1 1 1-1-1-1 1 1-1-1-1 1  
1 -1 -1 -1 1   
1 -1 -1 -1 1  
1 -1 -1 -1 1**

1. **Uzyskane wyniki**

Algorytm Kohonena, przedstawionej w sprawozdaniu implementacji uczy się aż osiągnie zadany 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 Rozkład liter dla błędu 0.0000001 oraz siatce 10x10



Mapa 2 Rozkład liter dla błędu 0.0000001 oraz siatce 5x5

|  |  |  |
| --- | --- | --- |
| A | 4 | 4 |
| B | 0 | 4 |
| C | 2 | 1 |
| D | 0 | 0 |
| E | 0 | 4 |
| F | 2 | 4 |
| G | 0 | 2 |
| H | 4 | 4 |
| I | 2 | 1 |
| J | 0 | 0 |
| K | 4 | 2 |
| L | 2 | 1 |
| M | 4 | 0 |
| N | 4 | 0 |
| O | 0 | 0 |
| P | 2 | 4 |
| R | 0 | 0 |
| S | 2 | 4 |
| T | 0 | 4 |
| U | 4 | 1 |

Tab. 1 Tabela współrzędnych rozkładu liter dla błędu 0. 0001 oraz siatce 5x5

|  |  |  |
| --- | --- | --- |
| A | 8 | 9 |
| B | 0 | 9 |
| C | 2 | 4 |
| D | 0 | 0 |
| E | 1 | 8 |
| F | 5 | 7 |
| G | 4 | 0 |
| H | 9 | 7 |
| I | 0 | 3 |
| J | 0 | 0 |
| K | 9 | 4 |
| L | 0 | 5 |
| M | 9 | 0 |
| N | 8 | 0 |
| O | 1 | 0 |
| P | 4 | 9 |
| R | 2 | 0 |
| S | 5 | 9 |
| T | 0 | 9 |
| U | 6 | 3 |

Tab. 2 Tabela współrzędnych rozkładu liter dla błędu 0. 0001 oraz siatce 10x10

|  |  |  |
| --- | --- | --- |
| A | 4 | 0 |
| B | 0 | 0 |
| C | 0 | 2 |
| D | 0 | 4 |
| E | 0 | 0 |
| F | 2 | 0 |
| G | 2 | 4 |
| H | 4 | 0 |
| I | 0 | 2 |
| J | 0 | 4 |
| K | 4 | 2 |
| L | 0 | 2 |
| M | 4 | 4 |
| N | 4 | 4 |
| O | 0 | 4 |
| P | 2 | 0 |
| R | 0 | 4 |
| S | 2 | 0 |
| T | 0 | 0 |
| U | 4 | 3 |

Tab. 3 Tabela współrzędnych rozkładu liter dla błędu 0. 01 oraz siatce 5x5

|  |  |  |
| --- | --- | --- |
| A | 0 | 9 |
| B | 9 | 9 |
| C | 8 | 4 |
| D | 9 | 0 |
| E | 8 | 8 |
| F | 4 | 7 |
| G | 6 | 1 |
| H | 0 | 8 |
| I | 9 | 4 |
| J | 9 | 0 |
| K | 0 | 5 |
| L | 9 | 5 |
| M | 1 | 0 |
| N | 0 | 0 |
| O | 8 | 1 |
| P | 5 | 9 |
| R | 9 | 1 |
| S | 4 | 9 |
| T | 9 | 9 |
| U | 2 | 3 |

Tab. 4 Tabela współrzędnych rozkładu liter dla błędu 0. 01 oraz siatce 10x10

1. **Wnioski**

Problem postawiony w wykonywanym laboratorium jest skomplikowany z dwóch powodów. Pierwszym problem jest określenie czy dana litera jest podobna do drugiej (jak określić czy algorytm grupuje poprawnie). Natomiast drugi problem to mała matryca liter która powoduje, że litery różnią się czasami jedynie 2 pozycjami w wektorze. Pomimo tych problemów sporządzono mapy dla błędu 0.0000001 oraz wygenerowano tabele z współrzędnymi z kolejnych prób. Doświadczalnie kolejny raz można stwierdzić, iż czym mniejszy zakładany błąd tym lepsze grupowanie. Dodatkowo przy tak małej grupie rekordów do nauczenia lepsza wydaję się mniejsza siatka (5 na 5). W przykładzie dla mapy 2 mogę zgodzić się, że litery B i E są podobne ale litera T według mnie powinna być inaczej zgrupowana. Litery M i N są prawidłowo zgrupowane. A i H są faktycznie podobne. D bardzo różni się od J. Jak widać algorytm dla grupowania liter nie działa w 100% dokładnie. Możliwe że użycie innej funkcji aktualizującej współczynniki poszczególnych neuronów dało by lepsze wyniki. Funkcja Gaussa została tu użyta ponieważ wydaję się najbardziej naturalna i uniwersalna.

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;

}

}

}