Imię Nazwisko Grupa Michał Słowikowski Gr 4	Temat Scenariusz 6	Data 22.12.2017r.

Celem ćwiczenia było napisanie zapoznanie się z działaniem sieci Kohonena przy wykorzystaniu reguły WTM do nauki istotnych cech liter

Syntetyczny opis algorytmu uczenia

Do wykonania ćwiczenia zbudowałem SOM (self organizing map) z użyciem algorymtmu WTM. Sieć składała się ze zmiennej ilości neuronów, które początkowo przybierały losowe wagi. Przygotowałem 1 zestaw danych wejściowych, na który składało się 20 liter oraz ich zaszumione wersje. Do uczenia wykorzystałem 20 poprawnych natomiast do testowania zaszumione wersje. Mapa składała się z siatki o różnej ilości neuronów, jednak do przedstawiania danych zawsze wypisywana była jako kwadrat.

Sieć Kohonena pomaga reprezentować wielowymiarowe dane w przestrzeni o mniejszym wymiarze. Algorytm polega na tym, że najpierw z zestawu danych losujemy losowy rekord. Następnie szukamy neuronu, który znajduje się “najbliżej” jego, będzie on tzw. “Zwycięzcą” . Obliczanie odległości odbywało się wg. Wzoru:

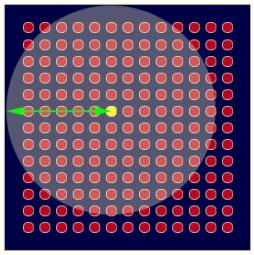
$$Dist = \sqrt{\sum_{i=0}^{i=n} (V_i - W_i)^2}$$

Equation 1

Następnie należało wyznaczyć nowe wagi tego neuronu wg wzoru.

$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) + \eta_i(k)[\mathbf{x} - \mathbf{w}_i(k)]$$

Dla neuronów leżących w sąsiedztwie



$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\lambda}\right) \qquad t = 1, 2, 3...$$

equation 2

Wzór na obliczanie promienia sąsiedztwa

$$W(t+1) = W(t) + \Theta(t)L(t)(V(t) - W(t))$$

Equation 5

Wzór na nowe wagi

$$\Theta(t) = \exp\left(-\frac{dist^2}{2\sigma^2(t)}\right) \qquad t = 1, 2, 3...$$

Equation 6

Wzór zależności wpływu odległości neuronu sąsiedniego.

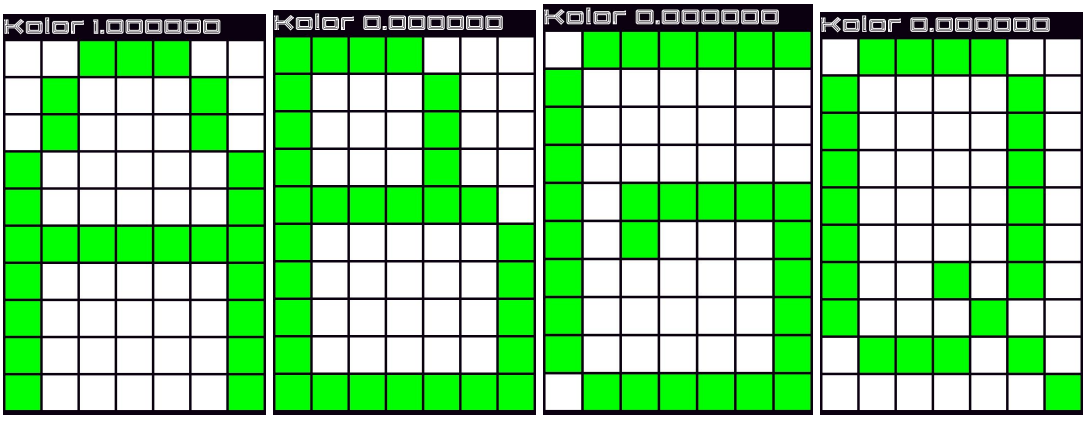
Trzeba również wziąć pod uwagę jego odległość podczas obliczania nowych wag. Bliższy sąsiad powinien zbliżyć się bardziej niż dalszy.

Oprócz zmiany ilości neuronów w sieci, zmiany współczynnika uczenia oraz spotkałem się z modyfikacją polegającą na zmniejszaniu w każdej epoce współczynnika uczenia. Wykonałem testy z i bez tej modyfikacji (z współczynnik uczenia dzieliłem przez 2).

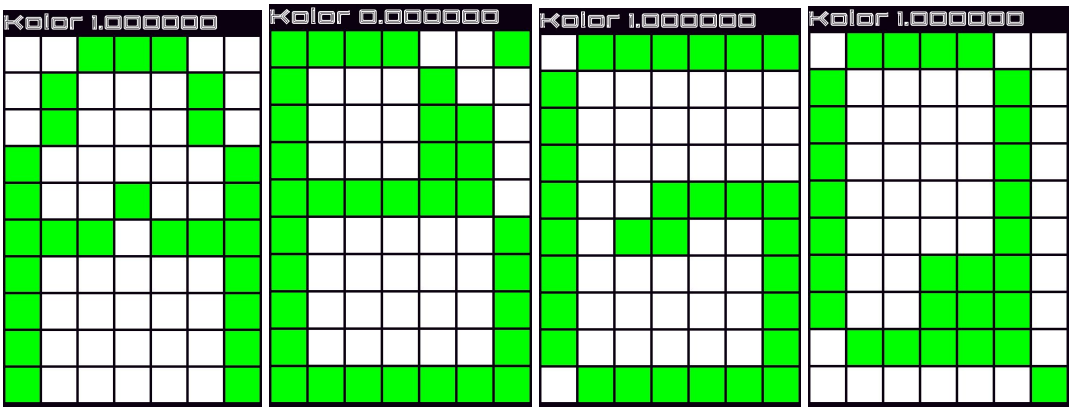
Dane do nauki

Jako dane do nauki przygotowałem litery, które zostały opisane za pomocą tablicy zawierającej piksele. Wielkość jeden litery to 7x10px czyli 70px (w pierwszej wersji użyłem liter z poprzedniego ćwiczenia, jednak zawierały one jedynie 35px przez co sieć nie potrafiła poprawnie rozpoznawać różnic w literach, zwiększenie rozmiaru tablicy znacząco poprawiło wyniki).

Przykładowe dane:



Zaszumione:



Przykładowy output:

```
Neuron: 8 Count: 1
Neuron: 10 Count: 1
Neuron: 11 Count: 1
Neuron: 14 Count: 1
Neuron: 21 Count: 1
Neuron: 23 Count: 1
Neuron: 26 Count: 1
Neuron: 33 Count: 1
Neuron: 39 Count: 1
Neuron: 44 Count: 1
Neuron: 49 Count: 1
Neuron: 52 Count: 3
Neuron: 59 Count: 1
Neuron: 63 Count: 1
Neuron: 81 Count: 1
Neuron: 82 Count: 1
Neuron: 91 Count: 1
Neuron: 93 Count: 1
.       .       .       .       .       .       .       .       O       .
J       T       .       .       C       .       .       .       .       .
.       Q       .       A       .       .       D       .       .       .
.       .       .       F       .       .       .       .       .       K
.       .       .       .       S       .       .       .       .       N
.       .       BEG     .       .       .       .       .       .       P
.       .       .       R       .       .       .       .       .       .
.       .       .       .       .       .       .       .       .       .
.       H       M       .       .       .       .       .       .       .
.       L       .       I       .       .       .       .       .       .
```

Lr = 0.2, ilość neuronów = 100, ilość cykli = 5000, radius = 2

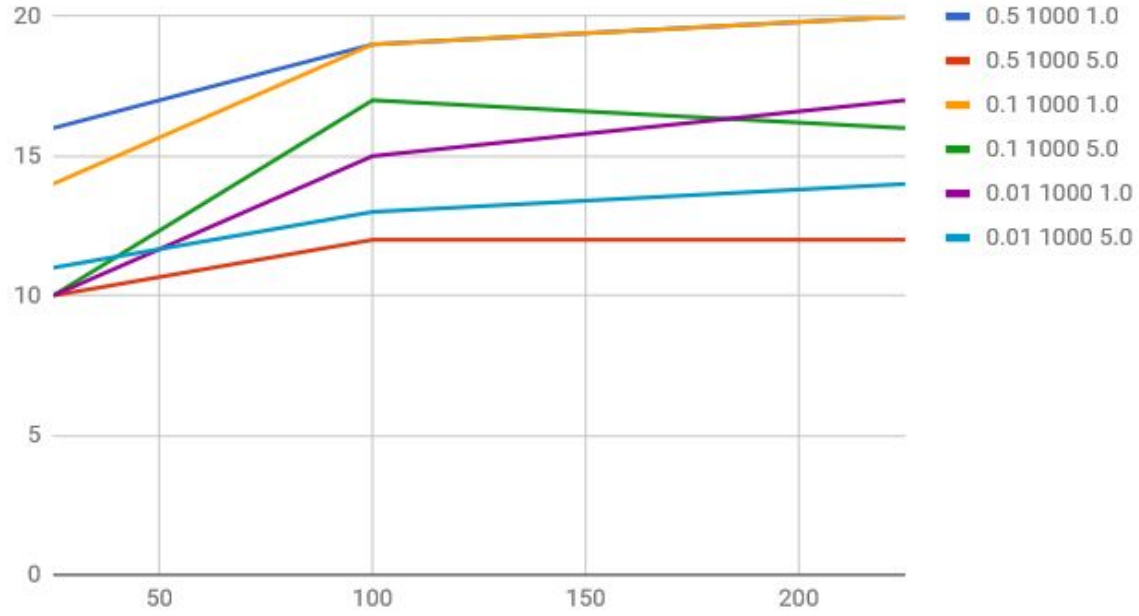
Wyniki

Sprawdzałem zachowanie sieci ze względu na współczynnik uczenia oraz promień sąsiedztwa.

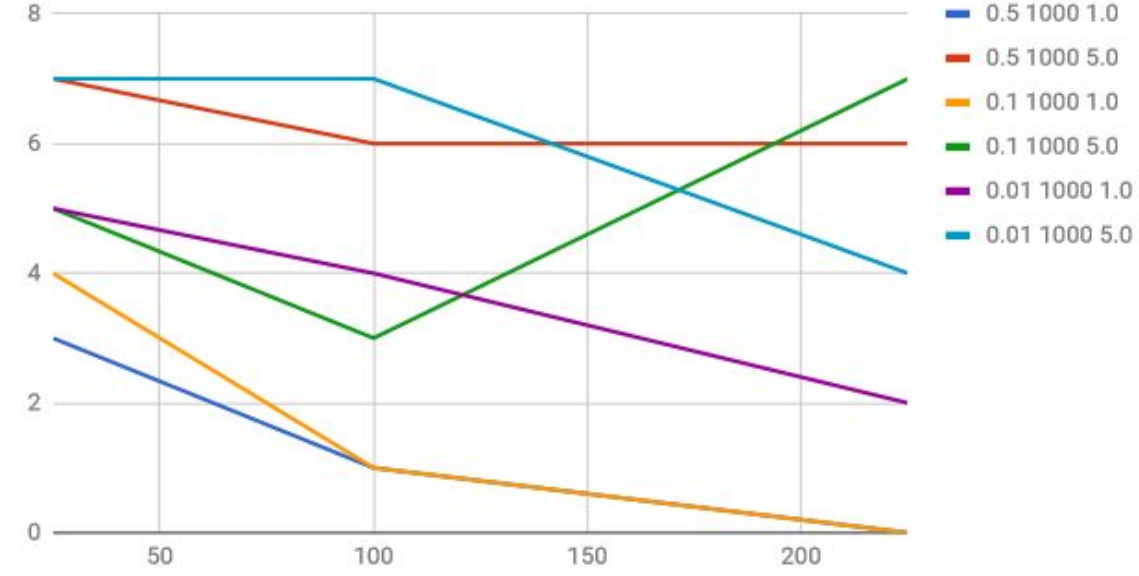
Parametry w legendzie po kolei 0.5 - współczynnik uczenia, 1000- ilość cykli, 1.0 - promień sąsiedztwa

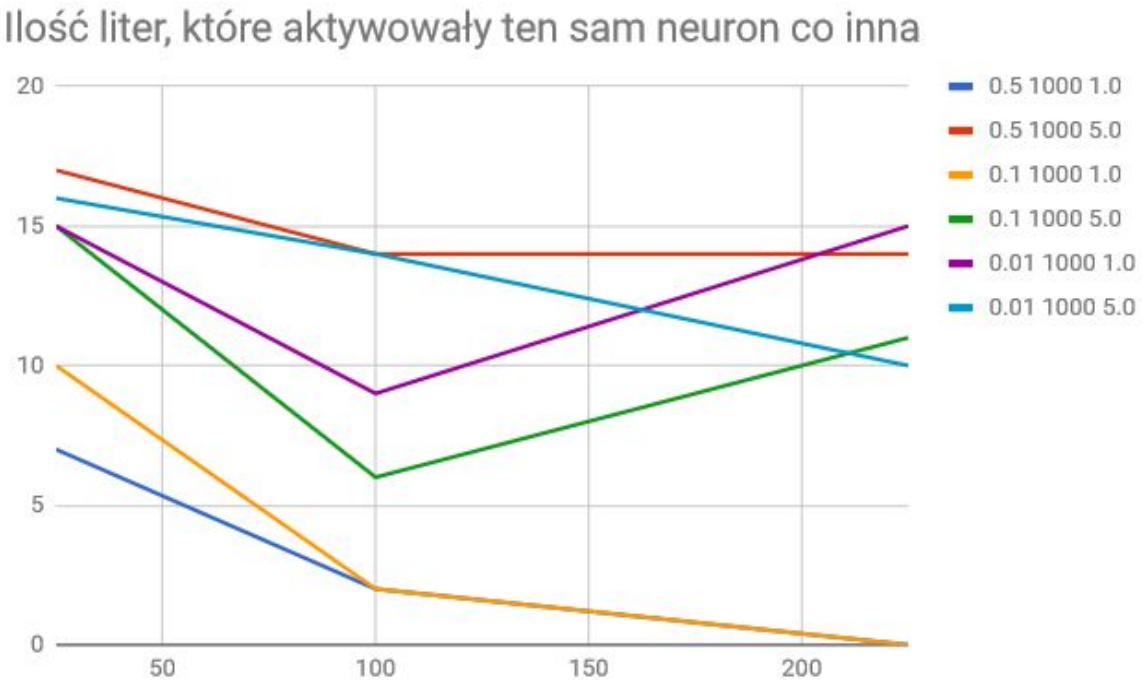
Oś x - ilość neuronów w siatce

Ilość neuronów, które były aktywowane



Ilość neuronów, które były aktywowane przez co najmniej 2 litery





Analiza Wyników

Na podstawie wykresów i porównywania otrzymanych outputów można zauważyć występowanie tworzenia się grup podobnych liter (wiele liter aktywowały jeden neuron).

Ze względu na współczynnik uczenia:
Jeżeli współczynnik uczenia był mniejszy litery miały tendencje do grupowania się w wiele małych grup. Neuron zwykle był aktywowany przez 2, rzadko 3, litery.

```
Neuron: 0 Count: 1
Neuron: 1 Count: 1
Neuron: 2 Count: 1
Neuron: 3 Count: 3
Neuron: 4 Count: 2
Neuron: 5 Count: 1
Neuron: 6 Count: 3
Neuron: 7 Count: 1
Neuron: 8 Count: 7
S       B       C
HMN     IT      Q
AOP     J       DEFGKLR
```

```
Neuron: 0 Count: 10
Neuron: 2 Count: 10
ACFIJKPQRT      .       BDEGHLMNOS
.       .       .
.       .       .
```

Ekstremalny przypadek dla 9 neuronów (po lewej lr = 0.5, po prawej = 0.01)

Ze względu na promień:
Wyniki dla promienia równego 1.0 były znacząco lepsze niż dla promienia równego 5.0. Większy promień powodował przesunięcie większej ilości neuronów. Powodowało to, że znacznie mniej neuronów było aktywowanych w porównaniu do wersji z mniejszym promieniem.

Wnioski

- Podobne litery często aktywowały ten sam neuron (np. H i N)
- Bardzo ważny był współczynnik uczenia. Mniejszy współczynnik uczenia powodował tworzenie się grup
- Większy promień powodował, że podczas testów aktywowało się mniej neuronów
- Można było poprawić jakość sieci zwiększając ilość danych (np utworzyć zaszumione dane, które służyłyby do nauki)

Kod

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Scen4ver2
{
    class Program
    {
        static void Main(string[] args)
        {
            Map m = new Map(0.01, 9, 1000, 2);
            m.Learn();
  //          m.Test(DataProvider.input);
  //          m.PrintMap(DataProvider.input);
            m.Test(DataProvider.testInput);
            m.PrintMap(DataProvider.testInput);
            Console.ReadLine();
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Scen4ver2
{
    class Neuron
    {
        double[] weights;
        public Neuron(double[] weights)                      //kopiowanie tablicy z wagami
        {
            this.weights = new double[weights.Length];
            Array.Copy(weights, this.weights, weights.Length);
        }

        public double CalculateEuclideanDistance(double[] input)    //obliczanie dystansu
        {
            double sum = 0.0;
            for (int i = 0; i < input.Length; i++)
            {
                sum += Math.Pow((input[i] - weights[i]), 2);
            }
            sum += Math.Pow((1 - weights[input.Length]), 2);
            return Math.Sqrt(sum);
        }

        public void PrintWeights()
        {
            foreach (double w in weights)
            {
                Console.WriteLine(w);
            }
        }

        public double[] GetWeights()
        {
            return weights;
        }

        public void CalculateNewWeights(double[] input, double lr)  //obliczanie nowych wag dla wygrywającego neuronu
        {
            for (int i = 0; i < input.Length; i++)
            {
                weights[i] += lr * (input[i] - weights[i]);
            }

            weights[input.Length] += lr * (1 - weights[input.Length]);
        }

        public void CalculateNewWeights(double[] input, double lr, double theta)    //obliczanie nowych wag dla neuronów znajdujących się w sąsiedztwie
        {
            for (int i = 0; i < input.Length; i++)
            {
                weights[i] += lr * theta * (input[i] - weights[i]);
            }

            weights[input.Length] += lr * theta * (1 - weights[input.Length]);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Scen4ver2
{
    class DataProvider
    {
        public static char[] Character = new char[] { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T' };
        public static double[,] input = new double[,]{
            {
                0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0,
                0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0,
                0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            }, //A
            {
                1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
                1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
            }, //B
            {
                0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0,
                0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0,
                0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0,
                0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0
            }, //C
            {
                1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
                1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0
            }, //D
            {
                1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
            }, //E
            {
                1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0,
```

```
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0
        }   //T
    };
    public static double[,] testInput = new double[,]{
        {
            0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
        },  //A
        {
            1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
        },  //B
        {
            0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0,
            0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0
        },  //C
        {
            1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0
        },  //D
        {
            1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
        },  //E
        {
            1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```

```
                1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
                0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
                0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0
            },  //S
            {
                1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
                0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0
            }   //T
        };

        public static double[] GetInput(int number, double[,] data)
        {
            double[] result = new double[data.GetLength(1)];

            for (int i = 0; i < result.Length; i++)
            {
                result[i] = data[number, i];
            }

            return result;
        }
        public static void RestoreInputToCorrectOrder()...
        public static void ShuffleInputData()
        {
            int inputDataCount = input.GetLength(0);
            int inputDataAttributionsCount = input.GetLength(1);
            Random r = new Random();
            int place;

            double[] tmp = new double[inputDataAttributionsCount];
            for (int i = 0; i < inputDataCount; i++)
            {

                for (int j = 0; j < inputDataAttributionsCount; j++)
                {
                    tmp[j] = input[i, j];
                }

                place = r.Next(i, inputDataCount);

                for (int j = 0; j < inputDataAttributionsCount; j++)
                {
                    input[i, j] = input[place, j];
                }

                for (int j = 0; j < inputDataAttributionsCount; j++)
                {
                    input[place, j] = tmp[j];
                }
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Scen4ver2
{
    class Map
    {
        private Neuron[] neuronMap;
        private int numberOfNeurons;
        private double lr;
        private double Max;
        private double neighborRadius;
        private double lambda;
        public Map(double learningRate, int numberOfNeurons, int iter, double neighbor)        //ustawienie wartości
        {
            this.lambda = iter / Math.Log(neighbor);
            this.neighborRadius = neighbor;
            this.Max = iter;
            this.numberOfNeurons = numberOfNeurons;
            lr = learningRate;
            neuronMap = new Neuron[this.numberOfNeurons];
            Random r = new Random();
            double[] weights = new double[DataProvider.input.GetLength(1) + 1];
            int size = (int)Math.Sqrt(numberOfNeurons);
            double x = 0;
            double y = 0;
            for (int j = 0; j < numberOfNeurons; j++)
            {

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

                        weights[i] = r.NextDouble();

                    }
                    neuronMap[j] = new Neuron(weights);
                }
            }

        public void PrintMap()
        {
            for (int i = 0; i < neuronMap.Length; i++)
            {
                Console.WriteLine("Neuron : " + i);
                neuronMap[i].PrintWeights();
            }
        }

        public void Learn()             //nauka
        {

            double lenght;
            double min;
            int counter = 0;
            int neuronNumber = 0;
            do
            {
                DataProvider.ShuffleInputData();                            //wymieszaj dane wejściowe
                for (int i = 0; i < DataProvider.input.GetLength(0); i++)
                {
                    double[] input = DataProvider.GetInput(i, DataProvider.input);
                    min = 0.0;
                    for (int j = 0; j < neuronMap.Length; j++)                //szukanie neuronu najbliższego
                    {
                        lenght = neuronMap[j].CalculateEuclideanDistance(input);
                        if (lenght < min || j == 0)
                        {
                            min = lenght;
                            neuronNumber = j;
                        }
                    }
                    neuronMap[neuronNumber].CalculateNewWeights(input, lr); //obliczanie wag neuronu
                }
                counter++;
                CalculateLearningRate();                    //obliczanie nowego lr
                CalculateNeighborRadius(counter);               //obliczanie nowego promienia
            } while (Max > counter);
            DataProvider.RestoreInputToCorrectOrder();
        }
```

```
private void CalculateNewWeights(int neuronNumber, double[] input)          //obliczanie wag neuronu wygrywającego i sąsiadów
{
    neuronMap[neuronNumber].CalculateNewWeights(input, lr);     //obliczanie wag zwycięzcy
    for (int j = 0; j < neuronMap.Length; j++)                    //szukanie sąsiadów
    {
        if(j!=neuronNumber)
        {
            double[] w = new double[70];
            Array.Copy(neuronMap[neuronNumber].GetWeights(), w, 70);
            double distance = neuronMap[j].CalculateEuclideanDistance(w);
            if (distance <= neighborRadius)                                     //gdy znajdzie sąsiada oblicza jego nowe wagi
            {
                double theta = CalculateTheta(distance);
                CalculateNewWeights(neuronNumber, input);
            }
        }

    }
}

private double CalculateTheta(double distance)                      //obliczanie wartości theta (uwzględniana w wagach sąsiada)
{
    return Math.Exp(-(Math.Pow(distance, 2) / (2 * Math.Pow(neighborRadius, 2))));
}

public void Test(double[,] inputArray)                  //testowanie mapy
{
    int neuronNumber;
    int[] responseCounter = new int[neuronMap.Length];
    responseCounter.Select(i => 0).ToArray();

    for (int i = 0; i < inputArray.GetLength(0); i++)
    {
        double[] input = DataProvider.GetInput(i, inputArray);
        neuronNumber = ClassifyInput(input);
        responseCounter[neuronNumber]++;
        Console.WriteLine(DataProvider.Character[i] + " Got: " + neuronNumber);
    }
    int n;
    for (int i = 0; i < responseCounter.Length; i++)
    {
        n = responseCounter[i];
        if (n != 0)
            Console.WriteLine("Neuron: " + i + " Count: " + n);
    }
}

private void CalculateNeighborRadius(int epoch)             //zmiana
{
    neighborRadius = neighborRadius * Math.Exp(-epoch / lambda);
}

public void PrintMap(double[,] inputArray)                  //wyświetl nową mapę
{
    int size = inputArray.GetLength(0) / DataProvider.Character.Length;
    int mapS = (int)Math.Sqrt(numberOfNeurons);
    int beg = 0;
    int end = size;
    String[] map = new String[numberOfNeurons];
    for (int i = 0; i < map.Length; i++)
    {
        map[i] = "";
    }

    for (int i = 0; i < inputArray.GetLength(0); i++)
    {
        beg = i * size;
        end = beg + size;
        map = calculateMap(map, beg, end, DataProvider.Character[i].ToString(), inputArray);
    }

    for (int i = 0; i < mapS; i++)
    {
        for (int j = 0; j < mapS; j++)
        {
            int k = i * mapS + j;
            if (map[k] == "")
            {
                Console.Write(".\t");
            }
            else
            {
                Console.Write(map[k] + "\t");
            }
        }
        Console.WriteLine();
    }
}
```

```
    private String[] calculateMap(String[] map, int beg, int end, String type, double[,] inputArray)       //metoda do wyświetlania która litera aktywuje który neuron w mapie
    {
        int nn;
        for (int i = beg; i < end; i++)
        {
            double[] input = DataProvider.GetInput(i, inputArray);
            nn = ClassifyInput(input);
            map[nn] += type;
        }
        return map;
    }

    private void CalculateLearningRate()            //obliczanie lr
    {
        lr /= 2;
    }

    private int ClassifyInput(double[] input)          //klasyfikacja który neuron zostanie aktywowany dla tego inputu
    {
        int classification = 0;
        double lenght;
        double min = 0.0;
        for (int j = 0; j < neuronMap.Length; j++)
        {
            lenght = neuronMap[j].CalculateEuclideanDistance(input);
            if (lenght < min || j == 0)
            {
                min = lenght;
                classification = j;
            }
        }
        return classification;
    }
}
```