**PODSTAWY SZUCZNEJ INTELIGENCJI**

Scenariusz 6

Maria Podkalicka, IS grupa 3

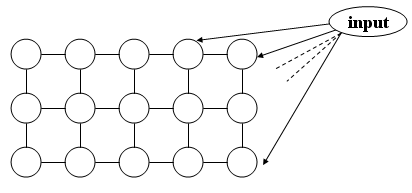
**Temat sieci:** Budowa i działanie sieci Kohonena dla WTM.

**Cel ćwiczenia**: Poznanie budowy i działania sieci Kohonena przy wykorzystaniu reguły WTM do odwzorowywania istotnych cech liter alfabetu.

**Opis sieci**

**Sieć Kohonena** – [sieć neuronowa](https://pl.wikipedia.org/wiki/Sie%C4%87_neuronowa) uczona w trybie bez nauczyciela w celu wytworzenia niskowymiarowej (przeważnie dwuwymiarowej) [zdyskretyzowanej](https://pl.wikipedia.org/wiki/Matematyka_dyskretna) reprezentacji przestrzeni wejściowej. Sieć Kohonena wyróżnia się tym od innych sieci, że zachowuje odwzorowanie sąsiedztwa przestrzeni wejściowej.

Topologia sieci Kohonena odpowiada topologii docelowej przestrzeni. Jeśli np. chcemy prezentować wynik na ekranie, rozsądnym modelem jest prostokątna siatka wezłów (im więcej, tym wyższą rozdzielczość będzie miała mapa):



Zasady działania sieci Kohonena:

* 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
* 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.

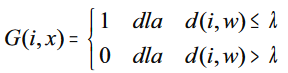
Inicjalizacja wag sieci Kohonena jest losowa. Wektory wejściowe stanowią próbę uczącą, podobnie jak w przypadku zwykłych sieci rozpatrywaną w pętli podczas budowy mapy. Wykorzystanie utworzonej w ten sposób mapy polega na tym, że zbiór obiektów umieszczamy na wejściu sieci i obserwujemy, które węzły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

**Algorytm WTM**

W metodzie WTA tylko zwycięski neuron ma prawo adaptacji swoich wag. Natomiast w algorytmie WTM(Winner Takes Most) oprócz zwycięskiego neuronu zmianie podlegają również wagi jego sąsiadów według reguły:

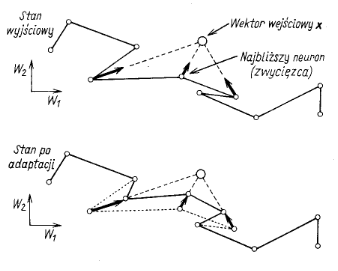


gdzie G(x,i) jest funkcją jego sąsiedztwa. W klasycznej sieci Kohonena funkcja G(x,i) zdefiniowana jest następująco:



gdzie d(i,w) jest odległością pomiędzy neuronami w przestrzeni sieci tzn. na siatce(a nie w przestrzeni danych wejściowych), a lambda jest promieniem sąsiedztwa malejącym do zera w trakcie nauki. Jest to tak zwane sąsiedztwo prostokątne.

Ilustracja procesu adaptacji wag w algorytmie WTM:



**Dane uczące i testujące**

Przygotowany przeze mnie zestaw uczący zawiera 20 dużych liter alfabetu łacińskiego utworzonych na matrycy 5x5 czyli na 25 polach.

Przykład : litera E

1 1 1 1 1  
1 0 0 0 0  
1 1 1 1 1  
1 0 0 0 0  
1 1 1 1 1

Liniowym rozwinięciem tych matryc będą wektory wejściowe ciągu uczącego. Przykładowy wektory uczące :

|  |
| --- |
| A = 1,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,0,0,0,1,1,0,0,0,1,A |
|  |

|  |
| --- |
| B = 1,1,1,0,0,1,0,0,1,0,1,1,1,0,0,1,0,0,1,0,1,1,1,0,0,B |
|  |

|  |
| --- |
| C = 1,1,1,1,1,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,1,1,1,C |
|  |

|  |
| --- |
| D = 1,1,1,0,0,1,0,1,0,0,1,0,1,0,0,1,0,1,0,0,1,1,1,0,0,D |
|  |

|  |
| --- |
| E = 1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,E |
|  |

F = 1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1,0,0,0,0,F

Na końcu podana została litera, która odpowiada danemu wektorowi. 1 oznacza że w danym punkcie istnieje zawartość, natomiast wartość 0 oznacza, że pole jest puste.

Dane testujące zawierają dwa ciągi liczb, które odpowiadają literom D i F:

|  |
| --- |
| 1,1,1,0,0,1,0,1,0,0,1,0,1,0,0,1,0,1,0,0,1,1,1,0,0 |
|  |

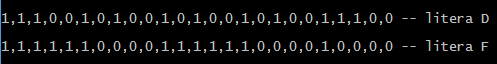
|  |
| --- |
|  |
|  |

1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1,0,0,0,0

**Analiza i zestawienie otrzymanych wyników**

W programie użyłam ogólno dostępnej biblioteki do tworzenia sztucznych sieci neuronowych w języku java script.

Na podstawie danych testujących program bada jaka litera odpowiada podanemu ciągowi liczb 0 i 1.



Na wyżej widocznym zdjęciu widać, że sieć została poprawnie nauczona i litery zostały poprawnie rozpoznane. Uczenie zostało przeprowadzone dla różnych współczynników uczenia. Współczynnik uczenia może przyjmować wartości z przedziału (0,1). Im większa wartość współczynnika uczenia tym mniejsza liczba iteracji, co ma wpływ na działanie programu i powoduje uzyskanie błędnych wyników. Aby zredukować liczbę błędnych wyników można zwiększyć liczbę iteracji.

Powyżej wykres który przedstawia odległość od sąsiada dla współczynnika uczenia 0.1 i dokładności 0.01

**Wnioski**

Zadanie zostało rozwiązane poprzez wykorzystanie algorytmu WTM, ponieważ algorytmy WTA, w których tylko jeden neuron może podlegać adaptacji w każdej iteracji, są algorytmami słabo zbieżnymi, szczególnie przy dużej liczbie neuronów. Dlatego w praktyce zostały one zastąpione algorytmami WTM (*Winner Takes Most*)*,* w których oprócz zwycięzcy uaktualniają swoje wagi również neurony z jego sąsiedztwa, przy czym im dalsza jest odległość od zwycięzcy, tym mniejsza jest zmiana wartości wag neuronu.

Dla sieci samoorganizującej Kohonena ilość danych uczących jest ważna – przy zbyt małej można otrzymać błędne wyniki. W programie 20 wektorów odpowiadających 20 dużym literom alfabetu łacińskiego okazała się wystarczająca i sieć została poprawnie nauczona.

Ważnym elementem dla zasady WTM jest określenie tzw. sąsiedztwa. Jest ono rozumiane w sensie geometrycznym jako położenie neuronu względem zwycięzcy. Zasięg sąsiedztwa określany jest jako tzw. promień sąsiedztwa. Przykładowo sąsiedztwo o promieniu zerowym obejmuje tylko jeden neuron, tj. zwycięzcę. Promień wynoszący jeden wyznacza obszar sąsiedztwa zawierający zwycięzcę i jego bezpośrednich sąsiadów

Listing kodu:

|  |
| --- |
| var synaptic = require('synaptic'); |
|  |

|  |
| --- |
| var fs = require('fs'); |
|  |

|  |
| --- |
| var Layer = synaptic.Layer, |
|  |

|  |
| --- |
| Network = synaptic.Network; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| var inputLayer = new Layer(25); |
|  |

|  |
| --- |
| var hiddenLayer = new Layer(5); |
|  |

|  |
| --- |
| var outputLayer = new Layer(20); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| inputLayer.project(hiddenLayer); |
|  |

|  |
| --- |
| hiddenLayer.project(outputLayer); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| var network = new Network({ |
|  |

|  |
| --- |
| input: inputLayer, |
|  |

|  |
| --- |
| hidden: [hiddenLayer], |
|  |

|  |
| --- |
| output: outputLayer |
|  |

|  |
| --- |
| }); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| var som = require('node-som'); |
|  |

|  |
| --- |
| var somInstance = new som({ |
|  |

|  |
| --- |
| inputLength: 25, |
|  |

|  |
| --- |
| maxClusters: 10, |
|  |

|  |
| --- |
| loggingEnabled:false, |
|  |

|  |
| --- |
| target: 0.1, |
|  |

|  |
| --- |
| scale:{ |
|  |

|  |
| --- |
| min:0, |
|  |

|  |
| --- |
| max:10 |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| }); |
|  |

|  |
| --- |
| somInstance.train(trainingData); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| var sample = [0,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,0,0,0,1,1,0,0,0,1]; |
|  |

|  |
| --- |
| var group = somInstance.classify(sample); |
|  |

|  |
| --- |
| console.log(getFlowerName2(group)); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| var trainingData = []; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| fs.readFile('learning\_data.txt','utf8',function(err, data){ |
|  |

|  |
| --- |
| if(err) throw err; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| var lines = data.split("\n"); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| for(var i = 0; i < lines.length; i++){ |
|  |

|  |
| --- |
| var line = lines[i].trim(); |
|  |

|  |
| --- |
| var splitLine = line.split(","); |
|  |

|  |
| --- |
| var input = splitLine.slice(0, 25); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| var output = splitLine[25]=='A' ? [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'B' ? [0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'C' ? [0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'D' ? [0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'E' ? [0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'F' ? [0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'G' ? [0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'H' ? [0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'I' ? [0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'J' ? [0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'K' ? [0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'L' ? [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'M' ? [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'N' ? [0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'O' ? [0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'P' ? [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'R' ? [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'S' ? [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0] |
|  |

|  |
| --- |
| : splitLine[25] == 'T' ? [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0] |
|  |

|  |
| --- |
| : [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1]; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| trainingData.push({ |
|  |

|  |
| --- |
| input: input, |
|  |

|  |
| --- |
| output: output |
|  |

|  |
| --- |
| }); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| var learningRate = .01; |
|  |

|  |
| --- |
| for (var i = 1; i <= 10000; i++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| for(var j = 0; j < trainingData.length; j++){ |
|  |

|  |
| --- |
| network.activate(trainingData[j].input); |
|  |

|  |
| --- |
| network.propagate(learningRate, trainingData[j].output); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| // if(i%1000 == 0) |
|  |

|  |
| --- |
| // console.log("Training... "+i/100+"% complete. "); |
|  |

|  |
| --- |
| //console.log(network.log); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| fs.readFile('test\_data.txt','utf8',function(err, data){ |
|  |

|  |
| --- |
| if(err) throw err; |
|  |

|  |
| --- |
| console.log("\n\nResults\n===============================\n"); |
|  |

|  |
| --- |
| var lines = data.split("\n"); |
|  |

|  |
| --- |
| for(var i = 0; i < lines.length; i++){ |
|  |

|  |
| --- |
| var input = lines[i].trim().split(","); |
|  |

|  |
| --- |
| var result = getFlowerName(network.activate(input)); |
|  |

|  |
| --- |
| console.log(lines[i].trim()+" => "+result); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| }); |
|  |

|  |
| --- |
| }); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| function getLargestIndex(arr){ |
|  |

|  |
| --- |
| var result = 0; |
|  |

|  |
| --- |
| for(var i = 1; i < arr.length; i++) |
|  |

|  |
| --- |
| if(arr[i] > arr[result]) |
|  |

|  |
| --- |
| result = i; |
|  |

|  |
| --- |
| return result; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| function getFlowerName(arr){ |
|  |

|  |
| --- |
| var index = getLargestIndex(arr); |
|  |

|  |
| --- |
| if(index == 0) return "A"; |
|  |

|  |
| --- |
| if(index == 1) return "B"; |
|  |

|  |
| --- |
| if(index == 2) return "C"; |
|  |

|  |
| --- |
| if(index == 3) return "D"; |
|  |

|  |
| --- |
| if(index == 4) return "E"; |
|  |

|  |
| --- |
| if(index == 5) return "F"; |
|  |

|  |
| --- |
| if(index == 6) return "G"; |
|  |

|  |
| --- |
| if(index == 7) return "H"; |
|  |

|  |
| --- |
| if(index == 8) return "I"; |
|  |

|  |
| --- |
| if(index == 9) return "J"; |
|  |

|  |
| --- |
| if(index == 10) return "K"; |
|  |

|  |
| --- |
| if(index == 11) return "L"; |
|  |

|  |
| --- |
| if(index == 12) return "M"; |
|  |

|  |
| --- |
| if(index == 13) return "N"; |
|  |

|  |
| --- |
| if(index == 14) return "O"; |
|  |

|  |
| --- |
| if(index == 15) return "P"; |
|  |

|  |
| --- |
| if(index == 16) return "R"; |
|  |

|  |
| --- |
| if(index == 17) return "S"; |
|  |

|  |
| --- |
| if(index == 18) return "T"; |
|  |

|  |
| --- |
| return "U"; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| function getFlowerName2(index){ |
|  |

|  |
| --- |
| if(index == 0) return "A"; |
|  |

|  |
| --- |
| if(index == 1) return "B"; |
|  |

|  |
| --- |
| if(index == 2) return "C"; |
|  |

|  |
| --- |
| if(index == 3) return "D"; |
|  |

|  |
| --- |
| if(index == 4) return "E"; |
|  |

|  |
| --- |
| if(index == 5) return "F"; |
|  |

|  |
| --- |
| if(index == 6) return "G"; |
|  |

|  |
| --- |
| if(index == 7) return "H"; |
|  |

|  |
| --- |
| if(index == 8) return "I"; |
|  |

|  |
| --- |
| if(index == 9) return "J"; |
|  |

|  |
| --- |
| if(index == 10) return "K"; |
|  |

|  |
| --- |
| if(index == 11) return "L"; |
|  |

|  |
| --- |
| if(index == 12) return "M"; |
|  |

|  |
| --- |
| if(index == 13) return "N"; |
|  |

|  |
| --- |
| if(index == 14) return "O"; |
|  |

|  |
| --- |
| if(index == 15) return "P"; |
|  |

|  |
| --- |
| if(index == 16) return "R"; |
|  |

|  |
| --- |
| if(index == 17) return "S"; |
|  |

|  |
| --- |
| if(index == 18) return "T"; |
|  |

|  |
| --- |
| return "U"; |
|  |

|  |
| --- |
| } |
|  |