**Sprawozdanie z projektu nr 4**

Monika Pawlik

**Temat ćwiczenia:**  
Uczenie sieci regułą Hebba.

**Cel ćwiczenia:**  
Celem ćwiczenia jest poznanie działania reguły Hebba dla sieci jednowarstwowej na przykładzie grupowania liter alfabetu.

1. **Syntetyczny opis budowy sieci jednowarstwowej**

Sieci jednowarstwowe – neurony w tej sieci ułożone są w jednej warstwie, zasilane jedynie z węzłów wejściowych. Węzły wejściowe nie tworzą warstwy neuronowej, ponieważ nie zachodzi w nich proces obliczeniowy.

Działanie sieci polega na tym, że pojawienie się określonego wektora wejściowego X powoduje powstanie sygnałów wyjściowych ym na wszystkich neuronach wchodzących w skład rozważanej warstwy. Oczekujemy przy tym maksymalnego sygnału wyjściowego ym na tym neuronie, którego wektor wag W(m) najbardziej przypomina X. Sieć tego typu może więc rozpoznawać k różnych klas obiektów, gdyż każdy neuron zapamiętuje jeden wzorcowy obiekt, na którego pojawienie się jest uczulony. Klasy obiektów jako wzorce są w postaci wektorów wag W(m).

Warstwę neuronów, z których każdy ma ten sam zestaw sygnałów wejściowych (wektor sygnałów):

𝑋=<𝑥1,𝑥2,…,𝑥𝑛>𝑇

Ponumerowane neurony mają wektor wag: 𝑊(𝑚)=<𝑤1(𝑚),𝑤2(𝑚),…,𝑤𝑛(𝑚) >𝑇

Sygnał wyjściowy m-tego neurony można wyznaczyć ze wzoru:

𝑦𝑚= 𝑊(𝑚)∗𝑋 =∑ 𝑤𝑖(𝑚)∗𝑥𝑖

1. **Algorytm Hebba**

Jest to jedna z najpopularniejszych metod samouczenia sieci neuronowych. Polega ona na tym, że sieci pokazuje się kolejne przykłady sygnałów wejściowych, nie podając żadnych informacji o tym, co z tymi sygnałami należy zrobić. Sieć obserwuje otoczenie i odbiera różne sygnały, nikt nie określa jednak, jakie znaczenie mają pokazujące się obiekty i jakie są pomiędzy nimi zależności. Sieć na podstawie obserwacji występujących sygnałów stopniowo sama odkrywa, jakie jest ich znaczenie i również sama ustala zachodzące między sygnałami zależności.

Po podaniu do sieci neuronowej każdego kolejnego zestawu sygnałów wejściowych tworzy się w tej sieci pewien rozkład sygnałów wyjściowych - niektóre neurony sieci są pobudzone bardzo silnie, inne słabiej, a jeszcze inne mają sygnały wyjściowe wręcz ujemne.

**Algorytm według którego przeprowadza się modyfikację wag:**

**wi(t+1) = wi(t) + nyxi** Oznaczenia:

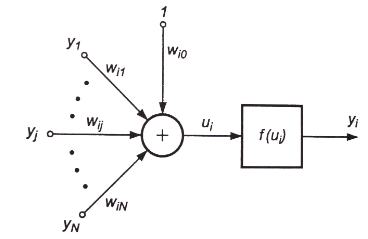
i – numer wagi neuronu  
t – numer iteracji w epoce  
y – sygnał wyjściowy neuronu  
x – wartość wejściowa neuronu  
n – współczynnik uczenia (0,1)

Uczenie neuronu z zastosowaniem reguły Hebba może odbywać się w trybie bez nauczyciela i z nauczycielem.

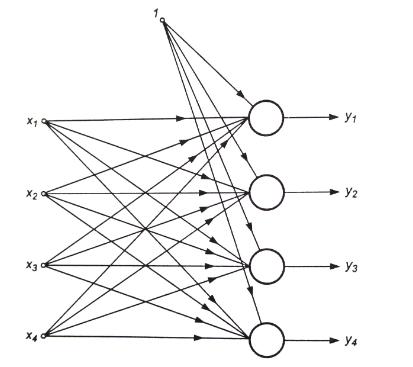
**Metoda uczenia bez nauczyciela** : w przypadku pojedynczego neuronu w trakcie uczenia będziemy modyfikować wartość wag proporcjonalnie zarówno do wartości sygnału podanego na i-te wejście, jak i sygnału wyjściowego y z uwzględnieniem współczynnika uczenia, nie podajemy wzorcowej wartości wyjściowej.

**Metoda uczenia z nauczycielem** : modyfikacja algorytmu - wi(t+1) = wi(t) + n**d**xi, d – sygnał wzorcowy

Ogólny model neuronu Hebba odpowiada standardowej postaci modelu neuronu:



**Rys. Model neuronu w oparciu o regułę Hebba**



**Rys. Schemat sieci Hebba**

1. **Wygenerowanie danych uczących i testujących**

Zestaw danych uczących zawiera 20 dużych liter alfabetu łacińskiego w postaci dwuwymiarowej tablicy.  
Przykładowe dane:

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

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

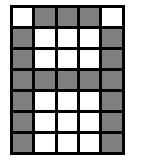
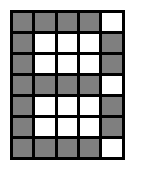
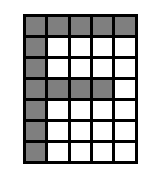
1 0 0 0 1 1 0 0 0 1 1 0 0 0 0

1 1 1 1 1 A 1 1 1 1 0 B 1 1 1 1 0 F

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

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

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



Białym pikselom jest przyporządkowana liczba - 0, szarym - 1.

Otrzymane zostały następujące wektory ciągu uczącego: (przykład: wybrane wektory liter A, B, i F)

x(A) = (0,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,1,1,0,0,0,1,1,0,0,0,1,0,0,0,1)

x(B) = (1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,0)

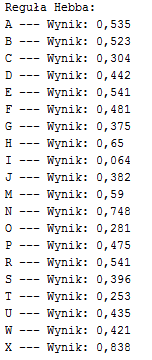
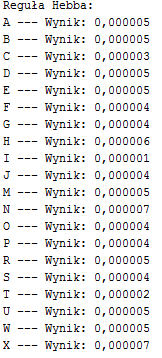
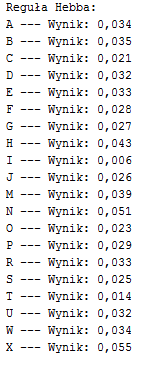
x(F) = (1,1,1,1,1,1,0,0,0,0,1,0,0,0,0,1,1,1,1,0,1,0,0,0,0,1,0,0,0,1,0,0,0,0)

1. **Analiza uczenia sieci**

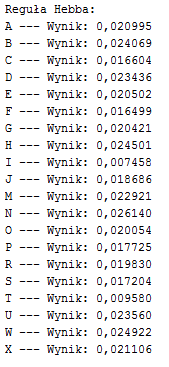
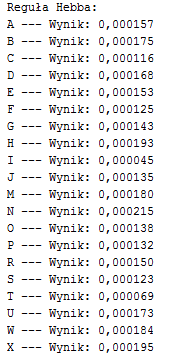
W celu zaprezentowania metody uczenia Hebba został napisany program implementujący powyższą regułę. Zadanie polegało na zapamiętaniu przez sieć 20 liter, a następnie testowanie i odtworzenie wzorców. Sieć została uczona dla różnych współczynników uczenia i zapominania oraz były modyfikowane wagi.

**Testowanie, fragmenty konsoli**

wspł. uczenia = 0.4 wspł. uczenia = 0.2 wspł. uczenia = 0.1  
wspł. zapominania = 0.1 wspł. zapominania = 0.1 wspł. zapominania = 0.1



wspł. uczenia = 0.1 wspł. uczenia = 0.1   
wspł. zapominania = 0.4 wspł. zapominania = 0.2



**Wykres: wspł. uczenia = 0.2, wspł. zapominania = 0.1**

**Wykres: wspł. uczenia = 0.2, wspł. zapominania = 0.1**

Dla większej czytelności wykresów zostały zaprezentowane litery A-J dla naprzemiennie zmieniających się współczynników: 0.1 i 0.2.

Dla współczynnika zapominania 0.1 wartości są bardziej stabilne, mniej zróżnicowane niż dla 0.2. Im wyższy współczynnik uczenia tym sieć lepiej się uczy, ale nie może być za wysoki.

Współczynnik zapominania stanowi niewielki procent stałej uczenia. Przyjęcie dużej wartości powoduje, że neuron zapomina większość tego, co zdołał się nauczyć w przeszłości. Najlepszą i typową wartością współczynnika jest y <= 0.1, przy której zachowuje większość informacji zgromadzonej w procesie uczenia, a jednocześnie możliwe jest ustabilizowanie wag w określonym poziomie.

**Wnioski**

W podstawowej metodzie Hebba wagi mają tendencję do przyjmowania dużych wartości, gdyż w każdym cyklu uczącym dodajemy przyrost delta(w). Zastosowanie współczynnika zapominania pozwala poprawić tą regułę. Współczynnik zapominania przyjmuje wartości od 0 do 1, zmniejsza on znaczenie aktualnych wag. Przy aktualizacji wag brana jest pod uwagę nie ostatnia wartość wij, ale wartość zmniejszona o współczynnik zapominania y.

Dobór wag ma bezpośredni wpływ na działanie perceptronów w sieci, w zależności od ich wartości zmienia się poprawność wyników i liczba iteracji. Wartości wag mają największy wpływ na efekt końcowy, często wagi są ustalane losowo.

Dane uczące wpływają na poprawność uczenia perceptronu, przy zbyt małej ilości otrzymujemy błędne wyniki. Aby uzyskać lepsze wyniki powinniśmy dostarczyć wystarczającą liczbę danych wejściowych

Listing kodu:

**public class** HebbRuleNeuralNetwork  
{  
  
 **public static final char**[] ***letters*** = {**'A'**,**'B'**,**'C'**,**'D'**,**'E'**,**'F'**,**'G'**,**'H'**,**'I'**,**'J'**,**'M'**,**'N'**,**'O'**,**'P'**,**'R'**,**'S'**,**'T'**,**'U'**,**'W'**,**'X'**};  
 **public static final int**[][] ***learning\_letters*** = {  
 {0,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},  
 {1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,0},  
 {0,1,1,1,0,1,0,0,0,1,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,1,0,1,1,1,0},  
 {1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,1,1,1,0},  
 {1,1,1,1,1,1,0,0,0,0,1,0,0,0,0,1,1,1,1,0,1,0,0,0,0,1,0,0,0,0,1,1,1,1,1},  
 {1,1,1,1,1,1,0,0,0,0,1,0,0,0,0,1,1,1,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0},  
 {0,1,1,1,0,1,0,0,0,1,1,0,0,0,0,1,0,1,1,1,1,0,0,0,1,1,0,0,0,1,0,1,1,1,0},  
 {1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,1,1,1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},  
 {0,1,1,1,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,1,1,1,0},  
 {1,1,1,1,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,0,0,0,1,0,1,1,1,0},  
 {1,0,0,0,1,1,1,0,1,1,1,0,1,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1},  
 {1,0,0,0,1,1,0,0,0,1,1,1,0,0,1,1,0,1,0,1,1,0,0,1,1,1,0,0,0,1,1,0,0,0,1},  
 {0,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,0,1,1,1,0},  
 {1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0},  
 {1,1,1,1,0,1,0,0,0,1,1,0,0,0,1,1,1,1,1,0,1,0,1,0,0,1,0,0,1,0,1,0,0,0,1},  
 {0,1,1,1,0,1,0,0,0,1,1,0,0,0,0,0,1,1,1,0,0,0,0,0,1,1,0,0,0,1,0,1,1,1,0},  
 {1,1,1,1,1,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0},  
 {1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,0,1,1,1,0},  
 {1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,0,0,1,1,0,1,0,1,1,0,1,0,1,0,1,0,1,0},  
 {1,0,0,0,1,1,0,0,0,1,0,1,0,1,0,0,0,1,0,0,0,1,0,1,0,1,0,0,0,1,1,0,0,0,1}  
 };  
 **private int**[] **inputs**;  
 **private double**[] **weights**;  
 **private double**[] **outputs**;  
 **private int maxCells**;  
 **private double epsilon**;  
  
 **public** HebbRuleNeuralNetwork(**int** cellCount)  
 {  
 **inputs** = **new int**[cellCount];  
 **weights** = **new double**[cellCount,cellCount];  
 **outputs** = **new double**[cellCount];  
 **maxCells** = cellCount;  
 **epsilon** = .3;  
 resetValues();  
 }  
  
 **public** string printWeights()  
 {  
 **int** maxValueSize = 2;  
  
 **for** (**int** index = 0; index < **maxCells**; index++)  
 {  
 **for** (**int** i = 0; i < **maxCells**; i++)  
 {  
 string s = **weights**[index, i].ToString();  
 **if** (s.Length > maxValueSize)  
 maxValueSize = s.Length;  
 }  
 }  
  
  
 StringBuilder dataBuilder = **new** StringBuilder();  
 dataBuilder.Append(**" "**);  
 **for** (**int** i = 0; i < **maxCells**; i++)  
 dataBuilder.Append(makeRightSize(i.ToString(), maxValueSize)).Append(**" "**);  
 dataBuilder.Append(**"\n"**);  
 **for** (**int** index = 0; index < **maxCells**; index++)  
 {  
 **if** (index < 10)  
 dataBuilder.Append(**"#"**).Append(index.ToString()).Append(**" "**);  
 **else** dataBuilder.Append(**"#"**).Append(index.ToString()).Append(**" "**);  
  
 **for** (**int** i = 0; i < 16; i++)  
 dataBuilder.Append(makeRightSize(**weights**[index, i].ToString(), maxValueSize)).Append(**" "**);  
  
 dataBuilder.Append(**"\n"**);  
 }  
  
 **return** dataBuilder.ToString();  
 }  
  
 **private** string makeRightSize(string number, **int** size)  
 {  
 StringBuilder sb = **new** StringBuilder();  
 **for** (**int** i = number.Length; i < size; i++)  
 sb.Append(**' '**);  
 sb.Append(number);  
 **return** sb.ToString();  
 }  
  
 **public** bool isOutputClicked(**int** i)  
 {  
 **return outputs**[i] == 1;  
 }  
  
 **public double** getWeight(**int** i, **int** j)  
 {  
 **return weights**[i, j];  
 }  
  
  
 **public void** resetValues()  
 {  
 **for** (**int** i = 0; i < **inputs**.Length; i++)  
 **inputs**[i] = 0;  
 **for** (**int** i = 0; i < **outputs**.Length; i++)  
 **outputs**[i] = 0.0;  
 **for** (**int** index = 0; index < **maxCells**; index++)  
 **for** (**int** i = 0; i < **maxCells**; i++)  
 **weights**[index,i] = 0.0;  
 }  
  
 **public void** definePattern(**int**[] inp)  
 {  
 **for** (**int** i = 0; i < **maxCells**; i++)  
 {  
 **inputs**[i] = inp[i];  
 **outputs**[i] = (**double**)inp[i];  
 }  
 }  
  
 **public void** computeActivations(**int** adjustWeights)  
 {  
 **for** (**int** i = 0; i < **maxCells**; i++)  
 {  
 **outputs**[i] = 0.0;  
 **for**(**int** weight = 0; weight<**maxCells**; weight++)  
 **outputs**[i] += (**weights**[i,weight] \* (**double**)**inputs**[weight]);  
  
 *//Click the outputs* **if** (**outputs**[i] >= 0.0)  
 **outputs**[i] = 1.0;  
 **else  
 outputs**[i] = -1.0;  
 }  
 }  
  
 **public void** adjustWeights()  
 {  
 **for** (**int** i = 0; i < **maxCells**; i++)  
 **for** (**int** weight = 0; weight < **maxCells**; weight++)  
 **weights**[i,weight] += **epsilon** \* (**outputs**[i] \* (**double**)**inputs**[weight]);  
 }  
}