Kamil Witek

**Sprawozdanie – scenariusz 4**

Temat ćwiczenia: Uczenie sieci regułą Hebba.

1. Cel ćwiczenia

Celem ćwiczenia jest poznanie działania reguły Hebba na przykładzie rozpoznawania emotikon

1. Realizacja ćwiczenia

Do zrealizowania ćwiczenia wykorzystałem neurony o strukturze modelu sigmoidalnego z metodą uczenia Hebba. Metoda uczenia występuje w dwóch wersjach – z nauczycielem oraz bez nauczyciela. Sposoby modyfikacji wag opisane są wzorami:

**Δwij = learning\_rate \* yj \* yi** gdzie:

1. Learning\_rate – współczynnik uczenia
2. yj – sygnał wejściowy
3. yi – sygnał wyjściowy
4. Ze współczynnikiem zapominania:

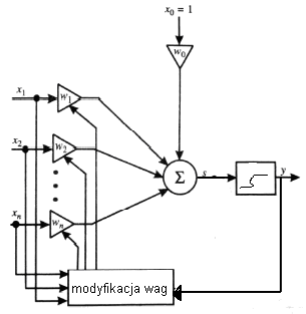
**wij(k+1) = (1-γ) \***  **wij(k) + Δwij** gdzie:

1. γ – współczynnik zapominania
2. Bez współczynnika zapominania:

**wij(k+1) = wij(k) + Δwij**

Modyfikacja wag jak widać w powyższych wzorach zależna jest od sygnału podanego na wejściu jak i sygnału wyjściowego.

Schemat modelu Hebba przedstawiony jest następująco:



Normalizacja wag (metoda normalize\_weights) polega na podziale każdej składowej wektora przez długość tego wektora, co zapobiega nadmiernemu wzrostowi wag. Wzór jest postaci:



Model Hebba wykorzystuje funkcje aktywacji postaci funkcji unipolarnej sigmoidalnej, która przedstawia się następująco:

Metoda sumująca klasy Hebb zwraca sumę iloczynów wag oraz sygnałów wejściowych:

**y = ∑ wj \* xj**

Metoda learnUnsupervised uczy poprzez modyfikacje wag neuronu ze współczynnikiem zapominania jak i również bez niego.

Metoda test zwraca sygnał wyjściowy.

Dane wejściowe w postaci wygenerowanych przeze mnie emotikonów przedstawiają się następująco:

Dodałem jeden pixel do każdej emotikony, który powodował zniekształcenie. Szablony z dodatkowym pixelem wyglądają następująco:

Każdy pixel to jeden sygnał wejściowy, więc dla każdego neuronu jest 8x8 + 1 (BIAS) = 65 wejść.

Proces uczenia oraz testów przeprowadziłem dla wersji ze współczynnikiem zapominania jak i bez niego. Oto wyniki testów dla różnych wartości współczynników uczenia i zapominania.

1. Wyniki

Tabela z modyfikacją wag wraz ze współczynnikiem zapominania.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Learning Rate | 0.001 | | 0.01 | | 0.1 | | 0.3 | |
| Lp. | Forgetting Rate | LR / 6 | LR / 3 | LR / 6 | LR / 3 | LR / 6 | LR / 3 | LR / 6 | LR / 3 |
| 1 | % poprawności [%] | 100 | 25 | 50 | 75 | 75 | 25 | 50 | 50 |
| Ilość epok | 123 | 101 | 6 | 3 | 2 | 15 | 1 | 1 |
| 2 | % poprawności [%] | 75 | 75 | 75 | 75 | 50 | 25 | 25 | 50 |
| Ilość epok | 65 | 192 | 114 | 9 | 2 | 2 | 6 | 1 |
| 3 | % poprawności [%] | 25 | 75 | 50 | 25 | 50 | 50 | 75 | 50 |
| Ilość epok | 119 | 43 | 17 | 115 | 3 | 2 | 2 | 1 |
| 4 | % poprawności [%] | 50 | 50 | 75 | 75 | 75 | 75 | 25 | 50 |
| Ilość epok | 111 | 125 | 14 | 7 | 2 | 2 | 1 | 3 |
| 5 | % poprawności [%] | 50 | 100 | 75 | 75 | 50 | 50 | 50 | 25 |
| Ilość epok | 121 | 127 | 16 | 7 | 3 | 2 | 2 | 1 |
| 6 | % poprawności [%] | 50 | 75 | 50 | 25 | 50 | 50 | 75 | 50 |
| Ilość epok | 112 | 98 | 4 | 14 | 1 | 1 | 6 | 6 |
| 7 | % poprawności [%] | 50 | 25 | 25 | 50 | 75 | 100 | 75 | 50 |
| Ilość epok | 77 | 82 | 8 | 117 | 1 | 1 | 1 | 1 |
| 8 | % poprawności [%] | 50 | 75 | 50 | 75 | 75 | 25 | 75 | 25 |
| Ilość epok | 90 | 156 | 8 | 10 | 1 | 14 | 1 | 2 |
| 9 | % poprawności [%] | 100 | 75 | 25 | 25 | 75 | 50 | 75 | 25 |
| Ilość epok | 78 | 81 | 25 | 13 | 3 | 2 | 2 | 1 |
| 10 | % poprawności [%] | 50 | 25 | 50 | 50 | 50 | 75 | 75 | 50 |
| Ilość epok | 97 | 75 | 112 | 6 | 14 | 2 | 1 | 1 |

Tabela modyfikacji wag bez współczynnika zapominania

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lp. | Learning Rate | 0.001 | 0.01 | 0.1 | 0.3 |
| 1 | % poprawności [%] | 100 | 75 | 75 | 75 |
| Ilość epok | 79 | 112 | 2 | 1 |
| 2 | % poprawności [%] | 50 | 25 | 75 | 75 |
| Ilość epok | 19 | 5 | 1 | 1 |
| 3 | % poprawności [%] | 75 | 100 | 75 | 50 |
| Ilość epok | 85 | 107 | 2 | 1 |
| 4 | % poprawności [%] | 75 | 75 | 75 | 50 |
| Ilość epok | 41 | 114 | 2 | 1 |
| 5 | % poprawności [%] | 50 | 75 | 50 | 75 |
| Ilość epok | 127 | 117 | 1 | 1 |
| 6 | % poprawności [%] | 50 | 75 | 75 | 75 |
| Ilość epok | 100 | 6 | 1 | 6 |
| 7 | % poprawności [%] | 50 | 50 | 75 | 100 |
| Ilość epok | 150 | 10 | 1 | 2 |
| 8 | % poprawności [%] | 75 | 50 | 75 | 50 |
| Ilość epok | 121 | 9 | 2 | 1 |
| 9 | % poprawności [%] | 50 | 75 | 50 | 75 |
| Ilość epok | 31 | 9 | 2 | 6 |
| 10 | % poprawności [%] | 100 | 75 | 25 | 75 |
| Ilość epok | 52 | 6 | 2 | 1 |

1. Analiza wyników

W przedstawionych wyżej wynikach analizując wersje ze współczynnikiem zapominania - ilość epok jaka była potrzebna do nauczenia sieci znacząco różniła się w poszczególnych przypadkach. Ciężko przy takiej rozbieżności jednoznacznie stwierdzić skuteczność nauki sieci na podstawie współczynnika nauczania. Losowanie za każdym razem różnych wag zdecydowanie nie ułatwia dokładnego porównania ilości potrzebnych epok. Jednak wraz ze wzrostem współczynnika nauczania ilość epok za każdym razem spada, jednak większa wartość powoduje też obniżenie poprawności wyników, co sugeruje, że model najlepiej uczyć jest powoli (tzn. mały współczynnik uczenia i wiele epok), wtedy poprawność wyników jest lepsza.

Wartość współczynnika uczenia 0,1 uzyskała najlepsze efekty tzn. dość wysoką poprawność wyników w stosunku do potrzebnej ilości epok. Współczynnik zapominania o wartości 1/6 współczynnika uczenia dał lepsze efekty niż 1/3.

Dla modelu bez współczynnika zapominania ilość epok potrzebna do wyuczenia była mniejsza niż w pierwszym przypadku. Również poprawność nauki prezentowała się nieco lepiej niż w przypadku pierwszym. Także tutaj wartość współczynnika uczenia miała znaczenie i wraz z jej wzrostem ilość potrzebnych epok malała.

1. Wnioski

Na podstawie powyższych wyników można wnioskować, iż najlepsze wyniki można uzyskać stosując metodę modyfikacji wag ze współczynnikiem zapominania. Najlepszy wynik osiągnęła sieć o współczynniku uczenia równym 0,1 oraz o współczynniku zapominania równym 1/6 wartości współczynnika uczenia.

Czasami sieć nie była w stanie nauczyć się wprowadzonych emotikon dlatego, aby zapobiec nieskończonemu wykonywaniu się programu narzuciłem limit maksymalnej ilości 1000 epok.

Rozmiar danych wejściowych ma dość istotny wpływ na rozróżnianie emotikon. Ze względu na niską rozdzielczość obrazków, emotikony są do siebie bardzo podobne i różnią się czasami tylko jednym pikselem co przy testach na zaszumionym obrazie daje odzwierciedlenie w wynikach. Sieć dla większej rozdzielczości uczyła by się bardziej poprawniej, gdyż wtedy emotikony coraz bardziej by się od siebie różniły.

Normalizacja wag zabezpiecza przed nieskończonym wzrostem wartości wag.

Dobranie odpowiedniej wartości współczynnika zapominania jest bardzo istotną kwestią, ważne, aby ta wartość nie była zbyt duża, ponieważ sieć zapominałaby szybko tego czego się nauczyła.

1. Listing kodu

**public class** Emoji {  
  
**public static double**[][] *emoji* = {

{ 1,

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

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

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

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

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

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

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

0, 0, 1, 1, 1, 1, 0, 0 },

{ 1,

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

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

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

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

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

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

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

0, 0, 1, 1, 1, 1, 0, 0 },

{ 1,

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

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

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

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

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

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

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

0, 0, 1, 1, 1, 1, 0, 0 },

{ 1,

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

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

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

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

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

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

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

0, 0, 1, 1, 1, 1, 0, 0 },

};

**public static double**[][] *emojiNoised* = {

{ 1,

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

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

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

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

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

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

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

0, 0, 1, 1, 1, 1, 0, 0 },

{ 1,

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

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

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

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

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

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

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

0, 0, 1, 1, 1, 1, 0, 0 },

{ 1,

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

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

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

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

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

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

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

0, 0, 1, 1, 1, 1, 0, 0 },

{ 1,

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

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

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

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

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

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

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

0, 0, 1, 1, 1, 1, 0, 0 },

};

**public class** main {  
  
 **static int** *numberOfInputs* = 64 + 1; *//ilość wejść (+1 bo bias)* **static double** *learningRate* = 0.01; *//współczynnik uczenia się* **static double** *forgettingRate* = *learningRate* / 6.0; *//współczynnik zapominania* **static int** *numberOfEmoji* = 4; *//liczba emotikonów* **static int** *numberOfNeurons* = 5; *//liczba neuronów* **public static void** main ( String[] args ) {  
  
 **int** winner;  
 Hebb[] hebbs = **new** Hebb[*numberOfNeurons*];  
 **for** ( **int** i = 0; i < *numberOfNeurons*; i++ )  
 hebbs[i] = **new** Hebb( *numberOfInputs* );  
  
 **int** ages = *learn*( hebbs );  
  
 System.***out***.println( **"PO UCZENIU"** );  
 **for** ( **int** i = 0; i < *numberOfEmoji*; i++ ) {  
 winner = *testHebb*( hebbs, Emoji.*emoji*[i] );  
 System.***out***.println( **"Emoji "** + Emoji.*emojiType*[i] + **" - winner neuron = "** + winner );  
 }  
  
 System.***out***.println( **"\nTESTOWANIE"** );  
 **for** ( **int** i = 0; i < *numberOfEmoji*; i++ ) {  
 winner = *testHebb*( hebbs, Emoji.*emojiNoised*[i] );  
 System.***out***.println( **"Emoji "** + Emoji.*emojiType*[i] + **" - winner neuron = "** + winner );  
 }  
  
 System.***out***.println( **"\nIlość epok = "** + ages );  
  
 }  
  
 *//uczenie neuronów* **public static int** learn ( Hebb[] hebbs ) {  
  
 **int** counter = 0;  
 **int** limit = 1000;  
 **int**[] winners = **new int**[*numberOfNeurons*];  
 **for** ( **int** i = 0; i < *numberOfNeurons*; i++ )  
 winners[i] = - 1;  
  
 **while** ( ! *isUnique*( winners ) ) {  
  
 **for** ( **int** j = 0; j < *numberOfNeurons*; j++ ) {  
  
 *//uczenie neuronów każdej emotikony* **for** ( **int** k = 0; k < *numberOfEmoji*; k++ )  
 hebbs[j].learnUnsupervised( Emoji.*emoji*[k], *learningRate*, *forgettingRate*, Hebb.*HEBB\_WITH\_FORGETTING* );  
  
 *//tesotowanie sieci celem sprawdzenia, czy sieć jest już nauczona* **for** ( **int** l = 0; l < *numberOfEmoji*; l++ )  
 winners[l] = *testHebb*( hebbs, Emoji.*emoji*[l] );  
 }  
  
 **if** ( ++ counter == limit )  
 **break**;  
 }  
  
 **return** counter;  
 }  
  
 *//funkcja pomocnicza w procesie uczenie  
 //zwraca true jeśli każdy element w tablicy jest unikalny* **public static boolean** isUnique ( **int**[] winners ) {  
  
 **for** ( **int** i = 0; i < *numberOfNeurons*; i++ )  
 **for** ( **int** j = 0; j < *numberOfNeurons*; j++ )  
 **if** ( i != j )  
 **if** ( winners[i] == winners[j] )  
 **return false**;  
  
 **return true**;  
 }  
  
 *//zwraca wartość zwycięzkiego neuronu dla podanej emotikony* **public static int** testHebb ( Hebb[] hebbs, **double**[] emoji ) {  
  
 **double** max = hebbs[0].test( emoji );  
 **int** winner = 0;  
  
 **for** ( **int** i = 1; i < *numberOfNeurons*; i++ ) {  
 **if** ( hebbs[i].test( emoji ) > max ) {  
 max = hebbs[i].test( emoji );  
 winner = i;  
 }  
 }  
  
 **return** winner;  
 }  
  
}

**import** java.util.Random;  
  
**public class** Hebb {  
  
 **private int noi**; *//ilość wejść* **private double**[] **w**; *//wagi* **public static boolean** *HEBB\_WITH\_FORGETTING* = **true**; *//flaga do uczenia ze współczynnikiem zapominania* **public static boolean** *HEBB\_WITHOUT\_FORGETTING* = **false**; *//flaga do uczenia bez współczynnika zapominania* **public** Hebb ( **int** numbers\_of\_inputs ) {  
 **noi** = numbers\_of\_inputs;  
 **w** = **new double**[**noi**];  
  
 **for** ( **int** i = 0; i < **noi**; i++ )  
 **w**[i] = **new** Random().nextDouble(); *//wagi początkowe sa losowane* normalizeWeights();  
 }  
  
 *//funkcja aktywacji* **private double** active ( **double** y\_p ) {  
 **return** ( 1.0 / ( 1 + Math.*pow*( Math.***E***, - y\_p ) ) ); *//unipolarna sigmoidalna* }  
  
 *//zwraca sumę iloczynów wag i sygnałów wejściowych* **private double** sumator ( **double**[] x ) {  
 **double** y\_p = 0.0;  
 **for** ( **int** i = 0; i < **noi**; i++ )  
 y\_p += x[i] \* **w**[i];  
  
 **return** y\_p;  
 }  
  
 *//uczenie* **public double** learnUnsupervised ( **double**[] x, **double** lr, **double** fr, **boolean** version ) {  
 **double** y\_p = active( sumator( x ) );  
  
 *//w zależności od podanej wersji, nauka będzie z lub bez współczynnika zapominania* **for** ( **int** i = 0; i < **noi**; i++ )  
 **if** ( version ) **w**[i] = ( 1 - fr ) \* **w**[i] + lr \* x[i] \* y\_p; *//ze współczynnikiem zapominania* **else w**[i] += lr \* x[i] \* y\_p; *//bez współczynnika zapominania* normalizeWeights();  
  
 **return** active( sumator( x ) );  
 }  
  
 *//zwraca output neuronu* **public double** test ( **double**[] x ) {  
 **return** active( sumator( x ) );  
 }  
  
 *//normalizuje wagi* **private void** normalizeWeights () {  
 **double** dl = 0.0;  
 **for** ( **int** i = 0; i < **w**.**length**; i++ )  
 dl += Math.*pow*( **w**[i], 2 );  
  
 dl = Math.*sqrt*( dl );  
  
 **for** ( **int** i = 0; i < **w**.**length**; i++ )  
 **if** ( **w**[i] > 0 && dl != 0 )  
 **w**[i] = **w**[i] / dl;  
 }  
  
}