Tomasz Tomala
Podstawy Sztucznej Inteligencji
Sprawozdanie z projektu nr 4

Celem ćwiczenia było poznanie działania reguły Hebba na przykładzie rozpoznawania emotikon.

1) Syntetyczny opis budowy użytej sieci i algorytmów uczenia

W ćwiczeniu wykorzystałem neurony o identycznej strukturze jak w przypadku modelu sigmoidalnego, jednak charakteryzujące się inną metodą uczenia, znaną pod nazwą *reguły Hebba*. Metoda ta występuje w dwóch wersjach: z nauczycielem oraz bez nauczyciela. Do wykonania ćwiczenia wykorzystałem wersję bez nauczyciela. Występują również dwa rodzaje sposobu modyfikacji wag:

$$\Delta w_{ij} = \eta\, y_j\, y_i$$

gdzie:
η – współczynnik uczenia
y_j – sygnał wejściowy
y_i – sygnał wyjściowy

Ze współczynnikiem zapominania:

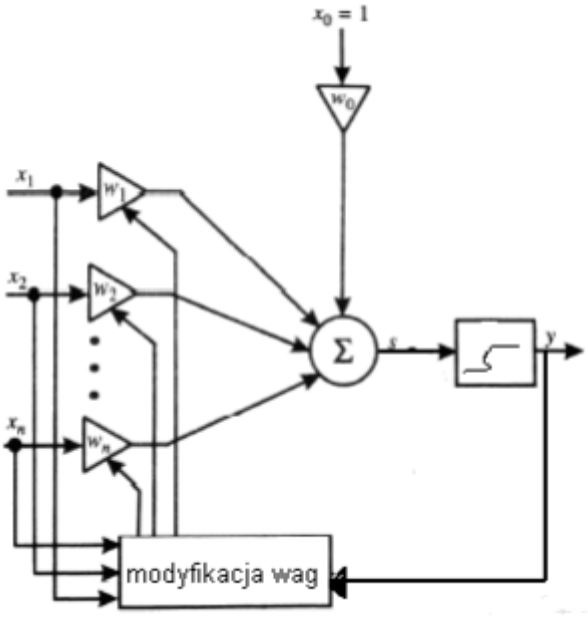
$$w_{ij}(k+1) = (1-\gamma) w_{ij}(k) + \Delta w_{ij}$$

gdzie:
γ – współczynnik zapominania

Bez współczynnika zapominania:

$$w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}$$

Tak więc w trakcie uczenia modyfikacja wagi zależna jest zarówno od sygnału podanego na wejście jak i sygnału wyjściowego. Poniżej schemat pojedynczego neuronu:



Schemat neuronu Hebba

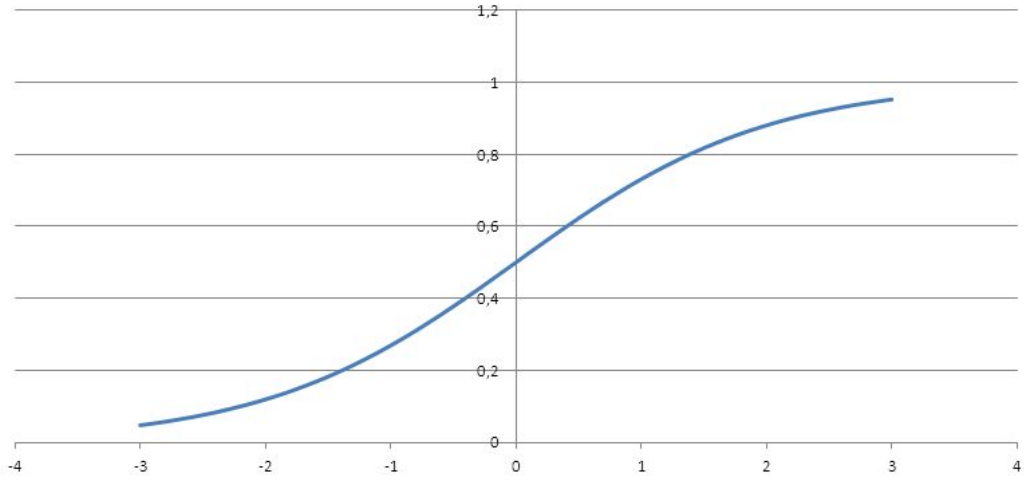
Dodatkowo podczas uczenia normalizuję wagi, aby zapobiec nieustającemu ich wzrostowi. Normalizacja polega na podzieleniu każdej ze składowej wektora przez długość tego wektora:

$$\hat{u} = \frac{\vec{u}}{||\vec{u}||}$$

Zaimplementowany przeze mnie klasa Hebb składa się z następujących metod:

Metoda ***active,*** wykorzystuje unipolarną sigmoidalną funkcję aktywacji:

$$f_\beta(x) = \frac{1}{1+e^{-\beta x}} \quad x \in \square .$$



Wykres dla β=1.

Metoda ***sumator,*** zwraca sumę iloczynów wag oraz sygnałów wejściowych:

$$y = \sum_j w_j x_j,$$

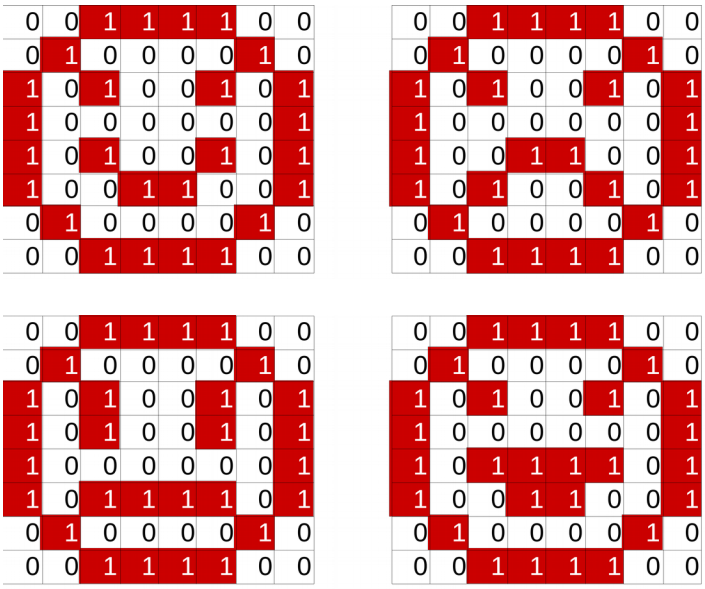
Metoda ***learnUnsupervised,*** uczenie poprzez modyfikację wag neuronu, zarówno w wersji ze współczynnikiem zapominania, jak i bez niego. Korzysta ona ze wzorów podanych już wcześniej.

Metoda ***test,*** zwraca sygnał wyjściowy neuronu.

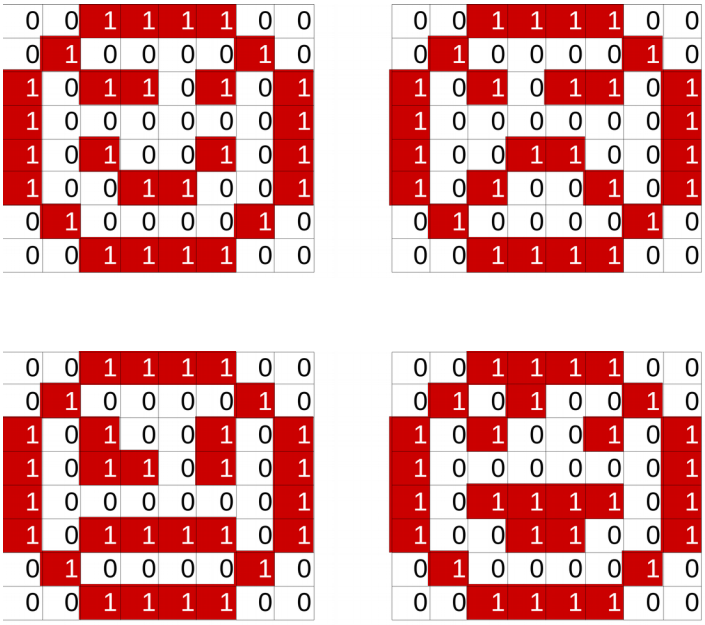
Metoda ***normalizeWeights,*** normalizuje wagi neuronu.

2) Zestawienie otrzymanych wyników

Jako dane uczące wykorzystałem własnoręcznie stworzone emotikony. Maja one wymiary 8x8 pikseli. Przedstawiam je poniżej:



Jako dane testujące wykorzystałem te same emotikony, jednak lekko zmodyfikowane, poprzez dodanie do nich szumu w postaci piksela w losowym miejscu:



Każdy piksel stanowi jedno wejście, tak więc każdy neuron otrzymuje po 64 sygnałów wejściowych.

Proces uczenia i testowania wykonałem zarówno dla wersji modyfikacji wag ze współczynnikiem zapominania jak i bez niego. Dla obu wersji wykonałem po kilka testów dla różnych współczynników uczenia oraz zapominania.

Emotikonów jest 4, tak więc aby uzyskać lepsze rezultaty w procesie uczenia wykorzystuję 5 neuronów ponumerowanych od 0 do 4. W trakcie moich testów zauważyłem, że sieć czasami nie była w stanie nauczyć się klasyfikować emotikony, tak więc do sprawozdania wybrałem najlepsze wyniki.

<u>Wyniki dla wersji modyfikacji wag ze współczynnikiem zapominania:</u>

Współczynnik uczenia = 0.1
Współczynnik zapominania = 0.1/3

PO UCZENIU
Emoji :) - winner neuron = 1
Emoji :(- winner neuron = 3
Emoji :| - winner neuron = 4
Emoji :D - winner neuron = 0

TESTOWANIE
Emoji :) - winner neuron = 1
Emoji :(- winner neuron = 3
Emoji :| - winner neuron = 4
Emoji :D - winner neuron = 0

Ilość epok = 17
Ilość błędów testowania = 0

Współczynnik uczenia = 0.1
Współczynnik zapominania = 0.1/6

PO UCZENIU
Emoji :) - winner neuron = 2
Emoji :(- winner neuron = 4
Emoji :| - winner neuron = 3
Emoji :D - winner neuron = 0

TESTOWANIE
Emoji :) - winner neuron = 2
Emoji :(- winner neuron = 1
Emoji :| - winner neuron = 3
Emoji :D - winner neuron = 0

Ilość epok = 1
Ilość błędów testowania = 1

Współczynnik uczenia = 0.01
Współczynnik zapominania = 0.01/3

PO UCZENIU
Emoji :) - winner neuron = 2
Emoji :(- winner neuron = 3
Emoji :| - winner neuron = 4
Emoji :D - winner neuron = 1

TESTOWANIE
Emoji :) - winner neuron = 2
Emoji :(- winner neuron = 3
Emoji :| - winner neuron = 4
Emoji :D - winner neuron = 3

Ilość epok = 10
Ilość błędów testowania = 1

Współczynnik uczenia = 0.01
Współczynnik zapominania = 0.01/6

PO UCZENIU
Emoji :) - winner neuron = 4
Emoji :(- winner neuron = 3
Emoji :| - winner neuron = 2
Emoji :D - winner neuron = 1

TESTOWANIE
Emoji :) - winner neuron = 4
Emoji :(- winner neuron = 3
Emoji :| - winner neuron = 1
Emoji :D - winner neuron = 4

Ilość epok = 11
Ilość błędów testowania = 2

Wyniki dla wersji modyfikacji wag bez współczynnikiem zapominania:

Współczynnik uczenia = 0.1

PO UCZENIU
Emoji :) - winner neuron = 3
Emoji :(- winner neuron = 0
Emoji :| - winner neuron = 2
Emoji :D - winner neuron = 4

TESTOWANIE
Emoji :) - winner neuron = 2
Emoji :(- winner neuron = 0
Emoji :| - winner neuron = 3
Emoji :D - winner neuron = 4

Ilość epok = 2
Ilość błędów testowania = 1

Współczynnik uczenia = 0.01

PO UCZENIU
Emoji :) - winner neuron = 3
Emoji :(- winner neuron = 4
Emoji :| - winner neuron = 2
Emoji :D - winner neuron = 1

TESTOWANIE
Emoji :) - winner neuron = 3
Emoji :(- winner neuron = 1
Emoji :| - winner neuron = 2
Emoji :D - winner neuron = 0

Ilość epok = 12
Ilość błędów testowania = 2

3) Analiza i dyskusja błędów uczenia i testowania oraz wyników rozpoznawania dla opracowanej sieci w zależności od wartości współczynnika uczenia i zapominania

Jak widać na powyższych wynikach, ilość epok jaka była potrzebna do nauczenia sieci znacząco różniła się w poszczególnych przypadkach. Niekiedy wynik wynosił tylko 1 epokę, a niekiedy było to aż 150 epok. Tak więc jeśli chodzi o szybkość uczenia, widać, że nie jest to dobra miara ocenienia jakości uczenia sieci, gdyż jest to spowodowane wyłącznie początkowymi wartościami wag neuronów, a te są losowe. Dodatkowo w przypadku sieci gdzie ilość epok była równa 150, ilość błędów testowania wynosiła aż 2. Obrazuje to, że nie powinno się zwracać uwagi na szybkość uczenia.

Jeśli jednak chodzi o dobór współczynników uczenia oraz współczynników zapominania, to widać zależności.

Dla wersji modyfikacji wag ze współczynnikiem zapominania, widać, że wraz ze zmniejszaniem wartości obu współczynników rośnie ilość błędów podczas testowania. To samo można zauważyć w wersji modyfikacji wag bez współczynnika zapominania – zmniejszenie współczynnika uczenia poskutkowało pogorszeniem wyników testowania.

4) Sformułowanie wniosków

Na podstawie powyższych wyników można wnioskować, iż lepsze wyniki można uzyskać stosując metodę modyfikacji wag ze współczynnikiem zapominania, ponieważ jedna sieć była w stanie nauczyć się poprawnie klasyfikować emotikony, a i podczas testowania nie wystąpił żaden błąd. Należy jednak rozważnie dobierać współczynnik zapominania, ponieważ jeśli będzie on zbyt duży, to sieć w trakcie nauki zbyt szybko zacznie zapominać tego czego dopiero się nauczyła. Jeśli jednak będzie on zbyt mały, również może negatywnie wpłynąć na otrzymane wyniki. Sieć ucząca się bez współczynnika zapominania również osiągała dobre wyniki, jednak nie były one aż tak obiecujące jak w pierwszym przypadku.

Należy również pamiętać o tym aby normalizować wagi neuronów. Podczas wykonywania ćwiczenia wykonywałem również testy dla sieci bez normalizacji wag, jednak wyniki były daleko od poprawnych. Dzieje się tak, ponieważ bez normalizacji, wagi mogą rosnąć w nieskończoność.

Jeśli chodzi o sam proces testowania zaszumionych emotikonów, widać, iż podczas testów prawie zawsze pojawiał się jakiś błąd. Spowodowane jest to tym, że emotikony są do siebie bardzo podobne – różnią się tylko praktycznie kilkoma pikselami. Dodatkowo sam rozmiar emotikonów pozostawia wiele do życzenia. Jest to bowiem rozmiar jedynie 8x8 pikseli, tak więc nie pozwala to na zbyt dużą różnorodność emotikonów. Gdyby emotikony miały większą rozdzielczość to podczas uczenia można by zastosować więcej wzorców jednej emotikony, które byłyby do siebie zbliżone. Wtedy sieć byłaby w sanie lepiej sklasyfikować emotikony, a i podczas testowania wyniki byłyby dokładniejsze.

5) Listing z komentarzami całego kodu programu

```
package main;

public class Emoji {
   public static double[][] emoji = {
         //pierwszy input to bias
         //zwykłe 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, 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, 1, 1,
0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 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, 1, 0, 0, 1, 0, 1, 1,
0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 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, 1, 1,
0, 1, 1, 1, 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 },    // :D
   };

   public static double[][] emojiNoised = {
         //pierwszy input to bias
         //zaszumione Emoji
         { 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 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, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1,
0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 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, 1, 1, 0, 1, 0, 1, 1,
0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 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, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1,
0, 1, 1, 1, 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 },    // :D
   };

   public static String[] emojiType = { ":)", ":(", ":|", ":D" };
}
```

```
package main;

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 są 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;
   }
}
```

```
package main;

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;
      }
}
```

Bibliografia:

http://pracownik.kul.pl/files/31717/public/Model_neuronu_Hebba.pdf
http://slideplayer.pl/slide/818119/2/images/11/sigma+(s)+w+greckim+alfabecie)..jpg
https://pl.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-vectors/a/vector-magnitude-normalization
https://en.wikipedia.org/wiki/Hebbian_theory#Generalization_and_stability