**Sprawozdanie 1**

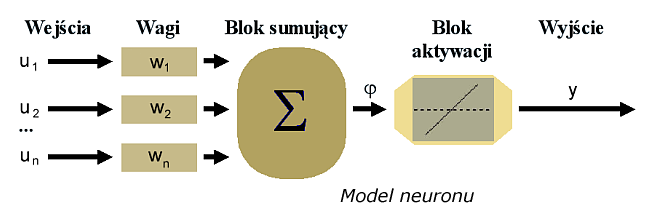
**Temat ćwiczenia: Budowa i działanie perceptronu Monika Pawlik, gr 05**

Celem ćwiczenia jest poznanie budowy i działanie perceptronu poprzez implementację oraz uczenie perceptronu realizującego wybraną funkcje logiczną dwóch zmiennych.

Niestety mam problem z pierwszym repozytorium, prawdopodobnie błędnie utworzone, więc utworzyłam nowe PSI\_semestr5. Zostawiłam pierwsze repozytorium, ponieważ jest tam folder, niestety bez plików w środku, które przepadły, ale jest data jego utworzenia.

1. **Syntetyczny opis budowy neuronu**

Na podstawie działania neuronu biologicznego można zbudować schemat działania neuronu sztucznego. Składa się on z wielu wejść z których sygnały są sumowane z odpowiednimi wagami a następnie poddawane działaniu funkcji aktywacji:

Wejścia to odpowiedniki dendrytów, lub ściślej: sygnały przez nie nadchodzące. Wagi to cyfrowe odpowiedniki modyfikacji dokonywanych na sygnałach przez synapsy. Blok sumujący to odpowiednik jądra, blok aktywacji to wzgórek aksonu, a wyjście - to akson.

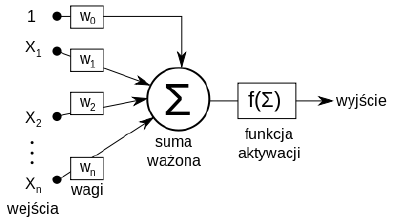
Proces przetwarzania sygnału w sztucznym neuronie w sposób ogólny można przedstawić następująco:  
Wejścia dostarczają sygnał, który następnie jest mnożony przez współczynniki wag, następnie w bloku sumowania następuje sumowanie pomnożonych sygnałów. Wynikiem tego otrzymujemy sygnał zwany potencjałem membranowym. Następnie sygnał przetworzony zostaje w bloku aktywacji, który w zależności od potrzeb może być opisany różnymi funkcjami - zwanymi funkcjami aktywacji. Wartość funkcji aktywacji jest sygnałem wyjściowym neuronu i propagowana jest do neuronów warstwy następnej.

**Neuron McCullocha-Pittsa** – jeden z matematycznych modeli neuronu. Posiada wiele wejść i jedno wyjście. Każdemu z wejść przyporządkowana jest liczba rzeczywista - waga wejścia. Neuron ten jest podstawowym budulcem sieci neuronowej perceptron. Wartość na wyjściu neuronu jest obliczana dwukrokowo:

* Obliczana jest suma iloczynów wartości xi podanych na wejścia i wag wi wejść:



* Na wyjście podawana jest wartość funkcji aktywacji f(s) dla obliczonej sumy



1. **Perceptron**

**Perceptron** – najprostsza sieć neuronowa, składająca się z jednego bądź wielu niezależnych neuronów McCullocha-Pittsa, implementująca algorytm uczenia nadzorowanego klasyfikatorów binarnych. Perceptron jest funkcją, która potrafi określić przynależność parametrów wejściowych do jednej z dwóch klas.

**Wykorzystany algorytm uczenia perceptronu**

Algorytm uczenia perceptronu - automatyczny dobór wag na podstawie napływających przykładów.

1) Wybieramy losowo początkowe wartości wag. Przyjmujemy wartości współczynnika c.

2) Podajemy na wejście neuronu wektor uczący

𝑥=𝑥 𝜏 =[𝑥0 𝜏 ,𝑥𝑖 𝜏 ,…,𝑥𝑖𝑖 𝜏 ,𝜏=1,2,…

3) Obliczenie wartości wyjściowej perceptronu y.

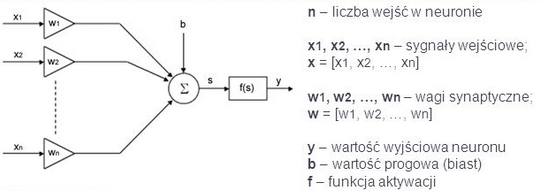
4) Porównanie wartości wyjściowej 𝑦 𝜏 z wartością wzorcową (uczącą) 𝑡=𝑡 𝑥 𝜏

5) Modyfikacja wag

𝑤𝑖 𝜏+1 =𝑤𝑖 𝜏 +𝑐 𝑡−𝑦 𝑥𝑖(𝜏)

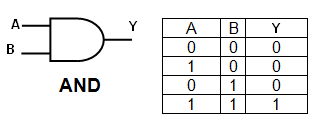
6) Powrót do punktu 2.

Algorytm powtarzany jest dopóty, dopóki błąd na wyjściu nie będzie mniejszy od założonego, dla wszystkich wektorów ciągu uczącego.



1. **Zestawienie otrzymanych wyników**

Wykorzystanie prostego układu funkcji logicznej do uczenia perceptronu. Wybrano funkcję logiczną AND. Operacja logiczna AND przyjmuje 2 parametry binarne i odpowiada wartością 0 lub 1. Symbol logiczny i tablica prawdy bramki AND:



Przedstawienie otrzymanych wyników uczenia w postaci tabel i wykresów.

Uczenie i testowanie opracowanego perceptronu w zależności od **wartości współczynnika uczenia**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| x1 | x2 | result | 0,01 | 0,001 | 0,5 | 0,8 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0,63 | 0,5 | 0,76 | 0,5 |
| 1 | 0 | 0 | 0,38 | 0,5 | 0,7 | 0,8 |
| 0 | 0 | 1 | 1,01 | 1 | 1,13 | 1,16 |

Współczynnik proporcjonalności: jeśli jest zbyt duży, w testowanym przykładzie = 0.8 powoduje przeskoczenie prawidłowego rozwiązania. Zbyt mały współczynnik proporcjonalności cechuje bardzo wolne, nieefektywne uczenie.

Uczenie i testowanie opracowanego perceptronu w zależności **od liczby danych uczących**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x1 | x2 | 10 | 100 | 1000 | 10000 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0,03 | 0,23 | 0,39 | 0,24 |
| 1 | 0 | 0,2 | 0,42 | 0,58 | 0,69 |
| 0 | 0 | 0,59 | 1 | 1,01 | 1,03 |



Im większa liczba danych uczących tym sieć szybciej się uczy. Podczas testów okazało się że dla 10 iteracji za każdym razem sieć dawała błędne odpowiedzi. Przy wzrastaniu danych uczących sieć uczyła się i poprawna była jej reakcja. Ale należy uważać aby sieć nie przeuczyć. Dla bardzo dużej liczby iteracji odpowiedzi sieci były różne, częściej dobre, ale nie zawsze.

1. **Analiza i dyskusja błędów uczenia i testowania opracowanego perceptronu w zależności od wartości współczynnika uczenia oraz liczby danych uczących**

Podczas testowania okazało się, że wybranie małego współczynnika uczenia doprowadziło do bardzo powolnego procesu uczenia (wagi były poprawiane w każdym kroku bardzo słabo, żeby więc osiągnęły pożądane wartości trzeba było wykonać bardzo dużo takich kroków).

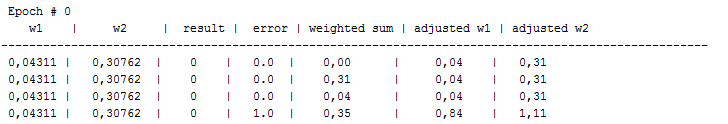
Z kolei wybór za dużego współczynnika uczenia powodował bardzo gwałtowne zmiany parametrów sieci, które w krańcowym przypadku prowadziły nawet do niestabilności procesu uczenia.

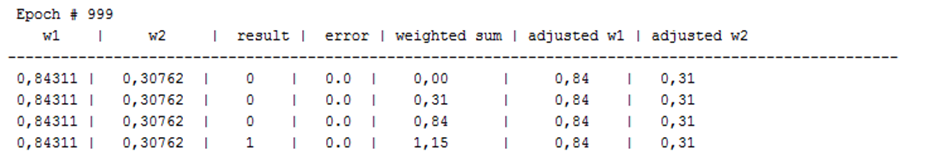
Testowany problem logiczny nie jest skomplikowany. W przypadku prostych problemów większe współczynniki poprawiają szybkość uczenia się sieci. W tak prostym przypadku sieć ma zbyt małą liczbę błędnego uczenia się – w skomplikowanych przypadkach bardzo duże współczynniki mogą prowadzić do uczenia się nowych rzeczy kosztem zapominania starych. Dla bardzo małej liczby danych uczących sieć nie uczyła się poprawnie. Można stwierdzić że wynik w pewnym stopniu zależy od stanu początkowego sieci.

**Opis implementacji**

Implementacja została wykonana w języku Java. Głównymi klasami są klasy Neuron i Perceptron. Taka struktura klas jest dosłowną interpretacją budowy sieci neuronowej. Program zawiera ponadto klasę Driver odpowiedzialną za wywołanie klas i jest związana z interfejsem użytkownika.

Przykładowe screeny z konsoli w JAVA





**WNIOSKI**

Sieci neuronowe mają zdolność do uczenia się, czyli zdolność do samodzielnego dostosowywania współczynników wagowych. Uczenie sieci to wymuszenie na niej określonego zareagowania na sygnały wejściowe. Sieci neuronowe potrafią samodzielnie dostosować się do zmieniających warunków. Im większa jest liczba danych uczących, tym sieć zachowuje się lepiej.

**Listing kodu**

**Kod jest pisany w Java.**

**Klasa Perceptron:**

**public class** Perceptron {  
 **public static final int** [][][] ***andData*** = {{{0,0},{0}},  
 {{0,1},{0}},  
 {{1,0},{0}},  
 {{1,1},{1}}};  
 **public static final double *learning\_rate*** = 0.8;  
 **public static final double**[] ***initial\_weights*** = { 0.5 - Math.*random*(),0.5 - Math.*random*()};  
 **public double** calculateWeightedSum( **int**[] data, **double**[] weights)  
 {  
 **double** weightedSum =0;  
 **for**(**int** x=0;x<data.**length**; x++) {  
 weightedSum +=data[x]\*weights[x];  
 }  
 **return** weightedSum;  
 }  
 **public int** applyActivationFunction(**double** weightedSum) *//zastosować aktywację funkcji* {  
 **int** result=0;  
 **if**(weightedSum > 1)  
 result = 1;  
 **return** result;  
 }  
 **public double**[] adjustedWeights(**int**[] data, **double**[] weights, **double** error) *//dostosuj wagi uczenie sieci* {  
 **double**[] adjustedWeights = **new double**[weights.**length**];  
 **for**(**int** x=0;x<weights.**length**;x++){  
 adjustedWeights[x]= ***learning\_rate*** \* error \* data[x] + weights[x];  
 }  
 **return** adjustedWeights;  
 }  
  
  
}

Klasa Neuron:

**public class** Neuron {  
 **private double threshold** = 0.5 - Math.*random*();  
 **private double**[] **weights** = {0.8 - Math.*random*(), 0.8 - Math.*random*()};  
 **private double output** = 0;  
 **private double error** = 0;  
  
 **public void** applyActivationFunction(**double** weightedSum) {**output**=1.0 /   
 (1+Math.*exp*(-1.0 \* weightedSum));}  
 **public double** derivative(){**return output** \* 1.0 - **output**;}  
 *//gettery and settery* **public double** getThreshold() {**return threshold**;}  
 **public void** setThreshold(**double** threshold) {**this**.**threshold** =   
 threshold;}  
 **public double**[] getWeights() {**return weights**;}  
 **public double** getOutput() {**return output**;}  
 **public void** setOutput(**double** output) {**this**.**output** = output;}  
 **public double** getError() {**return error**;}  
 **public void** setError(**double** error) {**this**.**error** = error;}  
  
}

**Klasa Driver:**

**public class** Driver {  
  
 **public static void** main(String[] args) {  
 **int**[][][] data = Perceptron.***andData***;  
 **double**[] weights = Perceptron.***initial\_weights***;  
 Driver driver = **new** Driver();  
 Perceptron perceptron = **new** Perceptron();  
 **int** epochNumber = 0;  
 **boolean** errorFlag = **true**;  
 **double** error = 0;  
 **double**[] adjustedWeights = **null**;  
  
 **int** iteration = 1000;  
 **for**(**int** j=0;j<iteration;j++)  
 {  
 driver.printHeading(epochNumber++);  
 errorFlag=**false**;  
 error=0;  
 **for**( **int** i=0;i<data.**length**;i++){  
 **double** weightedSum =   
 perceptron.calculateWeightedSum(data[i][0],weights);  
 **int** result =   
 perceptron.applyActivationFunction(weightedSum);  
 error = data[i][1][0] - result;  
 *// if(error != 0)  
 // errorFlag = true;* adjustedWeights =   
 perceptron.adjustedWeights(data[i][0],weights,error);  
 driver.printVector(data[i],weights,result,error,weightedSum,adjustedWeights);  
 weights = adjustedWeights;  
 }  
 }  
  
 }  
  
  
 **public void** printHeading(**int** epochNumber){ *//wydrukuj nagłówek* System.***out***.println(**"\n Epoch # "**+epochNumber);  
 *//System.out.println(" w1 | w2 | x1 | x2 | target result | result | error | weighted sum | adjusted w1 | adjusted w2");* System.***out***.println(**" w1 | w2 | result | error | weighted sum | adjusted w1 | adjusted w2"**);  
 System.***out***.println(**"-----------------------------------------------------------------------------------------------------"**);  
 }  
 **public void** printVector(**int**[][] data, **double**[] weights, **int** result, **double** error, **double** weightedSum, **double**[] adjustedWeights){  
  
 *//System.out.println(" "+String.format("%.2f",weights[0])+" | "+String.format("%.2f",weights[1])+" | "+data[0][0]+" | "+data[0][1]+" | "+data[1][0]+" | "+data[1][1]+" | "+result+" | "+error+" | "+String.format("%.2f",weightedSum)+" | "+String.format("%.2f",adjustedWeights[0])+" | "+String.format("%.2f",adjustedWeights[1]) );* System.***out***.println(**" "**+String.*format*(**"%.5f"**,weights[0])+**" | "**+String.*format*(**"%.5f"**,weights[1])+**" | "**+result+**" | "**+error+**" | "**+String.*format*(**"%.2f"**,weightedSum)+**" | "**+String.*format*(**"%.2f"**,adjustedWeights[0])+**" | "**+String.*format*(**"%.2f"**,adjustedWeights[1]) );  
  
 }  
  
  
}