**Scenariusz 2**

Białek Tomasz, gr. 1

**Temat ćwiczenia:** Budowa i działanie sieci jednowarstwowej

**1. Cel ćwiczenia**

Celem ćwiczenia jest poznanie budowy i działanie jednowarstwowych sieci neuronowych orazuczenie rozpoznawania wielkości liter.

**2. Opis budowy sieci i algorytmów uczenia.**

Celem budowanej sieci jest rozpoznawanie wielkości liter. Do tego celu utworzony został zestaw liter (10 dużych i 10 małych), które są reprezentowane w postaci dwuwymiarowej tablicy 5x7 pikseli dla jednej litery.

Ćwiczenie zostało wykonane w języku C++ (własna implementacja), w którym zastosowano uczenie nadzorowane (z pomocą nauczyciela). Wykorzystano model sieci Adaline oraz DeltaRule (sigmoidalna funkcja aktywacji).

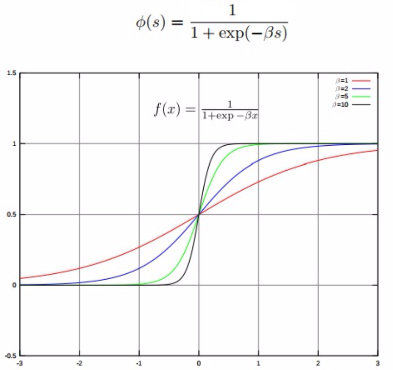
Duże litery wykorzystane w ćwiczeniu: ABCDEFGHIJ

Małe litery wykorzystane w ćwiczeniu: abcdefghij

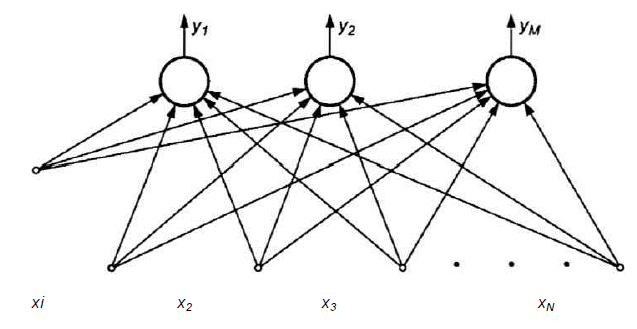
Tab. 1. Litery i ich reprezentacja

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

Ciąg przeznaczony do nauki składa się z 20 wektorów (tablic) zawierających 35 pozycje oraz ciąg wyjściowy (zawierający 1 – duża litera, 0 – mała litera). Sieć DeltaRule używa funkcji sigmoidalnej unipolarnej ze stałym współczynnikiem β = 0,5 jako funkcji aktywacji.



Rys. 1 Funkcja sigmoidalna unipolarna z różnymi współczynnikami β



Rys. 2 Schemat sieci neuronowej jednowarstwowej

Schemat uczenia sieci jednowarstwowej

Dane jest p par uczących: *{(y1,d1),(y2,d2),...,(yp,dp)},* gdzie *yi* ma rozmiar *J ×1, di* ma rozmiar *K ×1.* parametr *l* oznacza numer kroku cyklu uczenia.

Krok 1:

Wybór h > 0, Emax > 0.

Krok 2:

Wybór początkowych wartości elementów macierzy wag W jako niewielkich liczb losowych.

Macierz W ma wymiar K×J.

Krok 3:

Ustawienie wartości początkowej licznika kroków oraz wyzerowanie wartości błędu:

*1) l = 1*

*2) E = 0*

Krok 4:

Podanie danych na wejście i obliczenie sygnału wyjściowego y = yl, d = dl, zk = j(wkTy),

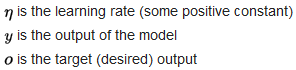
k = 1, 2, …, K (gdzie wkT jest k-tym wierszem macierzy W).

Krok 5:

Uaktualnienie wag według wzoru:

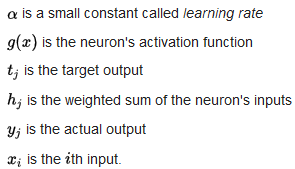
a) dla Adaline:

, gdzie:



b) dla DeltaRule:

, gdzie:



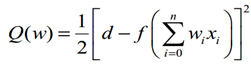
Krok 6:

Obliczenie błędu łącznego:

a) dla Adaline:



b) dla DeltaRule:



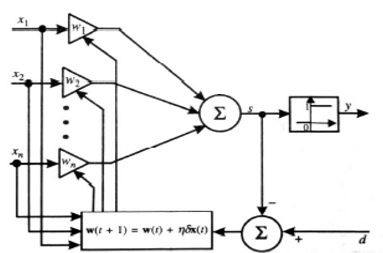
Krok 7:

Jeżeli *l < p,* to *l=l+1* oraz przejście do kroku 4.

Krok 8:

Cykl uczenia został zakończony jeżeli E < Emax (Threshold). W przeciwnym wypadku rozpoczęcie nowego cyklu uczenia (powrót do kroku 3.).

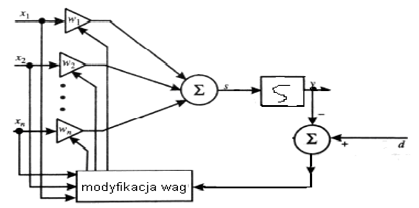
Schemat sieci Adaline



Model sieci Adaline

Model Adaline ma zbliżoną budowę do perceptronu. Różnią się one algorytmami uczenia. W modelu Adaline nie jest uwzględniana funkcja aktywacji przy porównywaniu sygnału wyjściowego z sygnałem wzorcowym.

Schemat sieci DeltaRule



Model sieci DeltaRule

Model DeltaRule ma analogiczną budowę do modelu Adaline, jednakże funkcją aktywacji jest funkcja sigmoidalna, a przy aktualizacji wag uwzględnia się pochodną tejże funkcji.

**3. Otrzymane wyniki i ich analiza**

*Przykładowe działanie programu*

|  |  |  |
| --- | --- | --- |
| Współczynnik  uczenia = 0.01 | Współczynnik  uczenia = 0.05 | Współczynnik  uczenia = 0.1 |
|  |  |  |

*Analiza*

Proces uczenia sieci jednowarstwowej przebiega analogicznie do uczenia perceptronu. Analizując wykres można zauważyć, że im wyższy współczynnik uczenia, tym liczba potrzebnych epok była niższa. Dla współczynnika uczenia > 0.01 sieć została wyćwiczona już po kilkunastu epokach. Dla współczynnika uczenia = 0.0001 proces uczenia zajął ponad 4000 epok (dla metody DeltaRule).

W przypadku metody Adaline sieć dla bardzo małych współczynników uczenia nie potrzebowała aż tak dużej liczby, lecz ich ilość była zauważalna.

Dla współczynnika uczenia = 0.01 liczba epok uczenia praktycznie się wyrównała.

Analizując powyższy wykres można zauważyć, że na efektywność uczenia sieci ma wpływ funkcja aktywacji. Im mniejsza wartość współczynnika β, tym potrzeba więcej epok nauki. Akceptowalne wartości współczynnika β wynoszą powyżej 0.5, gdzie uczenie przebiega w miarę na tym samym poziomie.

Z powyższego wykresu można odczytać, że współczynnik uczenia silnie wpływa na błąd średniokwadratowy tego uczenia. Im wyższy współczynnik uczenia, tym powstały błąd jest mniejszy.

Dla mniejszych współczynników uczenia błąd jest większy, co może wpływać na samo uczenie się sieci (nauka może trwać dłużej właśnie ze względu na powstały błąd).

Dla przypadku z DeltaRule błąd uczenia nie zmniejsza się jak przy modelu Adaline.

Z powyższego wykresu wynika, że sieć wykorzystująca model Adaline daje poprawniejsze wyniki aniżeli sieć z modelem DeltaRule. Model Adaline miał zawsze mniej lub taką samą ilość błędów co model DeltaRule.

**4. Podsumowanie**

Skuteczność procesu uczenia zależy od współczynnika uczenia. Wraz z jego wzrostem proces uczenia jest poprawniejszy. Jest to wytłumaczalne z tego względu, że im ta wartość jest większa tym przyrost wag, które na samym początku są niewielkie jest szybszy, więc proces uczenia przebiega szybciej. Sieć jednowarstwowa jest odporna na zaszumienie, jednakże tylko dla kilku bitów. Zbyt duże zaszumienie powoduje błędne odpowiedzi dawane przez sieć (w szczególności dla modelu DeltaRule). Przy projektowaniu sieci neuronowej trzeba wybrać odpowiedni jej model. Oprócz samego modelu na jej efektywność ma wpływ sama jej struktura, np. zastosowana funkcja aktywacji. Model Adaline pozwala na o wiele sprawniejsze uczenie sieci w porównaniu do modelu DeltaRule. W samym modelu DeltaRule wpływ na efektywność uczenia sieci ma współczynnik β w sigmoidalnej funkcji aktywacji (akceptowalną wartością współczynnika jest ten >=0.5, który powoduje, że sieć zachowuje się stabilnie i jej uczenie jest w miarę akceptowalne. Dla naszego przypadku wartość współczynnika, wynosząca 0.5, była wystarczająca, lecz dla bardziej zaawansowanych sieci współczynnik ten musi być wyższy.

**5. Kod programu**

„Source.cpp”

#include "Adaline.h"

#include "DeltaRule.h"

# define 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, 1, 0, 0, 0, 1

# define 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, 1, 0

# define C 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

# define D 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

# define E 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

# define 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, 0, 1, 0, 0, 0, 0

# define G 1, 1, 1, 1, 1, 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

# define H 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

# define I 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

# define J 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

# define a 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1

# define b 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0

# define c 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0

# define d 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1

# define e 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0

# define f 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0

# define g 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0

# define h 0, 0, 0, 0, 0, 0, 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

# define i 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0

# define j 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0

int main()

{

//Perceptron.h nie jest uzywana (sluzy tylko do zalaczania bibliotek i trzymania define)

double learningRate = 0.01; //wspolczynnik uczenia

cout << "Learning Rate: " << learningRate << endl;

//Adaline

Adaline adaline(learningRate);

adaline.learn();

adaline.test();

//DeltaRule

DeltaRule deltaRule(learningRate);

deltaRule.learn();

deltaRule.test();

system("PAUSE");

return 0;

}

„Adaline.h”

#pragma once

#include "Perceptron.h"

#define HOW\_MANY\_LETTERS 20

#define BITS\_OF\_ONE\_LETTER 35

class Adaline {

public:

double \* weights; //przechowuje wagi liter

double learningRate; //wspolczynnik uczenia

int inputData[HOW\_MANY\_LETTERS][BITS\_OF\_ONE\_LETTER]; //tablica przechowująca litery [20] i ich reprezentancje bitowe [35]

int expectedResults[HOW\_MANY\_LETTERS]; //tablica przechowująca oczekiwane wyniki [1 - duża liczba, 0 - mała liczba]

int numberOfSets; //liczba liter = 20

int numberOfWeights; //liczba wag = 35

double error; //blad globalny (po podniesieniu do kwadratu, podzieleniu przez 2 daje MSE)

double delta; //jest to wyliczona roznica (dodawana do wag)

int setTest[HOW\_MANY\_LETTERS][BITS\_OF\_ONE\_LETTER] = { //tablica danych testowych

{ 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 },

{ 1, 1, 1, 1, 1, 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 },

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

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

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

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

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

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

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

{ 0, 0, 0, 0, 0, 0, 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 },

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

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

};

//char'owe reprezentacje tablicy bitow

char setTestLetters[20] = { 'A','B','C','D','E','F','G','H','I','J','a','b','c','d','e','f','g','h','i','j' };

Adaline(double \_learningRate); //konstruktor (przyjmuje learningRate)

void readData(); //funkcja wczytuje dane do nauki z pliku

double getSum(int letter[], double \*weights); //funkcja zwraca sume wejscia danej litery

double getRandomDouble(); //funkcja losuje double z przedzialu <0; 1>

bool activationFunction(double sum); //funkcja aktywacji - funkcja progowa unipolarna

void learn(); //funkcja uczaca

void test(); //funkcja testujaca

};

„Adaline.cpp”

#include "Adaline.h"

Adaline::Adaline(double \_learningRate)

{

delta = 0;

numberOfWeights = BITS\_OF\_ONE\_LETTER;

numberOfSets = HOW\_MANY\_LETTERS;

learningRate = \_learningRate;

error = 0.0;

weights = new double[numberOfWeights];

for (int i = 0; i < BITS\_OF\_ONE\_LETTER; i++)

this->weights[i] = getRandomDouble();

//wczytanie z pliku danych uczacych

readData();

}

//wczytanie danych uczących z pliku

void Adaline::readData()

{

fstream file;

file.open("data\_for\_learning.txt");

if (!file.good()) {

cout << "--------- I can't open the file with learning data ---------" << endl;

system("PAUSE");

exit(0);

}

//wczytuj z pliku dopóki są dane

while (!file.eof())

for (int i = 0; i < HOW\_MANY\_LETTERS; i++) { // i oznacza indeks litery

for (int j = 0; j < BITS\_OF\_ONE\_LETTER; j++) // j oznacza ilosc bitow na daną litere

file >> this->inputData[i][j]; //wczytywanie do tablicy z wejsciami

file >> this->expectedResults[i]; //wczytanie z pliku czy dana litera jest duza (1) lub mala (0)

}

file.close();

}

//losuje double'a z przedzialu <0;1>

double Adaline::getRandomDouble()

{

double randValue = ((double)rand() / (double)RAND\_MAX);

return randValue;

}

//funkcja aktywacji - funkcja progowa unipolarna

bool Adaline::activationFunction(double sum)

{

if (sum > 0.5)

return true;

else

return false;

}

//zwraca sume danego wejscia

double Adaline::getSum(int letter[], double \* weights)

{

double sum = 0.0;

for (int i = 0; i < numberOfWeights; i++)

sum += letter[i] \* weights[i];

return sum;

}

//funkcja uczaca

void Adaline::learn()

{

cout << endl << "ADALINE - LEARNING" << endl;

bool acceptableError = false; //zmienna, stwierdzajaca czy blad jest mozliwy do zaakceptowania

int epoch = 0; //numer epoki

/\*

for (int i = 0; i < numberOfWeights; i++)

cout << "Weights are: w" << i+1 << " = " << weights[i] << endl;;

cout << endl;

\*/

do {

epoch++; // zwiekszenie epoki

error = 0.0; //zerowanie glownego bledu w celu sprawdzenia bledow podczas jednej iteracji

for (int i = 0; i<numberOfSets; i++) {

//obliczanie roznicy pomiedzy wynikiem oczekiwanym a wynikiem otrzymanym

delta = expectedResults[i] - getSum(inputData[i], weights);

//aktualizowanie wag

for (int j = 0; j < numberOfWeights; j++)

weights[j] += learningRate\*delta\*inputData[i][j];

//aktualizowanie bledu glownego

error += delta\*delta;

}

error /= 2;

//porownywanie bledu z progiem

if (error>0.01) {

acceptableError = false;

}

else {

acceptableError = true;

}

} while (!acceptableError);

cout << "Epoch: " << epoch << endl;

cout << "MSE error: " << error << endl;

/\*

for (int i = 0; i < numberOfWeights; i++)

cout << "Weights are: w" << i+1 << " = " << weights[i] << endl;

\*/

}

//funkcja testujaca

void Adaline::test()

{

cout << "Test letters:" << endl;

for (int i = 0; i<numberOfSets; i++) {

cout << "Letter " << setTestLetters[i] << " is: ";

if (activationFunction(getSum(setTest[i], weights))) {

cout << "big";

}

else {

cout << "small";

}

cout << endl;

}

}

„DeltaRule.h”

#pragma once

#include "Perceptron.h"

#define HOW\_MANY\_LETTERS 20

#define BITS\_OF\_ONE\_LETTER 35

class DeltaRule {

public:

double \* weights; //tablica wag

double learningRate; //wspolczynnik uczenia

int inputData[HOW\_MANY\_LETTERS][BITS\_OF\_ONE\_LETTER];

int expectedResults[HOW\_MANY\_LETTERS]; //tablica oczekiwanych wynikow

int numberOfSets; //liczba liter

int numberOfWeights; //liczba wag

double error; //blad globalny (po podniesieniu do kwadratu, podzieleniu przez 2 daje MSE)

double delta; //jest to wyliczona roznica(dodawana do wag)

double output; //przechowuje aktualna wartosc (porownywana z wartoscia oczekiwana)

int setTest[HOW\_MANY\_LETTERS][BITS\_OF\_ONE\_LETTER] = { //tablica danych testowych

{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},

{1, 1, 1, 1, 1, 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},

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

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

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

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

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

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

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

{0, 0, 0, 0, 0, 0, 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},

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

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

};

//char'owe reprezentacje tablicy bitow

char setTestLetters[20] = { 'A','B','C','D','E','F','G','H','I','J','a','b','c','d','e','f','g','h','i','j' };

DeltaRule(double \_learningRate); //konstruktor

void readData(); //wczytuje z pliku dane uczące

double getRandomDouble(); //losuje double z przedzialu <0; 1>

double activationFunction(double sum); //funkcja aktywacji - funkcja sigmoidalna

double derivativeActivationFunction(double sum); //pochodna funkcji aktywacji

double getSum(int letter[], double \* weights); //zwraca sume wejscia danej litery

void learn(); //funkcja uczaca

void test(); //funkcja testujaca

};

„DeltaRule.cpp”

#include "DeltaRule.h"

//konstruktor

DeltaRule::DeltaRule(double \_learningRate) {

delta = 0;

numberOfWeights = BITS\_OF\_ONE\_LETTER;

numberOfSets = HOW\_MANY\_LETTERS;

learningRate = \_learningRate;

error = 0;

output = 0;

weights = new double[numberOfWeights];

for (int i = 0; i < BITS\_OF\_ONE\_LETTER; i++)

this->weights[i] = getRandomDouble();

//wczytanie z pliku danych uczacych

readData();

}

//losuje double'a z przedzialu <0;1>

double DeltaRule::getRandomDouble()

{

double randValue = ((double)rand() / (double)RAND\_MAX);

return randValue;

}

//wczytanie danych uczących z pliku

void DeltaRule::readData()

{

fstream file;

file.open("data\_for\_learning.txt");

if (!file.good()) {

cout << "--------- I can't open the file with learning data ---------" << endl;

system("PAUSE");

exit(0);

}

//wczytuj z pliku dopoki są dane

while (!file.eof())

for (int i = 0; i < HOW\_MANY\_LETTERS; i++) { // i oznacza indeks litery

for (int j = 0; j < BITS\_OF\_ONE\_LETTER; j++) // j oznacza ilosc bitow na dana litere

file >> this->inputData[i][j]; //wczytywanie do tablicy z wejściami

file >> this->expectedResults[i]; //wczytanie z pliku czy dana litera jest duza (1) lub mala (0)

}

file.close();

}

//funkcja aktywacji - funkcja sigmoidalna

double DeltaRule::activationFunction(double sum) {

//Wspolczynnik beta = 1.0

return (1 / (1 + exp(-1.0 \* sum)));

}

//pochodna funkcji aktywcji

double DeltaRule::derivativeActivationFunction(double sum)

{

return (1.0\*exp(-1.0\*sum)) / (pow(exp(-1.0\*sum) + 1, 2));

}

//zwraca sume danego wejscia

double DeltaRule::getSum(int letter[], double \* weights)

{

double sum = 0.0;

for (int i = 0; i < numberOfWeights; i++)

sum += letter[i] \* weights[i];

return sum;

}

//funkcja uczaca

void DeltaRule::learn() {

cout << endl << "DELTARULE SIGMOID - LEARNING" << endl;

bool acceptableError = false; //zmienna, stwierdzajaca czy blad jest mozliwy do zaakceptowania

int epoch = 0; //numer epoki

/\*

for (int i = 0; i < numberOfWeights; i++)

cout << "Weights are: w" << i+1 << " = " << weights[i] << endl;

cout << endl;

\*/

do {

epoch++; // zwiekszenie numeru epoki

error = 0.0; //zerowanie glownego bledu w celu sprawdzenia bledow podczas jednej iteracji

for (int i = 0; i < numberOfSets; i++) {

//wynik otrzymany

output = activationFunction(getSum(inputData[i], weights));

//obliczanie roznicy pomiedzy wynikiem oczekiwanym a wynikiem otrzymanym

delta = expectedResults[i] - output;

//aktualizowanie wag

for (int j = 0; j < numberOfWeights; j++)

weights[j] += learningRate\*delta\*inputData[i][j] \* derivativeActivationFunction(getSum(inputData[i], weights));

//aktualizowanie bledu glownego

error += delta\*delta;

}

error /= 2;

//porownywanie bledu z progiem

if (error > 0.01)

acceptableError = false;

else

acceptableError = true;

} while (!acceptableError);

cout << "Epoch: " << epoch << endl;

cout << "MSE error: " << error << endl;

/\*

for (int i = 0; i < numberOfWeights; i++)

cout << "Weights are: w" << i+1 << " = " << weights[i] << endl;

\*/

}

//funkcja testujaca

void DeltaRule::test()

{

cout << "Test letters: " << endl;

for (int i = 0; i < numberOfSets; i++) {

cout << "Letter " << setTestLetters[i] << " is: ";

if (activationFunction(getSum(setTest[i], weights)) > 0.5) {

cout << "big";

}

else {

cout << "small";

}

cout << endl;

}

}

„data\_for\_learning”

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 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 1

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 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 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 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 1

1 1 1 1 1 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

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

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

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

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

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

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

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

0 0 0 0 0 0 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 0

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

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

„Perceptron.h”

#pragma once

#include <iostream>

#include <ctime>

#include <fstream>

using namespace std;

class Perceptron

{

};