Sprawozdanie Scenariusz 1 Aleksandra Karaś

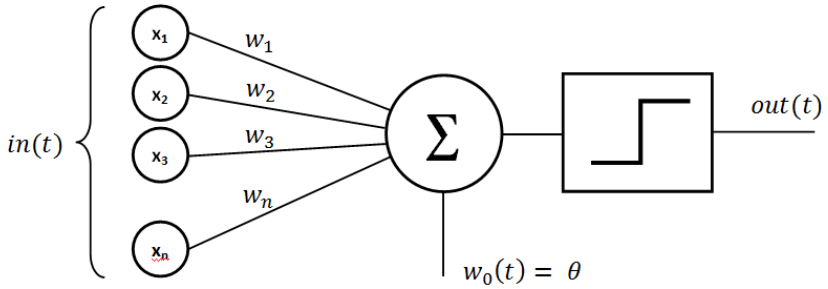
Budowa i działanie perceptronu

Cel ćwiczenia:

Głównym celem ćwiczenia było poznanie budowy i działania perceptronu poprzez implementację oraz uczenie perceptronu realizującego wybraną funkcję logiczną dwóch zmiennych.

Implementacja perceptronu:

Implementację w programie wykonałam na podstawie niżej załączonego obrazku



Kod:

```
static int Percepton(int threshold, double weights[], double x, double y) {
    double sum = x * weights[0] + y * weights[1] + weights[2];
    return sum >= threshold ? 1 : 0;
}
```

W implementacji użyłam jako funkcji aktywacji funkcję unipolarnej. Początkowa wartość progowa wynosiła a=0 (w moim programie threshold = 0).

$$y(x) = \begin{cases} 0 & dla \quad x < a \\ 1 & dla \quad x \geq a \end{cases}$$

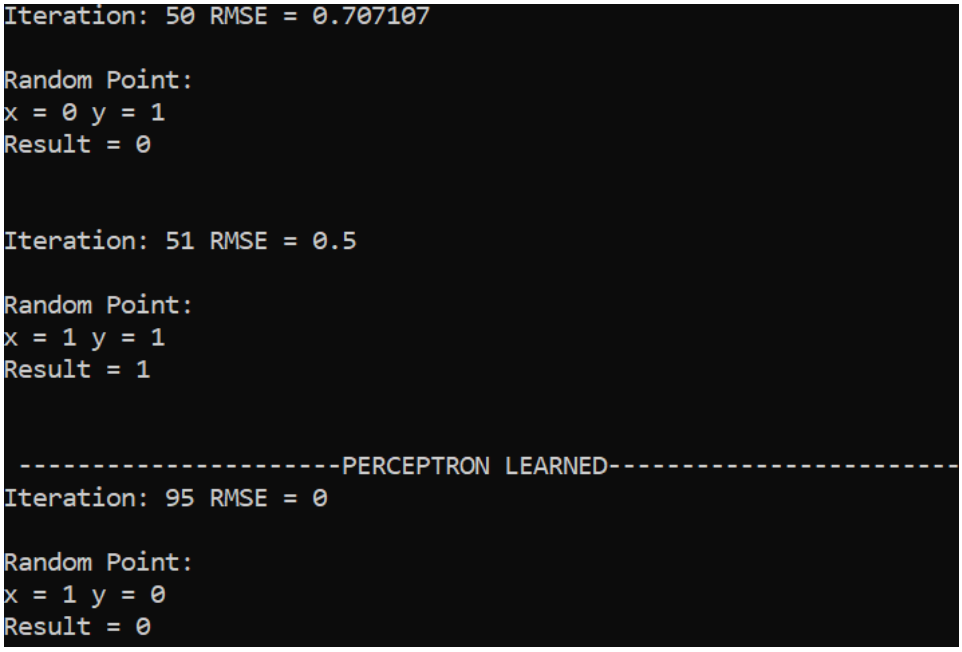
Implementacja algorytmu uczenia:

1. Zadeklarowanie współczynnika uczenia się, liczby danych oraz wartości progowej
2. Wprowadzenie danych wejściowych, wyjściowych (zgodnie z bramką logiczną AND)
3. Inicjacja wag, liczbami losowymi bliskimi 0
4. Rozpoczęcie nauczania:
    a. Wyznaczenie sumy iloczynu skalarnego wi*xi oraz sprawdzenie nierówności suma> thereshold -funkcja Percepton.
    b. Wyznaczenie błędu lokalnego, globalnego i średniej kwadratowej błędów - średniej różnicy między estymatorem a wartością estymowaną.
    c. Wyznaczenie nowych wag, poprzez dodanie iloczynu błędu lokalnego, danych wejściowych i współczynnika uczenia się.
    d. Porównanie wartości oczekiwanej z wartością perceptronu lub sprawdzenie czy nie skończyły się dane uczące.
5. Po nauczeniu RMSE = 0, sprawdzenie jeszcze 5 losowych punktów

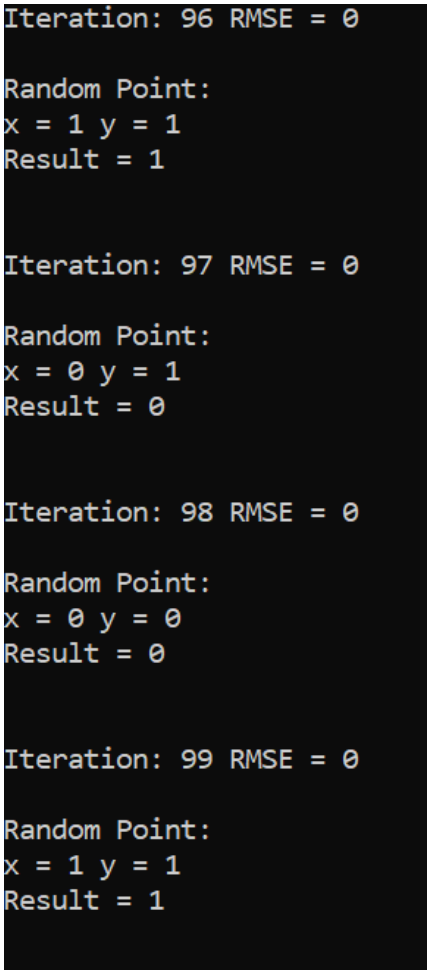
Przykład działania:

```
Iteration: 10 RMSE = 0.707107

Random Point:
x = 0 y = 0
Result = 1


Iteration: 11 RMSE = 0.707107

Random Point:
x = 0 y = 0
Result = 1


Iteration: 12 RMSE = 0.707107

Random Point:
x = 0 y = 0
Result = 1
```

Strzały perceptronu w trakcie nauki. Widzimy wysoką średnią kwadratową błędu i błędne wyniki bramki logicznej AND



W wyżej załączonym zdjęciu widzimy jak średnia kwadratowa błędu maleje, aż staje się równa 0, co świadczy o tym, że perceptron został nauczony. Następnie generujemy losowe punkty w celu sprawdzenia poprawności. Widzimy, że następna iteracja wykonała się poprawnie.

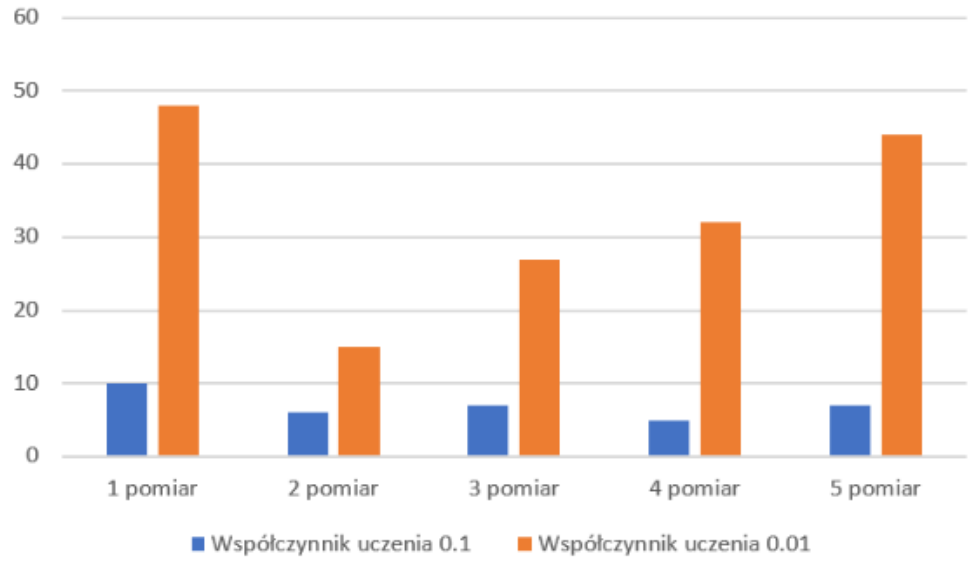


Przy kolejnych kontrolnych próbach widzimy ze perceptron bezbłędnie podaje wyniki bramki logicznej AND. Średnia kwadratowa błędu równa się w każdym wypadku 0.

Wyniki przeprowadzonego ćwiczenia:

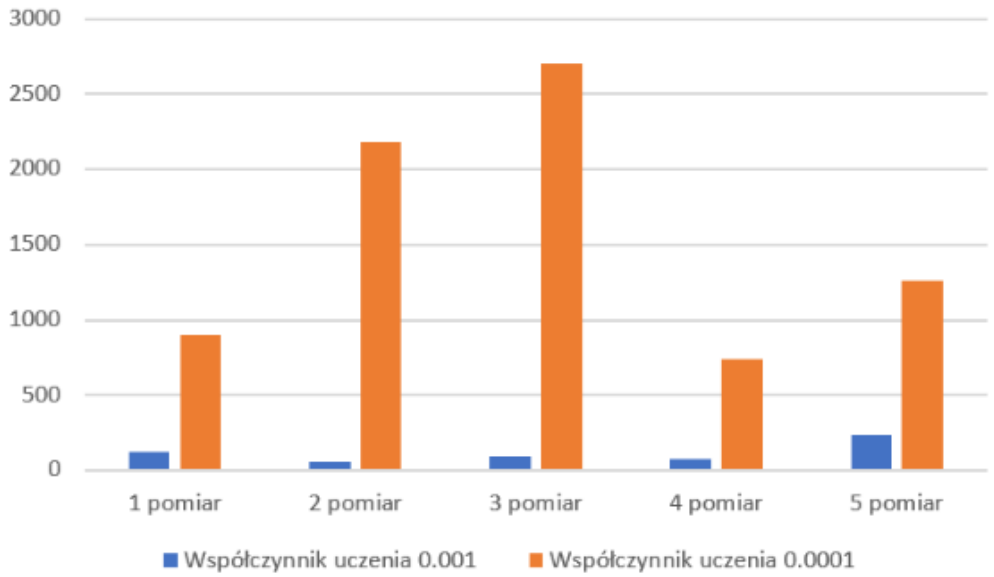
W zależności od przeprowadzonego testu, zmieniałam współczynnik uczenia. Każdy test zawierał 5 prób. Na podstawie tych prób powstały wykresy zależności wartości współczynnika uczenia a ilość iteracji potrzebnych do nauczenia prawidłowych odpowiedzi na zagadnienie bramki logicznej AND.

Współczynnik uczenia	Epoki	1 pomiar	2 pomiar	3 pomiar	4 pomiar	5 pomiar
0.1	100	10	6	7	5	7
0.01	100	48	15	27	32	44



W porównaniu współczynników uczenia się 0.1 i 0.01 widać, zmniejszenie tego współczynnika dziesięciokrotnie powoduje, że potrzeba znacznie więcej iteracji by z sukcesem nauczyć perceptron swojego zadania. Różnica w liczbie iteracji wynosi około od 2 – 4 razy.

Współczynnik uczenia	Epoki	1 pomiar	2 pomiar	3 pomiar	4 pomiar	5 pomiar
0.001	10000	121	59	97	74	234
0.0001	10000	902	2178	2705	425	617



Kolejne zmniejszenie współczynnika, pokazuje nam, że liczba iteracji diametralnie się zmienia. W niektórych przypadkach jest nawet kilkadziesiąt razy większa.

Niestety nie mogliśmy porównać współczynnika uczenia 0.0001 oraz 0.001 z liczbą iteracji jak przy współczynnikach 0.1 i 0.01 ze względu na to, że uczenie byłoby tak wolne, że by nie nastąpiło. Zatem w przypadku mniejszego współczynnika uczenia, musieliśmy zwiększyć ilość iteracji.

Wnioski oraz analiza:

W narzędziu jakim jest perceptron, musimy zwrócić szczególną uwagę na uważne dobieranie liczby epok, które w zależności od współczynnika uczenia może mieć duże wahania. Gdy zostanie zadanie zbyt mała liczba, może nie dojść do nauczenia się perceptronu i nie będzie w stanie poprawnie wykonywać zadania.

Kolejną rzeczą, na którą trzeba zwrócić uwagę jest dobieranie odpowiedniego współczynnika uczenia. To właśnie za jego sprawą, możemy kontrolować, jak szybko perceptron powinien się uczyć i odnieść sukces w uczeniu. Musimy jednak pamiętać, że współczynnik uczenia ma bezpośredni wpływ na liczbę epok.

Przedstawiony w zadaniu perceptron jednowarstwowy jest bardzo skutecznym narzędziem do klasyfikowania zbiorów liniowo separowalnych. Oznacza to, że uniemożliwia na przykład wytrenowanie złożonego z jednego neuronu perceptronu, który wykonywałby logiczną

operację XOR na wartościach wejść. W takim przypadku trzeba zbudować sieć z więcej niż jednego nautrona.

Listing kodu:

```
#include "stdafx.h"
#include <iostream>
#include <time.h>
#include <random>

using namespace std;

static int MaxIteration = 5000;
static double LearningRate = 0.0001;
static int NumInstances = 4;
static int threshold = 0;

static int Percepton(int threshold, double weights[], double x, double y) {
    double sum = x * weights[0] + y * weights[1] + weights[2];
    return sum >= threshold ? 1 : 0;
}
double fRand(double fMin, double fMax)
{
    double f = (double)rand() / RAND_MAX;
    return fMin + f * (fMax - fMin);
}
```

```
int main()
{
    srand((unsigned)time(NULL));
    int numInstances = 4;
    int* x = new int[numInstances];
    int* y = new int[numInstances];
    int* result = new int[numInstances];

    x[0] = 0; y[0] = 0; result[0] = 0;
    x[1] = 1; y[1] = 0; result[1] = 0;
    x[2] = 0; y[2] = 1; result[2] = 0;
    x[3] = 1; y[3] = 1; result[3] = 1;

    double* weights = new double[3];

    weights[0] = fRand(-0.5,0.5);
    weights[1] = fRand(-0.5,0.5);
    weights[2] = fRand(-0.5,0.5);

    double localError, globalError;
    int p, iteration, perceptonResult;
    int ifLearnd = 0;
    iteration = 0;
```

```
    do {
        iteration++;
        globalError = 0;
        //Perception learning
        for (p = 0; p < numInstances; p++) {
            perceptonResult = Percepton(threshold, weights, x[p], y[p]);
            localError = result[p] - perceptonResult;
            weights[0] += LearningRate * localError * x[p];
            weights[1] += LearningRate * localError * y[p];
            weights[2] += LearningRate * localError;
            globalError += (localError*localError);
        }
        //Checking if perception is learned
        if (ifLearnd == 0 && globalError == 0) {
            ifLearnd = 1;
            cout << "\n ----------------------PERCEPTRON LEARNED--------------------------------";
            iteration = MaxIteration - 5;
        }
        cout << "\nIteration: " << iteration << " RMSE = " << sqrt((globalError / numInstances));/*Root Mean Squared Erro
        int x1 = (int)rand() % 2;
        int y1 = (int)rand() % 2;
        perceptonResult = Percepton(threshold, weights, x1, y1);
        cout << "\n\nRandom Point:\n" << "x = "<< x1 << " y = " << y1<<"\nResult = "<< perceptonResult <<"\n\n";
    }while (iteration < MaxIteration);
```

```
    cout<<"\n\nDecision boundary equation:";
    cout<< weights[0] << "*x +" <<  weights[1] << "*y + " << weights[2] << " = 0\n";
    system("PAUSE");
}
```

Materiały źródłowe:

https://pl.wikipedia.org/wiki/Perceptron

http://www.algorytm.org/sztuczna-inteligencja/sztuczny-neuron.html

https://www.ii.uni.wroc.pl/~aba/teach/NN/w4.pdf

http://edu.pjwstk.edu.pl/wyklady/nai/scb/wyklad3/w3.htm

http://galaxy.uci.agh.edu.pl/~vlsi/AI/wstep/

http://www.cs.put.poznan.pl/rklaus/assn/percep.htm