**SCENARIUSZ NR 1**

**PERCEPTRON**

**Jan Pikulski**

1. **Temat ćwiczenia**

Budowa i działanie perceptronu

1. **Cel ćwiczenia**

Celem ćwiczenia jest poznanie budowy i działanie perceptronu poprzez implementację oraz

uczenie perceptronu realizującego wybraną funkcję logiczną dwóch zmiennych.

1. **Opis budowy i algorytmu uczenia**

W programie wykorzystano podstawowy schemat budowy w którym dane wejściowe wraz ze swoimi wagami, trafiają do perceptronu, gdzie są sumowane ∑x[i]w[i] (gdzie x jest wektorem danych wejściowych, a w jest wektorem ich wag).

Algorytm ma za zadanie wykonanie programu uczącego się logiki działania AND.

Na początku wybierane są losowe wartości z przedziału 0-1 i przypisywane do poszczególnych wag. Dla każdego zestawu danych wejściowych obliczana jest dana wyjściowa za pomocą wzoru: x\*w[0] + y\*w[1] + w[2]. Następnie jeśli ta suma jest większa od zmiennej *treshold*, perceptron zwraca wartość 1. W przeciwnym przypadku zwraca wartość 0.

Po wykonaniu powyższych operacji, obliczny jest błąd lokalny (różnica oczekiwanego i otrzymanego wyniku), który posłuży do zaktualizowania wag zmiennych:

w[0] += localError \* learnRate \* x[i];

w[1] += localError \* learnRate \* y[i];

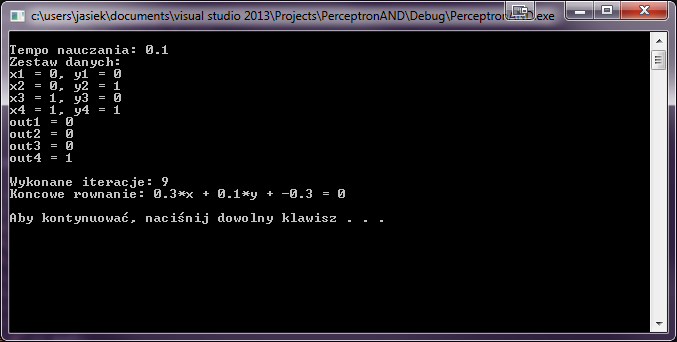
w[2] += learnRate \* localError;

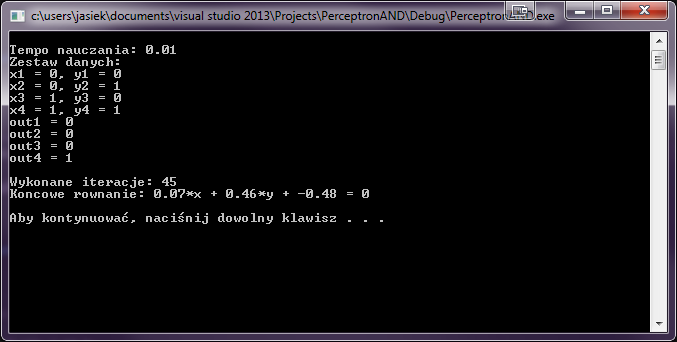
globalError += (localError\*localError);

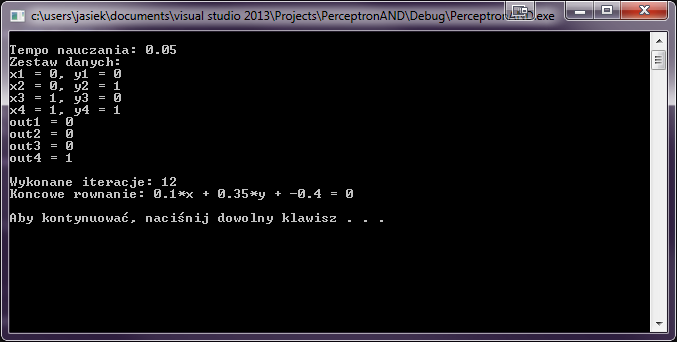
Warunkiem zakończenia pracy programu jest warunek: globalError != 0 && iteracja mniejsza od maksymalnej (druga część warunku jest potrzebna, aby uniknąć nieskończonej pętli).

Po zakończeniu nauki, program wyświetla wzór, według którego rozpoznaje klasę danego punktu (czy zestaw danych zwróci wartość 0 czy 1).

1. **Zestawienie i analiza wyników**







Jak widać, program nie potrzebuje zbyt wiele iteracji aby poradzić sobie z tym problemem. Oczywiście największy wpływ na szybkość wykonania całego zadania, miał współczynnik tempa uczenia. Przy jego zwiększeniu, program potrzebował mniej iteracji, aby znaleźć rozwiązanie, jednak lepiej było utrzymywać go w wartościach 0 – 1, ponieważ po ich przekroczeniu, zaczynały pojawiać się znaczące błędy.

1. **Wnioski:**

Dzięki temu ćwiczeniu można było lepiej zrozumieć działanie prostego perceptronu. Ważnym wnioskiem, który należało wyciągnąć jest to, że tego typu uczenia nie należy wykonywać „w nieskończoność” ponieważ od pewnego momentu, wartości błędu zaczną ponownie wzrastać. Dlatego też lepiej ograniczyć działanie programu za pomocą limitu iteracji i innych warunków granicznych.

Warto też pamiętać, że zwiększenie ilości danych uczących może poprawić dokładność przeprowadzonego uczenia, jednak nie należy zbytnio komplikować zagadnienia jeśli zależy nam na jak najsprawniejszym działaniu programu.

1. **Listing kodu:**

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

float outCalculate(float\* w, int x, int y);

int main(){

srand(time(NULL));

const int INST\_NUM = 4;

float learnRate = 0.4;

int iter = 0;

float localError, globalError;

int x[INST\_NUM];

int y[INST\_NUM];

int out[INST\_NUM];

float\* w = new float(3);

int MAX = 400;

for (int i = 0; i < 3; i++)

{

w[i] = (rand() % 10) / 10.0;

}

x[0] = 0; y[0] = 0;

x[1] = 0; y[1] = 1;

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

x[3] = 1; y[3] = 1;

out[0] = 0;

out[1] = 0;

out[2] = 0;

out[3] = 1;

do{

iter++;

globalError = 0;

for (int i = 0; i < INST\_NUM; i++){

float output = outCalculate(w, x[i], y[i]);

localError = out[i] - output;

w[0] += localError \* learnRate \* x[i];

w[1] += localError \* learnRate \* y[i];

w[2] += learnRate \* localError;

globalError += (localError\*localError);

}

} while (globalError != 0 && iter < MAX);

//cout << w[0] << endl << w[1] << endl << w[2] << endl;

cout << endl << "Tempo nauczania: " << learnRate << endl;

cout << "Zestaw danych: " << endl;

cout << "x1 = 0, y1 = 0" << endl;

cout << "x2 = 0, y2 = 1" << endl;

cout << "x3 = 1, y3 = 0" << endl;

cout << "x4 = 1, y4 = 1" << endl;

cout << "out1 = 0" << endl;

cout << "out2 = 0" << endl;

cout << "out3 = 0" << endl;

cout << "out4 = 1" << endl << endl;

cout << "Wykonane iteracje: " << iter << endl;

cout << "Koncowe rownanie: " << w[0] << "\*x + " << w[1] << "\*y + " << w[2] << " = 0" << endl << endl;

system("PAUSE");

}

float outCalculate(float\* w, int x, int y){

float treshold = 0.0;

float sum = (x\*w[0] + y\*w[1] + w[2]);

return (sum >= treshold) ? 1 : 0;

}