**Scenariusz 1**

Białek Tomasz, gr. 1

**Temat ćwiczenia:** Budowa i działanie perceptronu

**1. Cel ćwiczenia**

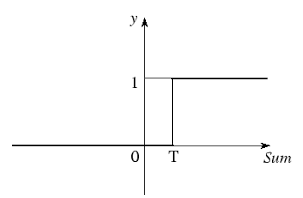
Celem ćwiczenia jest poznanie budowy i działanie perceptronu poprzez implementację oraz

uczenie perceptronu realizującego wybraną funkcję logiczną dwóch zmiennych.

**2. Syntetyczny opis budowy oraz wykorzystanego algorytmu ucznia.**

Perceptron został zaimplementowany zgodnie z modelem McCullocha-Pittsa (omówiony 19.10.2017 r. na wykładzie). Sygnały wejściowe xi (i = 1, 2, …, N) są sumowane z odpowiednimi wagami wi  w sumatorze. Sygnał wyjściowy neuronu yj można opisać wzorem:

gdzie funkcje wyrażamy wzorem:



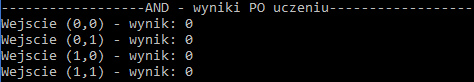
Perceptron według modelu McCullocha-Pittsa

Model ten jest modelem matematycznym, w którym stan neuronu określony jest na podstawie stanu sygnałów wejściowych neuronów w chwili poprzedniej a perceptron ma za zadanie nauczyć się funkcji logicznej AND. Na początku wagi dobrano w sposób losowy z zakresu (0;1). Dzięki nim obliczamy wartość sygnału wyjściowego yi. W wyniku porównania aktualnej wartości yi oraz wartości oczekiwanej di dokonywana jest aktualizacja wag (w przypadku, gdy wartość otrzymana i oczekiwana są takie same – wagi nie zmieniają się). Podczas uczenia wykorzystuje się jedynie informacje o aktualnej wartości sygnału wyjściowego neuronu oraz z wag.

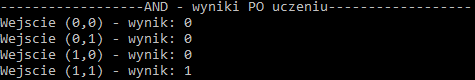
Ćwiczenie zostało wykonane w języku C++, w którym zastosowano uczenie nadzorowane (z pomocą nauczyciela). Funkcja aktywacji jest funkcją nieliniową unipolarną.

**3. Otrzymane wyniki i ich analiza**

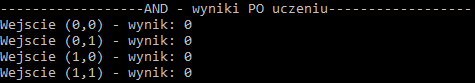
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.01, 1 epoka



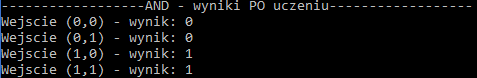
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.01, 5 epok



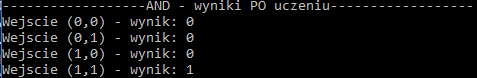
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.01, 10 epok



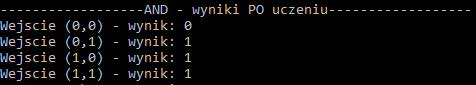
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.01, 20 epok



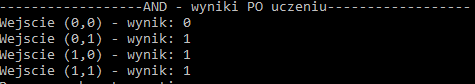
***Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.01, 50 epok***



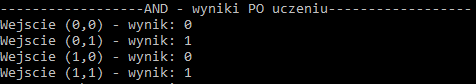
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.1, 1 epoka



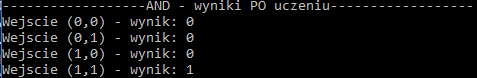
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.1, 5 epok



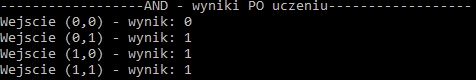
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.1, 10 epok



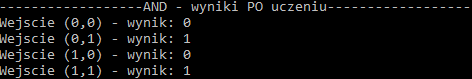
**Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.1, 20 epok**



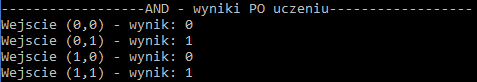
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.5, 1 epok



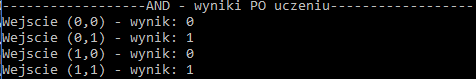
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.5, 5 epok



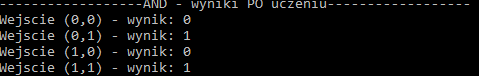
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.5, 10 epok



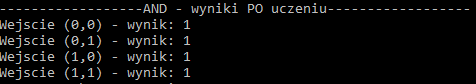
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.5, 20 epok



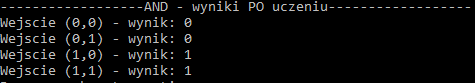
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 0.5, 50 epok



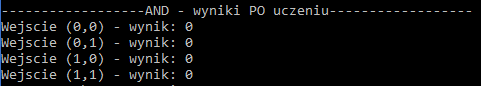
Jedna dana ucząca [(1,1) -> 1], współczynnik uczenia = 1, 10 epok



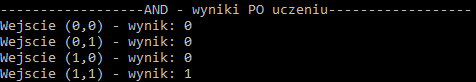
Dwie dane uczące [(1,1) -> 1, (1,0) ->0], współczynnik uczenia = 0.01, 1 epoka



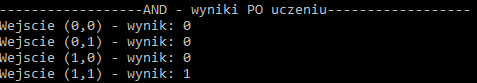
Dwie dane uczące [(1,1) -> 1, (1,0) ->0], współczynnik uczenia = 0.01, 5 epok



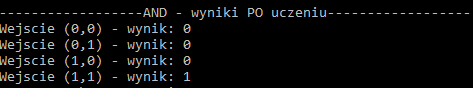
**Dwie dane uczące [(1,1) -> 1, (1,0) ->0], współczynnik uczenia = 0.01, 10 epok**



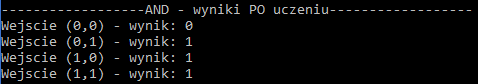
Dwie dane uczące [(1,1) -> 1, (1,0) ->0], współczynnik uczenia = 0.1, 1 epoka



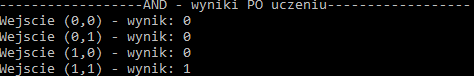
Dwie dane uczące [(1,1) -> 1, (1,0) ->0], współczynnik uczenia = 0.1, 5 epok



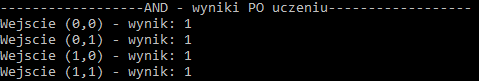
Dwie dane uczące [(1,1) -> 1, (1,0) ->0], współczynnik uczenia = 0.5, 1 epoka



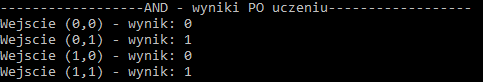
***Dwie dane uczące [(1,1) -> 1, (1,0) ->0], współczynnik uczenia = 0.5, 5 epok***



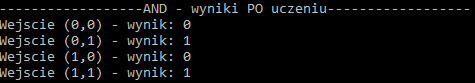
Dwie dane uczące [(1,1) -> 1, (1,0) ->0], współczynnik uczenia = 1, 1 epoka



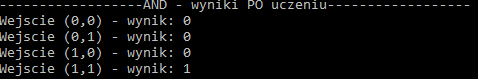
Dwie dane uczące [(1,1) -> 1, (1,0) ->0], współczynnik uczenia = 1, 5 epok



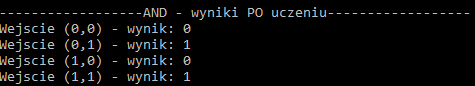
Dwie dane uczące [(1,1) -> 1, (1,0) ->0], współczynnik uczenia = 1, 10 epok



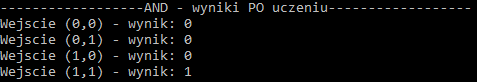
**Dwie dane uczące [(1,1) -> 1, (1,0) ->0], współczynnik uczenia = 1, 20 epok**



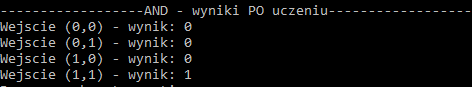
Trzy dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0], współczynnik uczenia = 0.01, 1 epoka



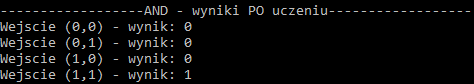
**Trzy dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0], współczynnik uczenia = 0.01, 5 epok**



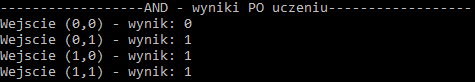
Trzy dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0], współczynnik uczenia = 0.1, 1 epoka



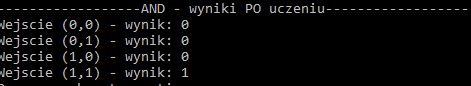
Trzy dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0], współczynnik uczenia = 0.1, 5 epok



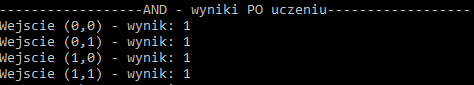
Trzy dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0], współczynnik uczenia = 0.5, 1 epoka



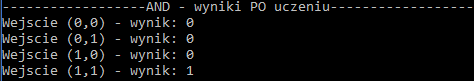
**Trzy dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0], współczynnik uczenia = 0.5, 5 epok**



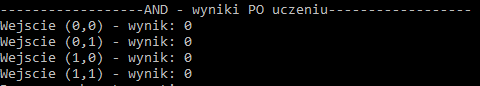
Trzy dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0], współczynnik uczenia = 1, 1 epoka



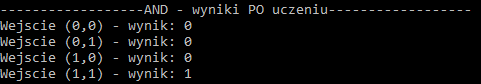
**Trzy dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0], współczynnik uczenia = 1, 5 epok**



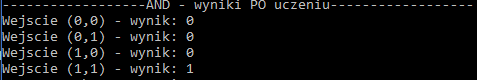
Cztery dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0, (0,0)->0], współczynnik uczenia = 0.01, 1 epoka



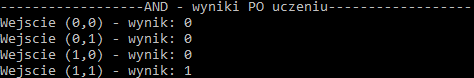
**Cztery dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0, (0,0)->0], współczynnik uczenia = 0.01, 5 epok**



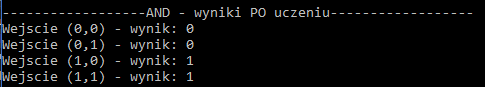
Cztery dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0, (0,0)->0], współczynnik uczenia = 0.1, 1 epoka



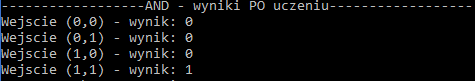
Cztery dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0, (0,0)->0], współczynnik uczenia = 0.1, 5 epok



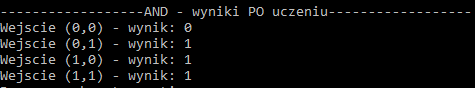
Cztery dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0, (0,0)->0], współczynnik uczenia = 0.5, 1 epoka



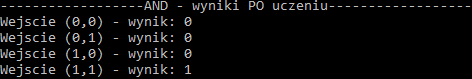
**Cztery dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0, (0,0)->0], współczynnik uczenia = 0.5, 5 epok**



Cztery dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0, (0,0)->0], współczynnik uczenia = 1, 1 epoka



**Cztery dane uczące [(1,1) -> 1, (1,0) ->0, (0,1) ->0, (0,0)->0], współczynnik uczenia = 1, 5 epok**



Z otrzymanych danych można zaobserwować, że na szybkość uczenia ma wpływ współczynnik uczenia oraz liczba danych. Efektywność uczenia zwiększa się dla małego współczynnika uczenia oraz większej ilości danych uczących – otrzymujemy większy stosunek poprawnych odpowiedzi podczas dania do nauki czterech danych uczących. Jednakże, gdy wybierze się skrajną wartość tylko jednego z nich (współczynnik uczenia lub liczba danych uczących), to efektywność już nie jest dobra. Dla czterech danych uczących niezależnie od współczynnika uczenia mamy praktycznie od razu poprawnie działający perceptron, który nie popełniał błędów w kolejnych testach.

Z powyższego wykresu wynika, że efektywność uczenia wzrasta wraz z ilością danych uczących oraz współczynnikiem uczenia. Dla konkretnej liczby danych uczących i najwyższym współczynniku uczenia równym 1.0 otrzymywano podobne wyniki jak dla zwiększonej o jeden liczby danych uczących o najmniejszym współczynniku uczenia równym 0.01.

**4. Podsumowanie**

Skuteczność procesu uczenia zależy od liczby danych uczących. Im ich więcej, tym sieć jest w stanie szybciej się nauczyć. Jeśli liczba danych uczących będzie za mała perceptron nie będzie w stanie nauczyć się i dać poprawnych odpowiedzi.

Na skuteczność uczenia ma także współczynnik uczenia. Najlepiej, gdy znajduje się on w przedziale (0,1 – 1). Inne wartości mogą spowodować złe uczenie się, które spowoduje sytuację, gdy perceptron nie będzie w stanie nauczyć się poprawnych odpowiedzi.

Jesteśmy w stanie tak manipulować współczynnikiem uczenia, aby nie było potrzebna duża liczba danych uczących, co spowoduje, że proces uczenia przebiegnie szybko (jednocześnie zachowując jego efektywność).

**5. Listing kodu**

**Perceptron.h**

class Perceptron {

public:

unsigned numberOfInputs;

double \*weights;

double learningRate;

double getRandomDouble();

double getValue(const double inputs[]);

int getResult(const double inputs[]);

double getWeightAt(unsigned index);

const double \*getWeights();

unsigned getNumOfInputs();

double getTrainingRate();

void setTrainingRate(double trainingRate);

void setWeightAt(unsigned index, double weight);

void setWeights(const double \*weights);

void changeWeights(int actualResult, int desiredResult, const double inputs[]);

void learn(const double inputs[], int expectedResult);

Perceptron(unsigned numOfInputs, double trainingRate);

};

**Perceptron.cpp**

#include <iostream>

#include <ctime>

#include "Perceptron.h"

//zwraca: suma wszytkich wejście\*waga

double Perceptron::getValue(const double inputs[]){

double sum = 0;

for (int i = 0; i < this->numberOfInputs; i++)

sum += this->weights[i] \* inputs[i];

return sum;

}

//ustaw: współczynnik uczenia

void Perceptron::setTrainingRate(double learningRate) {

this->learningRate = learningRate;

}

//ucz się

void Perceptron::learn(const double inputs[], int expectedResult){

int result = getResult(inputs);

if (result == expectedResult)

return;

changeWeights(result, expectedResult, inputs);

}

//zwraca: wynik końcowy dla pojedynczego wejścia <porównuje wartość wejścia z funkcją binarną z progiem>

int Perceptron::getResult(const double inputs[]){

return (getValue(inputs) > 1) ? 1 : 0;

}

//zwraca: waga dla pojedynczego wejścia

double Perceptron::getWeightAt(unsigned index){

return this->weights[index];

}

//zwraca: waga wszystkich wejść

const double \* Perceptron::getWeights(){

return this->weights;

}

//zwraca: ilość wejść

unsigned Perceptron::getNumOfInputs(){

return this->numberOfInputs;

}

//zwraca: współczynnik uczenia

double Perceptron::getTrainingRate(){

return this->learningRate;

}

//ustaw: waga dla danego wejścia

void Perceptron::setWeightAt(unsigned index, double weight){

this->weights[index] = weight;

}

//ustaw: wagi dla wszystkich wejść

void Perceptron::setWeights(const double \* weights){

unsigned i;

for (i = 0; i < this->numberOfInputs; i++)

this->weights[i] = weights[i];

}

//funkcja losująca dla wag

double Perceptron::getRandomDouble(){

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

return randValue;

}

//funkcja - zmienia wagi

void Perceptron::changeWeights(int actualResult, int desiredResult, const double inputs[]){

for (int i = 0; i < this->numberOfInputs; i++)

this->weights[i] += this->learningRate \* (desiredResult - actualResult) \* inputs[i];

}

//konstruktor

Perceptron::Perceptron(unsigned numOfInputs, double trainingRate){

srand(time(NULL));

this->learningRate = trainingRate;

this->numberOfInputs = numOfInputs;

if (this->numberOfInputs <= 0)

this->numberOfInputs = 1;

this->weights = new double[numberOfInputs];

for (int i = 0; i < this->numberOfInputs; i++)

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

}

**Source.cpp**

#include <iostream>

#include "Perceptron.h"

using namespace std;

int main() {

const double

ONE\_ONE[] = { 1, 1, 1 },

ZERO\_ONE[] = {1, 0, 1 },

ONE\_ZERO[] = {1, 1, 0 },

ZERO\_ZERO[] = { 1, 0, 0 };

int startEpoch = 0;

int numberOfEpochs = 15;

int numberOfInputs = 3;

double learningRate = 0.1;

Perceptron perceptron(numberOfInputs, learningRate);

cout << "----------------AND - wyniki PRZED uczeniem----------------" << endl;

cout << "Wejscie (0,0) - wynik: " << perceptron.getResult(ZERO\_ZERO) << endl;

cout << "Wejscie (0,1) - wynik: " << perceptron.getResult(ZERO\_ONE) << endl;

cout << "Wejscie (1,0) - wynik: " << perceptron.getResult(ONE\_ZERO) << endl;

cout << "Wejscie (1,1) - wynik: " << perceptron.getResult(ONE\_ONE) << endl;

while (startEpoch < numberOfEpochs) {

perceptron.learn(ZERO\_ZERO, 0);

perceptron.learn(ZERO\_ONE, 0);

perceptron.learn(ONE\_ZERO, 0);

perceptron.learn(ONE\_ONE, 1);

//cout << "Wagi: " << \*perceptron.getWeights() << endl;

//cout << "Wagi: " << perceptron.getWeightAt(0) << " " << perceptron.getWeightAt(1) << endl;

startEpoch++;

}

cout<< endl << "------------------AND - wyniki PO uczeniu------------------" << endl;

cout << "Wejscie (0,0) - wynik: " << perceptron.getResult(ZERO\_ZERO) << endl;

cout << "Wejscie (0,1) - wynik: " << perceptron.getResult(ZERO\_ONE) << endl;

cout << "Wejscie (1,0) - wynik: " << perceptron.getResult(ONE\_ZERO) << endl;

cout << "Wejscie (1,1) - wynik: " << perceptron.getResult(ONE\_ONE) << endl;

system("pause");

return 0;

}