Scenariusz 1

Temat ćwiczenia : Budowa i działanie perceptronu

Wykonał: Rafał Chołody IS, gr. 1, 286152

1. Cel ćwiczenia

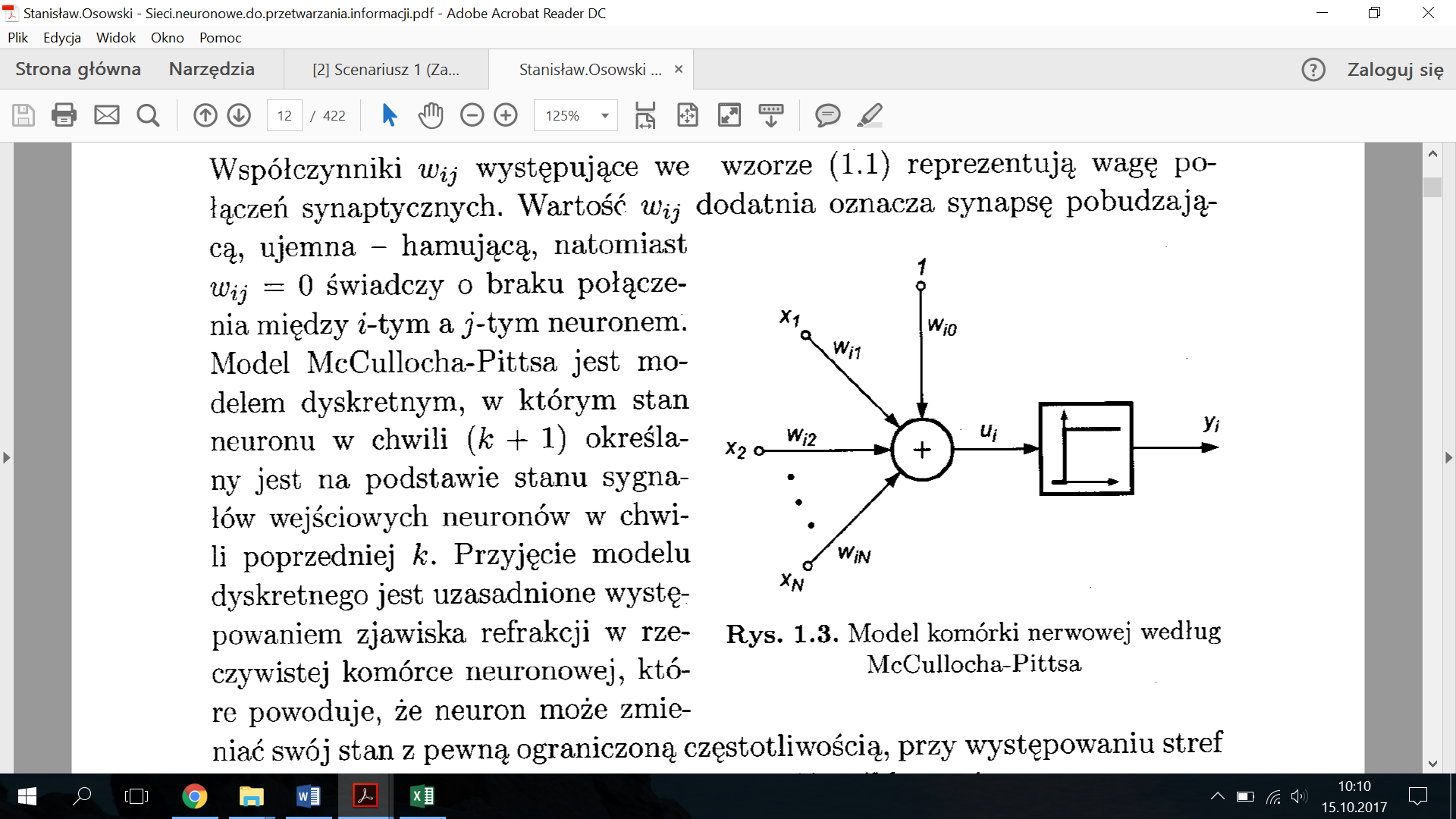
Celem ćwiczenia jest poznanie budowy i działanie perceptronu poprzez implementację oraz

uczenie perceptronu realizującego wybraną funkcję logiczną dwóch zmiennych.

1. **Opis budowy perceptronu i wykorzystanego algorytmu ucznia.**

Perceptron został zaimplementowany zgodnie z modelem McCullocha-Pittsa. Sygnały wejściowe xj (j = 1, 2, …, N) są sumowane z odpowiednimi wagami wij w sumatorze, a następnie jest dodawany jest sygnał polaryzacji x0 = 1 dla którego waga wynosiwio. Sygnał wyjściowy neuronu yi wyraża się, więc zależnością:

gdzie funkcje wyrażamy wzorem:



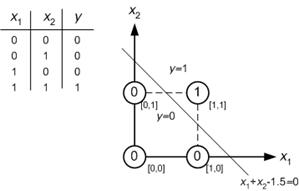
Rys. 1 Perceptron według modelu McCullocha-Pittsa[1]

Model ten jest modelem dyskretnym, w którym stan neuronu w chwili (k+1) określony jest na podstawie stanu sygnałów wejściowych neuronów w chwili poprzedniej k.

Do uczenia perceptronu wykorzystano regułę Widrowa-Hoffa uzupełnioną o współczynnik uczenia . Na wstępie wagi dobrano w sposób losowy z zakresu 0-1. Dzięki nim obliczamy wartość sygnału wyjściowego yi. W wyniku porównania aktualnej wartości yi oraz wartości zadanej di dokonuje się aktualizacji wag. Jeżeli wartości otrzymana i zadana się nie różnią to wagi pozostają niezmienione. A więc zgodnie z tą regułą dobór wag odbywa się według zależności:

Podczas uczenia wykorzystuje się jedynie informacje o aktualnej wartości sygnału wyjściowego neuronu oraz wartości zadanej[1].

Zaimplementowany perceptron składa się z jednej warstwy i jego zadaniem jest nauczyć się funkcji logicznej AND, a więc musi spełniać poniższe zależności.

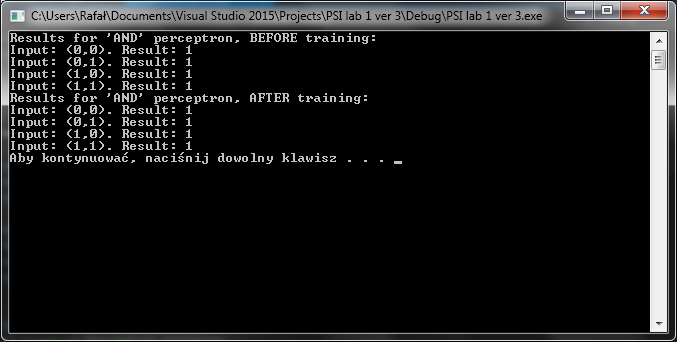


Rys. 2 Funkcja logiczna AND

Otrzymane wyniki: z nauczycielem

training iterations =1:

TRAINING\_RATE = 0.2;



training iterations = 5:

TRAINING\_RATE = 0.2;



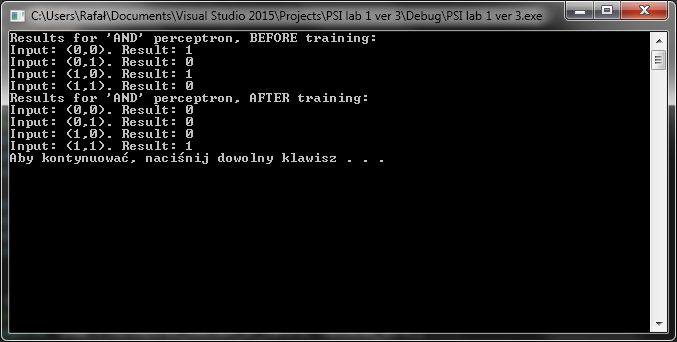
training iterations = 10

TRAINING\_RATE = 0.2;



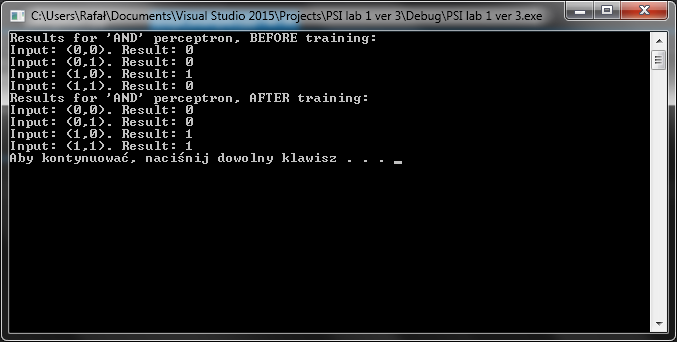
training\_iterations = 16:

TRAINING\_RATE = 0.2;



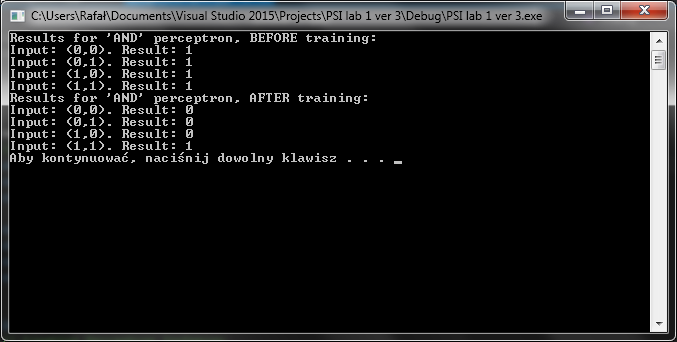
training iterations = 1

TRAINING RATE = 0,9



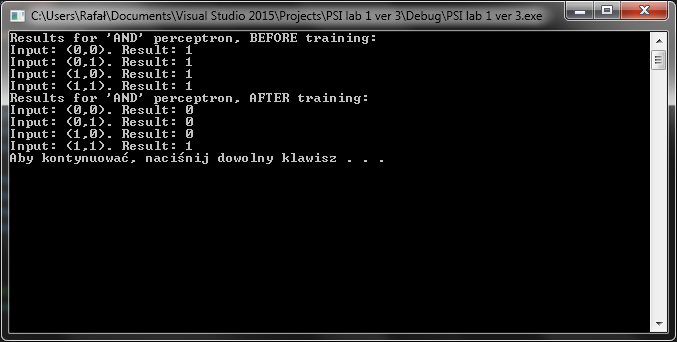
training iterations = 5

TRAINING RATE = 0,9



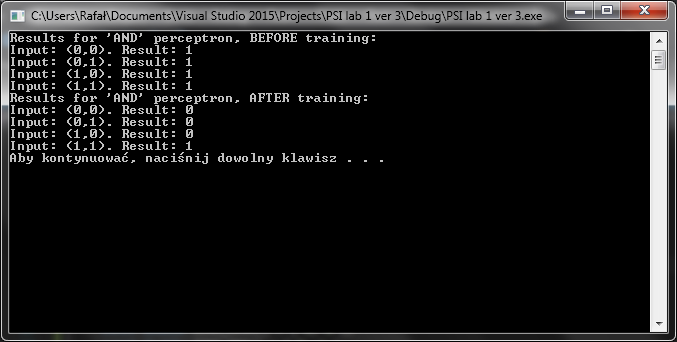
training iterations = 10

TRAINING RATE = 0,9



training iterations = 16

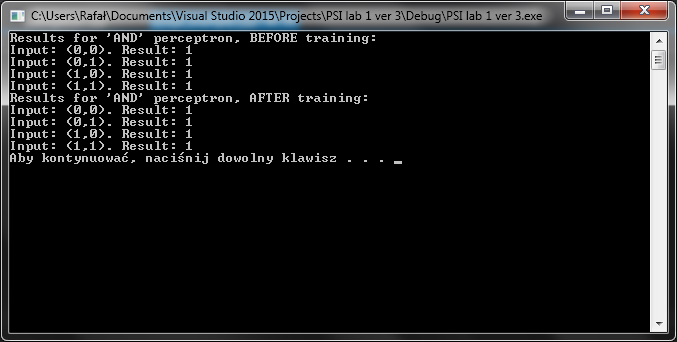
TRAINING RATE = 0,9



Otrzymane wyniki: bez nauczyciela

training iterations = 16

TRAINING RATE = 0,9



Wykresy:

Na podstawie otrzymanych wyników można stwierdzić, że szybkość uczenia zależna jest od współczynnika uczenia oraz liczby danych. Sama efektywność uczenia znacząco zwiększa się dla małego współczynnika uczenia oraz większej ilości danych uczących- w ten sposób otrzymujemy większy stosunek poprawnych odpowiedzi podczas dania do nauki czterech danych uczących. Można również zaobserwować, że gdy wybierze się skrajną wartość tylko jednego z nich (współczynnik uczenia lub liczba danych uczących) to efektywność już nie jest stosunkowo tak dobra. Dla czterech danych uczących niezależnie od współczynnika uczenia mamy praktycznie od razu poprawnie działający perceptron, który nie popełniał błędów w kolejnych testach.

Wnioski:

* Skuteczność procesu uczenia zależy od liczby danych uczących, które zostaną dostarczone do naszego programu. Kiedy ich ilość rośnie wtedy proces ten wykazuje większą skuteczność. Natomiast kiedy liczba ta będzie za mała to perceptron nie będzie działał poprawnie i nie będzie w stanie przeprowadzić udanej nauki. Dąży się do tego aby liczba niepoprawnie wykonany operacji była wartością najbliższą zeru
* Kolejnym aspektem wpływającym na skuteczność jest współczynnik uczenia(TRAINING RATE). Wraz z jego wzrostem proces uczenia jest poprawniejszy. Należy tak dobrać jego wartość, aby przy zadanej liczbie danych uczących.
* Możliwe jest takie manipulowanie współczynnikiem uczenia, aby nie było potrzebna duża liczba danych uczących, co za tym idzie proces uczenia przebiegał by szybciej a jego efektywność ciągle byłaby zachowana.
* Jednowarstwowe sieci neuronowe można stosować do wykonywania prostych funkcji logicznych. Po odpowiednio przeprowadzonym procesie uczenia otrzymano bezbłędnie działający perceptron wykonujący funkcję logiczną AND dwóch zmiennych.
* Ogromne znaczenie ma zastosowanie procesu uczenia z nauczycielem lub bez, wyniki procesu bez nauczyciela są zazwyczaj o wiele gorsze niżeli w sytuacji przeciwnej
* Spore znaczenie ma również kwestia danych uczących na których pracuje perceptron, przy podaniu wartości [0,0] było prawie niemożliwe osiągnięcie stanu w którym to program nauczył się poprawnych wartości dla operacji, gdzie wymagana była liczba 1.
* Ciekawostką może być to, że ten sam proces z użyciem cyfr [1,1] przebiegał dużo poprawniej i potrafił dojść do wartości 0. Ma to prawdopodobnie związek z wartościami punktów aktywacji

Kod programu:

#pragma once

#ifndef PERCEPTRON\_H

#define PERCEPTRON\_H

#ifdef \_\_cplusplus

extern "C" {

#endif

typedef struct Perceptron {

unsigned numInputs\_;

double \*weights\_;

double threshold\_;

double trainingRate\_;

} Perceptron;

// Public

Perceptron \*Perceptron\_new(unsigned numOfInputs, double trainingRate);

double Perceptron\_getValue(const Perceptron \*perceptron, const double inputs[]);

void Perceptron\_setTrainingRate(Perceptron \*perceptron, double trainingRate);

void Perceptron\_train(Perceptron \*perceptron, const double inputs[], int expectedResult);

int Perceptron\_getResult(const Perceptron \*perceptron, const double inputs[]);

double Perceptron\_getWeightAt(const Perceptron \*perceptron, unsigned index);

const double \*Perceptron\_getWeights(const Perceptron \*perceptron);

unsigned Perceptron\_getNumOfInputs(const Perceptron \*perceptron);

double Perceptron\_getThreshold(const Perceptron \*perceptron);

double Perceptron\_getTrainingRate(const Perceptron \*perceptron);

void Perceptron\_setWeightAt(Perceptron \*perceptron, unsigned index, double weight);

void Perceptron\_setWeights(Perceptron \*perceptron, const double \*weights);

void Perceptron\_setThreshold(Perceptron \*perceptron, double threshold);

// Private

double \_Perceptron\_getRandomDouble();

void \_Perceptron\_changeWeights(Perceptron \*perceptron, int actualResult, int desiredResult, const double inputs[]);

#ifdef \_\_cplusplus

}

#endif

#endif

#include <stdlib.h>

#include <time.h>

#include "perceptron.h"

#ifdef \_\_cplusplus

extern "C" {

#endif

Perceptron \*Perceptron\_new(unsigned numOfInputs, double trainingRate) {

unsigned i;

static int initializedRandomization\_ = 0;

Perceptron \*perc = (Perceptron\*)malloc(sizeof(Perceptron));

perc->numInputs\_ = numOfInputs;

if (perc->numInputs\_ <= 0) {

perc->numInputs\_ = 1;

}

perc->trainingRate\_ = trainingRate;

perc->weights\_ = (double\*)malloc(perc->numInputs\_ \* sizeof(double));

if (!initializedRandomization\_) {

srand(time(NULL));

initializedRandomization\_ = 1;

}

for (i = 0; i < perc->numInputs\_; i++) {

perc->weights\_[i] = \_Perceptron\_getRandomDouble();

}

perc->threshold\_ = \_Perceptron\_getRandomDouble();

return perc;

}

double Perceptron\_getValue(const Perceptron \*perceptron, const double inputs[]) {

unsigned i;

double ans = 0;

for (i = 0; i < perceptron->numInputs\_; i++) {

ans += perceptron->weights\_[i] \* inputs[i];

}

return ans;

}

//void Perceptron\_setTrainingRate(Perceptron \*perceptron, double trainingRate) {

// perceptron->trainingRate\_ = trainingRate;

//}

void Perceptron\_train(Perceptron \*perceptron, const double inputs[], int expectedResult) {

int result = Perceptron\_getResult(perceptron, inputs);

if (result == expectedResult) {

return;

}

\_Perceptron\_changeWeights(perceptron, result, expectedResult, inputs);

}

int Perceptron\_getResult(const Perceptron \*perceptron, const double inputs[]) {

return (Perceptron\_getValue(perceptron, inputs) >= perceptron->threshold\_);

}

double Perceptron\_getWeightAt(const Perceptron \*perceptron, unsigned index) {

return perceptron->weights\_[index];

}

const double \*Perceptron\_getWeights(const Perceptron \*perceptron) {

return perceptron->weights\_;

}

//unsigned Perceptron\_getNumOfInputs(const Perceptron \*perceptron) {

// return perceptron->numInputs\_;

//}

//double Perceptron\_getThreshold(const Perceptron \*perceptron) {

// return perceptron->threshold\_;

//}

//double Perceptron\_getTrainingRate(const Perceptron \*perceptron) {

// return perceptron->trainingRate\_;

//}

//void Perceptron\_setWeightAt(Perceptron \*perceptron, unsigned index, double weight) {

// perceptron->weights\_[index] = weight;

//}

//void Perceptron\_setWeights(Perceptron \*perceptron, const double \*weights) {

// unsigned i;

// for (i = 0; i < perceptron->numInputs\_; i++) {

// perceptron->weights\_[i] = weights[i];

// }

//}

//void Perceptron\_setThreshold(Perceptron \*perceptron, double threshold) {

// perceptron->threshold\_ = threshold;

//}

void \_Perceptron\_changeWeights(Perceptron \*perceptron, int actualResult, int desiredResult, const double inputs[]) {

unsigned i;

for (i = 0; i < perceptron->numInputs\_; i++) {

perceptron->weights\_[i] += perceptron->trainingRate\_ \* (desiredResult - actualResult) \* inputs[i];

}

perceptron->threshold\_ -= perceptron->trainingRate\_ \* (desiredResult - actualResult);

}

double \_Perceptron\_getRandomDouble() {

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

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

if (negativeRand < 0.5) {

randValue \*= -1.0;

}

return randValue;

}

#ifdef \_\_cplusplus

}

#endif

#include <stdio.h>

#include <iostream>

#include "perceptron.h"

int main() {

// Constants

const int TRAINING\_ITERATIONS = 16;

const int NUM\_OF\_INPUTS = 2;

const double TRAINING\_RATE = 0.2;

const double ZERO\_ZERO[] = { 0, 0 };

const double ZERO\_ONE[] = { 0, 1 };

const double ONE\_ZERO[] = { 1, 0 };

const double ONE\_ONE[] = { 1, 1 };

int i;

// Creating Perceptron instances

Perceptron \*pAND = Perceptron\_new(NUM\_OF\_INPUTS, TRAINING\_RATE);

// Printing the results of the randomly generated perceptrons (BEFORE TRAINING)

printf("Results for 'AND' perceptron, BEFORE training:\n");

printf("Input: (0,0). Result: %d\n", Perceptron\_getResult(pAND, ZERO\_ZERO));

printf("Input: (0,1). Result: %d\n", Perceptron\_getResult(pAND, ZERO\_ONE));

printf("Input: (1,0). Result: %d\n", Perceptron\_getResult(pAND, ONE\_ZERO));

printf("Input: (1,1). Result: %d\n", Perceptron\_getResult(pAND, ONE\_ONE));

// Training each of the perceptrons

for (i = 0; i < TRAINING\_ITERATIONS; i++) {

Perceptron\_train(pAND, ZERO\_ZERO, 0);

Perceptron\_train(pAND, ZERO\_ONE, 0);

Perceptron\_train(pAND, ONE\_ZERO, 0);

Perceptron\_train(pAND, ONE\_ONE, 1);

}

// Printing the results of the trained perceptrons (AFTER TRAINING)

printf("Results for 'AND' perceptron, AFTER training:\n");

printf("Input: (0,0). Result: %d\n", Perceptron\_getResult(pAND, ZERO\_ZERO));

printf("Input: (0,1). Result: %d\n", Perceptron\_getResult(pAND, ZERO\_ONE));

printf("Input: (1,0). Result: %d\n", Perceptron\_getResult(pAND, ONE\_ZERO));

printf("Input: (1,1). Result: %d\n", Perceptron\_getResult(pAND, ONE\_ONE));

system("pause");

return 0;

}