1.**Cel ćwiczenia:**

Celem ćwiczenia jest poznanie budowy i działania perceptronu poprzez implementację oraz uczenie perceptronu realizującego wybraną funkcję logiczną dwóch zmiennych.

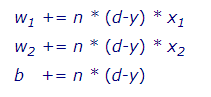
1. Syntetyczny opis budowy oraz wykorzystanego algorytmu uczenia

Perceptron w swojej budowie odwzorowuje budowę komórki nerwowej. Składa się z wejść (odwzorowania dendrytów), bloku sumacyjnego i aktywacji (odwzorowanie jądra komórki) i wyjścia (odwzorowanie aksonu).

Rysunek 1

Każde z wejść posiada swoją wagę. Blok sumacyjny odpowiada za zsumowanie iloczynów wartości wejść i wag tych wejść. Wartość wyliczona przez sumator jest przekazywana do bloku aktywacji, który wykonuje funkcję aktywacji (w przypadku perceptronu jest to funkcja progowa jak na Rysunku 1).

Perceptron może zawierać dodatkowe wejście zwane biasem o stałej wartości 1.

Mechanizm uczenia z nauczycielem polega na wprowadzaniu do neuronu wcześniej przygotowanych danych dla których znany jest właściwy wynik pracy neuronu i porównywaniu otrzymanych wyników z wynikiem prawidłowym. W razie nieprawidłowej pracy zmianie ulegają wagi poszczególnych wejść, zgodnie ze wzorem 1.

Równanie 1

W jest wagą wejścia, n - współczynnikiem uczenia, d – wartością otrzymaną na wyjściu, y – wartością oczekiwaną, x – wartością na wejściu.

**2.Podjęte kroki**

Stworzyłem Perception który działa jak bramka logiczna AND. Pobiera on dane z pliku test.txt, następnie iteruje po nich, za każdym razem aktualizując wagi. Ten proces powtarzany jest w pętli określoną ilość razy. Gdy już wagi są ustalone testuję poprawność wyuczonych wag licząc wynik dla każdej możliwości danych wejściowych ([0.0],[0.1],[1.0],[1.1])

Jako że funkcja AND ma maksymalną liczbę danych uczących równą 4, użyję następujących danych:

Zestaw 1: Zestaw 2:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| X1 | X2 | Y |  | X1 | X2 | Y |
| 0 | 0 | 0 |  | 0 | 0 | 0 |
| 1 | 0 | 0 |  |  |  |  |
| 0 | 1 | 0 |  | 0 | 1 | 0 |
| 1 | 1 | 1 |  |  |  |  |

**3.Analiza**

Wynik uczenia dla zestawu 1:

Wynik uczenia perceptronu dla danych z zestawu 2:

Gdy dla zestawu 1 zmienimy współczynnik uczenia na 0,1:

Gdy zmienimy go na 0,01

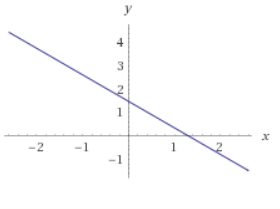
Oraz 0,001

Dla Learning rate określonego jako 0.1 i liczby iteracji 1000 równanie prostej przyjmuje wartość 0,78x+0,28y-0,93 =0 . W momencie zmniejszenia learning rate do 0.01 wagi unormowały się na poziomie ok. 0,5.

Zmniejszenie tudzież zwiększenie liczby maxymalnych iteracji nie wpływa na wynik, gdyż program kończy pracę w okolicach 6,7 iteracji, z powodu zniwelowania błędu.

Zwiększenie danych uczących wpływa na program jedynie w ten sposób, iż następuje więcej iteracji, aby dojść do tych samych rozwiązań, z powodu tego że dane się powtarzają.

Jak widać na poniższym rysunku, tylko dla zestawu danych 1,1 funkcja logiczna AND da 1 w wyniku, wiec równanie ma wiele możliwych postaci. W związku z tym dla tej funkcji, i takiego rozstrzału wartości danych wejściowych jest wiele możliwych wartości wag.



**4.Wnioski:**

Jak widać w analizie programu, dane uczące muszą mieć minimalny rozmiar który pokazuje wszystkie możliwe wyjścia. Gdy próbowałem użyć danych w których brakowało danych dla któ®ych perceptron zwraca 1, nigdy nie zniwelował swojego błędu. Gdy natomiast użyjemy większej ilości danych, i zaczną się one powtarzać, program znacząco spowolni podczas nauki, ale później będzie szybciej znajdywał odpowiedź.

Współczynnik uczenia liniowo przekłada się na szybsze uczenie. Gdy zmniejszyłem współczynnik uczenia program nigdy nie pozbył się błędu, z kolei gdy był on duży pozbywał się jego najszybciej.

**LISTING KODU PROGRAMU:**

#define \_CRT\_SECURE\_NO\_DEPRECATE

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#define LEARNING\_RATE 0.05

#define MAX\_ITERATION 100

float randomFloat()

{

return (float)rand() / (float)RAND\_MAX;

}

int calculateOutput(float weights[], float x, float y)

{

float sum = x \* weights[0] + y \* weights[1] + weights[2];

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

}

int main(int argc, char \*argv[])

{

srand(time(NULL));

float weights[3], localError, globalError;

int x[208], y[208], outputs[208], patternCount, i, p, iteration, output, proba[4];

FILE \*fp;

if ((fp = fopen("test.txt", "r")) == NULL) {

printf("Cannot open file.\n");

exit(1);

}

i = 0;

while (fscanf(fp, "%d %d %d", &x[i], &y[i], &outputs[i]) != EOF) {

if (outputs[i] == 0) {

outputs[i] = -1;

}

i++;

}

patternCount = i;

weights[0] = randomFloat();

weights[1] = randomFloat();

weights[2] = randomFloat();

iteration = 0;

do {

iteration++;

globalError = 0;

for (p = 0; p < patternCount; p++) {

output = calculateOutput(weights, x[p], y[p]);

localError = outputs[p] - output;

weights[0] += LEARNING\_RATE \* localError \* x[p];

weights[1] += LEARNING\_RATE \* localError \* y[p];

weights[2] += LEARNING\_RATE \* localError;

globalError += (localError\*localError);

}

/\* Root Mean Squared Error \*/

printf("Iteration %d : RMSE = %.4f\n", iteration,

sqrt(globalError / patternCount));

} while (globalError != 0 && iteration <= MAX\_ITERATION);

printf("asd %d", patternCount);

printf("\nDecision boundary (line) equation: %.2f\*x + %.2f\*y + %.2f = 0\n",

weights[0], weights[1], weights[2]);

proba[0] = calculateOutput(weights, 0, 0);

printf(" %d \n", proba[0]);

proba[1] = calculateOutput(weights, 0, 1);

printf(" %d \n", proba[1]);

proba[2] = calculateOutput(weights, 1, 0);

printf(" %d \n", proba[2]);

proba[3] = calculateOutput(weights, 1, 1);

printf(" %d \n", proba[3]);

system("pause");

return 0;

}