Ejercicio 2.3

IBERO
Ciudad de México • Tijuana

Neuronas (Hebb’s rule)

Allan Jair Escamilla Hernández

Profesor

César Ángeles

Materia

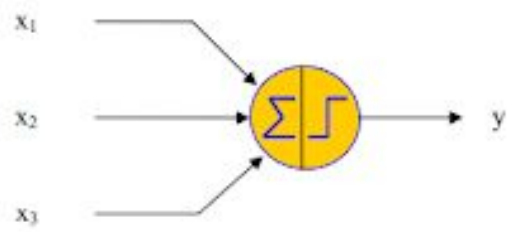
Taller de Desarrollo de Aplicaciones

Introducción:

Una de las características más significativas de las redes neuronales es su capacidad para aprender a partir de alguna fuente de información interactuando con su entorno. En 1958 el psicólogo Frank Ronsenblant desarrolló un modelo simple de neurona basado en el modelo de McCulloch y Pitts y en una regla de aprendizaje basada en la corrección del error. A este modelo le llamó Perceptrón. Una de las características que más interés despertó de este modelo fue su capacidad de aprender a reconocer patrones. El Perceptrón está constituido por conjunto de sensores de entrada que reciben los patrones de entrada a reconocer o clasificar y una neurona de salida que se ocupa de clasificar a los patrones de entrada en dos clases, según que la salida de la misma se 1 (activada) o 0 (desactivada). Sin embargo, dicho modelo tenía muchas limitaciones, como por ejemplo, no es capaz de aprender la función lógica XOR. Tuvieron que pasar unos años hasta que se propusiera la regla de aprendizaje de retropropagación del error para demostrarse que el Perceptrón multicapa es un aproximador universal.

Arquitectura de la neurona

$$f(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \text{si} \quad w_1 x_1 + w_2 x_2 + \ldots + w_n x_n \\ 0 & \text{si} \quad w_1 x_1 + w_2 x_2 + \ldots + w_n x_n \end{cases}$$



Aprendizaje de la neurona

Para la determinación de los pesos sinápticos y del umbral vamos a seguir un proceso adaptativo que consiste en comenzar con unos valores iniciales aleatorios e ir modificándolos iterativamente cuando la salida de la unidad no coincide con la salida deseada. La regla que vamos a seguir para modificar los pesos sinápticos se conoce con el nombre de regla de aprendizaje del Perceptrón simple y viene dada por la expresión:

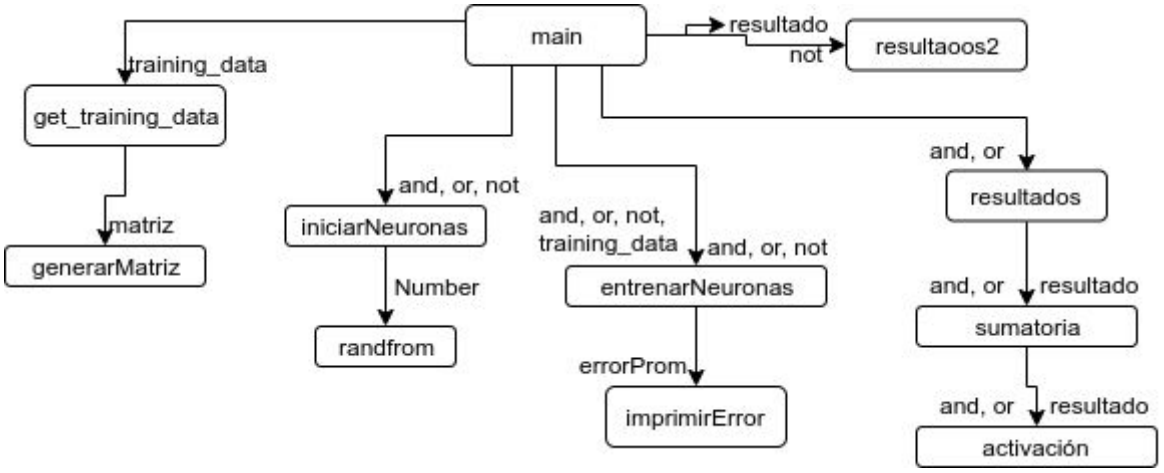
$$\Delta w_j(k) = \eta(k)[z(k) - y(k)]x_j(k)$$

Objetivos:

- Comprender el funcionamiento de una neurona y el proceso de aprendizaje.
- Implementar un algoritmo que entrene y pruebe el funcionamiento de neuronas que sean capaces de resolver los problemas que presentan las compuertas AND, OR y NOT.

Análisis

Diagrama IPO



Código

```
/*
* @author:  Allan Jair Escamilla Hernández
* @date:    18/octubre/2019
* @file:    neuronas.c
* @brief: Neuronas con valores predeterminados que resuelve el problema
de las compuertas and, or y not.
*/

// INCLUYENDO BIBLIOTECAS
#include<stdio.h>
#include<stdlib.h>
```

```c
#include<time.h>
#include<string.h>
// DEFINIENDO LA ESTRUCTURA QUE TENDRA UNA NEURONA
typedef struct Neuronas{
   double* w;
   double bias;
}Neurona;

// PROTOTIPOS DE LAS FUNCIONES
double sumatoria(double x[], Neurona n);
int activacion(double sum);
void iniciarNeuronas(Neurona* and, Neurona* or, Neurona* not);
double randfrom(double min, double max);
void entrenarNeurona(Neurona* neuron, double** training_data, char
gate[]);
void entrenarNeuronaNot(Neurona *neuron, double** training_data, char
gate[]);
double** get_training_data(char* gate, int numberOfPoints);
double **generarMatriz(int height, int weight);
void resultados(double x[], Neurona gate);
void resultados2(double x[], Neurona gate);
void imprimirError(double promError, int i, char gate[]);
void liberarMemoria(double **Matriz, int Ancho, int Alto);

// FUNCION PRINCIPAL
int main(){
   Neurona and, or, not; // Declarando las neuronas
   int op;
     double *x = (double*)malloc(sizeof(double)*2); // Asigno memoria
dinamica para las entradas
   // Genero apuntadores a funciones
   void (*funciones[2])(double [], Neurona);
   double** (*data)(char*, int) = get_training_data;
   void (*train[2])(Neurona*, double**, char[]);
   double **datasetNot = data("NOT", 50);
   double **datasetAND = data("AND", 50);
   double **datasetOR = data("OR", 50);
   funciones[0] = resultados;
   funciones[1] = resultados2;
   train[0] = entrenarNeurona;
```

```
    train[1] = entrenarNeuronaNot;
    // ASIGNANDO MEMORIA A LOS ARREGLOS QUE CONTENDRAN LOS PESOS DE LAS ENTRADAS
    and.w = (double*)malloc(sizeof(double)*2);
    or.w = (double*)malloc(sizeof(double)*2);
    not.w = (double*)malloc(sizeof(double));
     iniciarNeuronas(&and, &or, &not); // Inicializando los valores de las neuronas
    train[1](&not, datasetNot, "not");
    train[0](&and, datasetAND, "and");
    train[0](&or, datasetOR, "or");
    // MOSTRANDO MENU
    do{
        system("clear");
        printf("1.-and\n2.-or\n3.-not\n4.-salir\nElegir una opcion: ");
        scanf("%d", &op);
        if(op >= 1 && op <= 2){
            // Se llaman las funciones haciendo uso de los apuntadores
            if(op == 1)
                funciones[0](x, and);
            else
                funciones[0](x, or);
        }
        if(op == 3)
            funciones[1](x, not);
        if(op == 4)
            printf("Saliendo...\n");
        printf("Presionar enter para continuar...\n");
        __fpurge(stdin);
        getchar();
    } while (op != 4);
    // LIBERANDO MEMORIA ASIGNADA EN TIEMPO DE EJECUCION
    free(x);
    free(and.w);
    free(or.w);
    free(not.w);
    liberarMemoria(datasetNot, 50, 2);
    liberarMemoria(datasetAND, 50, 3);
    liberarMemoria(datasetOR, 50, 3);
}
```

```
// DESARROLLANDO LAS FUNCIONES

/* * Funcion que hace la sumatoria de los pesos y las entradas de la neurona
   * @param double x[] recibe un arreglo de las entradas dadas por el usuario.
   * @param Neurona n recibe la neurona que contiene los pesos y el valor del sesgo para realizar la sumatoria.
*/
double sumatoria(double x[], Neurona n){
   double sum = 0;
   for(int i = 0; i < 2; i++)
       sum+= x[i]*n.w[i];
   return sum + n.bias;
}

/* * Funcion que hace la sumatoria de los pesos y las entradas de la neurona para devolverle un resultado al usuario.
   * @param double x[] recibe un arreglo de las entradas dadas por el usuario.
   * @param Neurona n recibe la neurona que contiene los pesos y el valor del sesgo para realizar la sumatoria.
*/
void resultados(double x[], Neurona gate){
   for (int i = 0; i < 2; i++){
       printf("Ingresar entrada numero %d-> ", i);
       scanf("%le", &(x[i]));
   }
   printf("El resultado es %d\n", activacion(sumatoria(x, gate)));
}

/* * Funcion que hace la sumatoria de los pesos y las entradas de la neurona para devolverle un resultado al usuario (Neurona NOT).
   * @param double x[] recibe un arreglo de las entradas dadas por el usuario.
   * @param Neurona n recibe la neurona que contiene los pesos y el valor del sesgo para realizar la sumatoria.
*/
void resultados2(double x[], Neurona gate){
```

```
   printf("Ingresar entrada: ");
   scanf("%le", &(x[0]));
   x[1] = 0;
      printf("El resultado es %d\n", activacion(gate.w[0] * x[0] +
gate.bias));
}

/* * Funcion que inicializa los pesos y sesgos de las neuronas.
  * @param Neurona* and recibe una neurona para asignar pesos.
  * @param Neurona* or recibe una neurona para asignar pesos.
  * @param Neurona* not recibe una neurona para asignar pesos.
*/
void iniciarNeuronas(Neurona* and, Neurona* or, Neurona* not){
   and->w[0] = randfrom(0, 1);
   and->w[1] = randfrom(0, 1);
   and->bias = randfrom(0, 1);
   or->w[0] = randfrom(0, 1);
   or->w[1] = randfrom(0, 1);
   or->bias = randfrom(0, 1);
   not->w[0] = randfrom(0, 1);
   not->w[1] = 0;
   not->bias = randfrom(0, 1);
}

/* * Funcion de activacion de las neuronas.
    * @param double sum recibe el resultado de la sumatoria para
determinar la activacion o no de la neurona.
*/
int activacion(double sum){
   return sum >= 0;
}

/* * Funcion que devuelve un valor aleatorio entre dos numeros dados.
  * @param double min recibe el valor minimo.
  * @param double max recibe el valor maximo.
*/
double randfrom(double min, double max) {
   double range = (max - min);
   double div = RAND_MAX / range;
   return min + (rand() / div);
```

```
}

/* * Funcion que obtiene los valores de los pesos para que la neurona funcione adecuadamente.
   * @param Neurona* n recibe la neurona que contiene los pesos y el valor del sesgo para realizar la sumatoria.
   * @param double** training_data recibe una matriz de datos de prueba para entrenar a la neurona.
    * @param char gate[] recibe la compuerta logica que se esta entrenando.
*/
void entrenarNeurona(Neurona* neuron, double** training_data, char gate[]){
   char directorio[100];
   strcpy(directorio, gate);
   strcat(directorio, "/ecuacionPlano.dat");
   FILE* Arch = fopen(directorio, "wt");
   int epochs = 1000;
   int counter = 0;
   double salida, error, promError = 0.0;
   double sum = 0, lr = 0.2;
   for(int i = 0; i < epochs; i++){
       if(counter == 50)
           counter = 0;
                           error    =    training_data[counter][2]    -
sumatoria(training_data[counter], *neuron);
       promError+= error;
        if(i % 100 == 0){ // Cada 100 epocas se obtiene el promedio de errores.
           imprimirError(promError, i, gate);
           promError = 0.0;
       }
       for(int k = 0; k < 2; k++){
           neuron->w[k] += lr * error * training_data[counter][k];
       }
       neuron->bias += error * lr;
       counter++;
   }
```

```
    fprintf(Arch, "%f*x + %f * y + %f\n", neuron->w[0], neuron->w[1],
neuron->bias); // Se imprimie la ecuacion del plano que separa los
valores
   fclose(Arch);
}

/* * Funcion que obtiene los valores de los pesos para que la neurona
funcione adecuadamente (neurona NOT).
   * @param Neurona* n recibe la neurona que contiene los pesos y el
valor del sesgo para realizar la sumatoria.
  * @param double** training_data recibe una matriz de datos de prueba
para entrenar a la neurona.
    * @param char gate[] recibe la compuerta logica que se esta
entrenando.
*/
void entrenarNeuronaNot(Neurona *neuron, double** training_data, char
gate[]){
   char directorio[100];
   strcpy(directorio, gate);
   strcat(directorio, "/ecuacionPlano.dat");
   FILE *Arch = fopen(directorio, "wt");
   int epochs = 1000;
   int counter = 0;
   double salida, error, promError = 0.0;
   double sum = 0, lr = 0.01;
   for (int i = 0; i < epochs; i++){
       if (counter == 50)
           counter = 0;
               error = training_data[1][counter] - (neuron->w[0] *
training_data[0][counter] + neuron->bias);
       promError+= error;
        if (i % 100 == 0){ // Cada 100 epocas se obtiene el promedio de
error
           imprimirError(promError, i, gate);
           promError = 0.0;
       }
       neuron->w[0] += lr * error * training_data[0][counter];
       neuron->bias += error * lr;
       counter++;
   }
```

```
    fprintf(Arch, "%f*x + %f\n", neuron->w[0], neuron->bias);
    fclose(Arch);
}

/* * Funcion que devuelve una matriz de datos de prueba para entrenar las neuronas.
    * @param char gate[] recibe la compuerta logica que se esta entrenando.
    * int numberOfPoints recibe la cantidad de puntos que se van a generar.
*/
double **get_training_data(char* gate, int numberOfPoints){
    double dato, dato2, resultado;
    double **matriz;
    FILE* file, *file2;
    if(!strcmp("NOT", gate)){
        matriz = generarMatriz(numberOfPoints, 2);
        file = fopen("not/verdadero.dat", "wt");
        file2 = fopen("not/falso.dat", "wt");
        for(int i = 0; i < numberOfPoints; i++){
            dato = randfrom(-1, 1);
            if(dato <= 0){
                resultado = 1;
                fprintf(file, "%f, %f\n", dato, resultado);
            }else{
                resultado = -1;
                fprintf(file2, "%f, %f\n", dato, resultado);
            }
            matriz[0][i] = dato;
            matriz[1][i] = resultado;
        }
        fclose(file);
        fclose(file2);
    }
    if(!strcmp("AND", gate)){
        matriz = generarMatriz(3, numberOfPoints);
        file = fopen("and/verdadero.dat", "wt");
        file2 = fopen("and/falso.dat", "wt");
        for(int i = 0; i < numberOfPoints; i++){
            dato = randfrom(-1, 1);
```

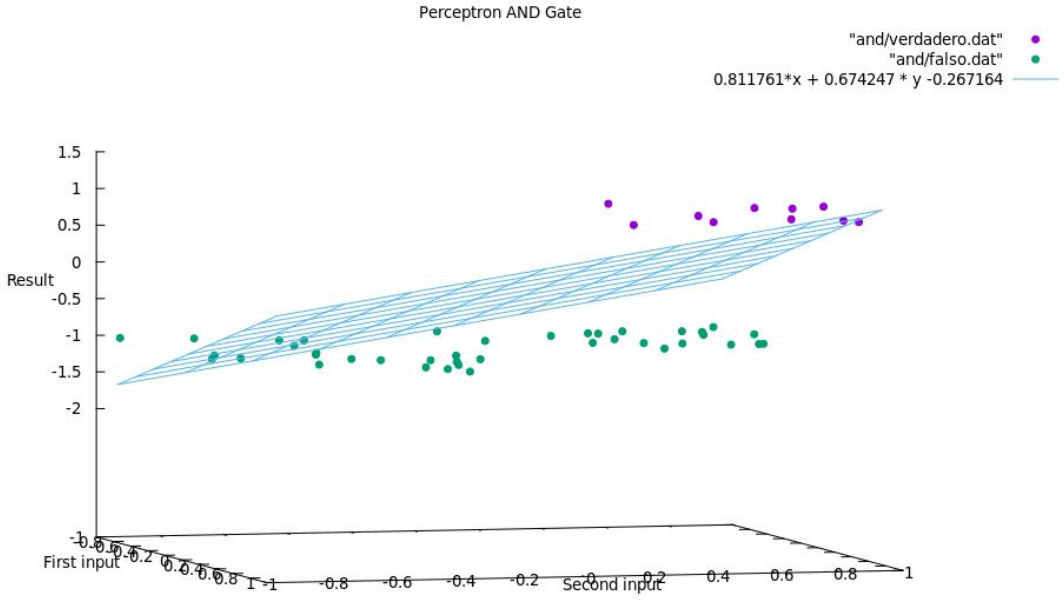
```
            dato2 = randfrom(-1, 1);
            if(dato > 0 && dato2 > 0){
                resultado = 1;
                fprintf(file, "%f, %f, %f\n", dato, dato2, resultado);
            }else{
                resultado = -1;
                fprintf(file2, "%f, %f, %f\n", dato, dato2, resultado);
            }
            matriz[i][0] = dato;
            matriz[i][1] = dato2;
            matriz[i][2] = resultado;
        }
        fclose(file);
        fclose(file2);
    }
    if(!strcmp("OR", gate)){
        matriz = generarMatriz(3, numberOfPoints);
        file = fopen("or/falso.dat", "wt");
        file2 = fopen("or/verdadero.dat", "wt");
        for(int i = 0; i < numberOfPoints; i++){
            dato = randfrom(-1, 1);
            dato2 = randfrom(-1, 1);
            if(dato <= 0 && dato2 <= 0){
                resultado = -1;
                fprintf(file, "%f, %f, %f\n", dato, dato2, resultado);
            }else{
                resultado = 1;
                fprintf(file2, "%f, %f, %f\n", dato, dato2, resultado);
            }
            matriz[i][0] = dato;
            matriz[i][1] = dato2;
            matriz[i][2] = resultado;
        }
        fclose(file);
        fclose(file2);
    }
    return matriz;
}
/* * Funcion que genera una matriz y devuelve su referencia en memoria.
  * @param int height recibe el alto de la matriz.
```

```
  * @param int weight recibe el ancho de la matriz.
*/
double** generarMatriz(int height, int weight){
   double** matriz = (double**)malloc(sizeof(double*) * weight);
   for(int i = 0; i < weight; i++)
       matriz[i] = (double*)malloc(sizeof(double) * height);
   return matriz;
}

/* * Funcion que imprime en un archivo el error de las neuronas.
  * @param double promError recibe la suma de los errores.
  * @param int i recibe la iteracion en la que se encuentra el proceso
de entreanamiento de la neurona,
   * @param char gate[] recibe el nombre de la compuerta que se esta
entrenando.
*/
void imprimirError(double promError, int i, char gate[]){
   char str[3];
   if(i == 100)
       strcpy(str, "wt");
   else
       strcpy(str, "at");
   char newString[100];
   strcpy(newString, gate);
   strcat(newString, "/errores.dat");
   FILE* Arch = fopen(newString, str);
   fprintf(Arch, "%d, %f\n", i, promError / 100);
   fclose(Arch);
}

/* * Funcion que libera memoria de las matrices generadas.
  * @param double** Matriz recibe la referencia en memoria de la matriz
a liberar.
  * @param int Ancho recibe el ancho de la matriz.
  * @param int Alto recibe el alto de la matriz.
*/
void liberarMemoria(double **Matriz, int Ancho, int Alto){
   for (int i = 0; i < Alto; i++)
       free(Matriz[i]); // Filas
   free(Matriz);        // Columnas
```

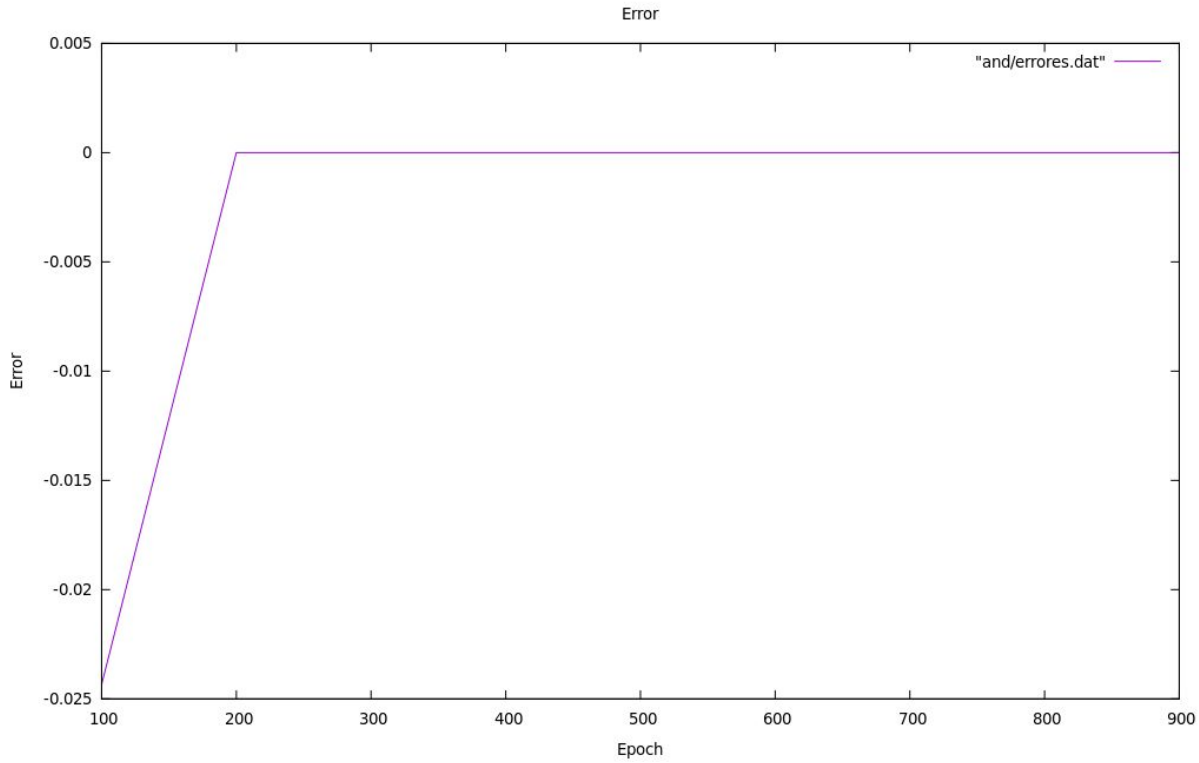
```
}
```

Gráficas de los resultados

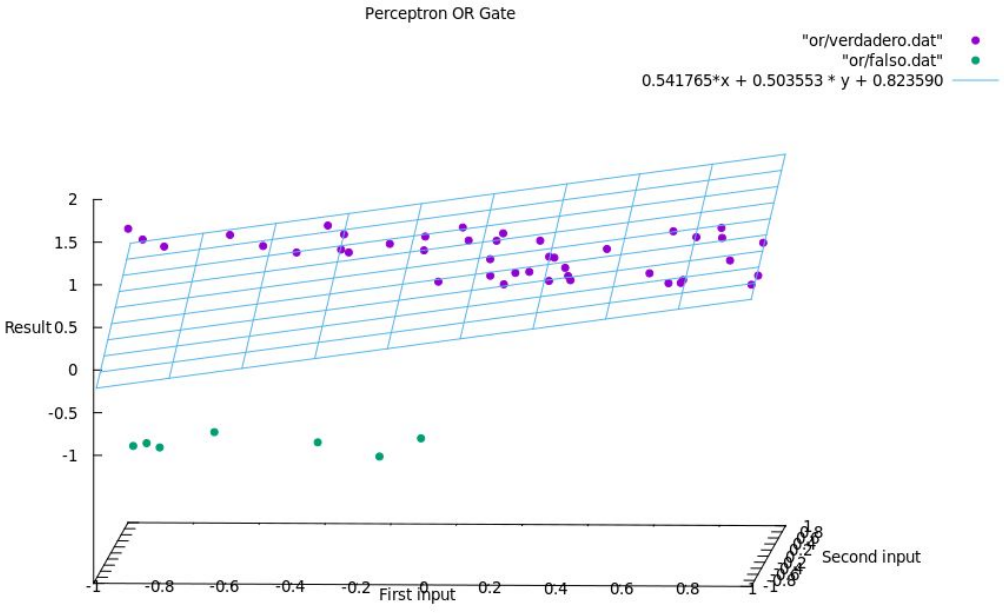
Perceptrón AND



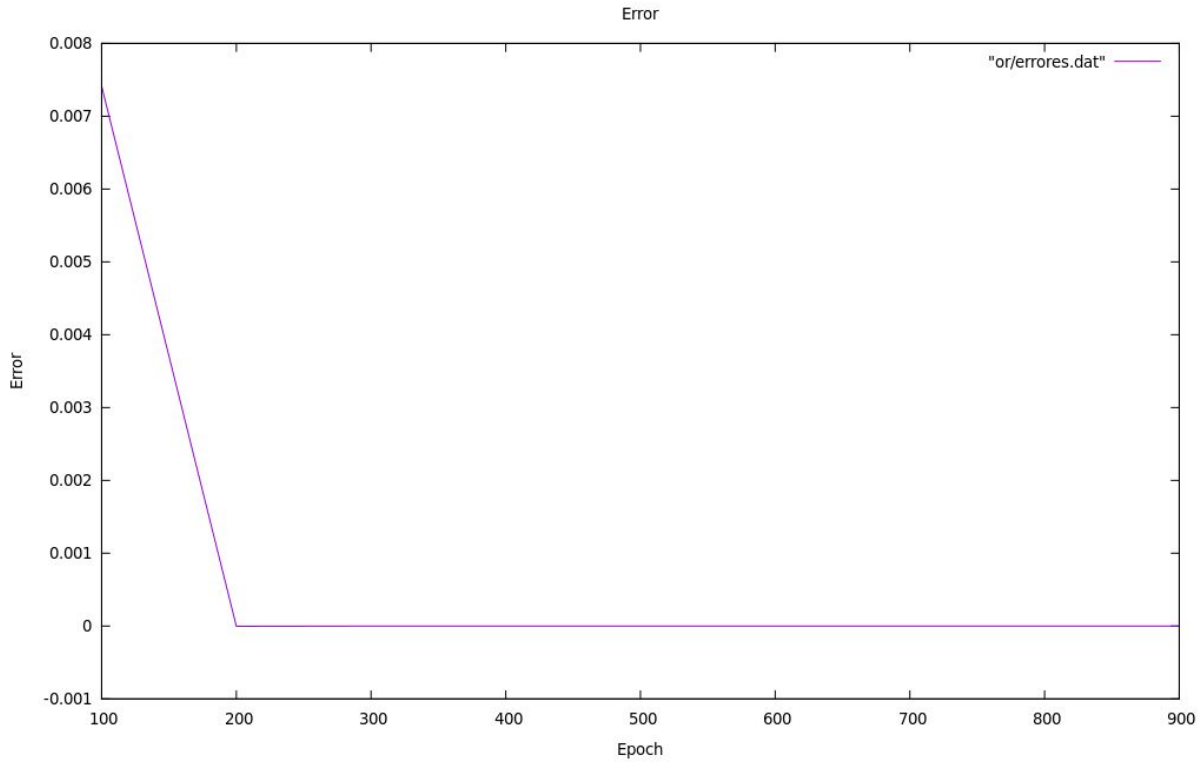
Error en el cálculo de los pesos del perceptrón AND



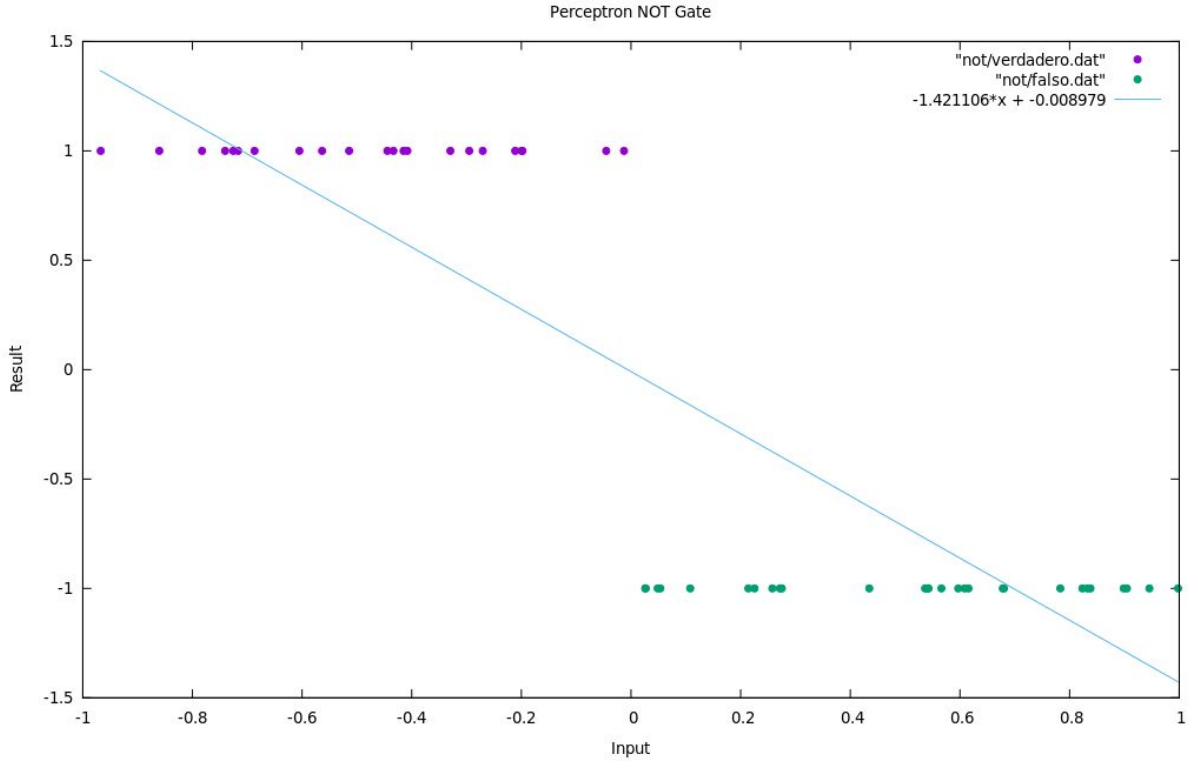
Perceptrón OR



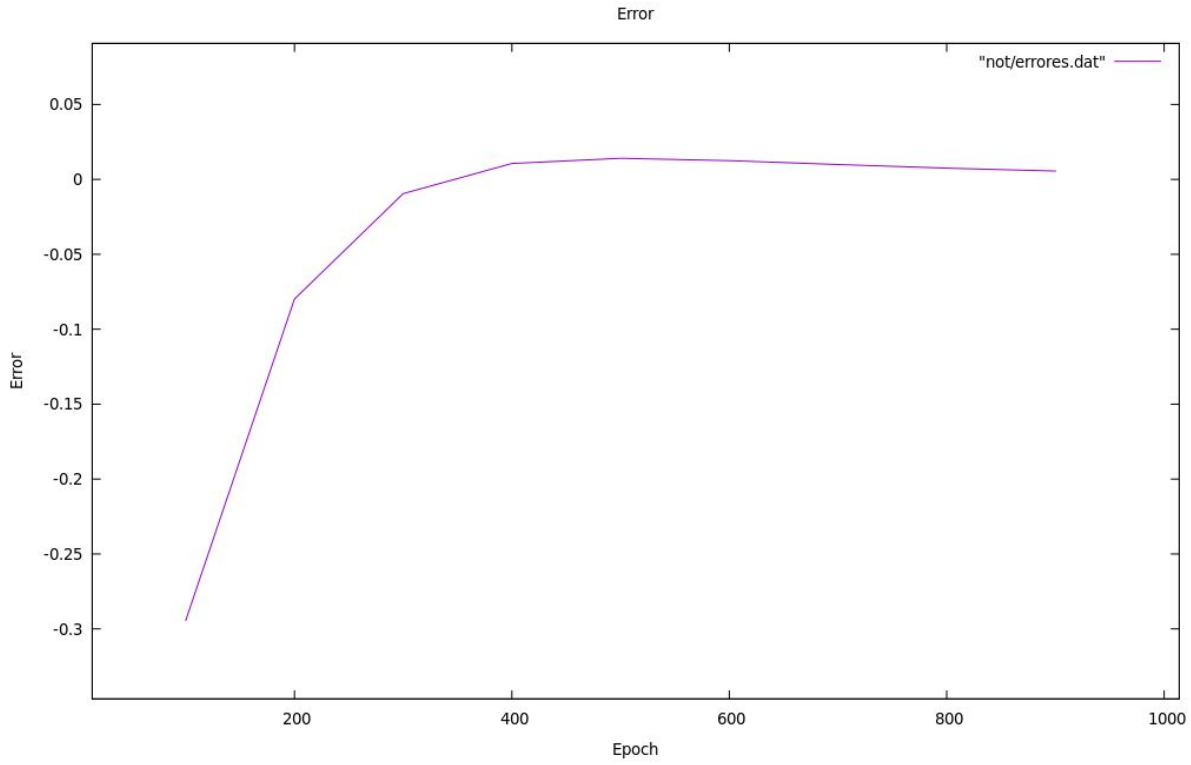
Error en el cálculo de los pesos del perceptrón OR



Perceptrón NOT



Error en el cálculo de los pesos del perceptrón NOT



Conclusión:

A lo largo del desarrollo de esta práctica he podido comprender el funcionamiento interno que tiene una neurona artificial, así como también el proceso de aprendizaje a través de la “Regla de Hebb”. Además, esto me ha servido para reforzar mis habilidades en el diseño e implementación de algoritmos en el lenguaje de programación de C.

Referencias:

Aguilar, R. "Red Neuronal de Topología Flexible". En: VI Congreso Nacional de Ciencias de la Computación, La Paz, Bolivia. Septiembre de 1999.