

**Experiencia educativa:**

TOPICOS AVANZADOS DE INEL I

**Académica:**

Cuellar Hernández Leticia

**Alumnos:**

*Abad Dolores Lázaro*

*Hernández Reyes Roberto Saúl*

*Sánchez López Luis Uriel*

*Velásquez Reyes Román Gabriel*

## Contenido

<b>Introducción:</b> .....	3
<b>Objetivo General:</b> .....	4
<b>Objetivos Específicos:</b> .....	4
<b>Marco teórico:</b> .....	4
<b>Función Tangente Hiperbólica</b> .....	6
<b>Forward Propagation</b> .....	7
<b>Backpropagation (cómputo del gradiente)</b> .....	8
<b>Fase 1: Propagar</b> .....	8
<b>Fase 2: Actualizar Pesos:</b> .....	8
<b>Componentes</b> .....	10
<b>Coche</b> .....	11
<b>Descripción de la practica</b> .....	12
<b>Resultados:</b> .....	21
<b>Conclusión:</b> .....	21

## **Introducción:**

Durante este curso hemos aprendido e implementado sobre el estudio de diversos aprendizajes de Inteligencia Artificial, para la resolución de problemas utilizamos algoritmos de agrupación (de aprendizaje no supervisado y método de puntos) hasta algoritmos de redes neuronales, los cuales nos enfocaremos para este documento, ya que estos se suelen utilizar para problemas de clasificación y con el potencial de resolver los problemas por medio de patrones. Las redes Neuronales Artificiales requieren almacenamiento y un exceso procesamiento, además que en su actualidad son las más usadas. Para finalizar este semestre hemos implementado los conocimientos del curso e implementarlos en un coche evita obstáculos, que funcionara con una neurona que implementamos y explicamos su funcionamiento, ya que su desarrollo es interesante y la base para cualquier dispositivo que utilice inteligencia artificial.

## Objetivo General:

Preparar una red neuronal e implementarla en un carrito evasor de obstáculos.

## Objetivos Específicos:

- Emplear un carrito evasor de obstáculos controlado por un microcontrolador, programado con redes neuronales y compuesto por motores y sensores.
- Implementar Redes neuronales que le permiten enseñar al carro por donde moverse.
- Utilizar distintos tipos de sensores ultrasónicos para poder detectar obstáculos.

## Definiciones:

*Python:* es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Y define este como un lenguaje multiparadigma, debido a que soporta orientación a objetos, programación imperativa y en menor medida programación funcional.

*Compilador Jupyter Notebook:* es una aplicación web que sirve a modo de puente constante entre el código y los textos explicativos. De este modo, los usuarios pueden crear y compartir en tiempo real código, ecuaciones, visualizaciones, etc.

*Función Sigmoide:* La función logística, curva logística o curva en forma de S es una función matemática que aparece en diversos modelos de crecimiento de poblaciones, propagación de enfermedades epidémicas y difusión en redes sociales. Dicha función constituye un refinamiento del modelo exponencial para el crecimiento de una magnitud.

## Marco teórico:

Como funciona la neurona:

Para explicar cómo funciona la red neuronal, primero se explicará brevemente la arquitectura de la red neuronal.

Nuestros datos de entrada serán:

### ➤ **Sensor de distancia** al obstáculo

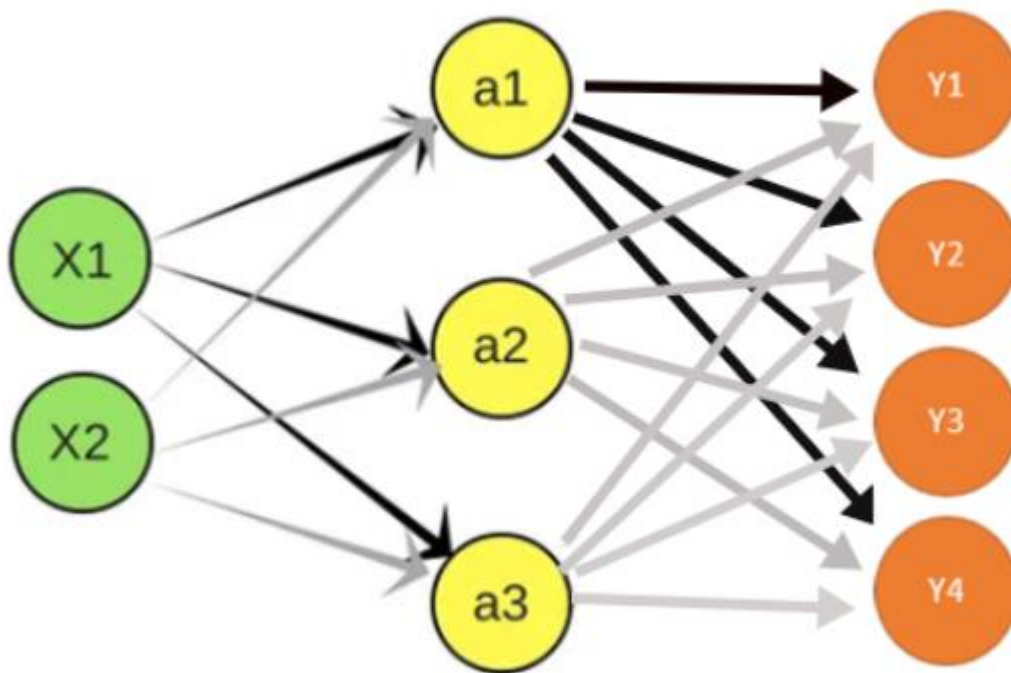
- Si es 0 no hay obstáculos a la vista
- Si es 0,5 se acerca a un obstáculo
- Si es 1 está demasiado cerca de un obstáculo

Entrada: Sensor Distancia	Entrada: Posición Obstáculo	Salida: Motor 1	Salida: Motor 2	Salida: Motor 3	Salida: Motor 4
-1	0	1	0	0	1
-1	1	1	0	0	1
-1	-1	1	0	0	1
0	-1	1	0	1	0
0	1	0	1	0	1
0	0	1	0	0	1
1	1	0	1	1	0
1	-1	0	1	1	0
1	0	0	1	1	0

Siendo el valor de los motores 0 y 1.

Acción	Motor 1	Motor 2	Motor 3	Motor 4
Avanzar	1	0	0	1
Retroceder	0	1	1	0
Giro Derecha	0	1	0	1
Giro Izquierda	1	0	1	0

La arquitectura de la red neuronal propuesta es la siguiente:

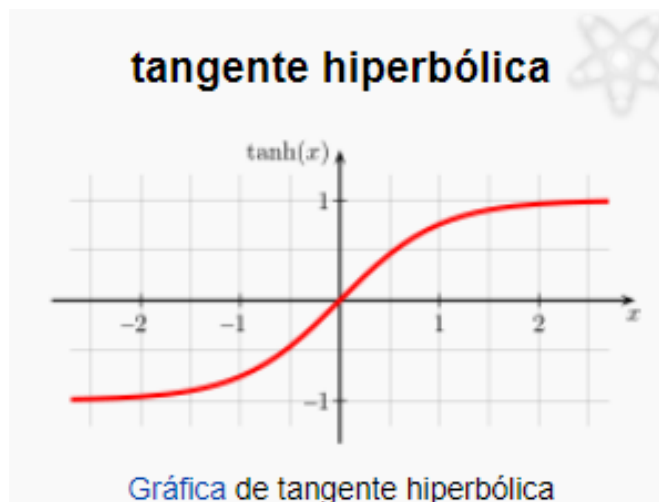


En la imagen anterior se ilustra la arquitectura de la red neuronal misma que esta compuestas de 3 capas, teniendo dos neuronas de entrada, tres capas ocultas y cuatro salidas, las salidas representan el encendido o apagado de los motores.

1.  $X(i)$  son las entradas
2.  $a(i)$  activación en la capa 2
3.  $y(i)$  son las salidas

## Función Tangente Hiperbólica

La tangente hiperbólica de un número real “ $x$ ” se designa mediante ***tanh x*** y se define como el cociente entre el seno hiperbólico y el coseno hiperbólico del número real  $X$ . Considerando que los límites de esta función van de  $-1$  a  $1$  nos resulta muy útil utilizar esta función para la solución a nuestra problemática pues los valores de entrada dispuestos corresponden a el mismo rango de aplicación que va de  $-1$  a  $1$ .



## Forward Propagation

Con **Feedforward** nos referimos al recorrido de “izquierda a derecha” que hace el algoritmo de la red, para calcular el valor de activación de las neuronas desde las entradas hasta obtener los valores de salida.

Si usamos notación matricial, las ecuaciones para obtener las salidas de la red serán:

$$X = [x_0 \ x_1 \ x_2]$$

$$z^{\text{layer2}} = O_1 X$$

$$a^{\text{layer2}} = g(z^{\text{layer2}})$$

$$z^{\text{layer3}} = O_2 a^{\text{layer2}}$$

$$y = g(z^{\text{layer3}})$$

Resumiendo: tenemos una red; tenemos 2 entradas, éstas se multiplican por los pesos de las conexiones y cada neurona en la capa oculta suma esos productos y les aplica la función de activación para “emitir” un resultado a la siguiente conexión (concepto conocido *en biología como sinapsis química*).

Como bien sabemos, los pesos iniciales se asignan con valores entre -1 y 1 de manera aleatoria. El desafío de este algoritmo, será que *las neuronas aprendan por sí mismas a ajustar el valor de los pesos para obtener las salidas correctas*.

## **Backpropagation (cálculo del gradiente)**

Al hacer backpropagation es donde **el algoritmo itera para aprender**. Esta vez iremos de “derecha a izquierda” en la red para mejorar la precisión de las predicciones. El algoritmo de backpropagation se divide en dos Fases: **Propagar y Actualizar Pesos**.

### **Fase 1: Propagar**

Esta fase implica 2 pasos:

1.1 Hacer forward propagation de un patrón de entrenamiento (recordemos que es este es un algoritmo supervisado, y conocemos las salidas) para generar las activaciones de salida de la red.

1.2 Hacer backward propagation de las salidas (activación obtenida) por la red neuronal usando las salidas “y” reales para generar los Deltas (error) de todas las neuronas de salida y de las neuronas de la capa oculta.

### **Fase 2: Actualizar Pesos:**

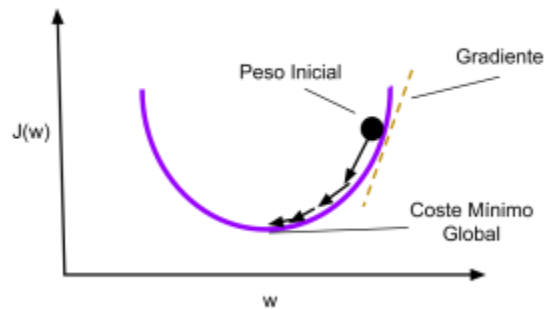
Para cada <<sinapsis>> de los pesos:

2.1 Multiplicar su delta de salida por su activación de entrada para obtener el gradiente del peso.

2.2 Substraer un porcentaje del gradiente de ese peso

*El porcentaje que utilizaremos en el paso 2.2 tiene gran influencia en la velocidad y calidad del aprendizaje del algoritmo y es llamado “learning rate” ó tasa de aprendizaje. Si es una tasa muy grande, el algoritmo aprende más rápido pero tendremos mayor imprecisión en el resultado. Si es demasiado pequeño, tardará mucho y podría no finalizar nunca.*





En esta gráfica vemos cómo utilizamos el gradiente paso a paso para descender y minimizar el coste total. Cada paso utilizará la Tasa de Aprendizaje **learning rate** que afectará la velocidad y calidad de la red.

Deberemos repetir las fases 1 y 2 hasta que la performance de la red neuronal sea satisfactoria.

Si denotamos al error en el layer “l” como **d(l)** para nuestras neuronas de salida en layer 3 la activación menos el valor actual será (usamos la forma vectorial):

$$d(3) = a^{\text{layer3}} - y$$

$$d(2) = O^T_2 d(3) \cdot g'(z^{\text{layer2}})$$

$$g'(z^{\text{layer2}}) = a^{\text{layer2}} \cdot (1 - a^{\text{layer2}})$$






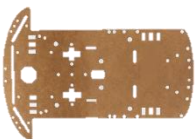

Nótese que no tendremos delta para la capa 1, puesto que son los valores X de entrada y no tienen error asociado.

El valor del error que es lo que queremos minimizar de nuestra red será

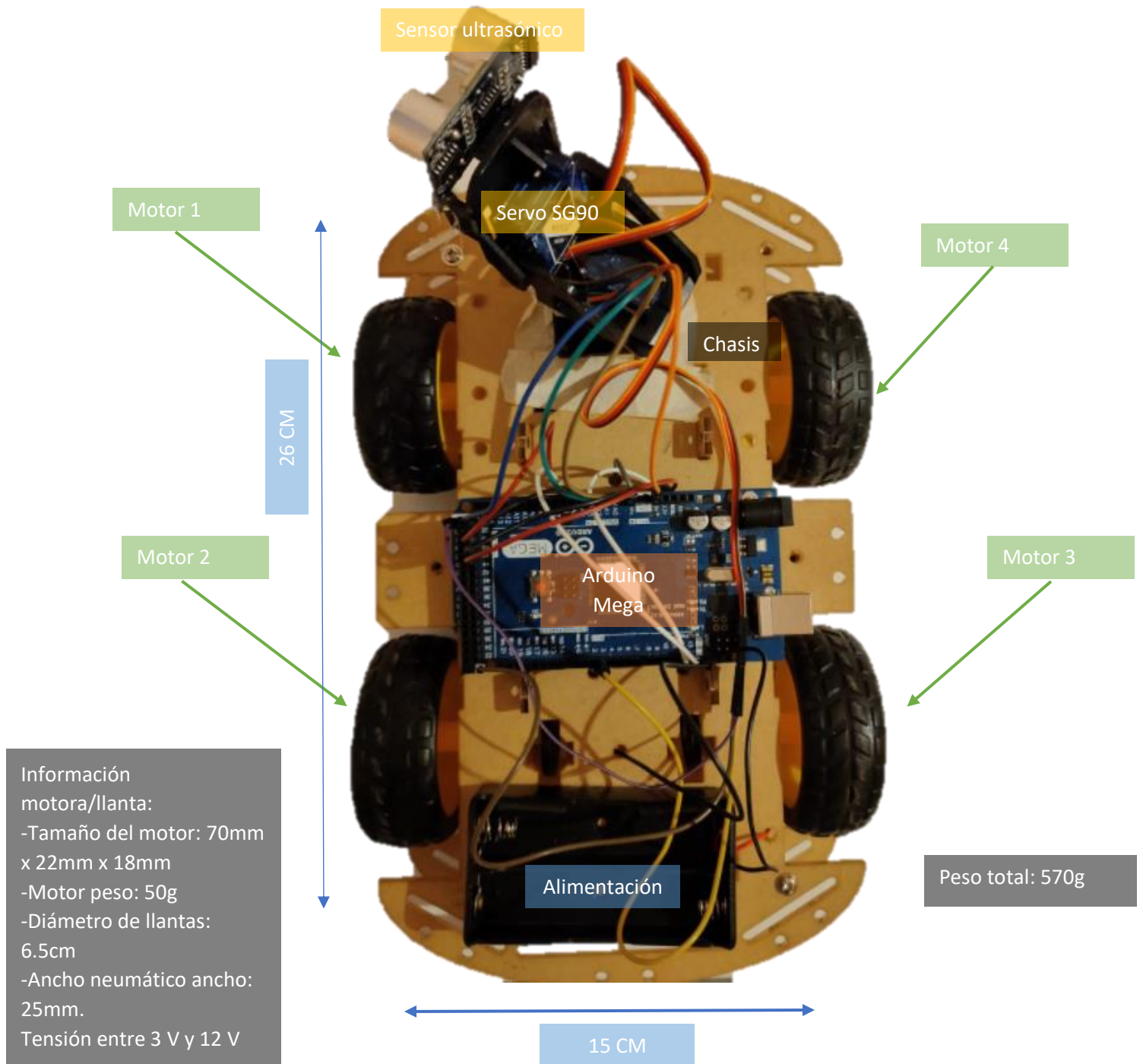
$$J = a^{\text{layer}} d^{\text{layer} + 1}$$

Usamos este valor y lo multiplicamos al learning rate antes de ajustar los pesos. Esto nos asegura que buscamos el gradiente, iteración a iteración “apuntando” hacia el mínimo global.

## Componentes

Componente	Descripción	Imagen
Placa Arduino Mega	Placa de desarrollo basada en el microcontrolador ATmega2560. Tiene 54 entradas/salidas digitales (de las cuales 15 pueden ser usadas como salidas PWM) un cristal de 16Mhz, conexión USB, jack para alimentación DC, conector ICSP, y un botón de reinicio.	
Controlador de motor L298N	Módulo controlador de motores L298N H-bridge nos permite controlar la velocidad y la dirección de dos motores de corriente continua o un motor paso a paso de una forma muy sencilla, gracias a los 2 los dos H-bridge que monta. Formado por 4 transistores que nos permite invertir el sentido de la corriente, y de esta forma podemos invertir el sentido de giro del motor.	
4 motores DC acompañada de una ruda	Conjunto sencillo y práctico para iniciarte con pequeños motores de corriente continua y ruedas a su medida.	
Servo Motor SG90	Es un servo miniatura de reducidas dimensiones, bajo consumo y muy económico. Tienen un rango de 180°	
Sensor Ultrasónico	Detectores de proximidad que trabajan libres de roces mecánicos y que detectan objetos a distancias que van desde pocos centímetros hasta varios metros. El sensor emite un sonido y mide el tiempo que la señal tarda en regresar.	
Chasis para el coche	Esta es la base de nuestro coche	
Soporte para sensor Ultrasónico	Ensamblado que va en conjunto con el servo motor y sirve como soporte para el sensor ultrasónico, el material que lo compone es plastico.	

## Coche



## Descripción de la practica

### 1. Obtención de los pesos

Una vez identificada el tipo de función a utilizar se crea la estructura de la red a utilizar que como se mencionó con anterioridad constará de 2 neuronas de entrada, 3 ocultas y 2 de salida. Debemos ir ajustando los parámetros de entrenamiento learning rate y la cantidad de iteraciones “epochs” para obtener buenas predicciones.

```
In [4]: # Red Coche para Evitar obstáculos
nn = NeuralNetwork([2,3,4],activation='tanh')
X = np.array([[-1, 0], # sin obstaculos
              [-1, 1], # sin obstaculos
              [-1, -1], # sin obstaculos
              [0, -1], # obstaculo detectado a derecha
              [0, 1], # obstaculo a izq
              [0, 0], # obstaculo centro
              [1, 1], # demasiado cerca a derecha
              [1, -1], # demasiado cerca a izq
              [1, 0] # demasiado cerca centro
            ])
# Las salidas 'y' se corresponden con encender (o no) Los motores
y = np.array([[1,0,0,1], # avanzar
              [1,0,0,1], # avanzar
              [1,0,0,1], # avanzar
              [0,1,0,1], # giro derecha
              [1,0,1,0], # giro izquierda (cambie izq y derecha)
              [1,0,0,1], # avanzar
              [0,1,1,0], # retroceder
              [0,1,1,0], # retroceder
              [0,1,1,0] # retroceder
            ])
nn.fit(X, y, learning_rate=0.03,epochs=40001)

def valNN(x):
    return (int)(abs(round(x)))

index=0
for e in X:
    prediccion = nn.predict(e)
    print("X:",e,"esperado:",y[index],"obtenido:", valNN(prediccion[0]),valNN(prediccion[1]),valNN(prediccion[2]),valNN(prediccion[3]))
    #print("X:",e,"y:",y[index],"Network:",prediccion)
    index=index+1
```

Las salidas obtenidas son: (comparar los valores “y” con los de “Network”) esto nos permitirá comparar los valores obtenidos, a partir de ciertos valores de entrada, con los valores que se habían fijado como salidas en un inicio, de tal manera que al ser similares significaría que la red neuronal funciona correctamente.

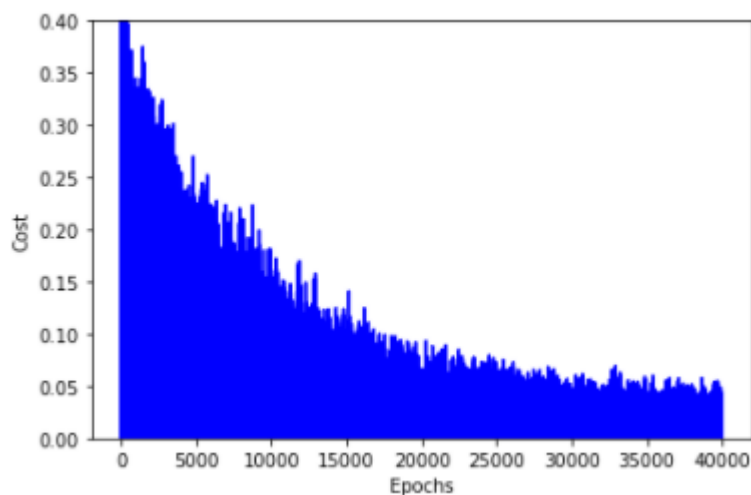
```
epochs: 0
epochs: 10000
epochs: 20000
epochs: 30000
epochs: 40000
X: [-1  0] esperado: [1 0 0 1] obtenido: 1 0 0 1
X: [-1  1] esperado: [1 0 0 1] obtenido: 1 0 0 1
X: [-1 -1] esperado: [1 0 0 1] obtenido: 1 0 0 1
X: [ 0 -1] esperado: [0 1 0 1] obtenido: 0 1 0 1
X: [0 1] esperado: [1 0 1 0] obtenido: 1 0 1 0
X: [0 0] esperado: [1 0 0 1] obtenido: 1 0 0 1
X: [1 1] esperado: [0 1 1 0] obtenido: 0 1 1 0
X: [ 1 -1] esperado: [0 1 1 0] obtenido: 0 1 1 0
X: [1 0] esperado: [0 1 1 0] obtenido: 0 1 1 0
```

Para un análisis más completo graficamos el comportamiento de error para visualizar como se iba reduciendo a partir del aumento de iteraciones tratando de llegar a el valor que fijamos, para este caso de 0.03.

```
In [5]: import matplotlib.pyplot as plt

deltas = nn.get_deltas()
valores=[]
index=0
for arreglo in deltas:
    valores.append(arreglo[1][0] + arreglo[1][1])
    index=index+1

plt.plot(range(len(valores)), valores, color='b')
plt.ylim([0, 0.4])
plt.ylabel('Cost')
plt.xlabel('Epochs')
plt.tight_layout()
plt.show()
```



Una vez en este punto ya es posible imprimir el valor que se ha dado a los pesos pues ya comprobamos que la red está funcionando correctamente.

Lo siguiente corresponderá a la inclusión de estos valores de pesos para el código en Arduino, mismo que controlará el móvil evasor de obstáculos.

```
In [7]: # Obtenermos Los pesos entrenados para poder usarlos en el codigo de arduino
pesos = nn.get_weights();

print('// SECCION PARA CAMBIAR PESOS :')
print('// float HiddenWeights ...')
print('// float OutputWeights ...')
print('// Con lo pesos entrenados.')
print('\n')
print(to_str('HiddenWeights', pesos[0]))
print(to_str('OutputWeights', pesos[1]))
```

```
// Reemplazar estas líneas en tu código arduino:
// float HiddenWeights ...
// float OutputWeights ...
// Con lo pesos entrenados.
```

```
float HiddenWeights[3][4] = {{-0.9618163922099295, -0.9793559932251648, 0.8190248365888009, -1.7082674910280957}, {4.7281827919
66169, 4.526141204231748, -0.561969085100709, -0.6977565392307244}, {-1.6329013706068534, 1.5477731824855092, 0.021427898725739
927, -0.1466448090123346}};
float OutputWeights[4][4] = {{-1.5759068372758238, 1.601885286580344, 0.29513981353842766, 0.605636698752538}, {0.3301170474153
5786, 0.28383622296344396, 1.6139682717467319, -1.7423137539434284}, {0.22812378563924726, 2.199227291808649, 2.204403971753740
7, 0.9391249728105223}, {-1.2057007850192127, 0.07441196080952706, 0.04902682287001095, -0.8935661492108808}};
```

## 2. Implementación del Código en Arduino

El código de Arduino consta de básicamente 4 partes. Cabecera, generamos los elementos a utilizar y cargamos las librerías necesarias para el proyecto, además de los parámetros iniciales del carrito.

En los parámetros iniciales son, la posición inicial del servo, el ángulo máximo y mínimo, los incrementos del ángulo de giro, el multiplicador, el sentido inicial del giro, la acción en curso y la velocidad del carrito.

Para este proyecto solo necesitamos la librería “Servo.h” (permite que las placas Arduino/Genuino controlen una variedad de servomotores), con ella controlaremos el movimiento de nuestro servomotor.

```
#include <Servo.h>
```

Posteriormente definimos los pines de nuestro sensor ultrasónico y los pines de control de nuestro puente h. Utilizamos las siguientes líneas de código:

```
int Echo = A1;
int Trig = A0;
#define ENA 13
#define ENB 12
#define IN1 46
#define IN2 48
#define IN3 50
#define IN4 52

unsigned long previousMillis = 0;
const long interval = 12;
int grados_servo = 90;
bool clockwise = true;
const long ANGULO_MIN = 0;
```

```

const long ANGULO_MAX = 180;
double distanciaMaxima = 60.0;
int incrementos =4;
int accionEnCurso = 1;
int multiplicador = 1000/interval;
const int SPEED =100;

```

Declaramos la red neuronal, para ello declaramos los nodos que serán para los resultados de las entradas (InputNodes, HiddenNodes, OutputNodes), de las capas ocultas, las salidas, también construimos la matriz de la red neuronal, además de dos acumuladores que se utilizarán en una función que evaluación las entradas en relación con los pesos para obtener las salidas.

```

const int InputNodes = 3;
const int HiddenNodes = 4;
const int OutputNodes = 4;
int i, j;
double Accum;
double Hidden[HiddenNodes];
double Output[OutputNodes];
float HiddenWeights[3][4] = { { 1.8991509504079183, -0.4769472541445052, -
0.6483690220539764, -0.38609165249078925}, {-0.2818610915467527,
4.040695699457223, 3.2291858058243843, -2.894301104732614},
{0.3340650864625773, -1.4016114422346901, 1.3580053902963762, -
0.981415976256285} };
float OutputWeights[4][4] = { { 1.136072297461121, 1.54602394937381,
1.6194612259569254, 1.8819066696635067}, {-1.546966506764457,
1.3951930739494225, 0.19393826092602756, 0.30992504138547006}, {-
0.7755982417649826, 0.9390808625728915, 2.0862510744685485, -
1.1229484266101883}, {-1.2357090352280826, 0.8583930286034466,
0.724702079881947, 0.9762852709700459} };

```

Segundo, el “void Setup”, definimos todas las entradas y salidas de nuestro movil y la posición inicial del servomotor.

```

myservo.attach(3);
pinMode(Echo, INPUT);
pinMode(Trig, OUTPUT);
pinMode(IN1, OUTPUT);
pinMode(IN2, OUTPUT);
pinMode(IN3, OUTPUT);
pinMode(IN4, OUTPUT);
pinMode(ENA, OUTPUT);
pinMode(ENB, OUTPUT);

stop();

myservo.write(90);

delay(2000);

```

En tercer lugar Tenemos el void loop en este únicamente definimos los intervalos de tiempo que nos ayudarán a determinar las salidas del sistema, además de obtener la salidas de la red neuronal.

```

    unsigned long currentMillis = millis();
    if (currentMillis - previousMillis >= interval) {
        previousMillis = currentMillis;

        if(grados_servo<=ANGULO_MIN || grados_servo>=ANGULO_MAX)
        {

            clockwise=!clockwise;
            grados_servo = constrain(grados_servo, ANGULO_MIN, ANGULO_MAX);

        }

        if(clockwise) {grados_servo=grados_servo+incrementos;}

        else {grados_servo=grados_servo-incrementos;}
        if(accionEnCurso>0){ accionEnCurso=accionEnCurso-1; }
        else{ conducir();}
        myservo.write(grados_servo);}

```



Por último, tenemos las subfunciones que son las que hacen el proceso de evalúa las entradas con los pesos para obtener las salidas. La función que se dedica enteramente a esto se llama InputToOutput la cual evalúa los parámetros recibido de las entradas (ángulo de giro y distancia del objeto) y los pesos de las matrices de la red neuronal.

```
void InputToOutput(double In1, double In2, double In3)
```

```
{  
    double TestInput[] = {0, 0,0};  
    TestInput[0] = In1;  
    TestInput[1] = In2;  
    TestInput[2] = In3;  
    for ( i = 0 ; i < HiddenNodes ; i++ )  
    {  
        Accum = 0;//HiddenWeights[InputNodes][i] ;  
        for ( j = 0 ; j < InputNodes ; j++ ) {  
            Accum += TestInput[j] * HiddenWeights[j][i] ;  
        }  
        //Hidden[i] = 1.0 / (1.0 + exp(-Accum)) ; //Sigmoid  
        Hidden[i] = tanh(Accum) ; //tanh  
    }  
}
```

En el ciclo anterior calculamos las activaciones en las capas ocultas, mientras que en las líneas siguientes calculamos el error en la capa de salida.

```
for ( i = 0 ; i < OutputNodes ; i++ ) {  
    Accum = 0;//OutputWeights[HiddenNodes][i];  
    for ( j = 0 ; j < HiddenNodes ; j++ ) {  
        Accum += Hidden[j] * OutputWeights[j][i] ;  
    }  
    Output[i] = tanh(Accum) ;//tanh  
}}
```

Ahora creamos una subfunción para impulsar los motores con la salida de la red neuronal.

```
int carSpeed = SPEED; //hacia adelante o atras
```

```

if((out1+out3)==2 || (out2+out4)==2){ // si es giro, necesita doble fuerza los motores
    carSpeed = SPEED * 2;
}
analogWrite(ENA, carSpeed);
analogWrite(ENB, carSpeed);
digitalWrite(IN1, out1 * HIGH);
digitalWrite(IN2, out2 * HIGH);
digitalWrite(IN3, out3 * HIGH);
digitalWrite(IN4, out4 * HIGH);
//poneos las salidas el los diferentes estados de salidas.
}

```

Por ultimo creamos el sistema que se dedica enteramente al control de la conducción del carrito.

```

void conducir()
{
    double TestInput[] = {0, 0,0};
    double entrada1=0, entrada2=0;

    //=====
    //===== OBTENER DISTANCIA DEL SENSOR =====
    //=====

    double distance = double(Distance_test()); // retoma el valor de la función de
prueba.

    distance= double(constrain(distance, 0.0, ditanciaMaxima)); //Restringe un número
para que esté dentro de un rango.

    entrada1= ((-2.0/ditanciaMaxima)*double(distance))+1.0; //uso una función lineal
para obtener cercanía.

    accionEnCurso = ((entrada1 +1) * multiplicador)+1; // si está muy cerca del
obstáculo, necesita más tiempo de reacción.

```

```

//=====
//===== OBTENER DIRECCION SEGUN ANGULO DEL SERVO =====
//=====

    entrada2 = map(grados_servo,  ANGULO_MIN,  ANGULO_MAX,  -100,
100);//Vuelve a mapear un número de un rango a otro
    entrada2 = double(constrain(entrada2, -100.00, 100.00));//Restringe un número
para que esté dentro de un rango.

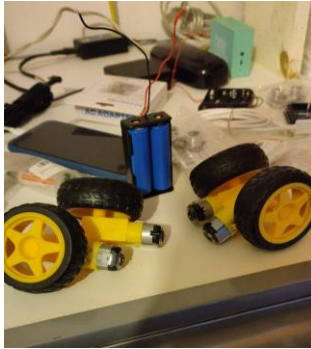
//=====
//=== LLAMAMOS A LA RED FEEDFORWARD CON LAS ENTRADAS ===
//=====

TestInput[0] = 1.0;//Unidad de las BIAS
TestInput[1] = entrada1;
TestInput[2] = entrada2/100.0;// manteien el rango entre -1 y 1.
//llamamos a la función InputToOutput, donde se evalúan las entradas con los pesos.
InputToOutput(TestInput[0], TestInput[1], TestInput[2]); //ENTRADAS a ANN
para obtener las SALIDAS.
//Obtenemos los valores de las salidas
int out1 = round(abs(Output[0]));
int out2 = round(abs(Output[1]));
int out3 = round(abs(Output[2]));
int out4 = round(abs(Output[3]));

```

## Ensamblado del carrito

### Armado de chasis:



1.-Se arman las llantas con el motor.



2.-Se juntan las llantas con el chasis inferior



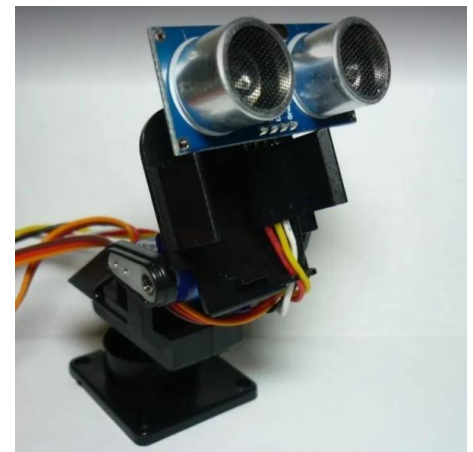
3.- Se conecta el chasis inferior con el superior.



1.-Se ensambla la pieza del medio



2.-Se conectan las piezas con ayuda de un servomotor que va a dirigir el eje X dándole movilidad en un rango de 180°.



3.-Se conecta por último el sensor ultrasónico y la base se conecta a la tabla superior del chasis.

## Resultados:

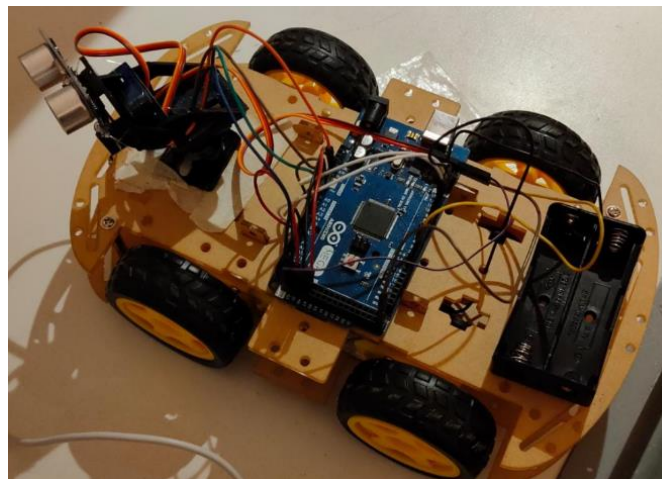
En primer paso logramos concluir lo que sería la red neuronal mostrando los siguientes pesos y con lo que posteriormente estos se ocuparon para ponerlos en el microcontrolador Arduino mega dando como resultado de pesos lo siguiente:

```
// Reemplazar estas líneas en tu código arduino:  
// float HiddenWeights ...  
// float OutputWeights ...  
// Con los pesos entrenados.  
  
float HiddenWeights[3][4] = {{-0.9618163922099295, -0.9793559932251648, 0.8190248365888009, -1.7082674910280957}, {4.7281827919  
66169, 4.526141204231748, -0.561969085100709, -0.6977565392307244}, {-1.6329013706068534, 1.5477731824855092, 0.021427898725739  
927, -0.1466448090123346}};  
float OutputWeights[4][4] = {{-1.5759068372758238, 1.601885286580344, 0.29513981353842766, 0.605636698752538}, {0.3301170474153  
5786, 0.28383622296344396, 1.6139682717467319, -1.7423137539434284}, {0.22812378563924726, 2.199227291808649, 2.204403971753740  
7, 0.9391249728105223}, {-1.2057007850192127, 0.07441196080952706, 0.04902682287001095, -0.8935661492108808}};
```

De ahí se pudo terminar satisfactoriamente el armado en físico para dar comienzo con la programación en Arduino y el entrenamiento de la red neuronal para que nuestro prototipo evasor de obstáculos funcione correctamente.

En el siguiente link se encuentra un video de nuestro proyecto funcionando y una foto de nuestro proyecto final.

Link de YouTube: <https://www.youtube.com/watch?v=Q9UMLYXuWuk>



## Conclusión:

Elaboramos en Python una red neuronal, así mismo comprender como funciona la red neuronal, interacción entre la función sigmoide y derivadas que nos permiten hacer Backpropagation, con estos valores calibrar sus iteraciones para reducir el error y obtener las salidas esperadas, haciendo que la red neuronal aprenda por sí misma, necesitando solo datos de entrada y salidas esperadas, mismos pesos obtenidos, colocarlos a nuestro carrito para así sea capaz de andar por sí mismo y que al detectar un obstáculo o evite.