CREAR UNA RED NEURONAL

INSTITUTO TECNOLOGICO DE IZTAPALAPA
MATRIA: INTELIGENCIA ARTIFICIAL
PROFESOR: ABIEL TOMAS PARRA H.

ALUMNA: SANTAMARIA CIRILIO NORMA NELLY
NUM. CONTROL: 171080086
GRUPO: ISC-8AV

INDICE

Pagina

Información sobre la red neuronal

¿Qué es una red neuronal?

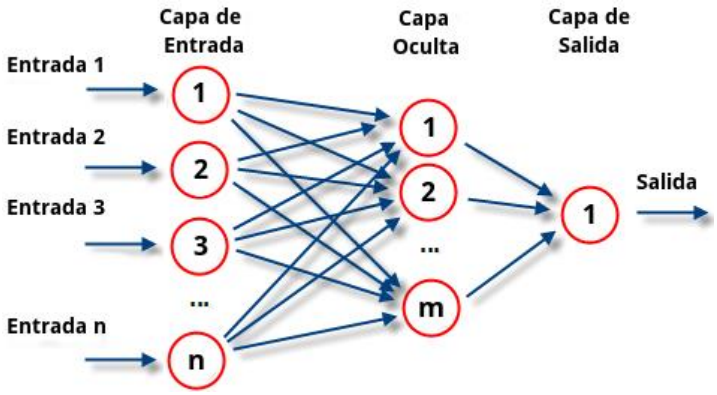
Las redes neuronales artificiales son un modelo inspirado en el funcionamiento del cerebro humano. Está formado por un conjunto de nodos conocidos como neuronas artificiales que están conectadas y transmiten señales entre sí. Estas señales se transmiten desde la entrada hasta generar una salida. Esto nos ayudara para darle uso al proyecto que se basara en un coche robótico, ya que este se le implantara, la red neuronal para que pueda conducir, evitando los obstáculos.

¿Cuál es el objetivo de las redes neuronales?

El objetivo principal de este modelo es aprender modificándose automáticamente a si mismo de forma que puede llegar a realizar tareas complejas que no podrían ser realizadas mediante la clásica programación basada en reglas. De esta forma se pueden automatizar funciones que en un principio solo podrían ser realizadas por personas. Pero con el paso de los años se ha podido desarrollar robots con esta función, como será en este proyecto con el carro robótico.

¿Cómo funcionan las redes neuronales?

Como se ha mencionado el funcionamiento de las redes se asemeja al del cerebro humano. Las redes reciben una serie de valores de entrada y cada una de estas entradas llega a un nodo llamado neurona. Las neuronas de la red están a su vez agrupadas en capas que forman la red neuronal. Cada una de las neuronas de la red posee a su vez un peso, un valor numérico, con el que modifica la entrada recibida. Los nuevos valores obtenidos salen de las neuronas y continúan su camino por la red. Este funcionamiento puede observarse de forma esquemática como en la siguiente imagen:



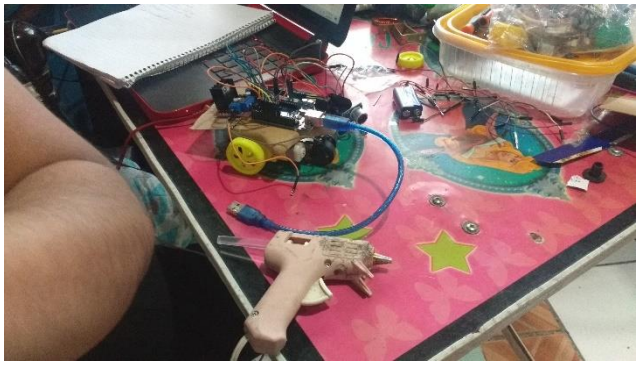
Esquema de funcionamiento de la red neuronal

Una vez que se ha alcanzado el final de la red se obtiene una salida que será la predicción calculada por la red. Cuantas más capas posea la red y más compleja sea, también serán más complejas las funciones que pueda realizar.

Desarrollo del carro

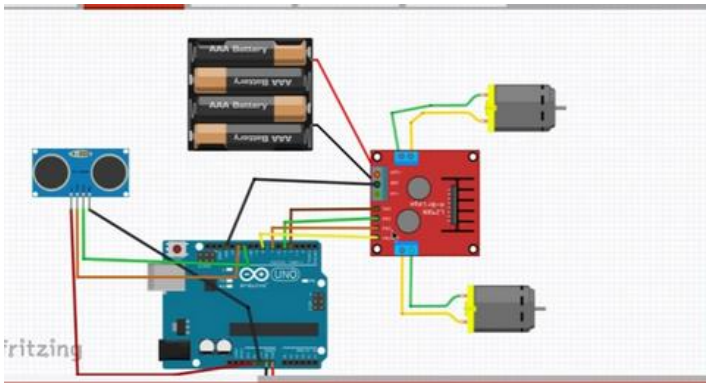
Materiales para el desarrollo del carro

- Una placa Arduino Uno
- Un Drivel L298N
- 2 motores DC y sus ruedas
- Servo Motor SG90
- Sensor Ultrasónico HC-SR04
- Batería 9v
- Un porta batería
- Jumpers



Como estará el orden de los materiales

En seguida se mostrara como se tiene que hacer el circuito de cada uno de los materiales:



Circuito del coche

Después se programara la tarjeta Arduino, el código Arduino controlará el servo motor con el sensor de distancia que se moverá de izquierda a derecha y nos proveerá las entradas de la red: Distancia y Dirección (giro).

¡El resto, lo hará la red neuronal! En realidad, la red ya «aprendió» (en Python) es decir, sólo hará multiplicaciones y sumas de los pesos para obtener salidas. Realizará el camino forward propagación. Y las salidas controlarán directamente los 4 motores.

```
#include <Servo.h>  //servo library
Servo myservo;      // create servo object to control servo

int Echo = A4;
int Trig = A5;
#define ENA 5
#define ENB 6
#define IN1 7
#define IN2 8
#define IN3 9
#define IN4 11

/*********************************************************************
   Network Configuration
 *********************************************************************/
const int InputNodes = 3; // incluye neurona de BIAS
const int HiddenNodes = 4; //incluye neurona de BIAS
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, -0.3860916524
float OutputWeights[4][4] = {{1.136072297461121, 1.54602394937381, 1.6194612259569254, 1.8819066696635067
/*********************************************************************
   End Network Configuration
 *********************************************************************/

void stop() {
  digitalWrite(ENA, LOW); //Desactivamos los motores
  digitalWrite(ENB, LOW); //Desactivamos los motores
  Serial.println("Stop!");
}
```

El objetivo de este desarrollo del carro, será para ver el funcionamiento de la red neuronal, en este sentido en todo momento, está activo la red neuronal ya que esta conformad en todo el circuito y también en el código que se utilizara. En este caso tenemos una red de tres capas con 2 neuronas de entrada 3 ocultas y 2 de salida: giro y dirección. Para este ejercicio haremos que la red neuronal tenga 4 salidas: una para cada motor. Además las salidas serán entre 0 y 1 (apagar o encender motor). También cambiaremos las entradas para que todas comprendan valores entre -1 y 1 y sean acordes a nuestra función tangente hiperbólica. Para poder entenderlo mejor está mejor explicada en la siguiente tabla:

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

Obteniendo los resultados se tiene que poner el código pero será en Python para que todo el circuito tenga la misma función.

```
# 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]))
    index=index+1
```

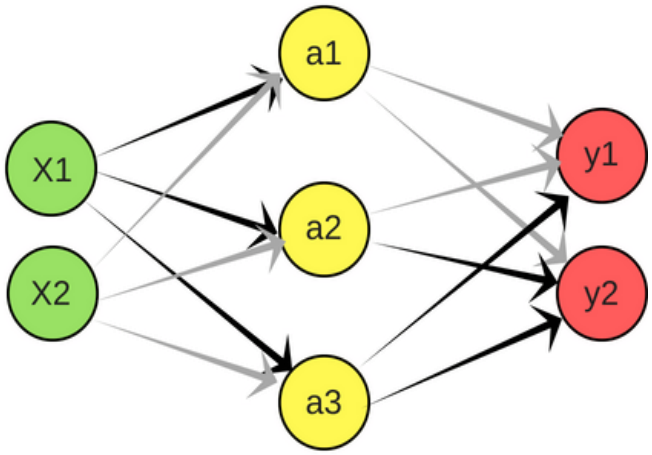
Nuestros datos de entrada serán:

- 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
- Posición del obstáculo (izquierda,derecha)
- El obstáculo es visto a la izquierda será -1
- visto a la derecha será 1
  Las salidas serán
- Girar derecha 1 / izquierda -1
- Dirección avanzar 1 / retroceder -1

La velocidad del vehículo podría ser una salida más (disminuir la velocidad si nos aproximamos a un objeto) y podríamos usar más sensores como entradas pero por simplificar el modelo y su implementación mantendremos estas 2 entradas y 2 salidas. Para entrenar la red tendremos las entradas y salidas que se ven en la tabla:

Entrada: Sensor Distancia	Entrada: Posición Obstáculo	Salida: Giro	Salida: Dirección	Acción de la Salida
0	0	0	1	Avanzar
0	1	0	1	Avanzar
0	-1	0	1	Avanzar
0.5	1	-1	1	Giro a la izquierda
0.5	-1	1	1	Giro a la derecha
0.5	0	0	1	Avanzar
1	1	0	-1	Retroceder
1	-1	0	-1	Retroceder
1	0	0	-1	Retroceder
-1	0	0	1	Avanzar
-1	-1	0	1	Avanzar
-1	1	0	1	Avanzar

Esta será la arquitectura de la red neuronal propuesta



En la imagen anterior -y durante el ejemplo- usamos la siguiente notación en las neuronas:

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

Y quedan implícitos, pero sin representación en la gráfica:

1. **O(j)** Los pesos de las conexiones entre neuronas será una matriz que mapea la capa j a la j+1
2. Recordemos que utilizamos 1 neurona extra en la capa 1 y una neurona extra en la capa 2 a modo de Bias -no están en la gráfica- para mejorar la precisión de la red neuronal, dandole mayor «libertad algebraica».

Los cálculos para obtener los valores de activación serán:

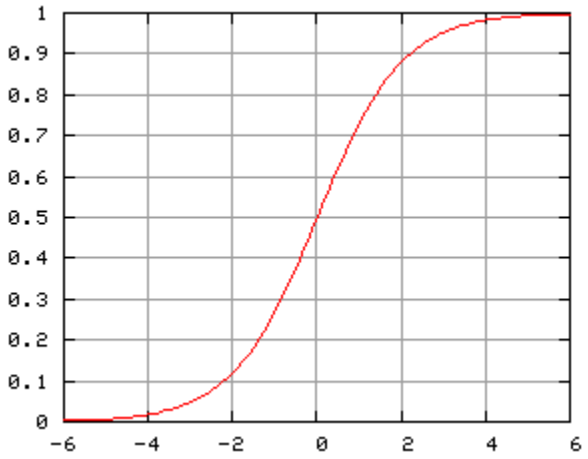
$a(1) = g(O^T_1 X)$
$a(2) = g(O^T_2 X)$
$a(3) = g(O^T_3 X)$

En las ecuaciones, la **g** es una *función Sigmoide* que refiere al caso especial de función logística y definida por la fórmula:

$$g(z) = 1/(1+e^{-z})$$

Funciones Sigmoide

Una de las razones para utilizar la función sigmoide –función Logística– es por sus propiedades matemáticas, en nuestro caso, sus derivadas. Cuando más adelante la red neuronal haga backpropagation para aprender y actualizar los pesos, haremos uso de su derivada. En esta función puede ser expresada como productos de f y 1-f . Entonces f'(t) = f(t)(1 – f(t)). Por ejemplo la función tangente y su derivada arco-tangente se utilizan normalizadas, donde su pendiente en el origen es 1 y cumplen las propiedades.



Forward Propagación o red Feedforward

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 = [x0 x1 x2]

zlayer2 = O1X

alayer2 = g (zlayer2)

zlayer3 = O2alayer2

y = g (zlayer3)

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).

Backpropagation (cómputo del gradiente)

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.

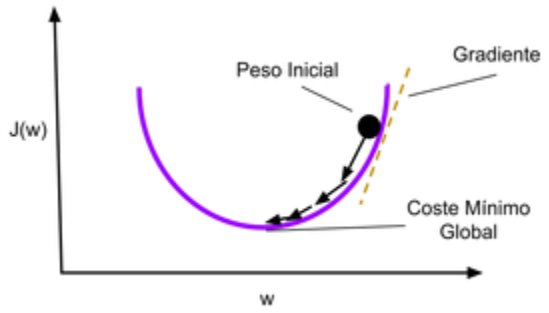
Fase 1: Propagar

Esta fase implica 2 pasos:
1.1 Hacer forward propagación 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.

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» o 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.



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) = alayer3 – y
D (2) = OT2 d (3). G'(zlayer2)
G'(zlayer2) = alayer2. (1 – alayer2)
Al fin aparecieron las derivadas 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 costo -que es lo que queremos minimizar- de nuestra red será
J = alayer dlayer + 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.

El Código Completo de la red Neuronal con Backpropagation

Primero, declaramos la clase NeuralNetwork

```
import numpy as np

def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

def sigmoid_derivada(x):
    return sigmoid(x)*(1.0-sigmoid(x))

def tanh(x):
    return np.tanh(x)

def tanh_derivada(x):
    return 1.0 - x**2


class NeuralNetwork:

    def __init__(self, layers, activation='tanh'):
        if activation == 'sigmoid':
            self.activation = sigmoid
            self.activation_prime = sigmoid_derivada
        elif activation == 'tanh':
            self.activation = tanh
            self.activation_prime = tanh_derivada

        # inicializo los pesos
        self.weights = []
        self.deltas = []
        # capas = [2,3,2]
        # rando de pesos varia entre (-1,1)
        # asigno valores aleatorios a capa de entrada y capa oculta
        for i in range(1, len(layers) - 1):
            r = 2*np.random.random((layers[i-1] + 1, layers[i] + 1)) -1
            self.weights.append(r)
        # asigno aleatorios a capa de salida
        r = 2*np.random.random( (layers[i] + 1, layers[i+1])) - 1
        self.weights.append(r)

    def fit(self, X, y, learning_rate=0.2, epochs=100000):
        # Agrego columna de unos a las entradas X
        # Con esto agregamos la unidad de Bias a la capa de entrada
        ones = np.atleast_2d(np.ones(X.shape[0]))
        X = np.concatenate((ones.T, X), axis=1)

        for k in range(epochs):
            i = np.random.randint(X.shape[0])
            a = [X[i]]

            for l in range(len(self.weights)):
                    dot_value = np.dot(a[l], self.weights[l])
                    activation = self.activation(dot_value)
                    a.append(activation)
            # Calculo la diferencia en la capa de salida y el valor obtenido
            error = y[i] - a[-1]
            deltas = [error * self.activation_prime(a[-1])]

            # Empezamos en el segundo layer hasta el ultimo
            # (Una capa anterior a la de salida)
            for l in range(len(a) - 2, 0, -1):
                deltas.append(deltas[-1].dot(self.weights[l].T)*self.activation_prime(a[l]))
            self.deltas.append(deltas)

            # invertir
            # [level3(output)->level2(hidden)]  => [level2(hidden)->level3(output)]
            deltas.reverse()

            # backpropagation
            # 1. Multiplcar los delta de salida con las activaciones de entrada
            #    para obtener el gradiente del peso.
            # 2. actualizo el peso restandole un porcentaje del gradiente
            for i in range(len(self.weights)):
                layer = np.atleast_2d(a[i])
                delta = np.atleast_2d(deltas[i])
                self.weights[i] += learning_rate * layer.T.dot(delta)

            if k % 10000 == 0: print('epochs:', k)

    def predict(self, x):
        ones = np.atleast_2d(np.ones(x.shape[0]))
        a = np.concatenate((np.ones(1).T, np.array(x)), axis=0)
        for l in range(0, len(self.weights)):
            a = self.activation(np.dot(a, self.weights[l]))
        return a

    def print_weights(self):
        print("LISTADO PESOS DE CONEXIONES")
        for i in range(len(self.weights)):
            print(self.weights[i])

    def get_deltas(self):
        return self.deltas
```

Y ahora creamos una red a nuestra medida, con 2 neuronas de entrada, 3 ocultas y 2 de salida. Deberemos ir ajustando los parámetros de entrenamiento learning rate y la cantidad de iteraciones «epochs» para obtener buenas predicciones.

```
# funcion Coche Evita obstáculos
nn = NeuralNetwork([2,3,2],activation ='tanh')
X = np.array([[0, 0],   # sin obstaculos
              [0, 1],   # sin obstaculos
              [0, -1],  # sin obstaculos
              [0.5, 1], # obstaculo detectado a derecha
              [0.5,-1], # obstaculo a izq
              [1,1],    # demasiado cerca a derecha
              [1,-1]])  # demasiado cerca a izq

y = np.array([[0,1],    # avanzar
              [0,1],    # avanzar
              [0,1],    # avanzar
              [-1,1],   # giro izquierda
              [1,1],    # giro derecha
              [0,-1],   # retroceder
              [0,-1]])  # retroceder
nn.fit(X, y, learning_rate=0.03,epochs=15001)

index=0
for e in X:
    print("X:",e,"y:",y[index],"Network:",nn.predict(e))
    index=index+1
```

La salidas obtenidas son: (comparar los valores «y» con los de «Network»)

```
X: [0. 0.]      y: [0 1]   Network: [0.00112476   0.99987346]
X: [0. 1.]      y: [0 1]   Network: [-0.00936178  0.999714  ]
X: [ 0. -1.]    y: [0 1]   Network: [0.00814966   0.99977055]
X: [0.5 1. ]    y: [-1  1] Network: [-0.92739127  0.96317035]
X: [ 0.5 -1. ] y: [1 1]   Network: [0.91719235   0.94992698]
X: [1. 1.]      y: [ 0 -1] Network: [-8.81827252e-04 -9.79524215e-01]
X: [ 1. -1.]    y: [ 0 -1] Network: [ 0.00806883  -0.96823086]
```

Como podemos ver son muy buenos resultados.
Aquí podemos ver como el coste de la función se va reduciendo y tiende a cero:

```
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, 1])
plt.ylabel('Cost')
plt.xlabel('Epochs')
plt.tight_layout()
plt.show()
```

Conclusión

Este proyecto fue realizado para la materia de inteligencia artificial, a mí me llamo más la atención una red neural ya que es una simulación de una neurona humana y ver como funciona y de esta manera lo simplifique en un carro es decir simularía la conducción de un coche sin un humano, me gustó mucho el proyecto ya que es algo ya de la vida real porque es lo que están creando.

Mas allá de ser este ejemplo la red neural se puede ocupar en muchas cosas