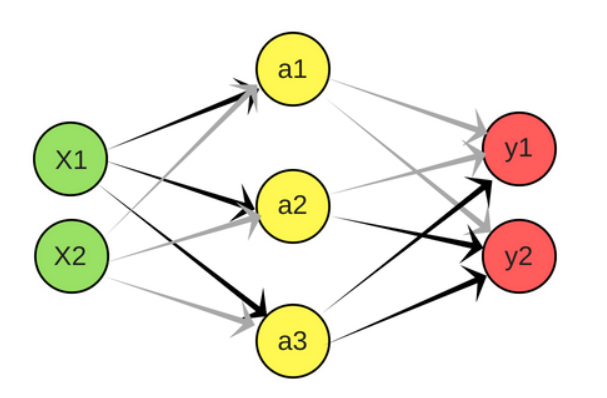
https://www.aprendemachinelearning.com/crear-una-red-neuronal-en-python-desde-cero/

| **Entrada: X1 Sensor Distancia** | **Entrada: X2 Posición Obstáculo** | **Salida: Giro Y1** | **Salida: Dirección Y1** | **Acción de la Salida  “a”** |
| --- | --- | --- | --- | --- |
| 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 |



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

Una de las razones para utilizar la función sigmoide –[función Logística](https://es.wikipedia.org/wiki/Funci%C3%B3n_log%C3%ADstica)– 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.

|  |
| --- |
| 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.

X1 X2 Y1 Y2 a

0 0 0 1 Avanzar

0 1 0 1 Avanzar

0 -1 0 1 Avanzar

0.5 0 0 1 Avanzar

0.5 1 -1 1 Giro Izq

0.5 -1 1 1 Giro Der

1 0 0 -1 Retroceder

1 1 0 -1 Retroceder

1 -1 0 -1 Retroceder

-1 0 0 1 Avanzar

-1 1 0 1 Avanzar

-1 -1 0 1 Avanzar

|  |  |
| --- | --- |
|  | # funcion Coche Evita obstáculos  nn = NeuralNetwork([2,3,2],activation ='tanh')  #INPUT X1, X2  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  #SALIDAS Y1, Y2  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() |

[Chart, histogram

Description automatically generated](https://www.aprendemachinelearning.com/wp-content/uploads/2018/07/descenso-gradiente.png)

Y podemos ver los pesos obtenidos de las conexiones con **nn.print\_weights()** pues estos valores serán los que usaremos en la red final que en [un próximo artículo implementaremos en Arduino para que un coche-robot conduzca sólo evitando obstáculos](https://www.aprendemachinelearning.com/programa-un-coche-arduino-con-inteligencia-artificial/).

Conclusión

Creamos una red neuronal en pocas líneas de código Python:

* comprendimos cómo funciona una red neuronal “básica”,
* el porqué de las funciones Sigmoides y sus derivadas que …
* nos permiten hacer Backpropagation,
* hallar el gradiente para minimizar el coste,
* reducir el error iterando y obtener las salidas buscadas,
* logrando que la red aprenda por sí misma en base a un conjunto de datos de entrada y sus salidas como “buen” Algoritmo Supervisado que es.