INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Introducción a las Redes Neuronales Artificiales

Dr. Juan Humberto Sossa Azuela

Reporte Práctica No.1

Acosta Rosales Jair Sebastián

Para realizar está práctica se hizo uso del lenguaje Python, un poderoso lenguaje de programación orientado a objetos de alto nivel que tiene múltiples frameworks y librerías que nos permiten hacer machine learning de una forma más sencilla, como lo es el caso de tensorflow, keras, numpy o pythorch, de esta forma podemos elegir varias formas de implementación, en este caso, se optó por la siguiente librería:

-Numpy

Una librería bastante optimizada que permite la operación entre matrices y vectores de una forma mucho más rápida que si lo hacemos de manera convencional, por lo tanto, y para una técnica como lo es backpropagation nos permite tener un tiempo de ejecución y entrenamiento más rápido, aunque, evidentemente esto depende del problema a resolver, en este caso, nuestro problema a resolver es bastante sencillo, al tener que entrenar una red que resuelva de una a cinco compuertas al mismo tiempo, las cuales son AND, OR, NAND, NOR, y XOR.

1. Inicializar los pesos y bias de forma aleatoria, usando una distribución aleatoria gaussiana normalizada.

Para realizar este primer punto se usó una función de la librería numpy, llamada:

random.normal(size=(rows,cols))

Donde:

rows = número de filas de la matriz

cols = número de columnas de la matriz

2. Modificar el programa anterior con el objetivo de poder variar el número de neuronas en la capa oculta a petición del usuario.

Para realizar esto se desarrolló una función que permitía al usuario ingresar el número de neuronas de la capa oculta, con este valor era posible inicializar los pesos y los bias de dicha capa, dado que, la dimensión de la matriz de pesos y de la matriz de bias de la capa oculta depende directamente del número de neuronas, donde podemos observar que:

rows_matrix = número de neuronas de la capa oculta.

cols_matrix = número de neuronas o entradas de la capa anterior.

Por lo tanto, si queremos 10 neuronas en la capa oculta, y tenemos una entrada de 5 vectores, la dimensión de la matriz de pesos será **dim_matrix = (10,5)** de modo que el número de pesos en la matriz es de **10*5=50,** pues cada neurona tendrá 5 pesos sinápticos.

A continuación, se muestra la imagen donde se inicializan los pesos y bias que dependen del número de neuronas de la capa oculta.

```
#MODEL FOR ONE HIDDEN LAYER WITH N-NEURONS

def Neural_network_model(X):

  n_hidden_layers = 1
  n_nodes_hl1 = int(input('Número de neuronas en capa oculta: '))
  # Initial weights
  #using a normal(gaussian)distribution
  network = []

  weights = {
      'W_hl0':np.random.normal(size=(n_nodes_hl1,X.shape[1])),
      'W_out':np.random.normal(size=(n_nodes_hl1))
  }



  biases = {
      'b_hl0':np.random.normal(size=(n_nodes_hl1)),
      'b_out':np.random.normal(size=(1))
  }


  layer_output = {}
  error_layer = {}

  network.append(weights)
  network.append(biases)
  network.append(layer_output)
  network.append(error_layer)
  network.append(n_hidden_layers)


  return network
```

En donde observamos en los recuadros de color rojo los pasos que se siguieron para inicializar los pesos y bias con base al número de neuronas del usuario, siendo:

n_nodes: Número de neuronas en la capa oculta.

X.shape[1]: Con X siendo la matriz que contiene el conjunto de vectores de entrenamiento, lo que hacemos es obtener la forma en la que está distribuida esa matriz, es decir, número de filas y columnas, con el atributo shape podemos obtener una tupla de la siguiente manera **(n_rows,n_cols)**, en este caso n_rows corresponde al número de vectores de entrenamiento que vamos a ingresar a la red, mientras que n_cols nos indica el tamaño de cada uno de esos vectores, por lo tanto obtenemos el segundo valor(n_cols) de la forma **shape[1].**

3. Modificar el programa anterior con el objetivo de poder variar el número de neuronas en la capa de salida a petición del usuario; con éste se podrán resolver m cantidades de compuertas lógicas simultáneamente.

En este punto había que modificar los targets correspondientes a nuestros vectores de entrenamiento, es decir, el número de salidas que se deben obtener ya no será el

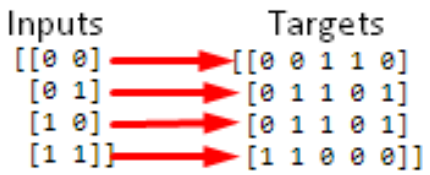
correspondiente a una sola neurona, lo será para cada compuerta que deseemos resolver, en este caso, se resolverán hasta un máximo de 5 compuertas lógicas, ya antes mencionadas, por lo tanto, al usuario se le debe preguntar cuántas compuertas lógicas pretende resolver, para poder inicializar los targets correspondientes a dicho número elegido por el usuario, para este caso, el formato de salida de las compuertas elegido es:

```
[AND, OR, NAND, NOR, XOR]
```

Como podemos ver la primer neurona de salida de la red corresponderá a la salida de la compuerta AND, la segunda a la OR, y así sucesivamente, de modo que si el usuario decide resolver tres compuertas, el vector estará conformado por [AND,OR,NAND], en la siguiente imagen el usuario decidió resolver las cinco compuertas lógicas:

```
¿Cuantas compuertas desea resolver?
[AND, OR, NAND, NOR, XOR]
5
[[0 0 1 1 0]
 [0 1 1 0 1]
 [0 1 1 0 1]
 [1 1 0 0 0]]
```

Por lo tanto, los targets para cada entrada serían:



4. Modificar el programa anterior con el objetivo de poder variar el número de capas ocultas. Cada capa oculta puede variar el número de neuronas en cada capa y la capa de salida puede resolver m cantidades de compuertas lógicas simultáneamente.

Para este punto se editó la anterior función de la arquitectura de la red neuronal, de modo que podamos decidir cuántas capas ocultas se desean y además el número de neuronas en cada una para su inicialización, para ello, hizo falta una nueva función que permitiera inicializar la matriz de pesos y el vector bias para cada capa oculta de la manera correcta de modo que:

```python
#MODEL FOR MLP WITH N-HIDDEN-LAYERS WITH N-NEURONS IN EACH ONE
def Neural_network_model(X,y):

  n_hidden_layers = int(input('Número de capas ocultas: '))
  #n_nodes_hl1 = int(input('Número de neuronas en capa oculta: '))
  # Initial weights
  #using a normal(gaussian)distribution
  network = []

  layer_output = {}
  error_layer = {}
  weights,biases = def_param(n_hidden_layers,X,y)
  network.append(weights)
  network.append(biases)
  network.append(layer_output)
  network.append(error_layer)
  network.append(n_hidden_layers)


  return network
```

La función que nos permite inicializar la matriz de pesos fue llamada **def_param(n_hidden_layers,x_train,y_train)** donde:

n_hidden_layers: Número de capas ocultas para la red.

x_train: Matriz de vectores de entrenamiento.

y_train: Matriz de etiquetas para cada vector de entrenamiento.

```python
#Defining architecture for the MLP
def def_param(n_layers,X,t):
  weights = {}
  bias = {}
  n_neurons_layer = {}
  """
  initializing weights and bias as follows:
  np.random.normal(size=(param1,param2)), size takes two arguments where:

  param1 = no. neurons in the current layer
  param2 = no. neurons in the previous layer
  """
  for i in range(n_layers+1):

    if i==0:#if its the first layer
      n_neurons = int(input("Numero de neuronas en la capa {} ".format(i)))
      weights['W_hl'+str(i)] = np.random.normal(size=(n_neurons,X.shape[1]))#the no. of neurons in the previous layers is e
      bias['b_hl'+str(i)] = np.random.normal(size=n_neurons)#the no. of bias is equal to the number of neurons
      n_neurons_layer['n_neurons'+str(i)] = n_neurons#stores the number of neurons in the current layer
    elif i==n_layers: #if its the output layer
      """
      the number of neurons in the output layer is equal to the target vector length
      """
      #weights['W_out'] = np.random.normal(size=(t.shape[1],n_neurons_layer['n_neurons'+str(i-1)]))
      #bias['b_out'] = np.random.normal(size=t.shape[1])
      if t.ndim == 1: #if its only one gate to solve
        weights['W_out'] = np.random.normal(size=(n_neurons_layer['n_neurons'+str(i-1)]))
        bias['b_out'] = np.random.normal(1)
        n_neurons_layer['n_neurons'+str(i)] =  1#stores the number of neurons in the current layer
      else:
        #t.shape[1] takes the number of columns of the target, this is the number of outputs in the output layer
        weights['W_out'] = np.random.normal(size=(t.shape[1],n_neurons_layer['n_neurons'+str(i-1)]))
        bias['b_out'] = np.random.normal(size=(t.shape[1]))
        n_neurons_layer['n_neurons'+str(i)] =  t.shape[1]#stores the number of neurons in the current layer


    else:#if its any other hidden layer
      n_neurons = int(input("Numero de neuronas en la capa {} ".format(i)))
      weights['W_hl'+str(i)] = np.random.normal(size=(n_neurons,n_neurons_layer['n_neurons'+str(i-1)]))
      bias['b_hl'+str(i)] = np.random.normal(size=n_neurons)
      n_neurons_layer['n_neurons'+str(i)] =  n_neurons#stores the number of neurons in the current layer

  return weights,bias
```

Usando diccionarios para almacenar cada matriz de pesos y bias en una misma variable para la red, preguntamos al usuario para cada capa, el número de neuronas que desee en dicha capa, de modo que, como ya se explicó anteriormente en el punto **2.** la matriz de pesos de esa capa se compone de **rows** = número de neuronas deseadas en esa capa y **cols** = número de neuronas en la capa anterior o número de entradas del vector de entrada en caso de ser la primera capa oculta, para el vector de bias, este es del mismo tamaño que el número de neuronas.

Se hizo uso de un diccionario adicional que guarda el número de neuronas de cada capa oculta que se inicializaba, de ese modo, para la capa siguiente era fácil acceder al número de neuronas de la capa anterior.

Entrenamiento

Una vez listas las funciones que permiten inicializar la arquitectura de la red neuronal pasamos a las funciones de entrenamiento, divididas en

- **Feedforward:** Dado un patrón(vector) de entrada, se realizan las operaciones correspondientes que recorren desde la primer hasta la última capa oculta para obtener un resultado en la capa de salida.
- **Backpropagation:** El error obtenido de la capa de salida es calculado, y se calculan los errores de cada capa oculta, partiendo desde la última capa, hasta la primera.
- **Update Weights and Bias:** Haciendo uso del principio de descenso del gradiente, una vez obtenidos los errores de cada capa, los usamos para actualizar los pesos de forma que nos permita disminuir el error con este cambio.

Feedforward

Esta operación está constituida por la propagación de la entrada hacia las capas ocultas, a través de las operaciones de producto punto de matrices y función de activación en cada neurona de cada capa, para ello se usa:

- **np.matmul(matriz1,matriz2):** que devuelve el vector resultante del producto punto de matriz1 por matriz2
- **sigmoid(x):** que dado el vector x, aplica la función sigmoide para saber que neuronas se están activando.

De este modo las funciones quedan:

```
# Sigmoid function
def sigmoid(x):

    sig = 1 / (1 + np.exp(-x))

    return(sig)
```

```python
# Feed forward
def feed_forward(x, network):

    n_layers = network[4]+1 #number of hidden layers plus the output layer
    outputD = {}
    """
    print("============================FEEDFORWARD============================")
    print("====================input {}====================".format(x))
    print("Weights {}".format(network[0]))
    print("Bias {}".format(network[1]))
    print("Output {}".format(outputD))
    """
    for i in range(n_layers):

      if i==0: #if its the first layer
        #print('Capa {}'.format(i))
        dotproduct = np.matmul(x,network[0]['W_hl'+str(i)].transpose()) + network[1]['b_hl'+str(i)] #does dot product
        output = sigmoid(dotproduct) #
        #print(output)
        #print(output.shape)
        #print(output.transpose().shape)
        outputD['output_hl'+str(i)]=output #stores the output of each layer hidden layer
      elif i==n_layers-1: #if it is the last layer
        #print('Capa {}'.format(i))
        dotproduct = np.matmul(outputD['output_hl'+str(i-1)],network[0]['W_out'].transpose()) + network[1]['b_out'] #does d
        output = sigmoid(dotproduct) #
        #print(output)
        #print(output.shape)
        #print(output.transpose().shape)
        outputD['output_out']=output #stores the output of output layer
      else: #if its any other hidden layer
        #print('Capa {}'.format(i))
        dotproduct = np.matmul(outputD['output_hl'+str(i-1)],network[0]['W_hl'+str(i)].transpose()) + network[1]['b_hl'+str
        output = sigmoid(dotproduct) #
        #print(output)
        #print(output.shape)
        #print(output.transpose().shape)
        outputD['output_hl'+str(i)]=output #stores the output of each layer hidden layer

      #print("Output {}".format(outputD))
    #print("============================FEEDFORWARD============================")
```

Se ingresa un vector x y este es responsable de comenzar la propagación hacia delante, a través de todas las capas de la red, es importante recalcar que para realizar el producto punto entre la matriz de pesos y la entrada que recibe cada capa, se debe hacer la transpuesta de cada matriz de pesos, pues esto nos permite respetar la regla para realizar el producto punto donde el número de columnas de la matriz1 debe ser igual al número de filas de la matriz2, el resultado es ingresado a la función de activación y este almacenado en un vector de salida que posteriormente será usado para el cálculo de los errores en cada capa.

En este caso se debe entender que cada neurona está representada por cada fila de la matriz de pesos de su respectiva capa, de modo que cada columna de cada fila corresponde al peso conectado a esa neurona con las neuronas de la capa anterior.

Backpropagation

La función backpropagation nos permite calcular el error que cada capa tuvo hasta la capa de salida con respecto a los pesos que se tienen hasta el momento, derivado de la regla de la cadena, podemos obtener la fórmula que nos permite dar con el error de cada capa de la red neuronal, en este caso se realiza un bucle que recorra la última capa de la red(capa de salida) hasta la primer capa oculta, pues los errores de cada capa dependen del error de la capa posterior.

La función queda:

```
#Calculating errors of each layer
for i in reversed(range(n_layers)):
  if i==n_layers-1: #if its the output layer
    # Output layer error (L)
    L_error = outputLayer_Error(tar,network)
    network[3]['e_output'] = L_error
  elif i==n_layers-2: #if its the last hidden layer before output layer
    if network[3]['e_output'].size!=1: #if its not only one neuron
      dotproduct = np.matmul(network[3]['e_output'],network[0]['W_out'])
      l_error = dotproduct * network[2]['output_hl'+str(i)] * (1 - network[2]['output_hl'+str(i)])
      network[3]['e_hl'+str(i)] = l_error #stores the error of the i-hidden-layer
    else:
      l_error = (network[3]['e_output'] * network[0]['W_out']) * (network[2]['output_hl'+str(i)] * (1 - network[2]['output_hl'+str(i)]))
      network[3]['e_hl'+str(i)] = l_error #stores the error of the i-hidden-layer
  else:#if its any othe hidden layer
    if network[3]['e_hl'+str(i+1)].size!=1:
      dotproduct = np.matmul(network[3]['e_hl'+str(i+1)],network[0]['W_hl'+str(i+1)])
      l_error = dotproduct * (network[2]['output_hl'+str(i)] * (1 - network[2]['output_hl'+str(i)]))
      network[3]['e_hl'+str(i)] = l_error #stores the error of the i-hidden-layer
    else:
      l_error = (network[3]['e_hl'+str(i+1)] * network[0]['W_hl'+str(i+1)]) * (network[2]['output_hl'+str(i)] * (1 - network[2]['output_hl'+str(i)]))
      network[3]['e_hl'+str(i)] = l_error #stores the error of the i-hidden-layer
```

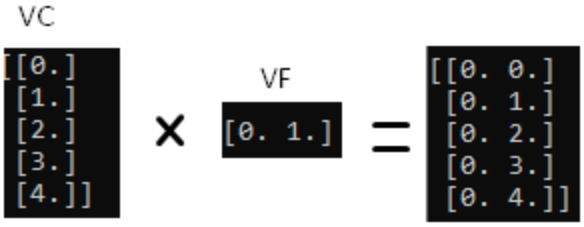
Update weights and bias

La función hace uso de los errores obtenidos en cada capa, y la salida de la capa que provoco ese error, todo esto obtenido gracias al principio del descenso del gradiente.

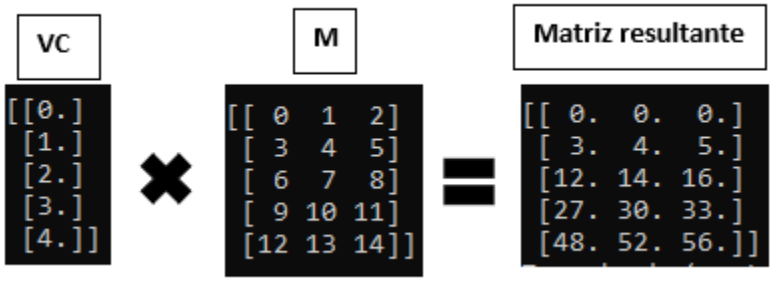
Una parte importante para destacar aquí es que se hizo uso de un concepto llamado “broadcasting” referido a la operación de vectores (sea suma, resta, multiplicación), debemos entender que estas operaciones entre matrices son elemento a elemento, lo cual exige que sea entre matrices del mismo tamaño, sin embargo, broadcasting nos evita el trabajo de siempre tener matrices del mismo tamaño.

Broadcasting tiene como concepto que estas operaciones podrán ser realizadas siempre en los siguientes casos:

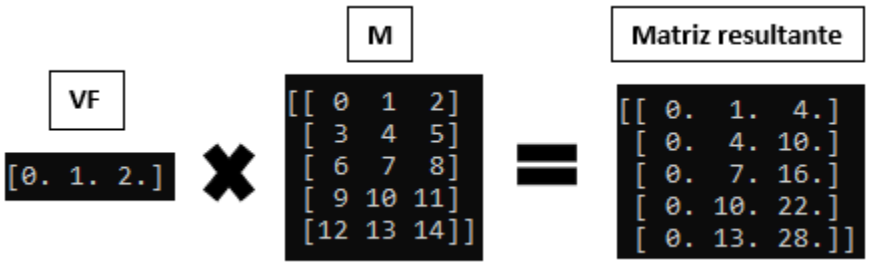
- Caso uno: un vector columna(VC) (n,1) es decir n filas y 1 columna que realiza una operación a un vector fila(VF) (1,m) 1 fila y m columnas, de modo que el vector resultante es una matriz de (n,m) dado que el vector VC se recorrió m veces a través del vector VF para cubrir las m componentes y el vector VF recorrió n veces para cubrir todas las componentes del VC.



- Caso dos: Realizar una operación aritmética entre Vector columna (VC) de tamaño (n,1) con una matriz(M) de tamaño (n,m), donde la matriz resultante de la operación será de tamaño (n,m).



- Caso tres: Realizar una operación aritmética entre Vector fila(VF) de tamaño (1,m) con una matriz(M) de tamaño (n,m), donde la matriz resultante de la operación será de tamaño (n,m).



Es importante recalcar que no importa el orden de los vectores o las matrices al realizar la operación, es decir:

VF * VC = VC * VF

VF * M = M * VF

VC * M = M * VC

Una vez aclarado esto, debemos analizar la fórmula del cálculo de pesos y bias.

- Para pesos la fórmula es la siguiente:

  n_weights = weights – alpha (error_layer * input_layer)

  Donde

  weights = pesos actuales

alpha = factor de aprendizaje o learning rate

error_layer = el error calculado en esa capa, calculado previamente

input_layer = el valor del vector de entrada que recibió esa capa en la etapa de feedforward.

Con esto tenemos una matriz de pesos de (m,n) con m = número de neuronas de la capa actual y n = número de neuronas de la capa anterior, por lo que para hacer la resta de ese matriz, con el resto de argumentos de la fórmula, tenemos que conseguir que en la multiplicación del vector **error_layer** por el vector **input_layer** consigamos una matriz de (m,n), para ello haremos uso del broadcasting.

Analizando, supongamos que tenemos estamos en una capa oculta cualquiera, y para ejemplificar suponemos que el tamaño de los vectores de error y entrada es:

shape_error_layer = (1,5)
shape_input_layer = (1,7)

Tenemos dos vectores fila, que nos indican o siguiente:

- El vector de error nos da el número de neuronas en esa capa, que es igual a 5.
- El vector de entrada nos da el número de neuronas en la capa anterior, que es igual a 7.
- Dados los dos valores anteriores (5 y 7) es fácil deducir que la matriz de pesos de esa capa tiene un tamaño de (5,7), por lo tanto, tenemos que usar esos vectores con ayuda de broadcasting para obtener una matriz de tamaño (5,7) y poder hacer la resta correspondiente a la matriz de pesos.

Dicho esto, notamos que para obtener dicha matriz, es necesario una operación que multiplique un vector columna por un vector fila por lo tanto, con el ejemplo del caso uno de broadcasting descrito anteriormente, notamos que es necesario que el vector de error sea transformado a un vector columna, para ello usamos la función de numpy reshape, de la forma:

np.reshape(error_layer,size(error_layer.shape[0],1))

De este modo indicamos que el vector de error tendrá ahora como filas el número de columnas que manejaba originalmente y ahora solo tendrá una columna, finalmente en la siguiente imagen podemos comprobarlo:

```
weights
[[ 0  1  2  3  4  5  6]
 [ 7  8  9 10 11 12 13]
 [14 15 16 17 18 19 20]
 [21 22 23 24 25 26 27]
 [28 29 30 31 32 33 34]]

Original error vector
[0. 1. 2. 3. 4.]

input vector
[0 1 2 3 4 5 6]

reshape error vector
[[0.]
 [1.]
 [2.]
 [3.]
 [4.]]

error_vector * input_vector
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  2.  3.  4.  5.  6.]
 [ 0.  2.  4.  6.  8. 10. 12.]
 [ 0.  3.  6.  9. 12. 15. 18.]
 [ 0.  4.  8. 12. 16. 20. 24.]]
```

Observamos que la matriz resultante de multiplicar el vector de errores con el vector de entradas coincide en tamaño con la matriz de pesos, por lo que hacer la resta de estos dos, ya es bastante sencillo.

Una forma analítica de ver esto es que nosotros necesitamos propagar el error que tuvimos de cada neurona(cada componente del vector) en cada componente de la entrada que hubo y de esta forma ver qué neuronas tuvieron más peso en el error obtenido y que estás tengan un mayor castigo en sus pesos(al restar esos valores de la matriz resultante a la matriz de pesos), por ejemplo, notemos que la primer fila(neurona) de la matriz resultante está llena de ceros, es decir esta neurona no se equivocó en lo absoluto, por ello al hacer la resta de esa matriz con la matriz de pesos, en la primer fila no habrá ningún cambio de pesos, pues esos son los indicados para que el error sea el mínimo.

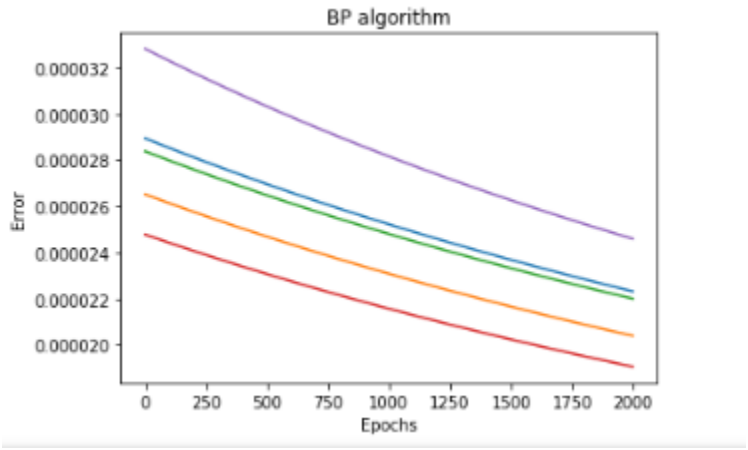
Finalmente, el código es el siguiente:

```
for i in reversed(range(n_layers)):
  if i==n_layers-1: #if its the output layer
    """
    print("Error layer {}".format(i))
    print("Error {}".format(network[3]['e_output']))
    print("previous output {}".format(network[2]['output_hl'+str(i-1)]))
    #n_W_out = network[0]['W_out'] - alpha * network[3]['e_output'] * network[2]['output_hl'+str(i-1)]
    #e_reshape = np.reshape(network[3]['e_output'],(network[3]['e_output'].size,1))
    #n_W_out = np.subtract(network[0]['W_out'] , (alpha * np.multiply(e_reshape , network[2]['output_hl'+str(i-1)])))
    """
    if network[3]['e_output'].size == 1: #If its only one gate to solve
        n_W_out = np.subtract(network[0]['W_out'] , (alpha * np.multiply(network[3]['e_output'] , network[2]['output_hl'+str(i-1)])))
        n_Weights['W_out'] = n_W_out
        n_b_out = network[1]['b_out'] - alpha * network[3]['e_output'] #calculates the new bias for the output layer
        n_Bias['b_out'] = n_b_out
    else:
        e_reshape = np.reshape(network[3]['e_output'],(network[3]['e_output'].size,1))
        n_W_out = np.subtract(network[0]['W_out'] , (alpha * np.multiply(e_reshape , network[2]['output_hl'+str(i-1)])))
        n_Weights['W_out'] = n_W_out
        n_b_out = network[1]['b_out'] - alpha * network[3]['e_output'] #calculates the new bias for the output layer
        n_Bias['b_out'] = n_b_out

  elif i!=0:#any other hidden layer
    """
    print("Error layer {}".format(i))
    print("Error {}".format(network[3]['e_hl'+str(i)]))
    print("previous output {}".format(network[2]['output_hl'+str(i-1)]))
    #n_W_hl = network[0]['W_hl'+str(i)] - (alpha * (network[3]['e_hl'+str(i)] * network[2]['output_hl'+str(i-1)])) #Actual we ights - a
    """
    e_reshape = np.reshape(network[3]['e_hl'+str(i)],(network[3]['e_hl'+str(i)].size,1))
    n_W_hl = np.subtract(network[0]['W_hl'+str(i)] , (alpha * np.multiply(e_reshape , network[2]['output_hl'+str(i-1)]))) #Ac tual weig
    n_Weights['W_hl'+str(i)] = n_W_hl
    n_b_hl = network[1]['b_hl'+str(i)] - (alpha * network[3]['e_hl'+str(i)])
    n_Bias['b_hl'+str(i)] = n_b_hl
  else:#If its the first hidden layer
    """
    print("Error layer {}".format(i))
    print("Error {}".format(network[3]['e_hl'+str(i)]))
    print("previous output {}".format(x))
    #n_W_hl = network[0]['W_hl'+str(i)] - (alpha * (network[3]['e_hl'+str(i)] * x)) #Actual weights - alpha *output_error * actual_inp
    """
    e_reshape = np.reshape(network[3]['e_hl'+str(i)],(network[3]['e_hl'+str(i)].size,1))
    n_W_hl = np.subtract(network[0]['W_hl'+str(i)] , (alpha * np.multiply(e_reshape , x))) #Actual weights - alpha *output_er ror * act
    n_Weights['W_hl'+str(i)] = n_W_hl
    n_b_hl = network[1]['b_hl'+str(i)] - (alpha * network[3]['e_hl'+str(i)]) #Actual bias - alpha * output_error
    n_Bias['b_hl'+str(i)] = n_b_hl

return(n_Weights,n_Bias)
```

Una vez que se tuvieron las funciones se procedió a realizar las pruebas de entrenamiento para clasificar 5 compuertas lógicas.

Usando una arquitectura de 5 capas ocultas con 5 neuronas cada una, un factor de aprendizaje de 0.7 y 3000 épocas de entrenamiento se obtuvo:

X	Target	Output
[0 0]	[0 0 1 1 0]	[0.0, 0.01, 1.0, 0.991, 0.007]
[0 1]	[0 1 1 0 1]	[0.007, 0.994, 0.993, 0.006, 0.993]
[1 0]	[0 1 1 0 1]	[0.007, 0.994, 0.993, 0.006, 0.993]
[1 1]	[1 1 0 0 0]	[0.991, 1.0, 0.009, 0.0, 0.008]



Con esto, podemos observar que la red converge bastante bien con la arquitectura, de modo que el error es muy pequeño.

5. Con la red programada en el paso 4 resolver el problema del Iris Dataset:

   a. Reportar la arquitectura utilizada, la exactitud de ésta y el número épocas que se necesitaron para resolver el problema.

   La arquitectura usada fue de 4 capas ocultas, con 4000 épocas y un Alpha de 0.2, además se hizo uso de solo los patrones de ancho del pétalo y el sépalo, es decir dos características, esto con la finalidad de elegir las mejores características que pueden hacer separable nuestro problema y finalmente se uso una codificación onehot definida como:

Codificación	**Clase**
[0,0,1]	Pertenece a clase 1
[0,1,0]	Pertenece a clase 2
[1,0,0]	Pertenece a clase 3

   A continuación, se muestra la tabla de los primeros 20 resultados por clase de la red una vez entrenada:

Clase 1

X	Target	Output
[3.5 0.2]	[0 0 1]	[0.001, 0.003, 0.997]
[3. 0.2]	[0 0 1]	[0.001, 0.003, 0.996]
[3.2 0.2]	[0 0 1]	[0.001, 0.003, 0.997]
[3.1 0.2]	[0 0 1]	[0.001, 0.003, 0.997]
[3.6 0.2]	[0 0 1]	[0.001, 0.002, 0.997]
[3.9 0.4]	[0 0 1]	[0.001, 0.003, 0.997]
[3.4 0.3]	[0 0 1]	[0.001, 0.003, 0.997]
[3.4 0.2]	[0 0 1]	[0.001, 0.003, 0.997]
[2.9 0.2]	[0 0 1]	[0.001, 0.003, 0.996]
[3.1 0.1]	[0 0 1]	[0.001, 0.003, 0.997]
[3.7 0.2]	[0 0 1]	[0.001, 0.002, 0.997]
[3.4 0.2]	[0 0 1]	[0.001, 0.003, 0.997]
[3. 0.1]	[0 0 1]	[0.001, 0.003, 0.997]
[3. 0.1]	[0 0 1]	[0.001, 0.003, 0.997]
[4. 0.2]	[0 0 1]	[0.001, 0.002, 0.997]
[4.4 0.4]	[0 0 1]	[0.001, 0.003, 0.997]
[3.9 0.4]	[0 0 1]	[0.001, 0.003, 0.997]
[3.5 0.3]	[0 0 1]	[0.001, 0.003, 0.997]
[3.8 0.3]	[0 0 1]	[0.001, 0.003, 0.997]
[3.8 0.3]	[0 0 1]	[0.001, 0.003, 0.997]

Clase 2

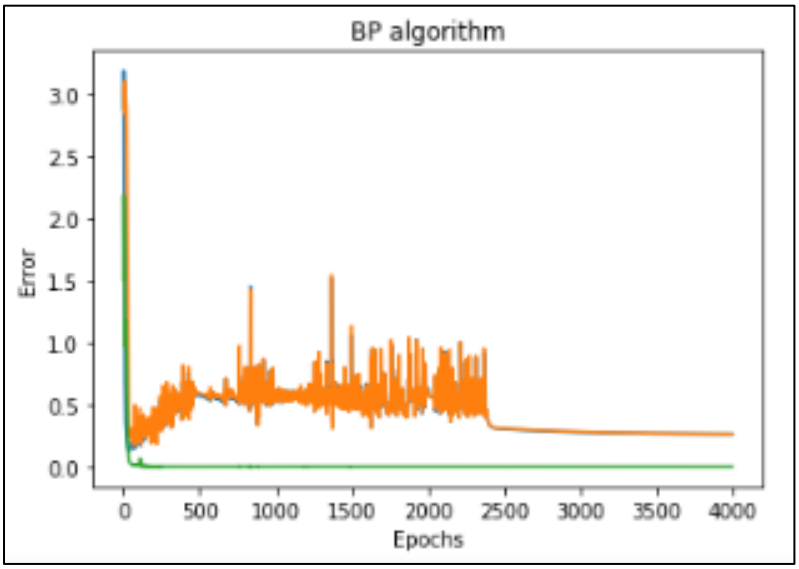
X	Target	Output
[3.2 1.4]	[0 1 0]	[0.005, 0.991, 0.004]
[3.2 1.5]	[0 1 0]	[0.006, 0.991, 0.004]
[3.1 1.5]	[0 1 0]	[0.007, 0.991, 0.004]
[2.3 1.3]	[0 1 0]	[0.028, 0.972, 0.0]
[2.8 1.5]	[0 1 0]	[0.8, 0.193, 0.0]
[2.8 1.3]	[0 1 0]	[0.006, 0.991, 0.004]
[3.3 1.6]	[0 1 0]	[0.008, 0.99, 0.004]
[2.4 1.]	[0 1 0]	[0.005, 0.991, 0.004]
[2.9 1.3]	[0 1 0]	[0.006, 0.991, 0.004]
[2.7 1.4]	[0 1 0]	[0.025, 0.968, 0.003]
[2. 1.]	[0 1 0]	[0.008, 0.992, 0.002]
[3. 1.5]	[0 1 0]	[0.009, 0.988, 0.004]
[2.2 1.]	[0 1 0]	[0.006, 0.992, 0.003]
[2.9 1.4]	[0 1 0]	[0.007, 0.991, 0.004]
[2.9 1.3]	[0 1 0]	[0.006, 0.991, 0.004]
[3.1 1.4]	[0 1 0]	[0.006, 0.991, 0.004]
[3. 1.5]	[0 1 0]	[0.009, 0.988, 0.004]
[2.7 1.]	[0 1 0]	[0.005, 0.975, 0.017]
[2.2 1.5]	[0 1 0]	[0.586, 0.411, 0.0]
[2.5 1.1]	[0 1 0]	[0.006, 0.992, 0.004]
[3.2 1.8]	[0 1 0]	[0.569, 0.443, 0.0]

Clase 3

X	Target	Output
[2.7 1.9]	[1 0 0]	[0.996, 0.003, 0.0]
[3. 2.1]	[1 0 0]	[0.997, 0.003, 0.0]
[2.9 1.8]	[1 0 0]	[0.996, 0.004, 0.0]
[3. 2.2]	[1 0 0]	[0.997, 0.003, 0.0]
[3. 2.1]	[1 0 0]	[0.997, 0.003, 0.0]
[2.5 1.7]	[1 0 0]	[0.993, 0.008, 0.0]
[2.9 1.8]	[1 0 0]	[0.996, 0.004, 0.0]
[2.5 1.8]	[1 0 0]	[0.995, 0.005, 0.0]
[3.6 2.5]	[1 0 0]	[0.997, 0.003, 0.0]
[3.2 2.]	[1 0 0]	[0.997, 0.003, 0.0]
[2.7 1.9]	[1 0 0]	[0.996, 0.003, 0.0]
[3. 2.1]	[1 0 0]	[0.997, 0.003, 0.0]
[2.5 2.]	[1 0 0]	[0.996, 0.004, 0.0]
[2.8 2.4]	[1 0 0]	[0.997, 0.003, 0.0]
[3.2 2.3]	[1 0 0]	[0.997, 0.003, 0.0]
[3. 1.8]	[1 0 0]	[0.995, 0.005, 0.0]
[3.8 2.2]	[1 0 0]	[0.997, 0.003, 0.0]
[2.6 2.3]	[1 0 0]	[0.996, 0.003, 0.0]
[2.2 1.5]	[1 0 0]	[0.586, 0.411, 0.0]
[3.2 2.3]	[1 0 0]	[0.997, 0.003, 0.0]
[2.8 2.]	[1 0 0]	[0.996, 0.003, 0.0]
[2.8 2.]	[1 0 0]	[0.996, 0.003, 0.0]
[2.7 1.8]	[1 0 0]	[0.996, 0.004, 0.0]
[3.3 2.1]	[1 0 0]	[0.997, 0.003, 0.0]
[3.2 1.8]	[1 0 0]	[0.569, 0.443, 0.0]

De los primeros 20 elementos clasificados de cada clase podemos notar algunos errores de clasificación entre la clase 2 y la clase 3

b. Presentar una gráfica de errores por épocas.



Con la gráfica nos aseguramos por completo de como los errores de la clase 2 y 3 tardaron mucho más en converger e incluso daban saltos bruscos, aún con un alpha de 0.2.

c. Resolver el problema utilizando 5 capas ocultas con 10 neuronas cada capa y reportar lo mismo que en a) y en b).

El problema fue resuelto en 4000 épocas con un alpha de 0.2

A continuación, se muestra la tabla de los primeros 20 resultados de cada clase una vez entrenada la red:

Clase 1

X	Target	Output
[3.5 0.2]	[0 0 1]	[0.002, 0.0, 0.999]
[3. 0.2]	[0 0 1]	[0.002, 0.0, 0.998]
[3.2 0.2]	[0 0 1]	[0.002, 0.0, 0.998]
[3.1 0.2]	[0 0 1]	[0.002, 0.0, 0.998]
[3.6 0.2]	[0 0 1]	[0.002, 0.0, 0.999]
[3.9 0.4]	[0 0 1]	[0.002, 0.0, 0.999]
[3.4 0.3]	[0 0 1]	[0.002, 0.0, 0.998]
[3.4 0.2]	[0 0 1]	[0.002, 0.0, 0.999]
[2.9 0.2]	[0 0 1]	[0.002, 0.0, 0.998]
[3.1 0.1]	[0 0 1]	[0.002, 0.0, 0.998]
[3.7 0.2]	[0 0 1]	[0.002, 0.0, 0.999]
[3.4 0.2]	[0 0 1]	[0.002, 0.0, 0.999]
[3. 0.1]	[0 0 1]	[0.002, 0.0, 0.998]
[3. 0.1]	[0 0 1]	[0.002, 0.0, 0.998]
[4. 0.2]	[0 0 1]	[0.002, 0.0, 0.999]
[4.4 0.4]	[0 0 1]	[0.002, 0.0, 0.999]
[3.9 0.4]	[0 0 1]	[0.002, 0.0, 0.999]
[3.5 0.3]	[0 0 1]	[0.002, 0.0, 0.999]
[3.8 0.3]	[0 0 1]	[0.002, 0.0, 0.999]
[3.8 0.3]	[0 0 1]	[0.002, 0.0, 0.999]

Clase 2

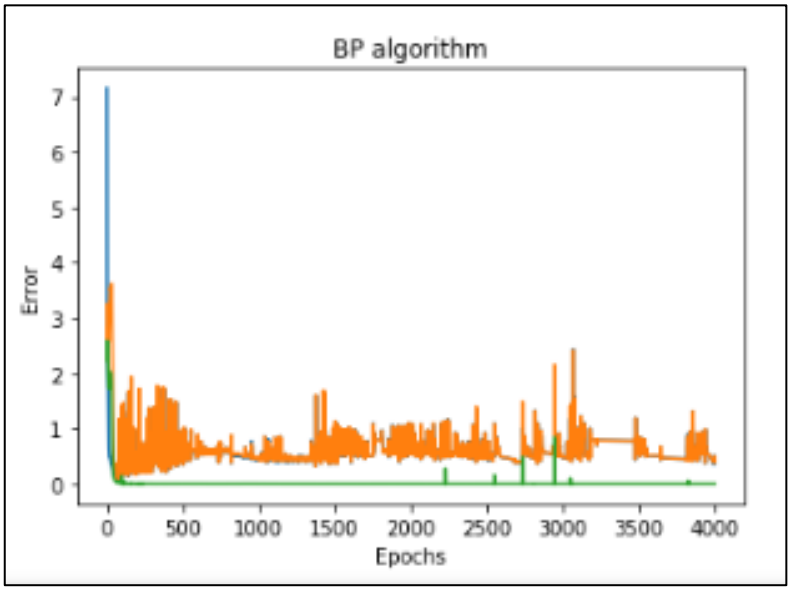
X	Target	Output
[3.2 1.4]	[0 1 0]	[0.002, 0.998, 0.0]
[3.2 1.5]	[0 1 0]	[0.002, 0.998, 0.0]
[3.1 1.5]	[0 1 0]	[0.003, 0.998, 0.0]
[2.3 1.3]	[0 1 0]	[0.394, 0.594, 0.0]
[2.8 1.5]	[0 1 0]	[0.65, 0.41, 0.0]
[2.8 1.3]	[0 1 0]	[0.002, 0.998, 0.0]
[3.3 1.6]	[0 1 0]	[0.003, 0.997, 0.0]
[2.4 1.]	[0 1 0]	[0.002, 0.998, 0.0]
[2.9 1.3]	[0 1 0]	[0.002, 0.998, 0.0]
[2.7 1.4]	[0 1 0]	[0.313, 0.79, 0.0]
[2. 1.]	[0 1 0]	[0.021, 0.991, 0.0]
[3. 1.5]	[0 1 0]	[0.008, 0.996, 0.0]
[2.2 1.]	[0 1 0]	[0.003, 0.998, 0.0]
[2.9 1.4]	[0 1 0]	[0.003, 0.998, 0.0]
[2.9 1.3]	[0 1 0]	[0.002, 0.998, 0.0]
[3.1 1.4]	[0 1 0]	[0.002, 0.998, 0.0]
[3. 1.5]	[0 1 0]	[0.008, 0.996, 0.0]
[2.7 1.]	[0 1 0]	[0.002, 0.998, 0.0]
[2.2 1.5]	[0 1 0]	[0.85, 0.148, 0.0]
[2.5 1.1]	[0 1 0]	[0.002, 0.998, 0.0]

Clase 3

X	Target	Output
[3.3 2.5]	[1 0 0]	[0.992, 0.008, 0.002]
[2.7 1.9]	[1 0 0]	[0.992, 0.008, 0.002]
[3. 2.1]	[1 0 0]	[0.992, 0.008, 0.002]
[2.9 1.8]	[1 0 0]	[0.991, 0.008, 0.002]
[3. 2.2]	[1 0 0]	[0.992, 0.008, 0.002]
[3. 2.1]	[1 0 0]	[0.992, 0.008, 0.002]
[2.5 1.7]	[1 0 0]	[0.992, 0.008, 0.002]
[2.9 1.8]	[1 0 0]	[0.991, 0.008, 0.002]
[2.5 1.8]	[1 0 0]	[0.992, 0.008, 0.002]
[3.6 2.5]	[1 0 0]	[0.992, 0.008, 0.002]
[3.2 2.]	[1 0 0]	[0.992, 0.008, 0.002]
[2.7 1.9]	[1 0 0]	[0.992, 0.008, 0.002]
[3. 2.1]	[1 0 0]	[0.992, 0.008, 0.002]
[2.5 2.]	[1 0 0]	[0.992, 0.008, 0.002]
[2.8 2.4]	[1 0 0]	[0.992, 0.008, 0.002]
[3.2 2.3]	[1 0 0]	[0.992, 0.008, 0.002]
[3. 1.8]	[1 0 0]	[0.991, 0.008, 0.002]
[3.8 2.2]	[1 0 0]	[0.992, 0.008, 0.002]
[2.6 2.3]	[1 0 0]	[0.992, 0.008, 0.002]
[2.2 1.5]	[1 0 0]	[0.85, 0.148, 0.0]

Analizando las tablas podemos notar que aún existe cierto margen de clasificación, pero ahora solo en algunos de los primeros 20 elementos de la clase 2.

La gráfica de entrenamiento es la siguiente:



d. Reportar que sucede con los errores de las capas ocultas cuando utilizando la arquitectura en c) y por qué.

En varios vectores de errores se puede notar saltos bastante drásticos que suceden de un patrón de entrenamiento a otro, esto seguramente se debe a que dichos patrones son los más difíciles de clasificar entre las tres clases, es decir, tienen características que muy bien pueden ser confundidas entre las tres clases, es por ese motivo que los errores al bajar para clasificar uno de esos para una clase, suben al querer clasificar otro que se parezca bastante.