```
#!/usr/bin/env python
# coding: utf-8

# In[12]:


from random import random,seed,randint
from pprint import pprint
def initialize(n_inputs,n_hidden,n_output):
    network=[]
    hidden_layer=[{'w':[random() for i in range(n_inputs+1)]} for i in
range(n_hidden)] #Initialize Weights and Biases for hidden layers
    network.append(hidden_layer)
    output_layer=[{'w':[random() for i in range(n_hidden+1)]} for i in
range(n_output)] #Initialize Weights and Biases for output layer
    network.append(output_layer)
    return network
def activate(w,i):
    activation=w[-1] #Bias value is -1
    for x in range(len(w)-1):
        activation+=w[x]*i[x] #WX similar to WiXi + Bias ie
activation=activation + Wixi
        return activation #WX+B
from math import exp
def sigmoid(a):
    return 1/(1+exp(-a))
def forward_prop(network,row):
    inputs=row
    for layer in network:
        new_inputs=[]
        for neuron in layer:
            activation=activate(neuron['w'],inputs) #Compute
Activations
            neuron['output']=sigmoid(activation) #Compute Sigmoid
            new_inputs.append(neuron['output']) #Adds it to the output
layer
        inputs=new_inputs #new_inputs values now becomes the input
    return inputs
def sigmoid_derivative(output):
    return output * (1-output) #Derivative of 1/(1+e^-x)
def backprop(network,expected): #expected is our expected output value
we'd use to compute the error
    for i in reversed(range(len(network))): #Prints the list ie
"Network" in reversed order
        layer=network[i] #network contains what? see below
        errors=[] #initialize error values to an empty list
        if i!=len(network)-1: #Output Layer
            for j in range(len(layer)):
                error=0 #Assign error values to 0
                for neuron in network[i+1]:
                    error+=(neuron['w'][j]*neuron['delta'])#Calculates
and Updates the error
                errors.append(error)
        else:
                for j in range(len(layer)):
                    neuron=layer[j]
```

```
                    errors.append(expected[j]-neuron['output'])
#Calculates and appends the errors
        for j in range(len(layer)):
                neuron=layer[j]

neuron['delta']=errors[j]*sigmoid_derivative(neuron['output'])
#Compute Gradients
def update_weights(network,row,lrate): #Gradient Descent
    for i in range(len(network)):
        inputs=row[:-1] #Takes all except last row
        if i!=0:
            inputs=[neuron['output'] for neuron in network[i-1]]
            for neuron in network[i]:
                for j in range(len(inputs)):
                    neuron['w'][j]+=lrate*neuron['delta']*inputs[j]
#Weights update similar to w5 + n*Edy*xi
                    neuron['w'][-1]+=lrate*neuron['delta'] #Bias is -1
and its updated
def train_network(network,train,lrate,epochs,n_output):
    for epoch in range(epochs):
        sum_err=0
        for row in train:
            outputs=forward_prop(network,row)
            expected=[0 for i in range(n_output)]
            expected[row[-1]]=1
            sum_err+=sum([(expected[i]-outputs[i])**2 for i in
range(len(expected))])#Computes the error
            backprop(network,expected)#Calls backpropagation
            update_weights(network,row,lrate)#Finally weights are
updated
        print('epoch=%d, lrate=%.3f,error=%.3f'%(epoch,lrate,sum_err))
seed(1)
data=[[2.7810836,2.550537003,0],
      [1.465489372,2.362125076,0],
      [3.396561688,4.400293529,0],
      [1.38807019,1.850220317,0],
      [3.06407232,3.005305973,0],
      [7.627531214,2.759262235,1],
      [5.332441248,2.088626775,1],
      [6.922596716,1.77106367,1],
      [8.675418651,-0.242068655,1],
      [7.673756466,3.508563011,1]] #This dataset contain 2 input inits
and 1 output unit
n_inputs=len(data[0])-1
n_outputs=len(set(row[-1] for row in data))
network=initialize(n_inputs,2,n_outputs)
pprint(network)
train_network(network,data,0.5,20,n_outputs)
for layer in network:
    pprint(layer)


# In[ ]:
```