```python
############################################
# Authors: Tanner Mengel and Ian Cox
# Date: 2/4/2022
# Project 1 - Artificial Neural Networks
# Description: This file contains the code for 3 classes
# Neuron, Layer, and NeuralNetwork. The Neuron class is used to create a
 single neuron in a Fully Connected layer of a neural network.
# The Layer class is used to create a single layer of a neural network.
# The NeuralNetwork class is used to create a neural network with multiple
 layers.
# The NeuralNetwork class is used to train and test the neural network.
# Main function is at the bottom of the file.
# To run the code, run the following command in the terminal:
 # python3 ANN.py [learning_rate] [example]
 # learning_rate: learning rate for the neural network
 # example: 'example', 'and' or 'xor'
############################################
# Imports
import numpy as np
import sys

class Neuron:
    '''
    Class for a single neuron in a neural network
    '''
    def __init__(self, input_dim, activation, learning_rate=0.1, weights=None,
     verbose=False):
        # Initialize the neuron with the following parameters
        self.input_dim = input_dim # Number of inputs plus 1 for bias (set
         eariler)
        self.activation = activation  # Activation function
        # Activation function can be either 'linear' or 'logistic'
        if activation not in ['linear', 'logistic']:
            raise ValueError(f'Activation function {activation} not supported')

        self.learning_rate = learning_rate # Learning rate
        # Weights are initialized randomly if not provided
        if weights is None: self.weights = np.random.rand(input_dim)
        else: self.weights = weights
        if self.weights.shape[0] != input_dim:
            raise ValueError(f'Input dim does not match the dim of weights,
             input dim: {input_dim}, weights dim: {self.weights.shape[0]}')
        # Initialize the output and inputs to None
        self.output = None
        self.inputs = []
        # Initialize the partial derivatives to None
        self.dW = []

        if verbose: # Print the parameters if verbose is True
            print('Initialized Neuron with the following architecture:')
            print(f'Input Dim: {input_dim}, Activation Function: {activation}')
            print(f'Learning Rate: {learning_rate}, Weights: {weights}')
            print(f'Weights Shape: {weights.shape}')

    # This method returns the output of the neuron given the input
    def active(self, input): # Activation function
        if self.activation == 'linear': return input
        elif self.activation == 'logistic': return 1/(1+np.exp(-input))
        else: raise ValueError(f'Activation function {self.activation} not
         supported')

    # This method calculates the output of the neuron given the input
    def calculate(self, X):
        self.inputs = X
        input = np.dot(self.weights, X) # Calculate the dot product of the
         weights and the inputs
        output = self.active(input) # Calculate the output of the neuron after
         applying the activation function
        self.output = output
        return output # Return the output of the neuron

    #This method returns the derivative of the activation function with
     respect to the net
    def activationderivative(self):
        if self.activation == "linear": return 1
        elif self.activation == "logistic": return self.output*(1-self.output)
        else : raise ValueError("Activation function not found")

    #This method calculates the partial derivative for each weight and returns
     the delta*w to be used in the previous layer
    def calcpartialderivative(self, wtimesdelta):
        # print('calcpartialderivative')
        delta = wtimesdelta*self.activationderivative()
        self.dW = self.inputs*delta #Calculate the partial derivative for each
         weight
        return delta*self.weights

    #Simply update the weights using the partial derivatives and the leranring
     weight
    def updateweight(self):
        for i in range(len(self.weights)): #Update the weights
            self.weights[i] -= self.learning_rate*self.dW[i]


class Layer:
    '''
    Class for a single layer in a neural network to manage the neurons in the
     layer
    '''
    def __init__(self, neuron_num, activation, input_num, learning_rate=0.1,
     weights=None, verbose=False):
        # Initialize the layer with the following parameters
        self.neuron_num = neuron_num   # Number of neurons in the layer
        self.activation = activation  # Activation function
        # Activation function can be either 'linear' or 'logistic'
        self.learning_rate = learning_rate # Learning rate
        self.input_num = input_num # Number of inputs to the layer (including
         bias)
        # Weights are initialized randomly if not provided
        if weights is None:
            weights = []
            for i in range(neuron_num):
                weights.append(np.random.rand(input_num)) # Initialize the
                 weights randomly
        self.weights = weights
        # Initialize the neurons in the layer
        self.neurons = []
        for i in range(neuron_num):
            weight = weights[i]
            inputdim = weight.shape[0]
            if inputdim != input_num:# Check if the input dimension matches
             the dimension of the weights
                raise ValueError(f'Input dim does not match the dim of
                 weights, input dim: {inputdim}, weights dim: {input_num}')
            self.neurons.append(Neuron(input_dim=inputdim,
             activation=activation, learning_rate=learning_rate,
             weights=weight))
        # Initialize the output and inputs to None
        self.outputs = []
        if verbose: # Print the parameters if verbose is True
            print('Initialized Layer with the following architecture:')
            print(f'Neuron Number: {neuron_num}, Activation Function:
             {activation}')
            print(f'Input Number: {input_num}, Learning Rate: {learning_rate}')
            print(f'Weights: {weights}')
            print(f'Weights Shape: {weights.shape}')
    # This method calculates the output of the layer given the input
    def calculate(self, X):
        input = X
        output = []
        for neuron in self.neurons:
            output.append(neuron.calculate(input)) # Calculate the output of
             each neuron in the layer
        return output
    #This method calculates the partial derivative for each weight and returns
     the delta*w to be used in the previous layer
    def calcwdeltas(self, dloss):
        #given the next layer's w*delta, should run through the neurons
         calling calcpartialderivative() for each (with the correct value),
         sum up its own w*delta, and then update the weights (using the
         updateweight() method). I should return the sum of w*delta.
        deltaw = np.zeros(self.input_num)
        for i in range(self.neuron_num):
            deltaw += self.neurons[i].calcpartialderivative(dloss[i])
             #Calculate the partial derivative sum for each weight
            self.neurons[i].updateweight()    #Update the weights
        return deltaw #Return the sum of w*delta


class NeuralNetwork:
    '''
    Class for a neural network to manage the layers and the training
    '''

    def __init__(self, num_of_layers, num_of_neurons, input_size, activation,
     loss, output_size=1, learning_rate=0.1, weights=None,verbose=False):
        # Initialize the neural network with the following parameters
        self.num_of_layers = num_of_layers # Number of layers in the neural
         network
        self.num_of_neurons = num_of_neurons  # Number of neurons in each layer
        # Number of neurons in each layer can be a list of integers
        if type(num_of_neurons) != list:
            raise ValueError(f'Number of neurons must be a list, not
             {type(num_of_neurons)}')
        # Number of neurons in each layer must match the number of layers
        if len(num_of_neurons) != num_of_layers:
            raise ValueError(f'Number of neurons does not match the number of
             layers, neurons: {len(num_of_neurons)}, layers: {num_of_layers}')

        self.input_size = input_size # Number of inputs to the neural network
        self.activation = activation # Activation function
        # Activation function can be either 'linear' or 'logistic'
        if type(activation) != list: # If activation function is not a list,
         use the same activation function for all layers
            print('Constant activation function for all layers')
            self.activation = []
            for i in range(num_of_layers):
                self.activation.append(activation)
            self.activation.append(activation)
        if len(self.activation) != num_of_layers + 1: # If activation function
         is a list, check if the number of activation functions matches the
         number of layers
            if len(self.activation) == num_of_layers: # If the number of
             activation functions is one less than the number of layers, use
             the last activation function for the output layer
                print("Output layer activation function not specified, using
                 previous activation function")
                self.activation.append(activation[-1])
            else:# If the number of activation functions does not match the
             number of layers, raise an error
                print("Activation function not specified for all layers, using
                 previous activation function")
                for i in range(num_of_layers - len(self.activation)):
                    self.activation.append(activation[-1])
        self.loss = loss # Loss function
        if loss not in ['mse', 'bce']: # Check if the loss function is
         supported
            raise ValueError(f'Loss function {loss} not supported')

        self.output_size = output_size # Number of outputs from the neural
         network
        self.learning_rate = learning_rate # Learning rate
        self.architecture = [input_size] + num_of_neurons + [output_size] #
         Architecture of the neural network
        # Weights are initialized randomly if not provided
        if weights is None:
            weights = []
            for i in range(len(self.architecture) - 1):
                weights.append(np.random.rand(self.architecture[i+1],
                 self.architecture[i]+1))   # Initialize the weights randomly
        self.weights = weights
        # Initialize the layers in the neural network
        self.layers = []
        for i in range(num_of_layers):
            weight = weights[i]
            neurons = weight.shape[0]# Number of neurons in the layer
            inputdim = weight.shape[1]# Number of inputs to the layer
            if neurons != num_of_neurons[i]:# Check if the number of neurons
             in the layer matches the number of neurons in the weights
                raise ValueError(f'Number of neurons in layer {i} does not
                 match the dim of weights, neurons: {neurons}, weights dim:
                 {num_of_neurons[i]}')
            self.layers.append(Layer(neuron_num=neurons,
             activation=activation[i], input_num=inputdim,
             learning_rate=learning_rate, weights=weight, verbose=verbose))
        # Initialize the output layer
        self.layers.append(Layer(neuron_num=weights[-1].shape[0],
         activation=activation[-1], input_num=weights[-1].shape[1],
         learning_rate=learning_rate, weights=weights[-1], verbose=verbose))
        # Initialize the outputs of the neural network
        self.outputs = []

        if verbose: # Print the architecture of the neural network
            print('Initialized Neural Network with the following
             architecture:')
            print(f'Input Size: {input_size}, Output Size: {output_size},
             Hidden Layers: {num_of_layers}')
            print(f'Neurons per Hidden Layer: {num_of_neurons}')
            print(f'Activation Function: {activation}, Loss Function: {loss}')
            print(f'Learning Rate: {learning_rate}')
            print(f'Weights: {weights}')
            for weight in self.weights:
                print(weight.shape)
    # Calculate the output of the neural network
    def calculate(self,X):
        input = X
        for layer in self.layers:
            input = np.append(input, 1) # Append a 1 to the input to account
             for the bias
            output = layer.calculate(input)# Calculate the output of the layer
            input = output
        return output   # Return the output of the neural network
    # Calculate the loss of the neural network with respect to the ground truth
    def calculateloss(self,yhat,y):
        if self.loss == 'mse':  # Calculate the mean squared error
            loss = 0
            for i in range(len(yhat)):
                loss += (yhat[i] - y[i])**2
            loss = loss/len(yhat) # Calculate the mean
            return loss
        elif self.loss == 'bce':   # Calculate the binary cross entropy
            loss = 0
            for i in range(len(yhat)):
                loss += -y[i]*np.log(yhat[i]) - (1-y[i])*np.log(1-yhat[i])
            loss = loss/len(yhat) # Calculate the mean
            return loss
        else : raise ValueError(f'Loss function {self.loss} not supported') #
         Raise an error if the loss function is not supported

    # Calculate the derivative of the loss function with respect to the output
     of the neural network
    def lossderiv(self,yhat,y):
        if len(yhat)==1:# If the output is a scalar, return a list
            if self.loss == 'mse':  # Calculate the derivative of the mean
             squared error
                dloss = 0
                for i in range(len(yhat)):
                    dloss = 2*(yhat[i] - y[i])
                dloss = dloss/len(yhat) # Calculate the mean
                return [dloss]
            elif self.loss == 'bce': # Calculate the derivative of the binary
             cross entropy
                dloss = 0
                for i in range(len(yhat)):
                    dloss = -y[i]/yhat[i] + (1-y[i])/(1-yhat[i])
                dloss = dloss/len(yhat) # Calculate the mean
                return [dloss]
            else : raise ValueError(f'Loss function {self.loss} not
             supported') # Raise an error if the loss function is not supported
        else: # If the output is a vector, return a vector
            dloss = []# Initialize the derivative of the loss
            if self.loss == 'mse': # Calculate the derivative of the mean
             squared error
                for i in range(len(yhat)):
```

```
                dloss.append(2*(yhat[i] - y[i]))
            dloss = np.array(dloss)/len(yhat)
            return dloss
        elif self.loss == 'bce': # Calculate the derivative of the binary
         cross entropy
            for i in range(len(yhat)):
                dloss.append(-y[i]/yhat[i] + (1-y[i])/(1-yhat[i]))
            dloss = np.array(dloss)/len(yhat)
            return dloss
        else : raise ValueError(f'Loss function {self.loss} not supported')

    # Train the neural network with the given inputs and outputs
    def train(self, X, Y, epochs=1, verbose=False):

        for epoch in range(epochs): # Iterate through the epochs
            loss = 0 # Initialize the loss
            for i in range(len(X)): # Iterate through the training examples
                yhat = self.calculate(X[i])     # Calculate the output of the
                 neural network
                dloss = self.lossderiv(yhat, Y[i]) # Calculate the derivative
                 of the loss function with respect to the output of the neural
                 network
                for j in range(len(self.layers)-1, -1, -1): # Iterate through
                 the layers in reverse order
                    dloss = self.layers[j].calcwdeltas(dloss)  # Calculate the
                     derivative of the loss function with respect to the
                     weights of the layer
                yhat = [] # Initialize the output of the neural network
                ytest = [] # Initialize the ground truth
                for i in range(len(X)): # Iterate through the training examples
                    yhat.append(self.calculate(X[i])) # Calculate the output
                     of the neural network
                    ytest.append(Y[i]) # Calculate the ground truth
                loss += self.calculateloss(yhat[i], Y[i])/len(X)
            if epoch % 10000 == 0: # Print the loss every 10000 epochs
                print(f'Epoch: {epoch}, Loss: {loss}') # Print the loss


if __name__=="__main__": # Run the main function
    learningRate = float(sys.argv[1])
    print('Using a learning rate of',learningRate)
    if (len(sys.argv)<3):
        print('a good place to test different parts of your code')

    elif (sys.argv[2]=='example'):
        numEpochs = 1
        print('run example from class (single step)')
        w=np.array([[[.15,.2,.35],[.25,.3,.35]],[[.4,.45,.6],[.5,.55,.6]]])
        x =np.array([[0.05,0.1]])
        y = np.array([[0.01,0.99]])
        example =
         NeuralNetwork(1,[2],len(x),
         ["logistic","logistic"],"mse",len(y),learningRate,w)
        example.train(x,y,numEpochs,True)
        print(example.weights)

    elif(sys.argv[2]=='and'):
        test = NeuralNetwork(num_of_layers=0, num_of_neurons=[], input_size=2,
         activation=['linear', 'logistic'], loss='bce', output_size=1,
         learning_rate=learningRate)
        x = np.array([[0,0],[0,1],[1,0],[1,1]])
        y = np.array([[0],[0],[0],[1]])
        print('training network')
        print(test.architecture)
        test.train(x,y,epochs=1,verbose=True)
        yhat = []
        for i in range(len(x)):
            yhat.append(*test.calculate(x[i]))
        print("And")
        for i in range(len(x)):
            print(f'X: {x[i]}, Y: {y[i]}, Yhat: {yhat[i]}')
        print('Total Loss: ', test.calculateloss(yhat, y))

    elif(sys.argv[2]=='xor'):
        test = NeuralNetwork(num_of_layers=1, num_of_neurons=[15],
         input_size=2, activation=['logistic', 'logistic'], loss='bce',
         output_size=1, learning_rate=0.1)
        x = np.array([[0,0],[1,0],[0,1],[1,1]])
        y = np.array([[0],[1],[1],[0]])
        print('training network')
        test.train(x,y,epochs=100000,verbose=True)
        yhat = []
        for i in range(len(x)):
            yhat.append(*test.calculate(x[i]))
        print("XOR")
        for i in range(len(x)):
            print(f'X: {x[i]}, Y: {y[i]}, Yhat: {yhat[i]}')
        print('Total Loss: ', test.calculateloss(yhat, y))

        print("Training single perceptron")
        test = NeuralNetwork(num_of_layers=0, num_of_neurons=[], input_size=2,
         activation=['logistic'], loss='bce', output_size=1, learning_rate=0.3)
        test.train(x,y,epochs=100000,verbose=True)
        yhat = []
        for i in range(len(x)):
            yhat.append(*test.calculate(x[i]))
        print("XOR Single Perceptron")
        for i in range(len(x)):
            print(f'X: {x[i]}, Y: {y[i]}, Yhat: {yhat[i]}')
        print('Total Loss: ', test.calculateloss(yhat, y))
```