```
import numpy as np
import sys
"""
For this entire file there are a few constants:
activation:
0 - linear
1 - logistic (only one supported)
loss:
0 - sum of square errors
1 - binary cross entropy
"""


# A class which represents a single neuron
class Neuron:
    #initilize neuron with activation type, number of inputs, learning rate, and
possibly with
    #set weights
    def __init__(self,activation, input_num, lr, weights=None):
        self.activation = activation
        self.input_num = input_num
        self.lr = lr
        self.weights = weights
        if weights is None:
            self.weights = np.random.rand(input_num+1)

    #This method returns the activation of the net
    def activate(self,net):
        if self.activation == 0:
            # Linear activation function
            return net
        else:
            # Logistic activation function
            return 1/(1+np.exp(-net))

    #Calculate the output of the neuron should save the input and output for
back-propagation.
    def calculate(self,input):
        self.input = np.append(input, 1)
        self.net = np.sum(self.input*self.weights)
        self.output = self.activate(self.net)
        return self.output

    #This method returns the derivative of the activation function with respect to
the net
    def activationderivative(self):
        if self.activation == 0:
            # Derivative of linear activation function
            return 1
        else:
```

```
            # Derivative of logistic activation function
            return self.output * (1 - self.output)

    #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):
        self.delta = wtimesdelta * self.activationderivative()
        return self.weights * self.delta

    #Simply update the weights using the partial derivatives and the leranring
weight
    def updateweight(self):
        self.weights -= self.lr * self.delta * self.input


#A fully connected layer
class FullyConnected:
    #initialize with the number of neurons in the layer, their activation,the input
size, the
    #leraning rate and a 2d matrix of weights (or else initilize randomly)
    def __init__(self,numOfNeurons, activation, input_num, lr, weights=None):
        self.numOfNeurons = numOfNeurons
        self.activation = activation
        self.input_num = input_num
        self.lr = lr
        self.weights = weights

        # weights is a 2D matrix; first index is neuron, second index is weight

        # Init weights randomly if necessary
        if weights is None:
            self.weights = np.random.rand(numOfNeurons, input_num + 1)
            bias = np.random.rand(1)
            self.weights[:,-1] = bias[0]

        # Create neuron objects in numpy array of all neurons in layer
        self.neurons = np.array([Neuron(activation, input_num, lr,
self.weights[neuronInd]) for neuronInd in range(numOfNeurons)])

    #calcualte the output of all the neurons in the layer and return a vector with
those values
    #(go through the neurons and call the calcualte() method)
    def calculate(self, input):
        return np.array([neuron.calculate(input) for neuron in self.neurons])


    #given the next layer's w*delta, should run through the neurons calling
    #calcpartialderivative() for each (with the correct value), sum up its
ownw*delta, and then
```

```
    #update the wieghts (using the updateweight() method). I should return the sum
of w*delta.
    def calcwdeltas(self, wtimesdelta):
        wtimesdeltaSum = 0
        for neuronInd, neuron in enumerate(self.neurons):
            wtimesdeltaSum += neuron.calcpartialderivative(wtimesdelta[neuronInd])
            neuron.updateweight()

        return wtimesdeltaSum


#An entire neural network
class NeuralNetwork:
    #initialize with the number of layers, number of neurons in each layer (vector),
input size,
    #activation (for each layer), the loss function, the learning rate and a 3d
matrix of weights
    #weights (or else initialize randomly)
    def __init__(self,numOfLayers,numOfNeurons, inputSize, activation, loss, lr,
weights=None):
        self.numOfLayers = numOfLayers
        self.numOfNeurons = numOfNeurons
        self.inputSize = inputSize
        self.activation = activation
        self.loss = loss
        self.lr = lr
        self.weights = weights

        # weights is a 3D matrix; first index is layer, second index is neuron,
third index is weight

        # Init weights randomly if necessary
        if weights is None:
            maxNumOfNeuronsInLayer = np.amax(numOfNeurons)
            maxNumOfWeightsPerNeuron = max(inputSize, maxNumOfNeuronsInLayer) + 1
            self.weights = np.random.rand(numOfLayers, maxNumOfNeuronsInLayer,
maxNumOfWeightsPerNeuron)

            # Set consistent bias values and pad unused weights with zeros
            biases = np.random.rand(numOfLayers)
            for layerInd, layerWeights in enumerate(self.weights):
                numOfNeuronsInLayer = numOfNeurons[layerInd]
                if layerInd != 0:
                    numOfNeuronsInPrevLayer = numOfNeurons[layerInd - 1]
                else:
                    numOfNeuronsInPrevLayer = inputSize
                layerWeights[:numOfNeuronsInLayer, numOfNeuronsInPrevLayer+1:] = 0
                layerWeights[:numOfNeuronsInLayer, numOfNeuronsInPrevLayer] =
biases[layerInd]
                layerWeights[numOfNeuronsInLayer:, :] = 0
```

```
        # Create FullyConnected layer objects in numpy array of all layers of
neurons
        self.layers = np.array([FullyConnected(numOfNeurons[layerInd], activation,
numOfNeurons[layerInd-1] if layerInd != 0 else inputSize, lr,
self.weights[layerInd][:numOfNeurons[layerInd], :numOfNeurons[layerInd-1]+1] if
layerInd != 0 else self.weights[layerInd][:numOfNeurons[layerInd], :inputSize+1])
for layerInd in range(numOfLayers)])

    #Given an input, calculate the output (using the layers calculate() method)
    def calculate(self,input):
        prevLayerOutput = input

        # Loop through every layer in NN and use output of previous layer as input
to next layer
        for layer in self.layers:
            prevLayerOutput = layer.calculate(prevLayerOutput)

        # Return output of NN
        return prevLayerOutput

    #Given a predicted output and ground truth output simply return the loss
(depending on the loss function)
    def calculateloss(self,yp,y):
        if self.loss == 0:
            # Sum of square errors loss function
            return np.sum(np.square(yp-y)) / len(yp)
        else:
            # Binary cross entropy loss function
            return np.sum((y*np.log(yp) + (1 - y)*np.log(1 - yp))) / -len(yp)

    #Given a predicted output and ground truth output simply return the derivative
of the loss
    #(depending on the loss function)
    def lossderiv(self,yp,y):
        if self.loss == 0:
            # Derivative of sum of square errors loss function
            return -(y-yp)
        else:
            # Derivative of binary cross entropy loss function
            return -((y/yp) - ((1-y)/(1-yp)))

    #Given a single input and desired output preform one step of backpropagation
(including a
    #forward pass, getting the derivative of the loss, and then calling calcwdeltas
for layers
    # with the right values
    def train(self,x,y):

        # Feed forward calculation
```

```
        output = self.calculate(x)

        # Backpropagation
        prevWtimesdelta = self.lossderiv(output, y)
        for layer in np.flip(self.layers):
            prevWtimesdelta = layer.calcwdeltas(prevWtimesdelta)

if __name__=="__main__":

    # Get the learning rate as a command line argument
    lr = 0
    if(len(sys.argv) >= 2):
        lr = float(sys.argv[1])
        print("Learning rate = " + str(lr))

    if (len(sys.argv)<3):
        print('a good place to test different parts of your code')

    elif (sys.argv[2]=='example'):
        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])

        # Build and train example neural net from class
        nn = NeuralNetwork(2, np.array([2, 2]), 2, 1, 0, lr, w)
        nn.train(x, y)

        # Print updated weights for example NN
        print("\n\nWeights after updating:")
        print(nn.weights)


    elif(sys.argv[2]=='and'):
        print('learn and')
        xData = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
        yData = np.array([[0], [0], [0], [1]])

        nn = NeuralNetwork(1, np.array([1]), 2, 1, 1, lr)

        for _ in range(10000):
            dataInd = np.random.randint(4, size=1)[0]
            nn.train(xData[dataInd], yData[dataInd])

        print("\n\nPerceptron results after training:")
        print("Input of (0,0) produces an output of:")
        print(nn.calculate(xData[0]))
        print("Input of (0,1) produces an output of:")
        print(nn.calculate(xData[1]))
        print("Input of (1,0) produces an output of:")
```

```
        print(nn.calculate(xData[2]))
        print("Input of (1,1) produces an output of:")
        print(nn.calculate(xData[3]))

    elif(sys.argv[2]=='xor'):
        print('learn xor')
        xData = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
        yData = np.array([[0], [1], [1], [0]])

        perceptron = NeuralNetwork(1, np.array([1]), 2, 1, 1, lr)
        nn = NeuralNetwork(2, np.array([2, 1]), 2, 1, 1, lr)

        for _ in range(10000):
            dataInd = np.random.randint(4, size=1)[0]
            perceptron.train(xData[dataInd], yData[dataInd])
            nn.train(xData[dataInd], yData[dataInd])

        print("\n\nPerceptron results after training:")
        print("Input of (0,0) produces an output of:")
        print(perceptron.calculate(xData[0]))
        print("Input of (0,1) produces an output of:")
        print(perceptron.calculate(xData[1]))
        print("Input of (1,0) produces an output of:")
        print(perceptron.calculate(xData[2]))
        print("Input of (1,1) produces an output of:")
        print(perceptron.calculate(xData[3]))

        print("\n\n")

        print("Neural network with one hidden layer results after training:")
        print("Input of (0,0) produces an output of:")
        print(nn.calculate(xData[0]))
        print("Input of (0,1) produces an output of:")
        print(nn.calculate(xData[1]))
        print("Input of (1,0) produces an output of:")
        print(nn.calculate(xData[2]))
        print("Input of (1,1) produces an output of:")
        print(nn.calculate(xData[3]))
```