```
import numpy as np
import sys
import matplotlib.pyplot as plt
"""
Matthew Walters - Steven Koprowicz
2/13/23

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.act = activation
        #for now, we're going to have self.numinps be the number of inputs,
disregarding the bias input,
        # which means the length of self.weights must be the length of self.numinps +
        self.numinps = input_num
        self.lr = lr
        self.weights = []
        #since I wrote the FullyConnectedLayer to randomly generate weights if not
given,
        # the neuron will be given weights, random or not
        for i in weights:
            self.weights.append(i)

    #This method returns the activation of the net
    def activate(self,net):
        if self.act == 0:
            return net
        elif self.act == 1:
            f = 1 / (1 + np.exp(-net))
            return f
        else:
            raise Exception("Activation Function not Supported")

    #Calculate the output of the neuron should save the input and output for back-
propagation.
    def calculate(self,inputs):
        self.input = inputs
        x = 0
        #   here, I put range(len(input)) + 1, so the bias would be included,
        #       and when it gets to the bias, the input is always 1, hence the if
statement
        for i in range(self.numinps + 1):
            if i >= self.numinps:
                x += 1*self.weights[i]
            else:
                x += self.input[i]*self.weights[i]
        self.output = self.activate(x)
        return self.output
```

```python

    #This method returns the derivative of the activation function with respect to the net
    def activationderivative(self):
        #derivative of log is output(1-output)
        #derivative of linear is the slope of the linear function, which I imagine jus returns 1 in this case,
        # since activate() just returns it's input
        if self.act == 1:
            return self.output * (1 - self.output)
        else:
            return 1

    #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()
        self.deltaw = []
        for i in range(len(self.weights)):
            self.deltaw.append(self.weights[i]*self.delta)
        return self.deltaw

    #Simply update the weights using the partial derivatives and the learning rate
    def updateweight(self):
        newweights = []
        for i in range(len(self.weights)):
            if i >= self.numinps:
                newweights.append(self.weights[i] - self.delta*1*self.lr)
            else:
                newweights.append(self.weights[i] - self.delta*self.input[i]*self.lr)
        self.weights = []
        for i in range(len(newweights)):
            self.weights.append(newweights[i])


#A fully connected layer
#for Matthew
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.numN = numOfNeurons
        self.activation = activation
        self.numinps = input_num
        self.lr = lr
        self.neurons = []
        self.weights = []
        # the length of each list of weights must be the number of inputs + 1 or it won't use them
        # it also won't accept the weights if there isn't a of list of weights for eac neuron, no more no less
        if weights is None or len(weights) != self.numN or len(weights[0]) != self.numinps + 1:
            # print("random weights")
            for i in range(self.numN):
                w = []
                for j in range(self.numinps + 1):
                    w.append(np.random.uniform(0.1, 0.9))
                self.weights.append(w)
```

```python
        else:
            for i in weights:
                w = []
                for j in i:
                    w.append(j)
                self.weights.append(w)
        for i in range(self.numN):
            x = Neuron(self.activation, self.numinps, self.lr, self.weights[i])
            self.neurons.append(x)

    #calculate 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, inputs):
        self.outputs = []
        for i in self.neurons:
            self.outputs.append(i.calculate(inputs))
        return self.outputs

    #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.
    def calcwdeltas(self, wtimesdelta):
        self.wdeltas = []
        for i in range(len(self.neurons)):
            x = self.neurons[i].calcpartialderivative(wtimesdelta[i])
            self.wdeltas.append(x)
        self.wtimesdeltas = []
        for i in range(len(self.weights[i])):
            y = 0
            for j in range(len(self.wdeltas)):
                y += self.wdeltas[j][i]
            self.wtimesdeltas.append(y)
        # wtimesdelta done
        # update weights
        for i in self.neurons:
            i.updateweight()
        return self.wtimesdeltas

#An entire neural network
#for Steven
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.numL = numOfLayers
        self.numN = numOfNeurons.copy()
        self.numinps = inputSize
        self.activation = activation.copy()
        self.loss = loss
        self.lr = lr
        self.layers = []
        isize = self.numinps
        if weights is not None and len(weights) == self.numL:
            for i in range(self.numL):
```

```python
                x = FullyConnected(self.numN[i], self.activation[i], isize, self.lr, weights[i])
                isize = self.numN[i]
                self.layers.append(x)
        else:
            for i in range(self.numL):
                x = FullyConnected(self.numN[i], self.activation[i], isize, self.lr)
                isize = self.numN[i]
                self.layers.append(x)

    #Given an input, calculate the output (using the layers calculate() method)
    def calculate(self,inputs):
        self.input = inputs
        output = inputs
        for i in self.layers:
            output = i.calculate(output)
        return output

    #Given a predicted output and ground truth output simply return the loss (dependir on the loss function)
    def calculateloss(self,yp,y):
        # self.loss == 0: sum of squared errors
        # self.loss == 1: binary cross entropy
        #the binary cross entropy makes the assumption you only have 1 output, sum squared errors assumes you have a list of outputs
        if self.loss == 0:
            loss = 0
            for i in range(len(yp)):
                loss += ((yp[i] - y[i])**2)
        elif self.loss == 1:
            loss = -(y * np.log(yp)) + ((1 - y) * np.log(1 - yp))
        else:
            raise Exception("Loss Function Not Supported")
        return loss

    #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 squared error
            ld = (yp - y)
        elif self.loss == 1:
            #derivative of binary cross entropy
            ld = -(y/yp) + ((1-y)/(1-yp))
        else:
            raise Exception("Loss Function Not Supported")
        return ld

    #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):
        #PSEUDOCODE
        output = self.calculate(x)
        ld = []
        for i in range(self.layers[self.numL - 1].numN):
            ld.append(self.lossderiv(output[i], y[i]))
        for i in reversed(range(len(self.layers))):
           ld = self.layers[i].calcwdeltas(ld)

```

```python
if __name__=="__main__":
    if (len(sys.argv)<2):
        print('a good place to test different parts of your code')

    elif (sys.argv[1]=='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])
        network = NeuralNetwork(2, [2,2], 2, [1,1], 0, .5, w)
        print(network.calculate(x))
        # points = []
        for i in range(1000):
            # point = network.calculate(x)
            # points.append(network.calculateloss(point, y))
            network.train(x, y)
        print(network.calculate(x))
        # plt.plot(points)
        # plt.xlabel("Iterations")
        # plt.ylabel("Loss")
        # plt.title("EXAMPLE: Loss over Time")
        # plt.show()

    elif(sys.argv[1]=='and'):
        print('learn AND')
        x = np.array([[0,0], [0,1], [1,0], [1,1]])
        y = np.array([[0],[0],[0],[1]])
        a = [0,0,0,0]
        # a single perceptron (logistic activation, binary cross entropy loss, .5 learning rate, random weights)
        network = NeuralNetwork(1, [1], 2, [1], 1, .5)
        count = 0
        points = []
        while (a != [1,1,1,1]):
            for i in range(len(x)):
                con = network.calculate(x[i])[0]
                points.append(network.calculateloss(con,y[i]))
                network.train(x[i], y[i])
                # checking for convergence, if the weight isn't changed at all for all sets of inputs, we're done
                if abs(con - network.calculate(x[i])[0]) < 10**(-2):
                    a[i] = 1
            count += 1
            if count == 10000:
                print("does not converge")
                break
            elif a == [1,1,1,1]:
                print("converged")
        for i in x:
            print(i,":", network.calculate(i))
        plt.plot(points)
        plt.xlabel("Iterations")
        plt.ylabel("Loss")
        plt.title("AND: Loss over Time")
        plt.show()

    elif(sys.argv[1]=='xor'):
        print('learn XOR')
        x = np.array([[0,0], [0,1], [1,0], [1,1]])
        y = np.array([[0], [1], [1], [0]])
```

```python
        # 1 perceptron (logistic activation, binary cross entropy loss, .5 learning rate, random weights)
        network = NeuralNetwork(1, [1], 2, [1], 1, .5)
        a = [0,0,0,0]
        print("Starting Training on 1 perceptron")
        for i in x:
            print(i,":", network.calculate(i))
        count = 0
        points = []
        while (a != [1,1,1,1]):
            for i in range(len(x)):
                con = network.calculate(x[i])[0]
                points.append(network.calculateloss(con, y[i]))
                network.train(x[i], y[i])
                # checking for convergence, if the weight isn't changed at all for all sets of inputs, we're done
                if abs(con - network.calculate(x[i])[0]) < 10**(-2):
                    a[i] = 1
            count += 1
            if count == 10000:
                print("does not converge")
                break
            if a == [1,1,1,1]:
                print("converged")
        plt.plot(points)
        plt.xlabel("Iterations")
        plt.ylabel("Loss")
        plt.title("XOR with 1 Perceptron: Loss over Time")
        plt.show()
        for i in x:
            print(i,":", network.calculate(i))
        # 1 output perceptron plus a hidden layer, also one perceptron
        # (logistic activation, binary cross entropy loss, .5 learning rate, random weights)
        # with less than 4 neurons in the hidden layer, it won't always converge
        network = NeuralNetwork(2, [4,1], 2, [1,1], 1, .5)
        print("Starting Training with a hidden layer")
        for i in x:
            print(i,":", network.calculate(i))
        count = 0
        points = []
        while (a != [1,1,1,1]):
            for i in range(len(x)):
                con = network.calculate(x[i])[0]
                points.append(network.calculateloss(con, y[i]))
                network.train(x[i], y[i])
                # checking for convergence, if the weight isn't changed at all for all sets of inputs, we're done
                if abs(con - network.calculate(x[i])[0]) < 10**(-2):
                    a[i] = 1
            count += 1
            if count == 10000:
                print("does not converge")
                break
            if a == [1,1,1,1]:
                print("converged")
        for i in x:
            print(i,":", network.calculate(i))
        plt.plot(points)
        plt.xlabel("Iterations")
```

```
    plt.ylabel("Loss")
    plt.title("XOR with a hidden layer: Loss over Time")
```