```python
"""
Devs: Gabriel Stowe and Eli Carter
Date: 2/10/23
"""

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 + 1
        self.lr = lr

#If weights are None, generate random values. Weights include the bias, therefore the
size of the weights is input_num + 1.
        if weights is None:
            self.weights = np.random.rand(self.input_num)
        else:
            if type(weights) != np.ndarray:
                weights = np.array(weights)

            self.weights = weights


        self.inputs = np.array(np.zeros(self.input_num))
        #Set last element to equal 1. This is for the bias.
        self.inputs[-1] = 1
        self.output = 0



    #This method returns the activation of the net
```

```
    def activate(self,net):
        #if activation function is logistic
        if self.activation == 1:
            return 1 / (1 + np.exp(-net))

        return net

    #Calculate the output of the neuron should save the input and output for back-
propagation.
    def calculate(self,input):
        #If the input is not inserted as a np.array, convert python list to  np.array
        if type(input) != np.ndarray:
            input = np.array(input)

        #set the self.inputs var to equal the input. Last element of inputs has to
remain 1 for the bias.
        self.inputs[:-1] = input

        #Calculate the net.
        net = np.dot(self.inputs,self.weights.T)

        self.output = self.activate(net)
        return self.output


    #This method returns the derivative of the activation function with respect to the
net
    def activationderivative(self):
        #derivative of sigmoid function is f(x)*(1-f(x))
        if self.activation == 1:
            return self.output * (1 - self.output)
        #derivative of linear function is 1
        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):
        #Calculate the partial derivative to update weights later
        self.pd = wtimesdelta * self.activationderivative() * self.inputs

        #Return delta * w
        return wtimesdelta * self.activationderivative() * self.weights

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

```
#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

        self.neurons = []

        for i in range(self.numOfNeurons):
            if self.weights is None:

self.neurons.append(Neuron(activation=self.activation,input_num=self.input_num,lr=self
.lr,weights=None))
            else:

self.neurons.append(Neuron(activation=self.activation,input_num=self.input_num,lr=self
.lr,weights=weights[i]))




    #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):
        #Stores the output
        output = np.array(np.zeros(self.numOfNeurons))

        #Check if input is a numpy array. If not, make it a numpy array.
        if type(input) != np.ndarray:
            input = np.array(input)

        #Loop through the neurons and calculate its output.
        for i,neuron in enumerate(self.neurons):
            output[i] = neuron.calculate(input)

        return output

    #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):
```

```
        # deltas = np.array(np.zeros(len(self.neurons)))
        deltas = []
        for i, neuron in enumerate(self.neurons):
            deltas.append(neuron.calcpartialderivative(wtimesdelta[i]))
            neuron.updateweight()

        return np.sum(deltas,axis=0)



#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, input_num, activation, loss, lr, weights=None):

        self.numOfLayers = numOfLayers
        self.numOfNeurons = numOfNeurons
        self.input_num = input_num
        self.activation = activation
        self.loss = loss
        self.lr = lr
        self.weights = weights

        self.layers = []
        for i in range(self.numOfLayers):
            if weights == None:
                self.layers.append(FullyConnected(numOfNeurons=self.numOfNeurons[i], activation=self.activation[i], input_num=self.input_num[i], lr=self.lr, weights=None))
            else:
                self.layers.append(FullyConnected(numOfNeurons=self.numOfNeurons[i], activation=self.activation[i], input_num=self.input_num[i], lr=self.lr, weights=self.weights[i]))

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

    #Given a predicted output and ground truth output simply return the loss (depending on the loss function)
    def calculateloss(self,yp,y):
        if type(yp)  != np.ndarray:
            yp = np.array(yp)
```

```
        if type(y) != np.ndarray:
            y = np.array(y)
        #Square Loss
        if self.loss == 0:
            return 0.5 * (y - yp)**2
        # Cross Entropy
        else:
            return -y*np.log(yp) - (1-y)*np.log(1-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):
        #Square Loss Derivative
        if self.loss == 0:
            return -(y- yp)
        # Cross Entropy Derivative
        else:
            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):
        #Check if x and y are numpy arrays. If not, convert them.
        if type(x) != np.ndarray:
            x = np.array(x)
        if type(y) != np.ndarray:
            y = np.array(y)

        #Calculate Prdicted Ouput
        yp = self.calculate(x)



        delta = self.lossderiv(yp,y)

        for r_layer in reversed(self.layers):
            delta = r_layer.calcwdeltas(delta)



        return self.calculateloss(yp,y)

if __name__=="__main__":
    if (len(sys.argv)<2):
        w=[[[.15,.2,.35],[.25,.3,.35]],[[.4,.45,.6],[.5,.55,.6]]]
```

```python
        N = NeuralNetwork( numOfLayers=2 , numOfNeurons=[2,2], input_num=[2,2],
activation=[1,1],loss=0, lr=0.5,weights=w)

        print("Inital Output: ",N.calculate([0.05,0.1]))
        print("Expected Output: [0.01,0.99]")

        for i in range(1000):
            N.train([0.05,0.1],[0.01,0.99])
        print("After 1000 Epochs Output: ",N.calculate([0.05,0.1]))

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

        w=[[[.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])

        N = NeuralNetwork( numOfLayers=2 , numOfNeurons=[2,2], input_num=[2,2],
activation=[1,1],loss=0, lr=float(sys.argv[1]),weights=w)
        N.train(x,y)

        print("After training Output: ",np.around(N.calculate(x),3))
        for i in range(N.numOfLayers):
            for j in range(N.layers[i].numOfNeurons):
                print(f"    Layer {i+1} Neuron {j+1} weights: {[round(w,2) for w in
N.layers[i].neurons[j].weights]}")

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

        input_nums = [2]
        num_neurons = [1]
        num_layers = 1
        activations = [1]

        N =
NeuralNetwork(num_layers,num_neurons,input_nums,activations,1,float(sys.argv[1]),
None)

        for i in range(1100):
            N.train([0,0],np.array([0]))
            N.train([1,0],np.array([0]))
            N.train([1,1],np.array([1]))
            N.train([0,1],np.array([0]))
            if i%100 == 0:
                print(f"Epoch {i}:")
                print(f"    0 and 0: {round(N.calculate([0,0])[0], 3)}")
                print(f"    0 and 1: {round(N.calculate([0,1])[0], 3)}")
                print(f"    1 and 0: {round(N.calculate([1,0])[0], 3)}")
                print(f"    1 and 1: {round(N.calculate([1,1])[0], 3)}")
```

```
        # print('After training:')
        # print(f"    0 and 0: {round(N.calculate([0,0])[0])}")
        # print(f"    0 and 1: {round(N.calculate([0,1])[0])}")
        # print(f"    1 and 0: {round(N.calculate([1,0])[0])}")
        # print(f"    1 and 1: {round(N.calculate([1,1])[0])}")

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

        input_nums = [2,2,3]
        num_neurons = [2,3,1]
        num_layers = 3
        activations = [1,1,1,1]

        N = NeuralNetwork(num_layers,
num_neurons,input_nums,activations,1,float(sys.argv[1]), None)

        for i in range(10000):
            N.train(np.array([0,0]),np.array([0]))
            N.train(np.array([0,1]),np.array([1]))
            N.train(np.array([1,0]),np.array([1]))
            N.train(np.array([1,1]),np.array([0]))
            if i%1000 == 0:
                print(f"Epoch {i}:")
                print(f"    0 and 0: {np.around(N.calculate(np.array([0,0])),3)[0]}")
                print(f"    0 and 1: {np.around(N.calculate(np.array([0,1])),3)[0]}")
                print(f"    1 and 0: {np.around(N.calculate(np.array([1,0])),3)[0]}")
                print(f"    1 and 1: {np.around(N.calculate(np.array([1,1])),3)[0]}")
```