Code

import numpy as np

import sys

import matplotlib.pyplot as plt

"""

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

self.net = 0

self.input = 0

self.output = 0 # output for neutron (after activation)

self.dEdw = 0 # partial deriv of weights for neuron

#This method returns the activation of the net

def activate(self,net):

if self.activation == 0:

return net

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.array(input)

self.net = np.matmul(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:

return 1

activationderiv = np.exp(-self.net)/(1+np.exp(-self.net))\*\*2

return activationderiv

#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):

delta = np.array(wtimesdelta\*self.activationderivative())

self.dEdw = delta\*self.input

wtimesdelta = delta\*self.weights

return wtimesdelta

#Simply update the weights using the partial derivatives and

# the learning weight

def updateweight(self):

self.weights = self.weights - self.lr\*self.dEdw

#A fully connected layer

class FullyConnected:

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):

self.Neurons.append(Neuron(self.activation,self.input\_num,self.lr,self.weights[i]))

#calculate the output of all the neurons in the layer and

# return a vector with those values (go through the neurons

# and call the calculate() method)

def calculate(self, input):

results = []

for neuron in self.Neurons:

results.append(neuron.calculate(input))

results.append(1)

return results

# 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 weights s

# (using the updateweight() method). I should return the sum of w\*delta.

def calcwdeltas(self, wtimesdelta):

wtimesdeltas = []

for i in range(self.numOfNeurons):

wtimesdeltas.append(self.Neurons[i].calcpartialderivative(wtimesdelta[i]))

self.Neurons[i].updateweight()

return np.sum(wtimesdeltas,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, 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

if self.weights is None:

self.weights = []

for i in range(numOfLayers):

weight = np.random.rand(numOfNeurons[i],inputSize[i])

self.weights.append(weight)

self.Layers = []

for i in range(self.numOfLayers):

self.Layers.append(FullyConnected(self.numOfNeurons[i],self.activation[i],

self.inputSize[i],self.lr,self.weights[i]))

#Given an input, calculate the output (using the layers calculate() method)

def calculate(self,input):

results = self.Layers[0].calculate(input)

for i in range(1,self.numOfLayers):

results = self.Layers[i].calculate(results)

return results[:-1]

#Given a predicted output and ground truth output simply

# return the loss (depending on the loss function)

def calculateloss(self,y,yp):

if self.loss == 0:

return 0.5\*np.sum((y-yp)\*\*2)

res = []

for k in range(len(y)):

res.append([-(i\*np.log2(j)+(1-i)\*np.log2(1-j)) for i,j in zip(y[k],yp[k])])

return np.mean(res)

#Given a predicted output and ground truth output simply

# return the derivative of the loss (depending on the loss function)

def lossderiv(self,y,yp):

if self.loss == 0:

return -(y-yp)

return [-i/j + (1-i)/(1-j) for i,j in zip(y,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):

# feedforward

yp = self.calculate(x)

# backpropagate

wtimesdelta = self.Layers[-1].calcwdeltas(self.lossderiv(y,yp))

for i in range(self.numOfLayers-2,-1,-1):

wtimesdelta = self.Layers[i].calcwdeltas(wtimesdelta)

if \_\_name\_\_=="\_\_main\_\_":

if (len(sys.argv)<2):

print('usage: python project1\_suann.py [example|and|or]')

elif (sys.argv[1]=='example'):

print('run example from class (1 steps)')

w=np.array([[[.15,.2,.35],[.25,.3,.35]],[[.4,.45,.6],[.5,.55,.6]]])

x=np.array([[0.05,0.1,1]]) # 1 input

y=np.array([[0.01,0.99]]) # 1 output

# Normalization of inputs

x = np.array([i/np.linalg.norm(i) for i in x])

y = np.array([i/np.linalg.norm(i) for i in y])

numOfLayers = len(w)

numOfNeurons = [len(w[i]) for i in range(numOfLayers)]

inputSize = [len(w[i][0]) for i in range(numOfLayers)]

# numOfLayers = 2

# numOfNeurons = [2,2]

# inputSize = [3,3]

activation = [0,0]

los = 0

lr = 0.5

# initialize neural network with the right layers, inputs, outputs

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,0.001)

losses = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses.append(loss)

print("yp=", yps)

print("y=",y)

print("loss=", loss)

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,0.01)

losses1 = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses1.append(loss)

print("yp=", yps)

print("y=",y)

print("loss1=", loss)

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,0.1)

losses2 = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses2.append(loss)

print("yp=", yps)

print("y=",y)

print("loss2=", loss)

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,1)

losses3 = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses3.append(loss)

print("yp=", yps)

print("y=",y)

print("loss3=", loss)

s = "Sigmoid function"

if NN.activation[0] == 0:

s = "Linear function"

# Plot loss vs epoch

plt.plot(losses, label="lr = 0.001")

plt.plot(losses1, label="lr = 0.01")

plt.plot(losses2, label="lr = 0.1")

plt.plot(losses3, label="lr = 1")

plt.title("The example problem\n Number of hidden layer = %d, Activation function = %s\nLoss function vs epoch for different learning rates" % (NN.numOfLayers-1, s) )

plt.xlabel("Epoch")

if NN.loss == 0:

plt.ylabel("Squared error loss")

else:

plt.ylabel("Binary cross entropy loss")

plt.legend(loc="upper right")

plt.show()

# print("yp=", yps)

# print("y=",y)

# print("loss=", loss)

# plt.plot(losses)

# plt.title("Loss function vs epoch")

# plt.xlabel("Epoch")

# plt.ylabel("Loss")

# plt.show()

elif(sys.argv[1]=='and'):

print('learn and')

w=np.array([[[1,1,-1.5]]]) # single layer

x=np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]]) # 4 different inputs

y=np.array([[0],[0],[0],[1]]) # 4 outputs

numOfLayers = 1

numOfNeurons = np.array([1])

inputSize = np.array([3])

activation = [0]

los = 0

lr = 1

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,0.001)

losses = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses.append(loss)

print("yp=", yps)

print("y=",y)

print("loss=", loss)

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,0.01)

losses1 = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses1.append(loss)

print("yp=", yps)

print("y=",y)

print("loss1=", loss)

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,0.1)

losses2 = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses2.append(loss)

print("yp=", yps)

print("y=",y)

print("loss2=", loss)

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,1)

losses3 = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses3.append(loss)

print("yp=", yps)

print("y=",y)

print("loss3=", loss)

s = "Sigmoid function"

if NN.activation[0] == 0:

s = "Linear function"

# Plot loss vs epoch

plt.semilogy(losses, label="lr = 0.001")

plt.semilogy(losses1, label="lr = 0.01")

plt.semilogy(losses2, label="lr = 0.1")

plt.semilogy(losses3, label="lr = 1")

plt.title("The AND problem\n Number of hidden layer = %d, Activation function = %s\nLoss function vs epoch for different learning rates" % (NN.numOfLayers-1, s) )

plt.xlabel("Epoch")

if NN.loss == 0:

plt.ylabel("Squared error loss")

else:

plt.ylabel("Binary cross entropy loss")

plt.legend(loc="upper right")

plt.show()

elif(sys.argv[1]=='xor'):

print('learn xor')

# w=np.array([]) # randomly initialize weights for each neuron in each layer

x=np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]]) # 4 different inputs

y=np.array([[0],[1],[1],[0]]) # 4 outputs

numOfLayers = 2

numOfNeurons = np.array([2,1])

inputSize = np.array([3,3])

activation = [0,0]

los = 0

lr = 0.001

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,0.001)

losses = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses.append(loss)

print("yp=", yps)

print("y=",y)

print("loss=", loss)

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,0.01)

losses1 = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses1.append(loss)

print("yp=", yps)

print("y=",y)

print("loss1=", loss)

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,0.1)

losses2 = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses2.append(loss)

print("yp=", yps)

print("y=",y)

print("loss2=", loss)

NN = NeuralNetwork(numOfLayers,numOfNeurons,inputSize,activation,los,1)

losses3 = []

num\_iters = 1000

for i in range(num\_iters):

yps = []

for j in range(len(x)):

NN.train(x[j],y[j])

yp = NN.calculate(x[j])

yps.append(yp)

loss = NN.calculateloss(y,yps)

losses3.append(loss)

print("yp=", yps)

print("y=",y)

print("loss3=", loss)

s = "Sigmoid function"

if NN.activation[0] == 0:

s = "Linear function"

# Plot loss vs epoch

plt.semilogy(losses, label="lr = 0.001")

plt.semilogy(losses1, label="lr = 0.01")

plt.semilogy(losses2, label="lr = 0.1")

# plt.semilogy(losses3, label="lr = 1")

plt.title("The XOR problem\n Number of hidden layer = %d, Activation function = %s\nLoss function vs epoch for different learning rates" % (NN.numOfLayers-1, s) )

plt.xlabel("Epoch")

if NN.loss == 0:

plt.ylabel("Squared error loss")

else:

plt.ylabel("Binary cross entropy loss")

plt.legend(loc="upper right")

plt.show()