**Introduction**

In this post ‘Deep Learning from first principles with Python, R and Octave-Part 7’, I implement optimization methods used in Stochastic Gradient Descent (SGD) to speed up the convergence. Specifically I discuss and implement the following gradient descent optimization techniques

a.Vanilla Stochastic Gradient Descent  
b.Learning rate decay  
c. Momentum method  
d. RMSProp  
e. Adaptive Moment Estimation (Adam)

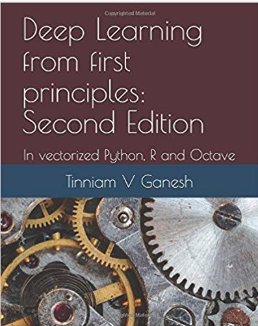
This post, further enhances my generic  L-Layer Deep Learning Network implementations in  vectorized Python, R and Octave to also include the Stochastic Gradient Descent optimization techniques.

**Note**: In the vectorized Python, R and Octave implementations below only a  1024 random training samples were used. This was to reduce the computation time. You are free to use the entire data set (60000 training data) for the computation.

All the above optimization techniques for Stochastic Gradient Descent are based on the technique of exponentially weighted average method. So for example if we had some time series data \theta_{1},\theta_{2},\theta_{3}... \theta_{t} then we we can represent the exponentially average value at time ‘t’ as a sequence of the the previous value v_{t-1}and \theta_{t}as shown below  
v_{t} = \beta v_{t-1} + (1-\beta)\theta_{t}

Here v_{t}represent the average of the data set over \frac {1}{1-\beta} By choosing different values of \beta, we can average over a larger or smaller number of the data points.  
We can write the equations as follows  
v_{t} = \beta v_{t-1} + (1-\beta)\theta_{t}  
v_{t-1} = \beta v_{t-2} + (1-\beta)\theta_{t-1}  
v_{t-2} = \beta v_{t-3} + (1-\beta)\theta_{t-2}  
and  
v_{t-k} = \beta v_{t-(k+1))} + (1-\beta)\theta_{t-k}  
By substitution we have  
v_{t} = (1-\beta)\theta_{t} + \beta v_{t-1}  
v_{t} = (1-\beta)\theta_{t} + \beta ((1-\beta)\theta_{t-1}) + \beta v_{t-2}  
v_{t} = (1-\beta)\theta_{t} + \beta ((1-\beta)\theta_{t-1}) + \beta ((1-\beta)\theta_{t-2}+ \beta v_{t-3} )

Hence it can be seen that the v_{t}is the weighted sum over the previous values \theta_{k}, which is an exponentially decaying function.

Checkout my book ‘Deep Learning from first principles: Second Edition – In vectorized Python, R and Octave’. My book starts with the implementation of a simple 2-layer Neural Network and works its way to a generic L-Layer Deep Learning Network, with all the bells and whistles. The derivations have been discussed in detail. The code has been extensively commented and included in its entirety in the Appendix sections.

**1.1a. Stochastic Gradient Descent (Vanilla) – Python**

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import sklearn.linear\_model

import pandas as pd

import sklearn

import sklearn.datasets

exec(open("DLfunctions7.py").read())

exec(open("load\_mnist.py").read())

# Read the training data

training=list(read(dataset='training',path=".\\mnist"))

test=list(read(dataset='testing',path=".\\mnist"))

lbls=[]

pxls=[]

for i in range(60000):

l,p=training[i]

lbls.append(l)

pxls.append(p)

labels= np.array(lbls)

pixels=np.array(pxls)

y=labels.reshape(-1,1)

X=pixels.reshape(pixels.shape[0],-1)

X1=X.T

Y1=y.T

# Create a list of 1024 random numbers.

permutation = list(np.random.permutation(2\*\*10))

# Subset 16384 from the data

X2 = X1[:, permutation]

Y2 = Y1[:, permutation].reshape((1,2\*\*10))

# Set the layer dimensions

layersDimensions=[784, 15,9,10]

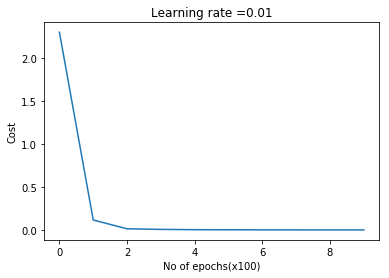
# Perform SGD with regular gradient descent

parameters = L\_Layer\_DeepModel\_SGD(X2, Y2, layersDimensions, hiddenActivationFunc='relu',

outputActivationFunc="softmax",learningRate = 0.01 ,

optimizer="gd",

mini\_batch\_size =512, num\_epochs = 1000, print\_cost = True,figure="fig1.png")



**1.1b. Stochastic Gradient Descent (Vanilla) – R**

source("mnist.R")

source("DLfunctions7.R")

#Load and read MNIST data

load\_mnist()

x <- t(train$x)

X <- x[,1:60000]

y <-train$y

y1 <- y[1:60000]

y2 <- as.matrix(y1)

Y=t(y2)

# Subset 1024 random samples from MNIST

permutation = c(sample(2^10))

# Randomly shuffle the training data

X1 = X[, permutation]

y1 = Y[1, permutation]

y2 <- as.matrix(y1)

Y1=t(y2)

# Set layer dimensions

layersDimensions=c(784, 15,9, 10)

# Perform SGD with regular gradient descent

retvalsSGD= L\_Layer\_DeepModel\_SGD(X1, Y1, layersDimensions,

hiddenActivationFunc='tanh',

outputActivationFunc="softmax",

learningRate = 0.05,

optimizer="gd",

mini\_batch\_size = 512,

num\_epochs = 5000,

print\_cost = True)

#Plot the cost vs iterations

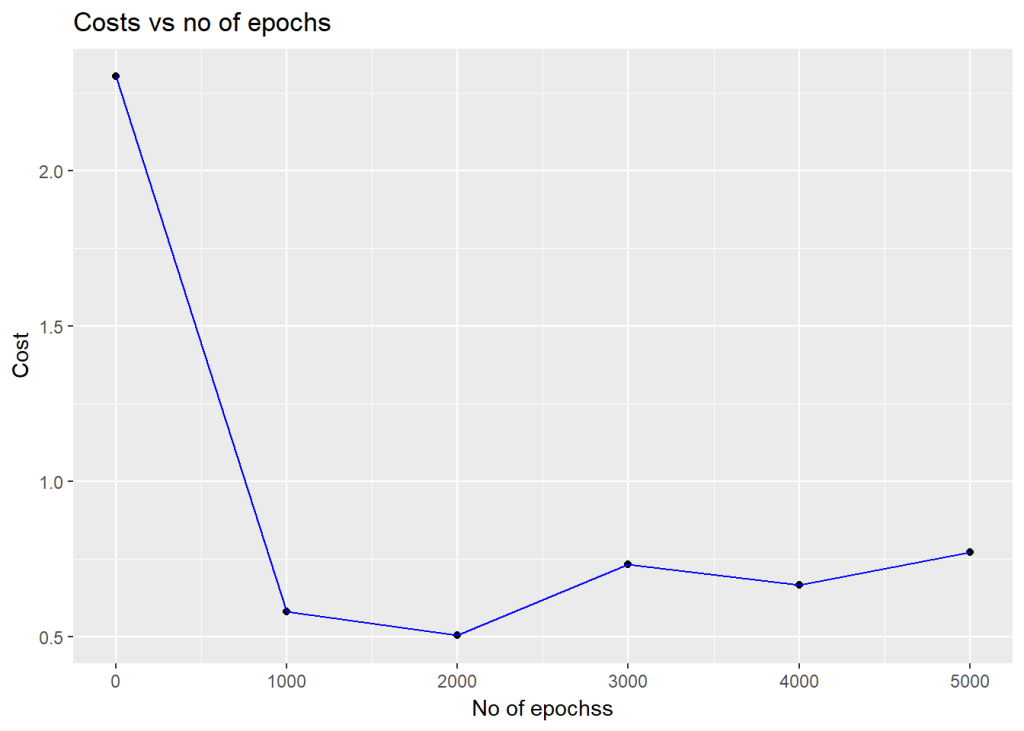
iterations <- seq(0,5000,1000)

costs=retvalsSGD$costs

df=data.frame(iterations,costs)

ggplot(df,aes(x=iterations,y=costs)) + geom\_point() + geom\_line(color="blue") +

ggtitle("Costs vs no of epochs") + xlab("No of epochss") + ylab("Cost")



**1.1c. Stochastic Gradient Descent (Vanilla) – Octave**

source("DL7functions.m")

#Load and read MNIST

load('./mnist/mnist.txt.gz');

#Create a random permutatation from 1024

permutation = randperm(1024);

disp(length(permutation));

# Use this 1024 as the batch

X=trainX(permutation,:);

Y=trainY(permutation,:);

# Set layer dimensions

layersDimensions=[784, 15, 9, 10];

# Perform SGD with regular gradient descent

[weights biases costs]=L\_Layer\_DeepModel\_SGD(X', Y', layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.005,

lrDecay=true,

decayRate=1,

lambd=0,

keep\_prob=1,

optimizer="gd",

beta=0.9,

beta1=0.9,

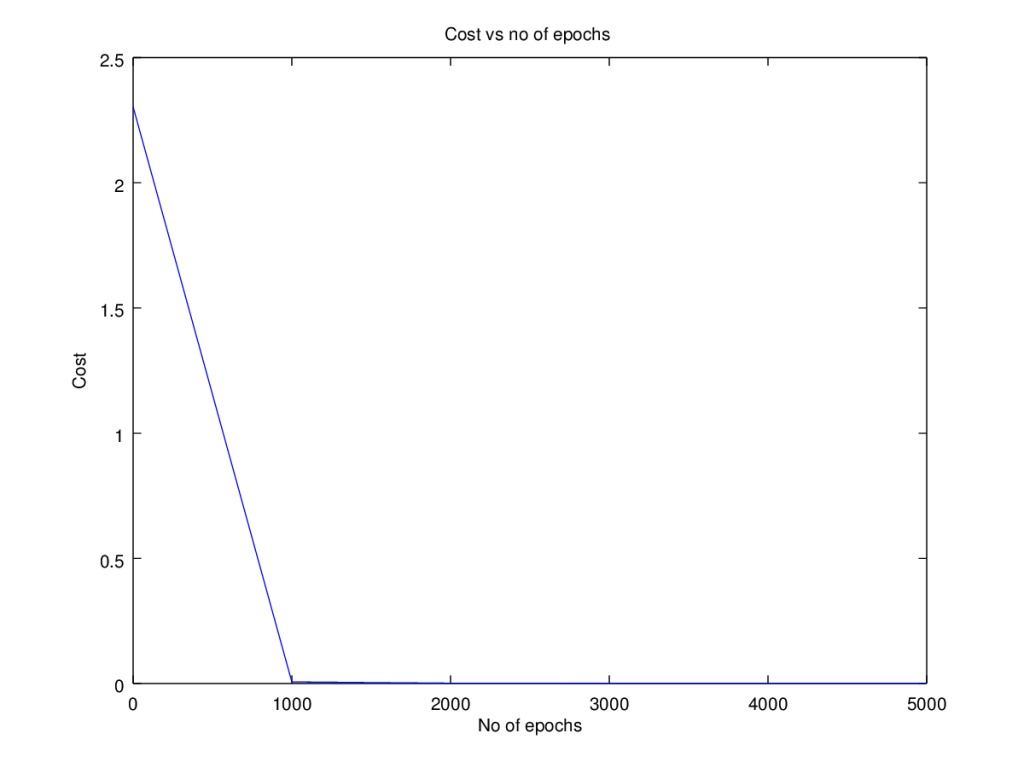
beta2=0.999,

epsilon=10^-8,

mini\_batch\_size = 512,

num\_epochs = 5000);

plotCostVsEpochs(5000,costs);



**2.1. Stochastic Gradient Descent with Learning rate decay**

Since in Stochastic Gradient Descent,with  each epoch, we use slight different samples, the gradient descent algorithm, oscillates across the ravines and wanders around the minima, when a fixed learning rate is used. In this technique of ‘learning rate decay’ the learning rate is slowly decreased with the number of epochs and becomes smaller and smaller, so that gradient descent can take smaller steps towards the minima.

There are several techniques employed in learning rate decay

a) Exponential decay: \alpha = decayRate^{epochNum} *\alpha_{0}  
b) 1/t decay : \alpha = \frac{\alpha_{0}}{1 + decayRate*epochNum}  
c) \alpha = \frac {decayRate}{\sqrt(epochNum)}*\alpha_{0}

In my implementation I have used the ‘exponential decay’. The code snippet for Python is shown below

if lrDecay == True:

learningRate = np.power(decayRate,(num\_epochs/1000)) \* learningRate

**2.1a. Stochastic Gradient Descent with Learning rate decay – Python**

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import sklearn.linear\_model

import pandas as pd

import sklearn

import sklearn.datasets

exec(open("DLfunctions7.py").read())

exec(open("load\_mnist.py").read())

# Read the MNIST data

training=list(read(dataset='training',path=".\\mnist"))

test=list(read(dataset='testing',path=".\\mnist"))

lbls=[]

pxls=[]

for i in range(60000):

l,p=training[i]

lbls.append(l)

pxls.append(p)

labels= np.array(lbls)

pixels=np.array(pxls)

y=labels.reshape(-1,1)

X=pixels.reshape(pixels.shape[0],-1)

X1=X.T

Y1=y.T

# Create a list of random numbers of 1024

permutation = list(np.random.permutation(2\*\*10))

# Subset 16384 from the data

X2 = X1[:, permutation]

Y2 = Y1[:, permutation].reshape((1,2\*\*10))

# Set layer dimensions

layersDimensions=[784, 15,9,10]

# Perform SGD with learning rate decay

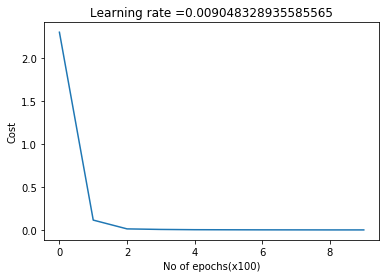
parameters = L\_Layer\_DeepModel\_SGD(X2, Y2, layersDimensions, hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.01 , lrDecay=True, decayRate=0.9999,

optimizer="gd",

mini\_batch\_size =512, num\_epochs = 1000, print\_cost = True,figure="fig2.png")



**2.1b. Stochastic Gradient Descent with Learning rate decay – R**

source("mnist.R")

source("DLfunctions7.R")

# Read and load MNIST

load\_mnist()

x <- t(train$x)

X <- x[,1:60000]

y <-train$y

y1 <- y[1:60000]

y2 <- as.matrix(y1)

Y=t(y2)

# Subset 1024 random samples from MNIST

permutation = c(sample(2^10))

# Randomly shuffle the training data

X1 = X[, permutation]

y1 = Y[1, permutation]

y2 <- as.matrix(y1)

Y1=t(y2)

# Set layer dimensions

layersDimensions=c(784, 15,9, 10)

# Perform SGD with Learning rate decay

retvalsSGD= L\_Layer\_DeepModel\_SGD(X1, Y1, layersDimensions,

hiddenActivationFunc='tanh',

outputActivationFunc="softmax",

learningRate = 0.05,

lrDecay=TRUE,

decayRate=0.9999,

optimizer="gd",

mini\_batch\_size = 512,

num\_epochs = 5000,

print\_cost = True)

#Plot the cost vs iterations

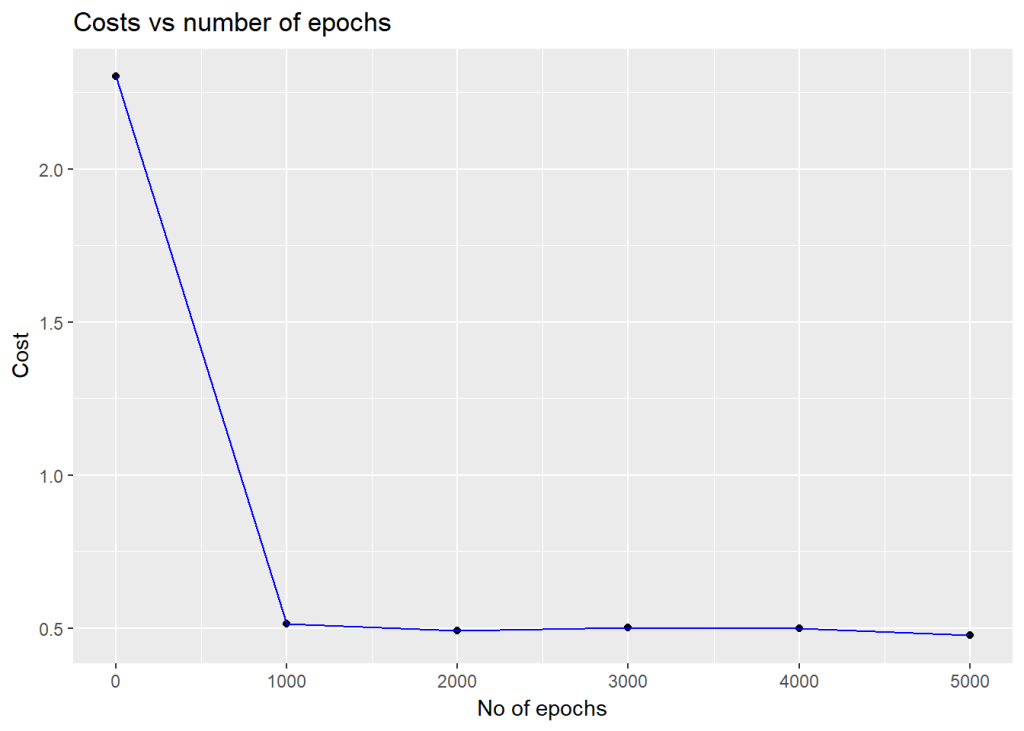
iterations <- seq(0,5000,1000)

costs=retvalsSGD$costs

df=data.frame(iterations,costs)

ggplot(df,aes(x=iterations,y=costs)) + geom\_point() + geom\_line(color="blue") +

ggtitle("Costs vs number of epochs") + xlab("No of epochs") + ylab("Cost")



**2.1c. Stochastic Gradient Descent with Learning rate decay – Octave**

source("DL7functions.m")

#Load and read MNIST

load('./mnist/mnist.txt.gz');

#Create a random permutatation from 1024

permutation = randperm(1024);

disp(length(permutation));

# Use this 1024 as the batch

X=trainX(permutation,:);

Y=trainY(permutation,:);

# Set layer dimensions

layersDimensions=[784, 15, 9, 10];

# Perform SGD with regular Learning rate decay

[weights biases costs]=L\_Layer\_DeepModel\_SGD(X', Y', layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.01,

lrDecay=true,

decayRate=0.999,

lambd=0,

keep\_prob=1,

optimizer="gd",

beta=0.9,

beta1=0.9,

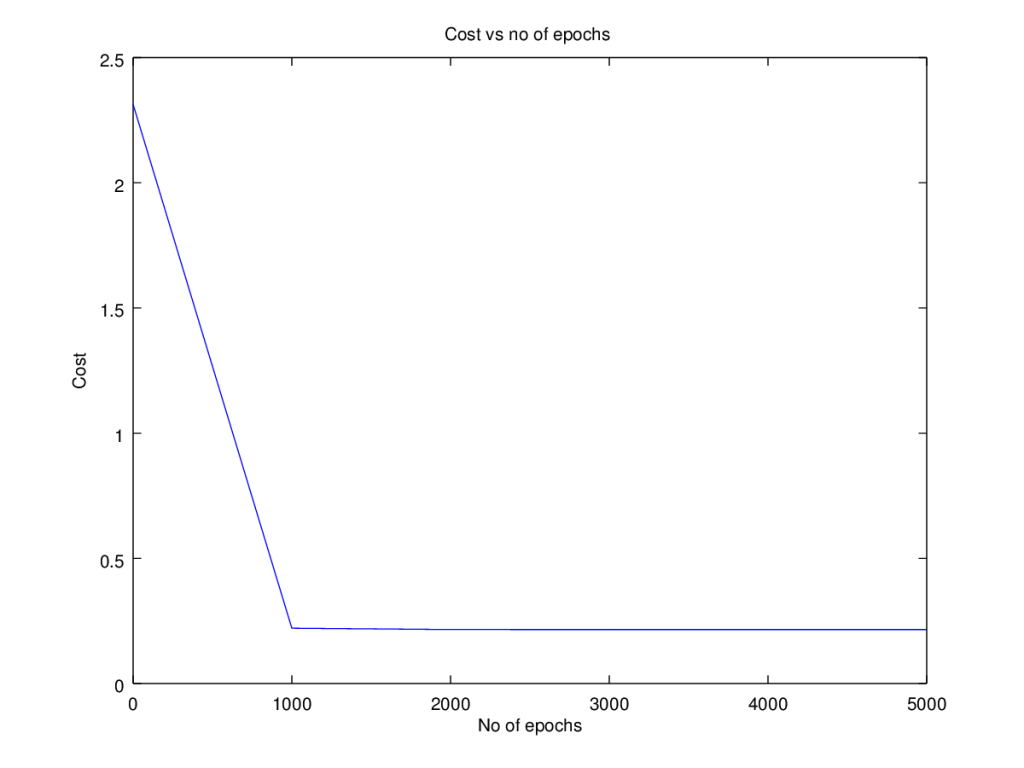
beta2=0.999,

epsilon=10^-8,

mini\_batch\_size = 512,

num\_epochs = 5000);

plotCostVsEpochs(5000,costs)



**3.1. Stochastic Gradient Descent with Momentum**

Stochastic Gradient Descent with Momentum uses the exponentially weighted average method discusses above and more generally moves faster into the ravine than across it. The equations are  
v_{dW}^l = \beta v_{dW}^l + (1-\beta)dW^{l}  
v_{db}^l = \beta v_{db}^l + (1-\beta)db^{l}  
W^{l} = W^{l} - \alpha v_{dW}^l  
b^{l} = b^{l} - \alpha v_{db}^lwhere  
v_{dW}and v_{db}are the momentum terms which are exponentially weighted with the corresponding gradients ‘dW’ and ‘db’ at the corresponding layer ‘l’ The code snippet for Stochastic Gradient Descent with momentum in R is shown below

# Perform Gradient Descent with momentum

# Input : Weights and biases

# : beta

# : gradients

# : learning rate

# : outputActivationFunc - Activation function at hidden layer sigmoid/softmax

#output : Updated weights after 1 iteration

gradientDescentWithMomentum <- function(parameters, gradients,v, beta, learningRate,outputActivationFunc="sigmoid"){

L = length(parameters)/2 # number of layers in the neural network

# Update rule for each parameter. Use a for loop.

for(l in 1:(L-1)){

# Compute velocities

# v['dWk'] = beta \*v['dWk'] + (1-beta)\*dWk

v[[paste("dW",l, sep="")]] = beta\*v[[paste("dW",l, sep="")]] +

(1-beta) \* gradients[[paste('dW',l,sep="")]]

v[[paste("db",l, sep="")]] = beta\*v[[paste("db",l, sep="")]] +

(1-beta) \* gradients[[paste('db',l,sep="")]]

parameters[[paste("W",l,sep="")]] = parameters[[paste("W",l,sep="")]] -

learningRate\* v[[paste("dW",l, sep="")]]

parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] -

learningRate\* v[[paste("db",l, sep="")]]

}

# Compute for the Lth layer

if(outputActivationFunc=="sigmoid"){

v[[paste("dW",L, sep="")]] = beta\*v[[paste("dW",L, sep="")]] +

(1-beta) \* gradients[[paste('dW',L,sep="")]]

v[[paste("db",L, sep="")]] = beta\*v[[paste("db",L, sep="")]] +

(1-beta) \* gradients[[paste('db',L,sep="")]]

parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] -

learningRate\* v[[paste("dW",l, sep="")]]

parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] -

learningRate\* v[[paste("db",l, sep="")]]

}else if (outputActivationFunc=="softmax"){

v[[paste("dW",L, sep="")]] = beta\*v[[paste("dW",L, sep="")]] +

(1-beta) \* t(gradients[[paste('dW',L,sep="")]])

v[[paste("db",L, sep="")]] = beta\*v[[paste("db",L, sep="")]] +

(1-beta) \* t(gradients[[paste('db',L,sep="")]])

parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] -

learningRate\* t(gradients[[paste("dW",L,sep="")]])

parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] -

learningRate\* t(gradients[[paste("db",L,sep="")]])

}

return(parameters)

}

**3.1a. Stochastic Gradient Descent with Momentum- Python**

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import sklearn.linear\_model

import pandas as pd

import sklearn

import sklearn.datasets

# Read and load data

exec(open("DLfunctions7.py").read())

exec(open("load\_mnist.py").read())

training=list(read(dataset='training',path=".\\mnist"))

test=list(read(dataset='testing',path=".\\mnist"))

lbls=[]

pxls=[]

for i in range(60000):

l,p=training[i]

lbls.append(l)

pxls.append(p)

labels= np.array(lbls)

pixels=np.array(pxls)

y=labels.reshape(-1,1)

X=pixels.reshape(pixels.shape[0],-1)

X1=X.T

Y1=y.T

# Create a list of random numbers of 1024

permutation = list(np.random.permutation(2\*\*10))

# Subset 16384 from the data

X2 = X1[:, permutation]

Y2 = Y1[:, permutation].reshape((1,2\*\*10))

layersDimensions=[784, 15,9,10]

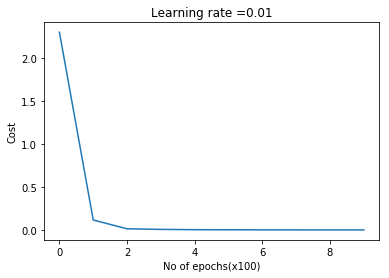
# Perform SGD with momentum

parameters = L\_Layer\_DeepModel\_SGD(X2, Y2, layersDimensions, hiddenActivationFunc='relu',

outputActivationFunc="softmax",learningRate = 0.01 ,

optimizer="momentum", beta=0.9,

mini\_batch\_size =512, num\_epochs = 1000, print\_cost = True,figure="fig3.png")



**3.1b. Stochastic Gradient Descent with Momentum- R**

source("mnist.R")

source("DLfunctions7.R")

load\_mnist()

x <- t(train$x)

X <- x[,1:60000]

y <-train$y

y1 <- y[1:60000]

y2 <- as.matrix(y1)

Y=t(y2)

# Subset 1024 random samples from MNIST

permutation = c(sample(2^10))

# Randomly shuffle the training data

X1 = X[, permutation]

y1 = Y[1, permutation]

y2 <- as.matrix(y1)

Y1=t(y2)

layersDimensions=c(784, 15,9, 10)

# Perform SGD with momentum

retvalsSGD= L\_Layer\_DeepModel\_SGD(X1, Y1, layersDimensions,

hiddenActivationFunc='tanh',

outputActivationFunc="softmax",

learningRate = 0.05,

optimizer="momentum",

beta=0.9,

mini\_batch\_size = 512,

num\_epochs = 5000,

print\_cost = True)

#Plot the cost vs iterations

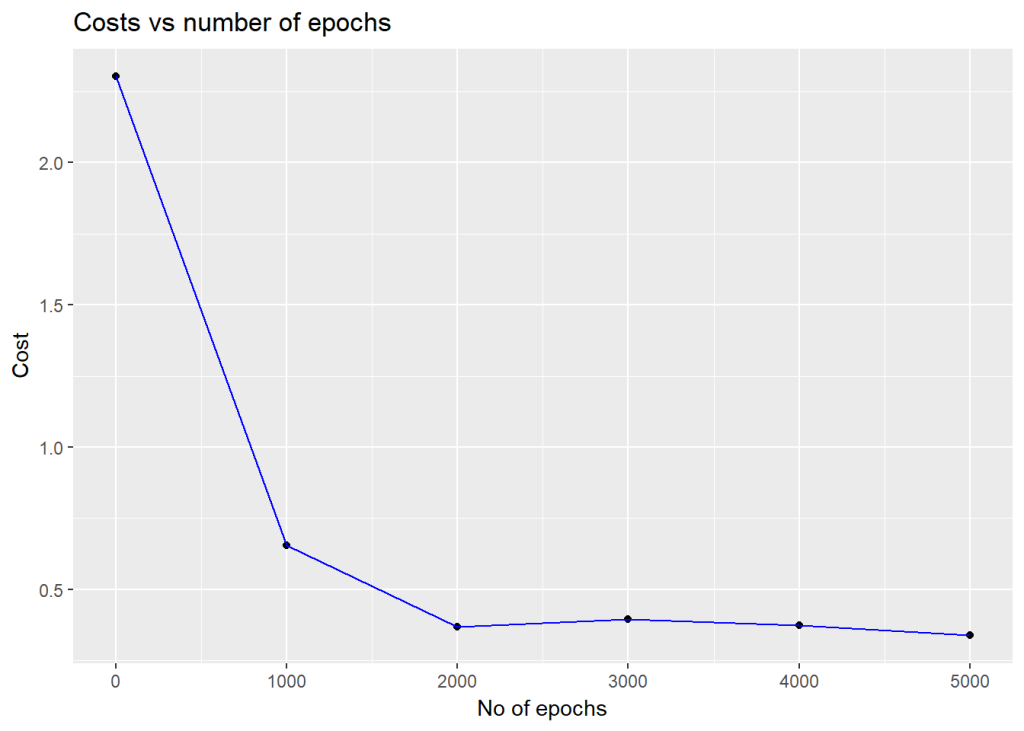
iterations <- seq(0,5000,1000)

costs=retvalsSGD$costs

df=data.frame(iterations,costs)

ggplot(df,aes(x=iterations,y=costs)) + geom\_point() + geom\_line(color="blue") +

ggtitle("Costs vs number of epochs") + xlab("No of epochs") + ylab("Cost")



**3.1c. Stochastic Gradient Descent with Momentum- Octave**

source("DL7functions.m")

#Load and read MNIST

load('./mnist/mnist.txt.gz');

#Create a random permutatation from 60K

permutation = randperm(1024);

disp(length(permutation));

# Use this 1024 as the batch

X=trainX(permutation,:);

Y=trainY(permutation,:);

# Set layer dimensions

layersDimensions=[784, 15, 9, 10];

# Perform SGD with Momentum

[weights biases costs]=L\_Layer\_DeepModel\_SGD(X', Y', layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.01,

lrDecay=false,

decayRate=1,

lambd=0,

keep\_prob=1,

optimizer="momentum",

beta=0.9,

beta1=0.9,

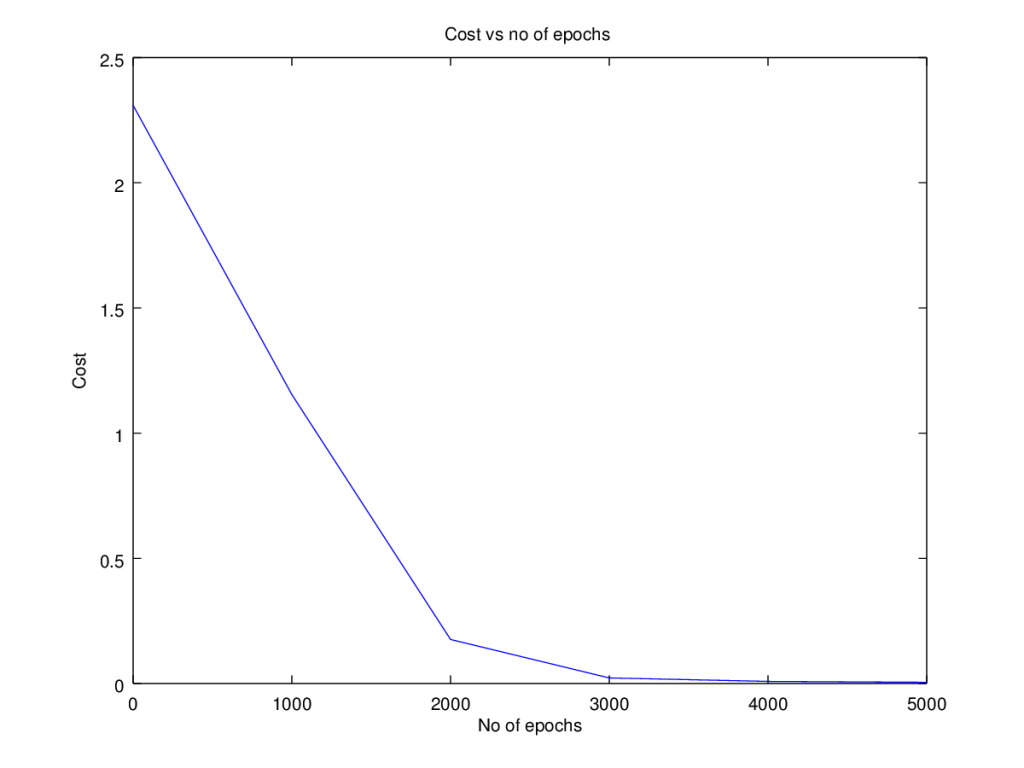
beta2=0.999,

epsilon=10^-8,

mini\_batch\_size = 512,

num\_epochs = 5000);

plotCostVsEpochs(5000,costs)



**4.1. Stochastic Gradient Descent with RMSProp**

Stochastic Gradient Descent with RMSProp tries to move faster towards the minima while dampening the oscillations across the ravine.  
The equations are

s_{dW}^l = \beta_{1} s_{dW}^l + (1-\beta_{1})(dW^{l})^{2}  
s_{db}^l = \beta_{1} s_{db}^l + (1-\beta_{1})(db^{l})^2  
W^{l} = W^{l} - \frac {\alpha s_{dW}^l}{\sqrt (s_{dW}^l + \epsilon) }  
b^{l} = b^{l} - \frac {\alpha s_{db}^l}{\sqrt (s_{db}^l + \epsilon) }  
where s_{dW}and s_{db}are the RMSProp terms which are exponentially weighted with the corresponding gradients ‘dW’ and ‘db’ at the corresponding layer ‘l’

The code snippet in Octave is shown below

# Update parameters with RMSProp

# Input : parameters

# : gradients

# : s

# : beta

# : learningRate

# :

#output : Updated parameters RMSProp

function [weights biases] = gradientDescentWithRMSProp(weights, biases,gradsDW,gradsDB, sdW, sdB, beta1, epsilon, learningRate,outputActivationFunc="sigmoid")

L = size(weights)(2); # number of layers in the neural network

# Update rule for each parameter.

for l=1:(L-1)

sdW{l} = beta1\*sdW{l} + (1 -beta1) \* gradsDW{l} .\* gradsDW{l};

sdB{l} = beta1\*sdB{l} + (1 -beta1) \* gradsDB{l} .\* gradsDB{l};

weights{l} = weights{l} - learningRate\* gradsDW{l} ./ sqrt(sdW{l} + epsilon);

biases{l} = biases{l} - learningRate\* gradsDB{l} ./ sqrt(sdB{l} + epsilon);

endfor

if (strcmp(outputActivationFunc,"sigmoid"))

sdW{L} = beta1\*sdW{L} + (1 -beta1) \* gradsDW{L} .\* gradsDW{L};

sdB{L} = beta1\*sdB{L} + (1 -beta1) \* gradsDB{L} .\* gradsDB{L};

weights{L} = weights{L} -learningRate\* gradsDW{L} ./ sqrt(sdW{L} +epsilon);

biases{L} = biases{L} -learningRate\* gradsDB{L} ./ sqrt(sdB{L} + epsilon);

elseif (strcmp(outputActivationFunc,"softmax"))

sdW{L} = beta1\*sdW{L} + (1 -beta1) \* gradsDW{L}' .\* gradsDW{L}';

sdB{L} = beta1\*sdB{L} + (1 -beta1) \* gradsDB{L}' .\* gradsDB{L}';

weights{L} = weights{L} -learningRate\* gradsDW{L}' ./ sqrt(sdW{L} +epsilon);

biases{L} = biases{L} -learningRate\* gradsDB{L}' ./ sqrt(sdB{L} + epsilon);

endif

end

**4.1a. Stochastic Gradient Descent with RMSProp – Python**

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import sklearn.linear\_model

import pandas as pd

import sklearn

import sklearn.datasets

exec(open("DLfunctions7.py").read())

exec(open("load\_mnist.py").read())

# Read and load MNIST

training=list(read(dataset='training',path=".\\mnist"))

test=list(read(dataset='testing',path=".\\mnist"))

lbls=[]

pxls=[]

for i in range(60000):

l,p=training[i]

lbls.append(l)

pxls.append(p)

labels= np.array(lbls)

pixels=np.array(pxls)

y=labels.reshape(-1,1)

X=pixels.reshape(pixels.shape[0],-1)

X1=X.T

Y1=y.T

print("X1=",X1.shape)

print("y1=",Y1.shape)

# Create a list of random numbers of 1024

permutation = list(np.random.permutation(2\*\*10))

# Subset 16384 from the data

X2 = X1[:, permutation]

Y2 = Y1[:, permutation].reshape((1,2\*\*10))

layersDimensions=[784, 15,9,10]

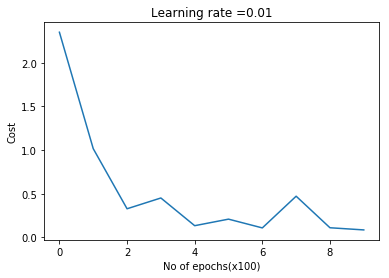
# Use SGD with RMSProp

parameters = L\_Layer\_DeepModel\_SGD(X2, Y2, layersDimensions, hiddenActivationFunc='relu',

outputActivationFunc="softmax",learningRate = 0.01 ,

optimizer="rmsprop", beta1=0.7, epsilon=1e-8,

mini\_batch\_size =512, num\_epochs = 1000, print\_cost = True,figure="fig4.png")



**4.1b. Stochastic Gradient Descent with RMSProp – R**

source("mnist.R")

source("DLfunctions7.R")

load\_mnist()

x <- t(train$x)

X <- x[,1:60000]

y <-train$y

y1 <- y[1:60000]

y2 <- as.matrix(y1)

Y=t(y2)

# Subset 1024 random samples from MNIST

permutation = c(sample(2^10))

# Randomly shuffle the training data

X1 = X[, permutation]

y1 = Y[1, permutation]

y2 <- as.matrix(y1)

Y1=t(y2)

layersDimensions=c(784, 15,9, 10)

#Perform SGD with RMSProp

retvalsSGD= L\_Layer\_DeepModel\_SGD(X1, Y1, layersDimensions,

hiddenActivationFunc='tanh',

outputActivationFunc="softmax",

learningRate = 0.001,

optimizer="rmsprop",

beta1=0.9,

epsilon=10^-8,

mini\_batch\_size = 512,

num\_epochs = 5000 ,

print\_cost = True)

#Plot the cost vs iterations

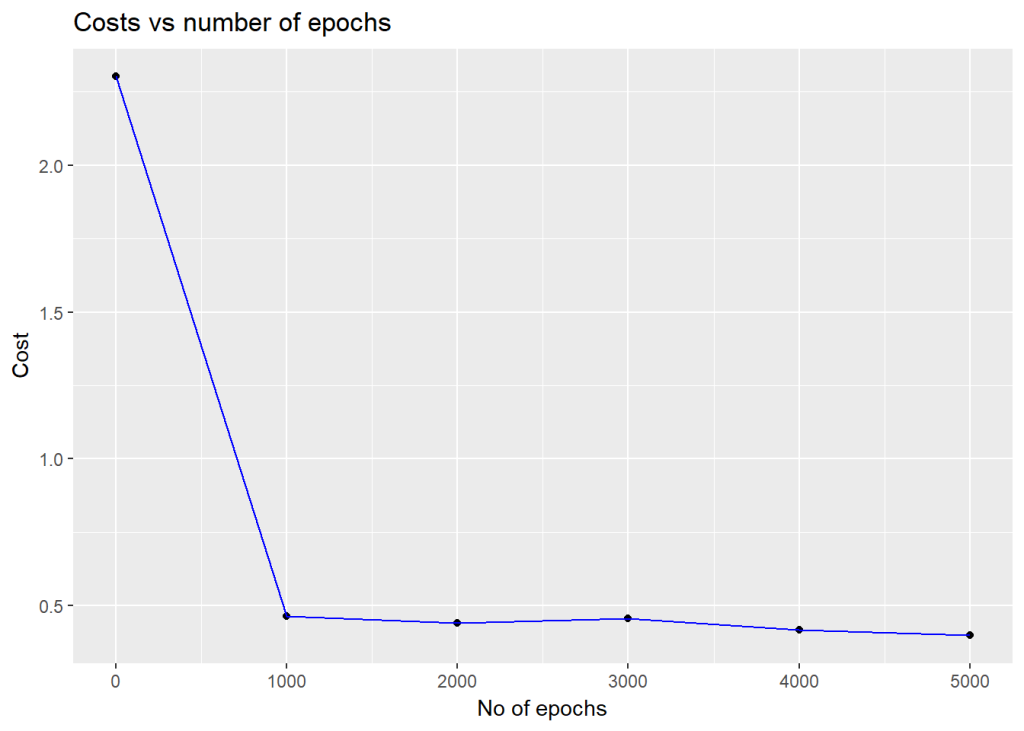
iterations <- seq(0,5000,1000)

costs=retvalsSGD$costs

df=data.frame(iterations,costs)

ggplot(df,aes(x=iterations,y=costs)) + geom\_point() + geom\_line(color="blue") +

ggtitle("Costs vs number of epochs") + xlab("No of epochs") + ylab("Cost")



**4.1c. Stochastic Gradient Descent with RMSProp – Octave**

source("DL7functions.m")

load('./mnist/mnist.txt.gz');

#Create a random permutatation from 1024

permutation = randperm(1024);

# Use this 1024 as the batch

X=trainX(permutation,:);

Y=trainY(permutation,:);

# Set layer dimensions

layersDimensions=[784, 15, 9, 10];

#Perform SGD with RMSProp

[weights biases costs]=L\_Layer\_DeepModel\_SGD(X', Y', layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.005,

lrDecay=false,

decayRate=1,

lambd=0,

keep\_prob=1,

optimizer="rmsprop",

beta=0.9,

beta1=0.9,

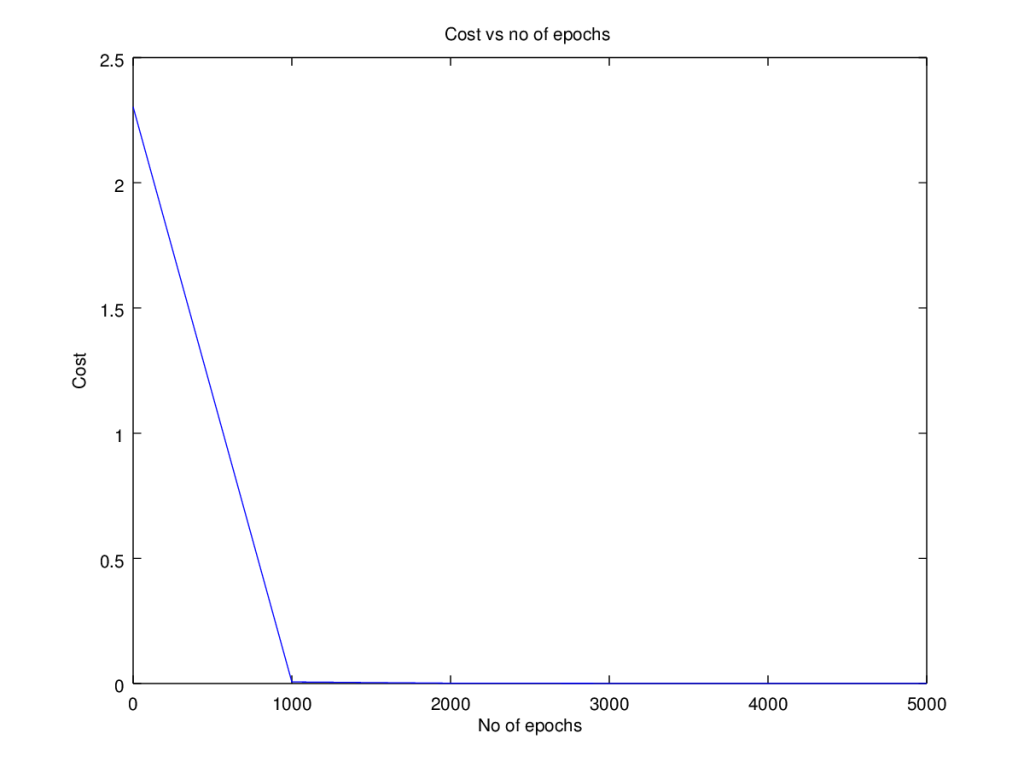
beta2=0.999,

epsilon=1,

mini\_batch\_size = 512,

num\_epochs = 5000);

plotCostVsEpochs(5000,costs)



**5.1. Stochastic Gradient Descent with Adam**

Adaptive Moment Estimate is a combination of the momentum (1st moment) and RMSProp(2nd moment). The equations for Adam are below  
v_{dW}^l = \beta_{1} v_{dW}^l + (1-\beta_{1})dW^{l}  
v_{db}^l = \beta_{1} v_{db}^l + (1-\beta_{1})db^{l}  
The bias corrections for the 1st moment  
vCorrected_{dW}^l= \frac {v_{dW}^l}{1 - \beta_{1}^{t}}  
vCorrected_{db}^l= \frac {v_{db}^l}{1 - \beta_{1}^{t}}

Similarly the moving average for the 2nd moment- RMSProp  
s_{dW}^l = \beta_{2} s_{dW}^l + (1-\beta_{2})(dW^{l})^2  
s_{db}^l = \beta_{2} s_{db}^l + (1-\beta_{2})(db^{l})^2  
The bias corrections for the 2nd moment  
sCorrected_{dW}^l= \frac {s_{dW}^l}{1 - \beta_{2}^{t}}  
sCorrected_{db}^l= \frac {s_{db}^l}{1 - \beta_{2}^{t}}

The Adam Gradient Descent is given by  
W^{l} = W^{l} - \frac {\alpha vCorrected_{dW}^l}{\sqrt (s_{dW}^l + \epsilon) }  
b^{l} = b^{l} - \frac {\alpha vCorrected_{db}^l}{\sqrt (s_{db}^l + \epsilon) }  
The code snippet of Adam in R is included below

# Perform Gradient Descent with Adam

# Input : Weights and biases

# : beta1

# : epsilon

# : gradients

# : learning rate

# : outputActivationFunc - Activation function at hidden layer sigmoid/softmax

#output : Updated weights after 1 iteration

gradientDescentWithAdam <- function(parameters, gradients,v, s, t,

beta1=0.9, beta2=0.999, epsilon=10^-8, learningRate=0.1,outputActivationFunc="sigmoid"){

L = length(parameters)/2 # number of layers in the neural network

v\_corrected <- list()

s\_corrected <- list()

# Update rule for each parameter. Use a for loop.

for(l in 1:(L-1)){

# v['dWk'] = beta \*v['dWk'] + (1-beta)\*dWk

v[[paste("dW",l, sep="")]] = beta1\*v[[paste("dW",l, sep="")]] +

(1-beta1) \* gradients[[paste('dW',l,sep="")]]

v[[paste("db",l, sep="")]] = beta1\*v[[paste("db",l, sep="")]] +

(1-beta1) \* gradients[[paste('db',l,sep="")]]

# Compute bias-corrected first moment estimate.

v\_corrected[[paste("dW",l, sep="")]] = v[[paste("dW",l, sep="")]]/(1-beta1^t)

v\_corrected[[paste("db",l, sep="")]] = v[[paste("db",l, sep="")]]/(1-beta1^t)

# Element wise multiply of gradients

s[[paste("dW",l, sep="")]] = beta2\*s[[paste("dW",l, sep="")]] +

(1-beta2) \* gradients[[paste('dW',l,sep="")]] \* gradients[[paste('dW',l,sep="")]]

s[[paste("db",l, sep="")]] = beta2\*s[[paste("db",l, sep="")]] +

(1-beta2) \* gradients[[paste('db',l,sep="")]] \* gradients[[paste('db',l,sep="")]]

# Compute bias-corrected second moment estimate.

s\_corrected[[paste("dW",l, sep="")]] = s[[paste("dW",l, sep="")]]/(1-beta2^t)

s\_corrected[[paste("db",l, sep="")]] = s[[paste("db",l, sep="")]]/(1-beta2^t)

# Update parameters.

d1=sqrt(s\_corrected[[paste("dW",l, sep="")]]+epsilon)

d2=sqrt(s\_corrected[[paste("db",l, sep="")]]+epsilon)

parameters[[paste("W",l,sep="")]] = parameters[[paste("W",l,sep="")]] -

learningRate \* v\_corrected[[paste("dW",l, sep="")]]/d1

parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] -

learningRate\*v\_corrected[[paste("db",l, sep="")]]/d2

}

# Compute for the Lth layer

if(outputActivationFunc=="sigmoid"){

v[[paste("dW",L, sep="")]] = beta1\*v[[paste("dW",L, sep="")]] +

(1-beta1) \* gradients[[paste('dW',L,sep="")]]

v[[paste("db",L, sep="")]] = beta1\*v[[paste("db",L, sep="")]] +

(1-beta1) \* gradients[[paste('db',L,sep="")]]

# Compute bias-corrected first moment estimate.

v\_corrected[[paste("dW",L, sep="")]] = v[[paste("dW",L, sep="")]]/(1-beta1^t)

v\_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-beta1^t)

# Element wise multiply of gradients

s[[paste("dW",L, sep="")]] = beta2\*s[[paste("dW",L, sep="")]] +

(1-beta2) \* gradients[[paste('dW',L,sep="")]] \* gradients[[paste('dW',L,sep="")]]

s[[paste("db",L, sep="")]] = beta2\*s[[paste("db",L, sep="")]] +

(1-beta2) \* gradients[[paste('db',L,sep="")]] \* gradients[[paste('db',L,sep="")]]

# Compute bias-corrected second moment estimate.

s\_corrected[[paste("dW",L, sep="")]] = s[[paste("dW",L, sep="")]]/(1-beta2^t)

s\_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-beta2^t)

# Update parameters.

d1=sqrt(s\_corrected[[paste("dW",L, sep="")]]+epsilon)

d2=sqrt(s\_corrected[[paste("db",L, sep="")]]+epsilon)

parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] -

learningRate \* v\_corrected[[paste("dW",L, sep="")]]/d1

parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] -

learningRate\*v\_corrected[[paste("db",L, sep="")]]/d2

}else if (outputActivationFunc=="softmax"){

v[[paste("dW",L, sep="")]] = beta1\*v[[paste("dW",L, sep="")]] +

(1-beta1) \* t(gradients[[paste('dW',L,sep="")]])

v[[paste("db",L, sep="")]] = beta1\*v[[paste("db",L, sep="")]] +

(1-beta1) \* t(gradients[[paste('db',L,sep="")]])

# Compute bias-corrected first moment estimate.

v\_corrected[[paste("dW",L, sep="")]] = v[[paste("dW",L, sep="")]]/(1-beta1^t)

v\_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-beta1^t)

# Element wise multiply of gradients

s[[paste("dW",L, sep="")]] = beta2\*s[[paste("dW",L, sep="")]] +

(1-beta2) \* t(gradients[[paste('dW',L,sep="")]]) \* t(gradients[[paste('dW',L,sep="")]])

s[[paste("db",L, sep="")]] = beta2\*s[[paste("db",L, sep="")]] +

(1-beta2) \* t(gradients[[paste('db',L,sep="")]]) \* t(gradients[[paste('db',L,sep="")]])

# Compute bias-corrected second moment estimate.

s\_corrected[[paste("dW",L, sep="")]] = s[[paste("dW",L, sep="")]]/(1-beta2^t)

s\_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-beta2^t)

# Update parameters.

d1=sqrt(s\_corrected[[paste("dW",L, sep="")]]+epsilon)

d2=sqrt(s\_corrected[[paste("db",L, sep="")]]+epsilon)

parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] -

learningRate \* v\_corrected[[paste("dW",L, sep="")]]/d1

parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] -

learningRate\*v\_corrected[[paste("db",L, sep="")]]/d2

}

return(parameters)

}

**5.1a. Stochastic Gradient Descent with Adam – Python**

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import sklearn.linear\_model

import pandas as pd

import sklearn

import sklearn.datasets

exec(open("DLfunctions7.py").read())

exec(open("load\_mnist.py").read())

training=list(read(dataset='training',path=".\\mnist"))

test=list(read(dataset='testing',path=".\\mnist"))

lbls=[]

pxls=[]

print(len(training))

#for i in range(len(training)):

for i in range(60000):

l,p=training[i]

lbls.append(l)

pxls.append(p)

labels= np.array(lbls)

pixels=np.array(pxls)

y=labels.reshape(-1,1)

X=pixels.reshape(pixels.shape[0],-1)

X1=X.T

Y1=y.T

# Create a list of random numbers of 1024

permutation = list(np.random.permutation(2\*\*10))

# Subset 16384 from the data

X2 = X1[:, permutation]

Y2 = Y1[:, permutation].reshape((1,2\*\*10))

layersDimensions=[784, 15,9,10]

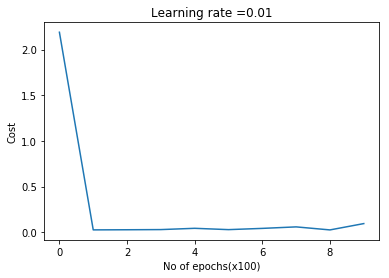
#Perform SGD with Adam optimization

parameters = L\_Layer\_DeepModel\_SGD(X2, Y2, layersDimensions, hiddenActivationFunc='relu',

outputActivationFunc="softmax",learningRate = 0.01 ,

optimizer="adam", beta1=0.9, beta2=0.9, epsilon = 1e-8,

mini\_batch\_size =512, num\_epochs = 1000, print\_cost = True, figure="fig5.png")



**5.1b. Stochastic Gradient Descent with Adam – R**

source("mnist.R")

source("DLfunctions7.R")

load\_mnist()

x <- t(train$x)

X <- x[,1:60000]

y <-train$y

y1 <- y[1:60000]

y2 <- as.matrix(y1)

Y=t(y2)

# Subset 1024 random samples from MNIST

permutation = c(sample(2^10))

# Randomly shuffle the training data

X1 = X[, permutation]

y1 = Y[1, permutation]

y2 <- as.matrix(y1)

Y1=t(y2)

layersDimensions=c(784, 15,9, 10)

#Perform SGD with Adam

retvalsSGD= L\_Layer\_DeepModel\_SGD(X1, Y1, layersDimensions,

hiddenActivationFunc='tanh',

outputActivationFunc="softmax",

learningRate = 0.005,

optimizer="adam",

beta1=0.7,

beta2=0.9,

epsilon=10^-8,

mini\_batch\_size = 512,

num\_epochs = 5000 ,

print\_cost = True)

#Plot the cost vs iterations

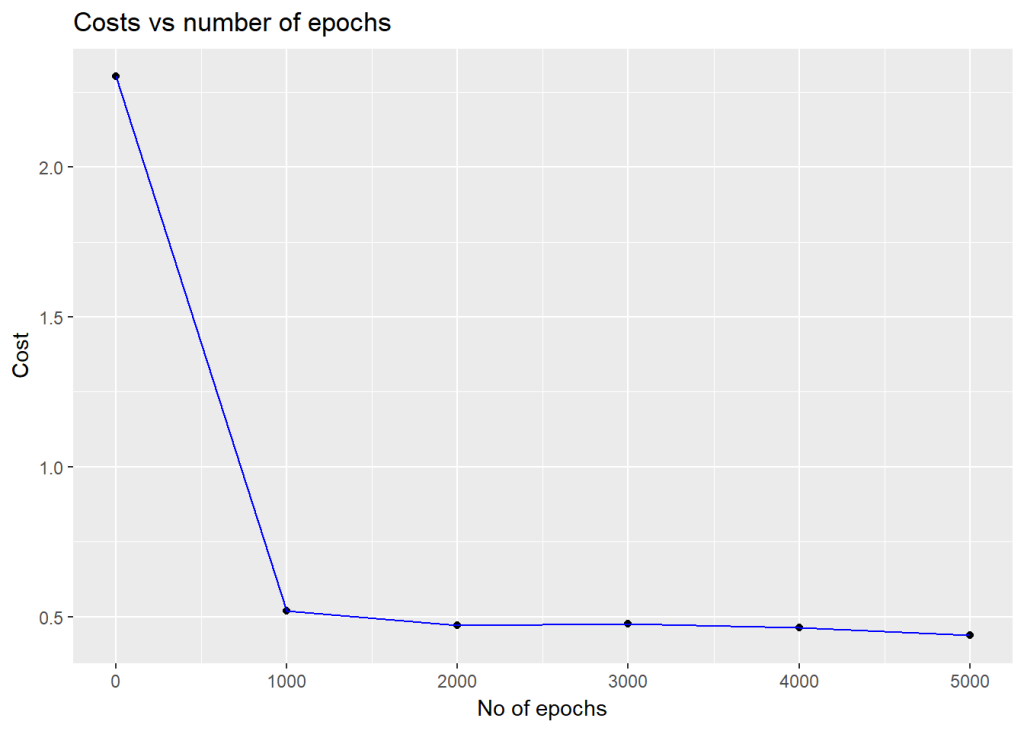
iterations <- seq(0,5000,1000)

costs=retvalsSGD$costs

df=data.frame(iterations,costs)

ggplot(df,aes(x=iterations,y=costs)) + geom\_point() + geom\_line(color="blue") +

ggtitle("Costs vs number of epochs") + xlab("No of epochs") + ylab("Cost")



**5.1c. Stochastic Gradient Descent with Adam – Octave**

source("DL7functions.m")

load('./mnist/mnist.txt.gz');

#Create a random permutatation from 1024

permutation = randperm(1024);

disp(length(permutation));

# Use this 1024 as the batch

X=trainX(permutation,:);

Y=trainY(permutation,:);

# Set layer dimensions

layersDimensions=[784, 15, 9, 10];

# Note the high value for epsilon.

#Otherwise GD with Adam does not seem to converge

# Perform SGD with Adam

[weights biases costs]=L\_Layer\_DeepModel\_SGD(X', Y', layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.1,

lrDecay=false,

decayRate=1,

lambd=0,

keep\_prob=1,

optimizer="adam",

beta=0.9,

beta1=0.9,

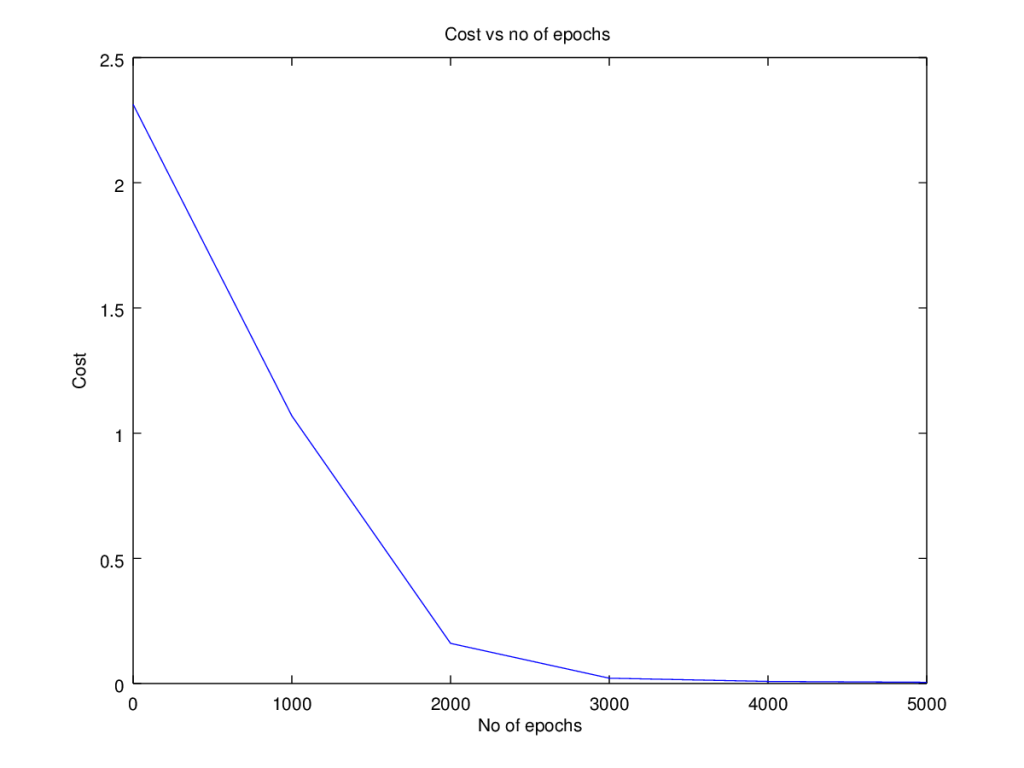
beta2=0.9,

epsilon=100,

mini\_batch\_size = 512,

num\_epochs = 5000);

plotCostVsEpochs(5000,costs)



**DLFunctions7.R**

|  |
| --- |
| library(ggplot2) |
|  | library(PRROC) |
|  | library(dplyr) |
|  |  |
|  | # Compute the sigmoid of a vector |
|  | sigmoid <- function(Z){ |
|  | A <- 1/(1+ exp(-Z)) |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  |  |
|  | } |
|  |  |
|  | # This is the older version. Very performance intensive |
|  | reluOld <-function(Z){ |
|  | A <- apply(Z, 1:2, function(x) max(0,x)) |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the Relu of a vector |
|  | relu <-function(Z){ |
|  | # Perform relu. Set values less that equal to 0 as 0 |
|  | Z[Z<0]=0 |
|  | A=Z |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the tanh activation of a vector |
|  | tanhActivation <- function(Z){ |
|  | A <- tanh(Z) |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Conmpute the softmax of a vector |
|  | softmax <- function(Z){ |
|  | # get unnormalized probabilities |
|  | exp\_scores = exp(t(Z)) |
|  | # normalize them for each example |
|  | A = exp\_scores / rowSums(exp\_scores) |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the detivative of Relu |
|  | # g'(z) = 1 if z >0 and 0 otherwise |
|  | reluDerivative <-function(dA, cache){ |
|  | Z <- cache |
|  | dZ <- dA |
|  | # Create a logical matrix of values > 0 |
|  | a <- Z > 0 |
|  | # When z <= 0, you should set dz to 0 as well. Perform an element wise multiply |
|  | dZ <- dZ \* a |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Compute the derivative of sigmoid |
|  | # Derivative g'(z) = a\* (1-a) |
|  | sigmoidDerivative <- function(dA, cache){ |
|  | Z <- cache |
|  | s <- 1/(1+exp(-Z)) |
|  | dZ <- dA \* s \* (1-s) |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Compute the derivative of tanh |
|  | # Derivative g'(z) = 1- a^2 |
|  | tanhDerivative <- function(dA, cache){ |
|  | Z = cache |
|  | a = tanh(Z) |
|  | dZ = dA \* (1 - a^2) |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Populate a matrix of 1s in rows where Y==1 |
|  | # This may need to be extended for K classes. Currently |
|  | # supports K=3 & K=10 |
|  | popMatrix <- function(Y,numClasses){ |
|  | a=rep(0,times=length(Y)) |
|  | Y1=matrix(a,nrow=length(Y),ncol=numClasses) |
|  | #Set the rows and columns as 1's where Y is the class value |
|  | if(numClasses==3){ |
|  | Y1[Y==0,1]=1 |
|  | Y1[Y==1,2]=1 |
|  | Y1[Y==2,3]=1 |
|  | } else if (numClasses==10){ |
|  | Y1[Y==0,1]=1 |
|  | Y1[Y==1,2]=1 |
|  | Y1[Y==2,3]=1 |
|  | Y1[Y==3,4]=1 |
|  | Y1[Y==4,5]=1 |
|  | Y1[Y==5,6]=1 |
|  | Y1[Y==6,7]=1 |
|  | Y1[Y==7,8]=1 |
|  | Y1[Y==8,9]=1 |
|  | Y1[Y==9,0]=1 |
|  | } |
|  | return(Y1) |
|  | } |
|  |  |
|  | softmaxDerivative <- function(dA, cache ,y,numTraining,numClasses){ |
|  | # Note : dA not used. dL/dZ = dL/dA \* dA/dZ = pi-yi |
|  | Z <- cache |
|  | # Compute softmax |
|  | exp\_scores = exp(t(Z)) |
|  | # normalize them for each example |
|  | probs = exp\_scores / rowSums(exp\_scores) |
|  | # Create a matrix of zeros |
|  | Y1=popMatrix(y,numClasses) |
|  | #a=rep(0,times=length(Y)) |
|  | #Y1=matrix(a,nrow=length(Y),ncol=numClasses) |
|  | #Set the rows and columns as 1's where Y is the class value |
|  | dZ = probs-Y1 |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Initialize the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  |  |
|  |  |
|  | # Initialize model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | initializeDeepModel <- function(layerDimensions){ |
|  | set.seed(2) |
|  |  |
|  | # Initialize empty list |
|  | layerParams <- list() |
|  |  |
|  | # Note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Indices in R start from 1 |
|  | for(l in 2:length(layersDimensions)){ |
|  | # Initialize a matrix of small random numbers of size l x l-1 |
|  | # Create random numbers of size l x l-1 |
|  | w=rnorm(layersDimensions[l]\*layersDimensions[l-1])\*0.01 |
|  | # Create a weight matrix of size l x l-1 with this initial weights and |
|  | # Add to list W1,W2... WL |
|  | layerParams[[paste('W',l-1,sep="")]] = matrix(w,nrow=layersDimensions[l], |
|  | ncol=layersDimensions[l-1]) |
|  | layerParams[[paste('b',l-1,sep="")]] = matrix(rep(0,layersDimensions[l]), |
|  | nrow=layersDimensions[l],ncol=1) |
|  | } |
|  | return(layerParams) |
|  | } |
|  |  |
|  |  |
|  |  |
|  | # He Initialization model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | # He initilization multiplies the random numbers with sqrt(2/layerDimensions[previouslayer]) |
|  | HeInitializeDeepModel <- function(layerDimensions){ |
|  | set.seed(2) |
|  |  |
|  | # Initialize empty list |
|  | layerParams <- list() |
|  |  |
|  | # Note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Indices in R start from 1 |
|  | for(l in 2:length(layersDimensions)){ |
|  | # Initialize a matrix of small random numbers of size l x l-1 |
|  | # Create random numbers of size l x l-1 |
|  | w=rnorm(layersDimensions[l]\*layersDimensions[l-1]) |
|  |  |
|  | # Create a weight matrix of size l x l-1 with this initial weights and |
|  | # Add to list W1,W2... WL |
|  | # He initialization - Divide by sqrt(2/layerDimensions[previous layer]) |
|  | layerParams[[paste('W',l-1,sep="")]] = matrix(w,nrow=layersDimensions[l], |
|  | ncol=layersDimensions[l-1])\*sqrt(2/layersDimensions[l-1]) |
|  | layerParams[[paste('b',l-1,sep="")]] = matrix(rep(0,layersDimensions[l]), |
|  | nrow=layersDimensions[l],ncol=1) |
|  | } |
|  | return(layerParams) |
|  | } |
|  |  |
|  | # XavInitializeDeepModel Initialization model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | # He initilization multiplies the random numbers with sqrt(1/layerDimensions[previouslayer]) |
|  | XavInitializeDeepModel <- function(layerDimensions){ |
|  | set.seed(2) |
|  |  |
|  | # Initialize empty list |
|  | layerParams <- list() |
|  |  |
|  | # Note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Indices in R start from 1 |
|  | for(l in 2:length(layersDimensions)){ |
|  | # Initialize a matrix of small random numbers of size l x l-1 |
|  | # Create random numbers of size l x l-1 |
|  | w=rnorm(layersDimensions[l]\*layersDimensions[l-1]) |
|  |  |
|  | # Create a weight matrix of size l x l-1 with this initial weights and |
|  | # Add to list W1,W2... WL |
|  | # He initialization - Divide by sqrt(2/layerDimensions[previous layer]) |
|  | layerParams[[paste('W',l-1,sep="")]] = matrix(w,nrow=layersDimensions[l], |
|  | ncol=layersDimensions[l-1])\*sqrt(1/layersDimensions[l-1]) |
|  | layerParams[[paste('b',l-1,sep="")]] = matrix(rep(0,layersDimensions[l]), |
|  | nrow=layersDimensions[l],ncol=1) |
|  | } |
|  | return(layerParams) |
|  | } |
|  |  |
|  | # Initialize velocity |
|  | # Input : parameters |
|  | # Returns: Initial velocity v |
|  | initializeVelocity <- function(parameters){ |
|  |  |
|  | L <- length(parameters)/2 |
|  | v <- list() |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for(l in 1:L){ |
|  | # Get the size of weight matrix |
|  | sz <- dim(parameters[[paste('W',l,sep="")]]) |
|  | v[[paste('dW',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | #Get the size of bias matrix |
|  | sz <- dim(parameters[[paste('b',l,sep="")]]) |
|  | v[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | } |
|  |  |
|  | return(v) |
|  | } |
|  |  |
|  |  |
|  | # Initialize RMSProp |
|  | # Input : parameters |
|  | # Returns: Initial RMSProp s |
|  | initializeRMSProp <- function(parameters){ |
|  |  |
|  | L <- length(parameters)/2 |
|  | s <- list() |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for(l in 1:L){ |
|  | # Get the size of weight matrix |
|  | sz <- dim(parameters[[paste('W',l,sep="")]]) |
|  | s[[paste('dW',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | #Get the size of bias matrix |
|  | sz <- dim(parameters[[paste('b',l,sep="")]]) |
|  | s[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | } |
|  |  |
|  | return(s) |
|  | } |
|  |  |
|  | # Initialize Adam |
|  | # Input : parameters |
|  | # Returns: Initial RMSProp s |
|  | initializeAdam <- function(parameters){ |
|  |  |
|  | L <- length(parameters)/2 |
|  | v <- list() |
|  | s <- list() |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for(l in 1:L){ |
|  | # Get the size of weight matrix |
|  | sz <- dim(parameters[[paste('W',l,sep="")]]) |
|  | v[[paste('dW',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | s[[paste('dW',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | #Get the size of bias matrix |
|  | sz <- dim(parameters[[paste('b',l,sep="")]]) |
|  | v[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | s[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]\*sz[2]), |
|  | nrow=sz[1],ncol=sz[2]) |
|  | } |
|  | retvals <- list("v"=v,"s"=s) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the activation at a layer 'l' for forward prop in a Deep Network |
|  | # Input : A\_prec - Activation of previous layer |
|  | # W,b - Weight and bias matrices and vectors |
|  | # activationFunc - Activation function - sigmoid, tanh, relu etc |
|  | # Returns : The Activation of this layer |
|  | # : |
|  | # Z = W \* X + b |
|  | # A = sigmoid(Z), A= Relu(Z), A= tanh(Z) |
|  | layerActivationForward <- function(A\_prev, W, b, activationFunc){ |
|  |  |
|  | # Compute Z |
|  | z = W %\*% A\_prev |
|  | # Broadcast the bias 'b' by column |
|  | Z <-sweep(z,1,b,'+') |
|  |  |
|  | forward\_cache <- list("A\_prev"=A\_prev, "W"=W, "b"=b) |
|  | # Compute the activation for sigmoid |
|  | if(activationFunc == "sigmoid"){ |
|  | vals = sigmoid(Z) |
|  | } else if (activationFunc == "relu"){ # Compute the activation for relu |
|  | vals = relu(Z) |
|  | } else if(activationFunc == 'tanh'){ # Compute the activation for tanh |
|  | vals = tanhActivation(Z) |
|  | } else if(activationFunc == 'softmax'){ |
|  | vals = softmax(Z) |
|  | } |
|  | # Create a list of forward and activation cache |
|  | cache <- list("forward\_cache"=forward\_cache, "activation\_cache"=vals[['Z']]) |
|  | retvals <- list("A"=vals[['A']],"cache"=cache) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the forward propagation for layers 1..L |
|  | # Input : X - Input Features |
|  | # paramaters: Weights and biases |
|  | # hiddenActivationFunc - elu/sigmoid/tanh |
|  | # outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # Returns : AL |
|  | # caches |
|  | # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and sigmoid actiovation at layer L |
|  | forwardPropagationDeep <- function(X, parameters,keep\_prob=1, hiddenActivationFunc='relu', |
|  | outputActivationFunc='sigmoid'){ |
|  | caches <- list() |
|  | dropoutMat <- list() |
|  | # Set A to X (A0) |
|  | A <- X |
|  | L <- length(parameters)/2 # number of layers in the neural network |
|  | # Loop through from layer 1 to upto layer L |
|  | for(l in 1:(L-1)){ |
|  | A\_prev <- A |
|  | # Zi = Wi x Ai-1 + bi and Ai = g(Zi) |
|  | # Set W and b for layer 'l' |
|  | # Loop throug from W1,W2... WL-1 |
|  | W <- parameters[[paste("W",l,sep="")]] |
|  | b <- parameters[[paste("b",l,sep="")]] |
|  | # Compute the forward propagation through layer 'l' using the activation function |
|  | actForward <- layerActivationForward(A\_prev, |
|  | W, |
|  | b, |
|  | activationFunc = hiddenActivationFunc) |
|  | A <- actForward[['A']] |
|  | # Append the cache A\_prev,W,b, Z |
|  | caches[[l]] <-actForward |
|  |  |
|  | # Randomly drop some activation units |
|  | # Create a matrix as the same shape as A |
|  | set.seed(1) |
|  | i=dim(A)[1] |
|  | j=dim(A)[2] |
|  | a<-rnorm(i\*j) |
|  | # Normalize a between 0 and 1 |
|  | a = (a - min(a))/(max(a) - min(a)) |
|  | # Create a matrix of D |
|  | D <- matrix(a,nrow=i, ncol=j) |
|  | # Find D which is less than equal to keep\_prob |
|  | D <- D < keep\_prob |
|  | # Remove some A's |
|  | A <- A \* D |
|  | # Divide by keep\_prob to keep expected value same |
|  | A <- A/keep\_prob |
|  | dropoutMat[[paste("D",l,sep="")]] <- D |
|  | } |
|  |  |
|  | # Since this is binary classification use the sigmoid activation function in |
|  | # last layer |
|  | # Set the weights and biases for the last layer |
|  | W <- parameters[[paste("W",L,sep="")]] |
|  | b <- parameters[[paste("b",L,sep="")]] |
|  | # Last layer |
|  | actForward = layerActivationForward(A, W, b, activationFunc = outputActivationFunc) |
|  | AL <- actForward[['A']] |
|  | # Append the output of this forward propagation through the last layer |
|  | caches[[L]] <- actForward |
|  | # Create a list of the final output and the caches |
|  | fwdPropDeep <- list("AL"=AL,"caches"=caches,"dropoutMat"=dropoutMat) |
|  | return(fwdPropDeep) |
|  |  |
|  | } |
|  |  |
|  | pickColumns <- function(AL,Y,numClasses){ |
|  | if(numClasses==3){ |
|  | a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3]) |
|  | } |
|  | else if (numClasses==10){ |
|  | a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3],AL[Y==3,4],AL[Y==4,5], |
|  | AL[Y==5,6],AL[Y==6,7],AL[Y==7,8],AL[Y==8,9],AL[Y==9,10]) |
|  | } |
|  | return(a) |
|  | } |
|  |  |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # :outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : numClasses |
|  | # Output: cost |
|  | computeCost <- function(AL,Y,outputActivationFunc="sigmoid",numClasses=3){ |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | m= length(Y) |
|  | cost=-1/m\*sum(Y\*log(AL) + (1-Y)\*log(1-AL)) |
|  |  |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | # Select the elements where the y values are 0, 1 or 2 and make a vector |
|  | # Pick columns |
|  | #a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3]) |
|  | m= length(Y) |
|  | a =pickColumns(AL,Y,numClasses) |
|  | #a = c(A2[y=k,k+1]) |
|  | # Take log |
|  | correct\_probs = -log(a) |
|  |  |
|  | # Compute loss |
|  | cost= sum(correct\_probs)/m |
|  | } |
|  | return(cost) |
|  | } |
|  |  |
|  |  |
|  | # Compute the cost with Regularization |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # :outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : numClasses |
|  | # Output: cost |
|  | computeCostWithReg <- function(parameters, AL,Y,lambd, outputActivationFunc="sigmoid",numClasses=3){ |
|  |  |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | m= length(Y) |
|  | cost=-1/m\*sum(Y\*log(AL) + (1-Y)\*log(1-AL)) |
|  |  |
|  | # Regularization cost |
|  | L <- length(parameters)/2 |
|  | L2RegularizationCost=0 |
|  | for(l in 1:L){ |
|  | L2RegularizationCost = L2RegularizationCost + |
|  | sum(parameters[[paste("W",l,sep="")]]^2) |
|  | } |
|  | L2RegularizationCost = (lambd/(2\*m))\*L2RegularizationCost |
|  | cost = cost + L2RegularizationCost |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | # Select the elements where the y values are 0, 1 or 2 and make a vector |
|  | # Pick columns |
|  | #a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3]) |
|  | m= length(Y) |
|  | a =pickColumns(AL,Y,numClasses) |
|  | #a = c(A2[y=k,k+1]) |
|  | # Take log |
|  | correct\_probs = -log(a) |
|  | # Compute loss |
|  | cost= sum(correct\_probs)/m |
|  |  |
|  | # Regularization cost |
|  | L <- length(parameters)/2 |
|  | L2RegularizationCost=0 |
|  | # Add L2 norm |
|  | for(l in 1:L){ |
|  | L2RegularizationCost = L2RegularizationCost + |
|  | sum(parameters[[paste("W",l,sep="")]]^2) |
|  | } |
|  | L2RegularizationCost = (lambd/(2\*m))\*L2RegularizationCost |
|  | cost = cost + L2RegularizationCost |
|  | } |
|  | return(cost) |
|  | } |
|  |  |
|  | # Compute the backpropagation through a layer |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # # activationFunc |
|  | # # numClasses |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  |  |
|  | layerActivationBackward <- function(dA, cache, Y, activationFunc,numClasses){ |
|  | # Get A\_prev,W,b |
|  | forward\_cache <-cache[['forward\_cache']] |
|  | activation\_cache <- cache[['activation\_cache']] |
|  | A\_prev <- forward\_cache[['A\_prev']] |
|  | numtraining = dim(A\_prev)[2] |
|  | # Get Z |
|  | activation\_cache <- cache[['activation\_cache']] |
|  | if(activationFunc == "relu"){ |
|  | dZ <- reluDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "sigmoid"){ |
|  | dZ <- sigmoidDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "tanh"){ |
|  | dZ <- tanhDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "softmax"){ |
|  | dZ <- softmaxDerivative(dA, activation\_cache,Y,numtraining,numClasses) |
|  | } |
|  |  |
|  | if (activationFunc == 'softmax'){ |
|  | W <- forward\_cache[['W']] |
|  | b <- forward\_cache[['b']] |
|  | dW = 1/numtraining \* A\_prev%\*%dZ |
|  | db = 1/numtraining\* matrix(colSums(dZ),nrow=1,ncol=numClasses) |
|  | dA\_prev = dZ %\*%W |
|  | } else { |
|  | W <- forward\_cache[['W']] |
|  | b <- forward\_cache[['b']] |
|  | numtraining = dim(A\_prev)[2] |
|  |  |
|  | dW = 1/numtraining \* dZ %\*% t(A\_prev) |
|  | db = 1/numtraining \* rowSums(dZ) |
|  | dA\_prev = t(W) %\*% dZ |
|  | } |
|  | retvals <- list("dA\_prev"=dA\_prev,"dW"=dW,"db"=db) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the backpropagation through a layer with Regularization |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # # activationFunc |
|  | # # numClasses |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  |  |
|  | layerActivationBackwardWithReg <- function(dA, cache, Y, lambd, activationFunc,numClasses){ |
|  | # Get A\_prev,W,b |
|  | forward\_cache <-cache[['forward\_cache']] |
|  | activation\_cache <- cache[['activation\_cache']] |
|  | A\_prev <- forward\_cache[['A\_prev']] |
|  | numtraining = dim(A\_prev)[2] |
|  | # Get Z |
|  | activation\_cache <- cache[['activation\_cache']] |
|  | if(activationFunc == "relu"){ |
|  | dZ <- reluDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "sigmoid"){ |
|  | dZ <- sigmoidDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "tanh"){ |
|  | dZ <- tanhDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "softmax"){ |
|  | dZ <- softmaxDerivative(dA, activation\_cache,Y,numtraining,numClasses) |
|  | } |
|  |  |
|  | if (activationFunc == 'softmax'){ |
|  | W <- forward\_cache[['W']] |
|  | b <- forward\_cache[['b']] |
|  | # Add the regularization factor |
|  | dW = 1/numtraining \* A\_prev%\*%dZ + (lambd/numtraining) \* t(W) |
|  | db = 1/numtraining\* matrix(colSums(dZ),nrow=1,ncol=numClasses) |
|  | dA\_prev = dZ %\*%W |
|  | } else { |
|  | W <- forward\_cache[['W']] |
|  | b <- forward\_cache[['b']] |
|  | numtraining = dim(A\_prev)[2] |
|  | # Add the regularization factor |
|  | dW = 1/numtraining \* dZ %\*% t(A\_prev) + (lambd/numtraining) \* W |
|  | db = 1/numtraining \* rowSums(dZ) |
|  | dA\_prev = t(W) %\*% dZ |
|  | } |
|  | retvals <- list("dA\_prev"=dA\_prev,"dW"=dW,"db"=db) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the backpropagation for 1 cycle through all layers |
|  | # Input : AL: Output of L layer Network - weights |
|  | # # Y Real output |
|  | # # caches -- list of caches containing: |
|  | # every cache of layerActivationForward() with "relu"/"tanh" |
|  | # #(it's caches[l], for l in range(L-1) i.e l = 0...L-2) |
|  | # #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1]) |
|  | # hiddenActivationFunc - Activation function at hidden layers - relu/tanh/sigmoid |
|  | # outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # |
|  | # Returns: |
|  | # gradients -- A dictionary with the gradients |
|  | # gradients["dA" + str(l)] = ... |
|  | # |
|  | backwardPropagationDeep <- function(AL, Y, caches,dropoutMat, lambd=0, keep\_prob=0, hiddenActivationFunc='relu', |
|  | outputActivationFunc="sigmoid",numClasses){ |
|  | #initialize the gradients |
|  | gradients = list() |
|  | # Set the number of layers |
|  | L = length(caches) |
|  | numTraining = dim(AL)[2] |
|  |  |
|  | if(outputActivationFunc == "sigmoid") |
|  | # Initializing the backpropagation |
|  | # dl/dAL= -(y/a) - ((1-y)/(1-a)) - At the output layer |
|  | dAL = -( (Y/AL) -(1 - Y)/(1 - AL)) |
|  | else if(outputActivationFunc == "softmax"){ |
|  | dAL=0 |
|  | Y=t(Y) |
|  | } |
|  |  |
|  | # Get the gradients at the last layer |
|  | # Inputs: "AL, Y, caches". |
|  | # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"] |
|  | # Start with Layer L |
|  | # Get the current cache |
|  | current\_cache = caches[[L]]$cache |
|  | if (lambd==0){ |
|  | retvals <- layerActivationBackward(dAL, current\_cache, Y, |
|  | activationFunc = outputActivationFunc,numClasses) |
|  | } else { |
|  | retvals = layerActivationBackwardWithReg(dAL, current\_cache, Y, lambd, |
|  | activationFunc = outputActivationFunc,numClasses) |
|  | } |
|  |  |
|  |  |
|  |  |
|  | #Note: Take the transpose of dA |
|  | if(outputActivationFunc =="sigmoid") |
|  | gradients[[paste("dA",L,sep="")]] <- retvals[['dA\_prev']] |
|  | else if(outputActivationFunc =="softmax") |
|  | gradients[[paste("dA",L,sep="")]] <- t(retvals[['dA\_prev']]) |
|  |  |
|  | gradients[[paste("dW",L,sep="")]] <- retvals[['dW']] |
|  | gradients[[paste("db",L,sep="")]] <- retvals[['db']] |
|  |  |
|  |  |
|  |  |
|  | # Traverse in the reverse direction |
|  | for(l in (L-1):1){ |
|  | # Compute the gradients for L-1 to 1 for Relu/tanh |
|  | # Inputs: "gradients["dA" + str(l + 2)], caches". |
|  | # Outputs: "gradients["dA" + str(l + 1)] , gradients["dW" + str(l + 1)] , gradients["db" + str(l + 1)] |
|  | current\_cache = caches[[l]]$cache |
|  | if (lambd==0){ |
|  | # Get the dropout matrix |
|  | D <-dropoutMat[[paste("D",l,sep="")]] |
|  | # Multiply gradient with dropout matrix |
|  | gradients[[paste('dA',l+1,sep="")]] = gradients[[paste('dA',l+1,sep="")]] \*D |
|  | # Divide by keep\_prob to keep expected value same |
|  | gradients[[paste('dA',l+1,sep="")]] = gradients[[paste('dA',l+1,sep="")]]/keep\_prob |
|  | retvals = layerActivationBackward(gradients[[paste('dA',l+1,sep="")]], |
|  | current\_cache, Y, |
|  | activationFunc = hiddenActivationFunc) |
|  | } else { |
|  | retvals = layerActivationBackwardWithReg(gradients[[paste('dA',l+1,sep="")]], |
|  | current\_cache, Y, lambd, |
|  | activationFunc = hiddenActivationFunc) |
|  | } |
|  |  |
|  | gradients[[paste("dA",l,sep="")]] <-retvals[['dA\_prev']] |
|  | gradients[[paste("dW",l,sep="")]] <- retvals[['dW']] |
|  | gradients[[paste("db",l,sep="")]] <- retvals[['db']] |
|  | } |
|  |  |
|  |  |
|  |  |
|  | return(gradients) |
|  | } |
|  |  |
|  |  |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | gradientDescent <- function(parameters, gradients, learningRate,outputActivationFunc="sigmoid"){ |
|  |  |
|  | L = length(parameters)/2 # number of layers in the neural network |
|  | # Update rule for each parameter. Use a for loop. |
|  | for(l in 1:(L-1)){ |
|  | parameters[[paste("W",l,sep="")]] = parameters[[paste("W",l,sep="")]] - |
|  | learningRate\* gradients[[paste("dW",l,sep="")]] |
|  | parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] - |
|  | learningRate\* gradients[[paste("db",l,sep="")]] |
|  | } |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* gradients[[paste("dW",L,sep="")]] |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* gradients[[paste("db",L,sep="")]] |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("dW",L,sep="")]]) |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("db",L,sep="")]]) |
|  |  |
|  | } |
|  | return(parameters) |
|  | } |
|  |  |
|  | # Perform Gradient Descent with momentum |
|  | # Input : Weights and biases |
|  | # : beta |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | gradientDescentWithMomentum <- function(parameters, gradients,v, beta, learningRate,outputActivationFunc="sigmoid"){ |
|  |  |
|  | L = length(parameters)/2 # number of layers in the neural network |
|  |  |
|  | # Update rule for each parameter. Use a for loop. |
|  | for(l in 1:(L-1)){ |
|  | # Compute velocities |
|  | # v['dWk'] = beta \*v['dWk'] + (1-beta)\*dWk |
|  | v[[paste("dW",l, sep="")]] = beta\*v[[paste("dW",l, sep="")]] + |
|  | (1-beta) \* gradients[[paste('dW',l,sep="")]] |
|  | v[[paste("db",l, sep="")]] = beta\*v[[paste("db",l, sep="")]] + |
|  | (1-beta) \* gradients[[paste('db',l,sep="")]] |
|  |  |
|  | parameters[[paste("W",l,sep="")]] = parameters[[paste("W",l,sep="")]] - |
|  | learningRate\* v[[paste("dW",l, sep="")]] |
|  | parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] - |
|  | learningRate\* v[[paste("db",l, sep="")]] |
|  | } |
|  |  |
|  | # Compute for the Lth layer |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | v[[paste("dW",L, sep="")]] = beta\*v[[paste("dW",L, sep="")]] + |
|  | (1-beta) \* gradients[[paste('dW',L,sep="")]] |
|  | v[[paste("db",L, sep="")]] = beta\*v[[paste("db",L, sep="")]] + |
|  | (1-beta) \* gradients[[paste('db',L,sep="")]] |
|  |  |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* v[[paste("dW",l, sep="")]] |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* v[[paste("db",l, sep="")]] |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | v[[paste("dW",L, sep="")]] = beta\*v[[paste("dW",L, sep="")]] + |
|  | (1-beta) \* t(gradients[[paste('dW',L,sep="")]]) |
|  | v[[paste("db",L, sep="")]] = beta\*v[[paste("db",L, sep="")]] + |
|  | (1-beta) \* t(gradients[[paste('db',L,sep="")]]) |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("dW",L,sep="")]]) |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("db",L,sep="")]]) |
|  | } |
|  | return(parameters) |
|  | } |
|  |  |
|  |  |
|  | # Perform Gradient Descent with RMSProp |
|  | # Input : Weights and biases |
|  | # : beta1 |
|  | # : epsilon |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | gradientDescentWithRMSProp <- function(parameters, gradients,s, beta1, epsilon, learningRate,outputActivationFunc="sigmoid"){ |
|  | L = length(parameters)/2 # number of layers in the neural network |
|  | # Update rule for each parameter. Use a for loop. |
|  | for(l in 1:(L-1)){ |
|  | # Compute RMSProp |
|  | # s['dWk'] = beta1 \*s['dWk'] + (1-beta1)\*dWk\*\*2/sqrt(s['dWk']) |
|  | # Element wise multiply of gradients |
|  | s[[paste("dW",l, sep="")]] = beta1\*s[[paste("dW",l, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('dW',l,sep="")]] \* gradients[[paste('dW',l,sep="")]] |
|  | s[[paste("db",l, sep="")]] = beta1\*s[[paste("db",l, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('db',l,sep="")]] \* gradients[[paste('db',l,sep="")]] |
|  |  |
|  | parameters[[paste("W",l,sep="")]] = parameters[[paste("W",l,sep="")]] - |
|  | learningRate \* gradients[[paste('dW',l,sep="")]]/sqrt(s[[paste("dW",l, sep="")]]+epsilon) |
|  | parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] - |
|  | learningRate\*gradients[[paste('db',l,sep="")]]/sqrt(s[[paste("db",l, sep="")]]+epsilon) |
|  | } |
|  |  |
|  | # Compute for the Lth layer |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | s[[paste("dW",L, sep="")]] = beta1\*s[[paste("dW",L, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('dW',L,sep="")]] \*gradients[[paste('dW',L,sep="")]] |
|  | s[[paste("db",L, sep="")]] = beta1\*s[[paste("db",L, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('db',L,sep="")]] \* gradients[[paste('db',L,sep="")]] |
|  |  |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* gradients[[paste('dW',l,sep="")]]/sqrt(s[[paste("dW",L, sep="")]]+epsilon) |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* gradients[[paste('db',l,sep="")]]/sqrt( s[[paste("db",L, sep="")]]+epsilon) |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | s[[paste("dW",L, sep="")]] = beta1\*s[[paste("dW",L, sep="")]] + |
|  | (1-beta1) \* t(gradients[[paste('dW',L,sep="")]]) \* t(gradients[[paste('dW',L,sep="")]]) |
|  | s[[paste("db",L, sep="")]] = beta1\*s[[paste("db",L, sep="")]] + |
|  | (1-beta1) \* t(gradients[[paste('db',L,sep="")]]) \* t(gradients[[paste('db',L,sep="")]]) |
|  |  |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("dW",L,sep="")]])/sqrt(s[[paste("dW",L, sep="")]]+epsilon) |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* t(gradients[[paste("db",L,sep="")]])/sqrt( s[[paste("db",L, sep="")]]+epsilon) |
|  | } |
|  | return(parameters) |
|  | } |
|  |  |
|  |  |
|  | # Perform Gradient Descent with Adam |
|  | # Input : Weights and biases |
|  | # : beta1 |
|  | # : epsilon |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | gradientDescentWithAdam <- function(parameters, gradients,v, s, t, |
|  | beta1=0.9, beta2=0.999, epsilon=10^-8, learningRate=0.1,outputActivationFunc="sigmoid"){ |
|  |  |
|  | L = length(parameters)/2 # number of layers in the neural network |
|  | v\_corrected <- list() |
|  | s\_corrected <- list() |
|  | # Update rule for each parameter. Use a for loop. |
|  | for(l in 1:(L-1)){ |
|  | # v['dWk'] = beta \*v['dWk'] + (1-beta)\*dWk |
|  | v[[paste("dW",l, sep="")]] = beta1\*v[[paste("dW",l, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('dW',l,sep="")]] |
|  | v[[paste("db",l, sep="")]] = beta1\*v[[paste("db",l, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('db',l,sep="")]] |
|  |  |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | v\_corrected[[paste("dW",l, sep="")]] = v[[paste("dW",l, sep="")]]/(1-beta1^t) |
|  | v\_corrected[[paste("db",l, sep="")]] = v[[paste("db",l, sep="")]]/(1-beta1^t) |
|  |  |
|  |  |
|  | # Element wise multiply of gradients |
|  | s[[paste("dW",l, sep="")]] = beta2\*s[[paste("dW",l, sep="")]] + |
|  | (1-beta2) \* gradients[[paste('dW',l,sep="")]] \* gradients[[paste('dW',l,sep="")]] |
|  | s[[paste("db",l, sep="")]] = beta2\*s[[paste("db",l, sep="")]] + |
|  | (1-beta2) \* gradients[[paste('db',l,sep="")]] \* gradients[[paste('db',l,sep="")]] |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | s\_corrected[[paste("dW",l, sep="")]] = s[[paste("dW",l, sep="")]]/(1-beta2^t) |
|  | s\_corrected[[paste("db",l, sep="")]] = s[[paste("db",l, sep="")]]/(1-beta2^t) |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(s\_corrected[[paste("dW",l, sep="")]]+epsilon) |
|  | d2=sqrt(s\_corrected[[paste("db",l, sep="")]]+epsilon) |
|  |  |
|  | parameters[[paste("W",l,sep="")]] = parameters[[paste("W",l,sep="")]] - |
|  | learningRate \* v\_corrected[[paste("dW",l, sep="")]]/d1 |
|  | parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] - |
|  | learningRate\*v\_corrected[[paste("db",l, sep="")]]/d2 |
|  | } |
|  |  |
|  | # Compute for the Lth layer |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | v[[paste("dW",L, sep="")]] = beta1\*v[[paste("dW",L, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('dW',L,sep="")]] |
|  | v[[paste("db",L, sep="")]] = beta1\*v[[paste("db",L, sep="")]] + |
|  | (1-beta1) \* gradients[[paste('db',L,sep="")]] |
|  |  |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | v\_corrected[[paste("dW",L, sep="")]] = v[[paste("dW",L, sep="")]]/(1-beta1^t) |
|  | v\_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-beta1^t) |
|  |  |
|  |  |
|  | # Element wise multiply of gradients |
|  | s[[paste("dW",L, sep="")]] = beta2\*s[[paste("dW",L, sep="")]] + |
|  | (1-beta2) \* gradients[[paste('dW',L,sep="")]] \* gradients[[paste('dW',L,sep="")]] |
|  | s[[paste("db",L, sep="")]] = beta2\*s[[paste("db",L, sep="")]] + |
|  | (1-beta2) \* gradients[[paste('db',L,sep="")]] \* gradients[[paste('db',L,sep="")]] |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | s\_corrected[[paste("dW",L, sep="")]] = s[[paste("dW",L, sep="")]]/(1-beta2^t) |
|  | s\_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-beta2^t) |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(s\_corrected[[paste("dW",L, sep="")]]+epsilon) |
|  | d2=sqrt(s\_corrected[[paste("db",L, sep="")]]+epsilon) |
|  |  |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate \* v\_corrected[[paste("dW",L, sep="")]]/d1 |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\*v\_corrected[[paste("db",L, sep="")]]/d2 |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | v[[paste("dW",L, sep="")]] = beta1\*v[[paste("dW",L, sep="")]] + |
|  | (1-beta1) \* t(gradients[[paste('dW',L,sep="")]]) |
|  | v[[paste("db",L, sep="")]] = beta1\*v[[paste("db",L, sep="")]] + |
|  | (1-beta1) \* t(gradients[[paste('db',L,sep="")]]) |
|  |  |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | v\_corrected[[paste("dW",L, sep="")]] = v[[paste("dW",L, sep="")]]/(1-beta1^t) |
|  | v\_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-beta1^t) |
|  |  |
|  |  |
|  | # Element wise multiply of gradients |
|  | s[[paste("dW",L, sep="")]] = beta2\*s[[paste("dW",L, sep="")]] + |
|  | (1-beta2) \* t(gradients[[paste('dW',L,sep="")]]) \* t(gradients[[paste('dW',L,sep="")]]) |
|  | s[[paste("db",L, sep="")]] = beta2\*s[[paste("db",L, sep="")]] + |
|  | (1-beta2) \* t(gradients[[paste('db',L,sep="")]]) \* t(gradients[[paste('db',L,sep="")]]) |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | s\_corrected[[paste("dW",L, sep="")]] = s[[paste("dW",L, sep="")]]/(1-beta2^t) |
|  | s\_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-beta2^t) |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(s\_corrected[[paste("dW",L, sep="")]]+epsilon) |
|  | d2=sqrt(s\_corrected[[paste("db",L, sep="")]]+epsilon) |
|  |  |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate \* v\_corrected[[paste("dW",L, sep="")]]/d1 |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\*v\_corrected[[paste("db",L, sep="")]]/d2 |
|  | } |
|  | return(parameters) |
|  | } |
|  |  |
|  | # Execute a L layer Deep learning model |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : num of iterations |
|  | #output : Updated weights after each iteration |
|  |  |
|  | L\_Layer\_DeepModel <- function(X, Y, layersDimensions, |
|  | hiddenActivationFunc='relu', |
|  | outputActivationFunc= 'sigmoid', |
|  | learningRate = 0.5, |
|  | lambd=0, |
|  | keep\_prob=1, |
|  | numIterations = 10000, |
|  | initType="default", |
|  | print\_cost=False){ |
|  | #Initialize costs vector as NULL |
|  | costs <- NULL |
|  |  |
|  | # Parameters initialization. |
|  | if (initType=="He"){ |
|  | parameters =HeInitializeDeepModel(layersDimensions) |
|  | } else if (initType=="Xav"){ |
|  | parameters =XavInitializeDeepModel(layersDimensions) |
|  | } |
|  | else{ |
|  | parameters = initializeDeepModel(layersDimensions) |
|  | } |
|  |  |
|  |  |
|  | # Loop (gradient descent) |
|  | for( i in 0:numIterations){ |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID/SOFTMAX. |
|  | retvals = forwardPropagationDeep(X, parameters,keep\_prob, hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc) |
|  | AL <- retvals[['AL']] |
|  | caches <- retvals[['caches']] |
|  | dropoutMat <- retvals[['dropoutMat']] |
|  |  |
|  | # Compute cost. |
|  | if(lambd==0){ |
|  | cost <- computeCost(AL, Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  | } else { |
|  | cost <- computeCostWithReg(parameters, AL, Y,lambd, outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  | } |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd, keep\_prob, hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  |  |
|  |  |
|  | if(i%%1000 == 0){ |
|  | costs=c(costs,cost) |
|  | print(cost) |
|  | } |
|  | } |
|  |  |
|  | retvals <- list("parameters"=parameters,"costs"=costs) |
|  |  |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Execute a L layer Deep learning model with Stochastic Gradient descent |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : mini\_batch\_size |
|  | # : num of epochs |
|  | #output : Updated weights after each iteration |
|  | L\_Layer\_DeepModel\_SGD <- function(X, Y, layersDimensions, |
|  | hiddenActivationFunc='relu', |
|  | outputActivationFunc= 'sigmoid', |
|  | learningRate = .3, |
|  | lrDecay=FALSE, |
|  | decayRate=1, |
|  | lambd=0, |
|  | keep\_prob=1, |
|  | optimizer="gd", |
|  | beta=0.9, |
|  | beta1=0.9, |
|  | beta2=0.999, |
|  | epsilon=10^-8, |
|  | mini\_batch\_size = 64, |
|  | num\_epochs = 2500, |
|  | print\_cost=False){ |
|  |  |
|  |  |
|  |  |
|  | print("hello") |
|  | cat("learningRate= ",learningRate) |
|  | cat("\n") |
|  | cat("lambd=",lambd) |
|  | cat("\n") |
|  | cat("keep\_prob=",keep\_prob) |
|  | cat("\n") |
|  | cat("optimizer=",optimizer) |
|  | cat("\n") |
|  | cat("lrDecay=",lrDecay) |
|  | cat("\n") |
|  | cat("decayRate=",decayRate) |
|  | cat("\n") |
|  | cat("beta=",beta) |
|  | cat("\n") |
|  | cat("beta1=",beta1) |
|  | cat("\n") |
|  | cat("beta2=",beta2) |
|  | cat("\n") |
|  | cat("epsilon=",epsilon) |
|  | cat("\n") |
|  | cat("mini\_batch\_size=",mini\_batch\_size) |
|  | cat("\n") |
|  | cat("num\_epochs=",num\_epochs) |
|  | cat("\n") |
|  | set.seed(1) |
|  | #Initialize costs vector as NULL |
|  | costs <- NULL |
|  | t <- 0 |
|  | # Parameters initialization. |
|  | parameters = initializeDeepModel(layersDimensions) |
|  |  |
|  |  |
|  | #Initialize the optimizer |
|  |  |
|  | if(optimizer == "momentum"){ |
|  | v <-initializeVelocity(parameters) |
|  | } else if(optimizer == "rmsprop"){ |
|  | s <-initializeRMSProp(parameters) |
|  | } else if (optimizer == "adam"){ |
|  | adamVals <-initializeAdam(parameters) |
|  | } |
|  |  |
|  | seed=10 |
|  |  |
|  | # Loop for number of epochs |
|  | for( i in 0:num\_epochs){ |
|  | seed=seed+1 |
|  | minibatches = random\_mini\_batches(X, Y, mini\_batch\_size, seed) |
|  |  |
|  | for(batch in 1:length(minibatches)){ |
|  |  |
|  | mini\_batch\_X=minibatches[[batch]][['mini\_batch\_X']] |
|  | mini\_batch\_Y=minibatches[[batch]][['mini\_batch\_Y']] |
|  | # Forward propagation: |
|  | retvals = forwardPropagationDeep(mini\_batch\_X, parameters,keep\_prob, hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc) |
|  | AL <- retvals[['AL']] |
|  | caches <- retvals[['caches']] |
|  | dropoutMat <- retvals[['dropoutMat']] |
|  |  |
|  | # Compute cost. |
|  | # Compute cost. |
|  | if(lambd==0){ |
|  | cost <- computeCost(AL, mini\_batch\_Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  | } else { |
|  | cost <- computeCostWithReg(parameters, AL, Y,lambd, outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  | } |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, mini\_batch\_Y, caches, dropoutMat, lambd, keep\_prob, hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  |  |
|  | if(optimizer == "gd"){ |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  | }else if(optimizer == "momentum"){ |
|  | # Update parameters with Momentum |
|  | parameters = gradientDescentWithMomentum(parameters, gradients,v,beta, learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  |  |
|  | } else if(optimizer == "rmsprop"){ |
|  | # Update parameters with RMSProp |
|  | parameters = gradientDescentWithRMSProp(parameters, gradients,s,beta1, epsilon,learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  |  |
|  | } else if(optimizer == "adam"){ |
|  | # Update parameters with Adam |
|  | #Get v and s |
|  | t <- t+1 |
|  | v <- adamVals[['v']] |
|  | s <- adamVals[['s']] |
|  | parameters = gradientDescentWithAdam(parameters, gradients,v, s,t, beta1,beta2, epsilon,learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  | } |
|  | } |
|  |  |
|  | if(i%%1000 == 0){ |
|  | costs=c(costs,cost) |
|  | print(cost) |
|  | } |
|  | if(lrDecay==TRUE){ |
|  | learningRate = decayRate^(num\_epochs/1000) \* learningRate |
|  | } |
|  | } |
|  |  |
|  | retvals <- list("parameters"=parameters,"costs"=costs) |
|  |  |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Predict the output for given input |
|  | # Input : parameters |
|  | # : X |
|  | # Output: predictions |
|  | predict <- function(parameters, X,keep\_prob=1, hiddenActivationFunc='relu'){ |
|  |  |
|  | fwdProp <- forwardPropagationDeep(X, parameters,keep\_prob, hiddenActivationFunc) |
|  | predictions <- fwdProp$AL>0.5 |
|  |  |
|  | return (predictions) |
|  | } |
|  |  |
|  | # Plot a decision boundary |
|  | # This function uses ggplot2 |
|  | plotDecisionBoundary <- function(z,retvals,keep\_prob=1,hiddenActivationFunc="sigmoid",lr=0.5){ |
|  | # Find the minimum and maximum for the data |
|  | xmin<-min(z[,1]) |
|  | xmax<-max(z[,1]) |
|  | ymin<-min(z[,2]) |
|  | ymax<-max(z[,2]) |
|  |  |
|  | # Create a grid of values |
|  | a=seq(xmin,xmax,length=100) |
|  | b=seq(ymin,ymax,length=100) |
|  | grid <- expand.grid(x=a, y=b) |
|  | colnames(grid) <- c('x1', 'x2') |
|  | grid1 <-t(grid) |
|  | # Predict the output for this grid |
|  | q <-predict(retvals$parameters,grid1,keep\_prob=1, hiddenActivationFunc) |
|  | q1 <- t(data.frame(q)) |
|  | q2 <- as.numeric(q1) |
|  | grid2 <- cbind(grid,q2) |
|  | colnames(grid2) <- c('x1', 'x2','q2') |
|  |  |
|  | z1 <- data.frame(z) |
|  | names(z1) <- c("x1","x2","y") |
|  | atitle=paste("Decision boundary for learning rate:",lr) |
|  | # Plot the contour of the boundary |
|  | ggplot(z1) + |
|  | geom\_point(data = z1, aes(x = x1, y = x2, color = y)) + |
|  | stat\_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2), alpha = 0.9)+ |
|  | ggtitle(atitle) + scale\_colour\_gradientn(colours = brewer.pal(10, "Spectral")) |
|  | } |
|  |  |
|  | # Predict the probability scores for given data set |
|  | # Input : parameters |
|  | # : X |
|  | # Output: probability of output |
|  | computeScores <- function(parameters, X,hiddenActivationFunc='relu'){ |
|  |  |
|  | fwdProp <- forwardPropagationDeep(X, parameters,hiddenActivationFunc) |
|  | scores <- fwdProp$AL |
|  |  |
|  | return (scores) |
|  | } |
|  |  |
|  |  |
|  | random\_mini\_batches <- function(X, Y, miniBatchSize = 64, seed = 0){ |
|  |  |
|  |  |
|  | set.seed(seed) |
|  | # Get number of training samples |
|  | m = dim(X)[2] |
|  | # Initialize mini batches |
|  | mini\_batches = list() |
|  |  |
|  | # Create a list of random numbers < m |
|  | permutation = c(sample(m)) |
|  | # Randomly shuffle the training data |
|  | shuffled\_X = X[, permutation] |
|  | shuffled\_Y = Y[1, permutation] |
|  |  |
|  | # Compute number of mini batches |
|  | numCompleteMinibatches = floor(m/miniBatchSize) |
|  | batch=0 |
|  | for(k in 0:(numCompleteMinibatches-1)){ |
|  | batch=batch+1 |
|  | # Set the lower and upper bound of the mini batches |
|  | lower=(k\*miniBatchSize)+1 |
|  | upper=((k+1) \* miniBatchSize) |
|  | mini\_batch\_X = shuffled\_X[, lower:upper] |
|  | mini\_batch\_Y = shuffled\_Y[lower:upper] |
|  | # Add it to the list of mini batches |
|  | mini\_batch = list("mini\_batch\_X"=mini\_batch\_X,"mini\_batch\_Y"=mini\_batch\_Y) |
|  | mini\_batches[[batch]] =mini\_batch |
|  |  |
|  |  |
|  | } |
|  |  |
|  | # If the batch size does not divide evenly with mini batc size |
|  | if(m %% miniBatchSize != 0){ |
|  | p=floor(m/miniBatchSize)\*miniBatchSize |
|  | # Set the start and end of last batch |
|  | q=p+m %% miniBatchSize |
|  | mini\_batch\_X = shuffled\_X[,(p+1):q] |
|  | mini\_batch\_Y = shuffled\_Y[(p+1):q] |
|  | } |
|  | # Return the list of mini batches |
|  | mini\_batch = list("mini\_batch\_X"=mini\_batch\_X,"mini\_batch\_Y"=mini\_batch\_Y) |
|  | mini\_batches[[batch]]=mini\_batch |
|  |  |
|  | return(mini\_batches) |
|  | } |
|  |  |
|  | # Plot a decision boundary |
|  | # This function uses ggplot2 |
|  | plotDecisionBoundary1 <- function(Z,parameters,keep\_prob=1){ |
|  | xmin<-min(Z[,1]) |
|  | xmax<-max(Z[,1]) |
|  | ymin<-min(Z[,2]) |
|  | ymax<-max(Z[,2]) |
|  |  |
|  | # Create a grid of points |
|  | a=seq(xmin,xmax,length=100) |
|  | b=seq(ymin,ymax,length=100) |
|  | grid <- expand.grid(x=a, y=b) |
|  | colnames(grid) <- c('x1', 'x2') |
|  | grid1 <-t(grid) |
|  |  |
|  | retvals = forwardPropagationDeep(grid1, parameters,keep\_prob, "relu", |
|  | outputActivationFunc="softmax") |
|  |  |
|  |  |
|  | AL <- retvals$AL |
|  | # From the softmax probabilities pick the one with the highest probability |
|  | q= apply(AL,1,which.max) |
|  |  |
|  | q1 <- t(data.frame(q)) |
|  | q2 <- as.numeric(q1) |
|  | grid2 <- cbind(grid,q2) |
|  | colnames(grid2) <- c('x1', 'x2','q2') |
|  |  |
|  | Z1 <- data.frame(Z) |
|  | names(Z1) <- c("x1","x2","y") |
|  | atitle=paste("Decision boundary") |
|  | ggplot(Z1) + |
|  | geom\_point(data = Z1, aes(x = x1, y = x2, color = y)) + |
|  | stat\_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2), alpha = 0.9)+ |
|  | ggtitle(atitle) + scale\_colour\_gradientn(colours = brewer.pal(10, "Spectral")) |
|  | } |

**DLFunctions.m**

|  |
| --- |
| 1; |
|  | # Define sigmoid function |
|  | function [A,cache] = sigmoid(Z) |
|  | A = 1 ./ (1+ exp(-Z)); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Relu function |
|  | function [A,cache] = relu(Z) |
|  | A = max(0,Z); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Relu function |
|  | function [A,cache] = tanhAct(Z) |
|  | A = tanh(Z); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Softmax function |
|  | function [A,cache] = softmax(Z) |
|  | # get unnormalized probabilities |
|  | exp\_scores = exp(Z'); |
|  | # normalize them for each example |
|  | A = exp\_scores ./ sum(exp\_scores,2); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Softmax function |
|  | function [A,cache] = stableSoftmax(Z) |
|  | # Normalize by max value in each row |
|  | shiftZ = Z' - max(Z',[],2); |
|  | exp\_scores = exp(shiftZ); |
|  | # normalize them for each example |
|  | A = exp\_scores ./ sum(exp\_scores,2); |
|  | #disp("sm") |
|  | #disp(A); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Relu Derivative |
|  | function [dZ] = reluDerivative(dA,cache) |
|  | Z = cache; |
|  | dZ = dA; |
|  | # Get elements that are greater than 0 |
|  | a = (Z > 0); |
|  | # Select only those elements where Z > 0 |
|  | dZ = dZ .\* a; |
|  | end |
|  |  |
|  | # Define Sigmoid Derivative |
|  | function [dZ] = sigmoidDerivative(dA,cache) |
|  | Z = cache; |
|  | s = 1 ./ (1+ exp(-Z)); |
|  | dZ = dA .\* s .\* (1-s); |
|  | end |
|  |  |
|  | # Define Tanh Derivative |
|  | function [dZ] = tanhDerivative(dA,cache) |
|  | Z = cache; |
|  | a = tanh(Z); |
|  | dZ = dA .\* (1 - a .^ 2); |
|  | end |
|  |  |
|  | # Populate a matrix with 1s in rows where Y=1 |
|  | # This function may need to be modified if K is not 3, 10 |
|  | function [Y1] = popMatrix(Y,numClasses) |
|  | Y1=zeros(length(Y),numClasses); |
|  | if(numClasses==3) # For 3 output classes |
|  | Y1(Y==0,1)=1; |
|  | Y1(Y==1,2)=1; |
|  | Y1(Y==2,3)=1; |
|  | elseif(numClasses==10) # For 10 output classes |
|  | Y1(Y==0,1)=1; |
|  | Y1(Y==1,2)=1; |
|  | Y1(Y==2,3)=1; |
|  | Y1(Y==3,4)=1; |
|  | Y1(Y==4,5)=1; |
|  | Y1(Y==5,6)=1; |
|  | Y1(Y==6,7)=1; |
|  | Y1(Y==7,8)=1; |
|  | Y1(Y==8,9)=1; |
|  | Y1(Y==9,10)=1; |
|  |  |
|  | endif |
|  | end |
|  |  |
|  | # Define Softmax Derivative |
|  | function [dZ] = softmaxDerivative(dA,cache,Y, numClasses) |
|  | Z = cache; |
|  | # get unnormalized probabilities |
|  | shiftZ = Z' - max(Z',[],2); |
|  | exp\_scores = exp(shiftZ); |
|  |  |
|  | # normalize them for each example |
|  | probs = exp\_scores ./ sum(exp\_scores,2); |
|  | # dZ = pi- yi |
|  | yi=popMatrix(Y,numClasses); |
|  | dZ=probs-yi; |
|  |  |
|  | end |
|  |  |
|  | # Define Softmax Derivative |
|  | function [dZ] = stableSoftmaxDerivative(dA,cache,Y, numClasses) |
|  | Z = cache; |
|  | # get unnormalized probabilities |
|  | exp\_scores = exp(Z'); |
|  | # normalize them for each example |
|  | probs = exp\_scores ./ sum(exp\_scores,2); |
|  | # dZ = pi- yi |
|  | yi=popMatrix(Y,numClasses); |
|  | dZ=probs-yi; |
|  |  |
|  | end |
|  |  |
|  | # Initialize the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  |  |
|  |  |
|  | # Initialize model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | function [W b] = initializeDeepModel(layerDimensions) |
|  | rand ("seed", 3); |
|  | # note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Create cell arrays for Weights and biases |
|  |  |
|  | for l =2:size(layerDimensions)(2) |
|  | W{l-1} = rand(layerDimensions(l),layerDimensions(l-1))\*0.01; # Multiply by .01 |
|  | b{l-1} = zeros(layerDimensions(l),1); |
|  |  |
|  | endfor |
|  | end |
|  |  |
|  |  |
|  | # He Initialization the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  |  |
|  |  |
|  | # He Initialization for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | function [W b] = HeInitializeDeepModel(layerDimensions) |
|  | rand ("seed", 3); |
|  | # note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Create cell arrays for Weights and biases |
|  |  |
|  | for l =2:size(layerDimensions)(2) |
|  | W{l-1} = rand(layerDimensions(l),layerDimensions(l-1))\* sqrt(2/layerDimensions(l-1)); # Multiply by .01 |
|  | b{l-1} = zeros(layerDimensions(l),1); |
|  |  |
|  | endfor |
|  | end |
|  |  |
|  | # Xavier Initialization for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | function [W b] = XavInitializeDeepModel(layerDimensions) |
|  | rand ("seed", 3); |
|  | # note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Create cell arrays for Weights and biases |
|  |  |
|  | for l =2:size(layerDimensions)(2) |
|  | W{l-1} = rand(layerDimensions(l),layerDimensions(l-1))\* sqrt(1/layerDimensions(l-1)); # Multiply by .01 |
|  | b{l-1} = zeros(layerDimensions(l),1); |
|  |  |
|  | endfor |
|  | end |
|  |  |
|  | # Initialize velocity |
|  | # Input : parameters |
|  | # Returns: Initial velocity v |
|  | function[vdW vdB] = initializeVelocity(weights, biases) |
|  |  |
|  | L = size(weights)(2) # Create an integer |
|  | # Initialize a cell array |
|  | v = {} |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for l=1:L |
|  | sz = size(weights{l}); |
|  | vdW{l} = zeros(sz(1),sz(2)); |
|  | sz = size(biases{l}); |
|  | vdB{l} =zeros(sz(1),sz(2)); |
|  | endfor; |
|  | end |
|  |  |
|  | # Initialize RMSProp |
|  | # Input : parameters |
|  | # Returns: Initial RMSProp |
|  | function[sdW sdB] = initializeRMSProp(weights, biases) |
|  |  |
|  | L = size(weights)(2) # Create an integer |
|  | # Initialize a cell array |
|  | s = {} |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for l=1:L |
|  | sz = size(weights{l}); |
|  | sdW{l} = zeros(sz(1),sz(2)); |
|  | sz = size(biases{l}); |
|  | sdB{l} =zeros(sz(1),sz(2)); |
|  | endfor; |
|  | end |
|  |  |
|  | # Initialize Adam |
|  | # Input : parameters |
|  | # Returns: Initial Adam |
|  | function[vdW vdB sdW sdB] = initializeAdam(weights, biases) |
|  |  |
|  | L = size(weights)(2) # Create an integer |
|  | # Initialize a cell array |
|  | s = {} |
|  |  |
|  | # Initialize velocity with the same dimensions as W |
|  | for l=1:L |
|  | sz = size(weights{l}); |
|  | vdW{l} = zeros(sz(1),sz(2)); |
|  | sdW{l} = zeros(sz(1),sz(2)); |
|  | sz = size(biases{l}); |
|  | sdB{l} =zeros(sz(1),sz(2)); |
|  | vdB{l} =zeros(sz(1),sz(2)); |
|  | endfor; |
|  | end |
|  |  |
|  | # Compute the activation at a layer 'l' for forward prop in a Deep Network |
|  | # Input : A\_prec - Activation of previous layer |
|  | # W,b - Weight and bias matrices and vectors |
|  | # activationFunc - Activation function - sigmoid, tanh, relu etc |
|  | # Returns : The Activation of this layer |
|  | # : |
|  | # Z = W \* X + b |
|  | # A = sigmoid(Z), A= Relu(Z), A= tanh(Z) |
|  | function [A forward\_cache activation\_cache] = layerActivationForward(A\_prev, W, b, activationFunc) |
|  |  |
|  | # Compute Z |
|  | Z = W \* A\_prev +b; |
|  | # Create a cell array |
|  | forward\_cache = {A\_prev W b}; |
|  | # Compute the activation for sigmoid |
|  | if (strcmp(activationFunc,"sigmoid")) |
|  | [A activation\_cache] = sigmoid(Z); |
|  | elseif (strcmp(activationFunc, "relu")) # Compute the activation for Relu |
|  | [A activation\_cache] = relu(Z); |
|  | elseif(strcmp(activationFunc,'tanh')) # Compute the activation for tanh |
|  | [A activation\_cache] = tanhAct(Z); |
|  | elseif(strcmp(activationFunc,'softmax')) # Compute the activation for tanh |
|  | #[A activation\_cache] = softmax(Z); |
|  | [A activation\_cache] = stableSoftmax(Z); |
|  | endif |
|  |  |
|  | end |
|  |  |
|  | # Compute the forward propagation for layers 1..L |
|  | # Input : X - Input Features |
|  | # paramaters: Weights and biases |
|  | # hiddenActivationFunc - Activation function at hidden layers Relu/tanh |
|  | # outputActivationFunc- sigmoid/softmax |
|  | # Returns : AL |
|  | # caches |
|  | # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and sigmoid actiovation at layer L |
|  | function [AL forward\_caches activation\_caches dropoutMat] = forwardPropagationDeep(X, weights,biases, keep\_prob=1, |
|  | hiddenActivationFunc='relu', outputActivationFunc='sigmoid') |
|  | # Create an empty cell array |
|  | forward\_caches = {}; |
|  | activation\_caches = {}; |
|  | droputMat ={}; |
|  | # Set A to X (A0) |
|  | A = X; |
|  | L = length(weights); # number of layers in the neural network |
|  | # Loop through from layer 1 to upto layer L |
|  | for l =1:L-1 |
|  | A\_prev = A; |
|  | # Zi = Wi x Ai-1 + bi and Ai = g(Zi) |
|  | W = weights{l}; |
|  | b = biases{l}; |
|  | [A forward\_cache activation\_cache] = layerActivationForward(A\_prev, W,b, activationFunc=hiddenActivationFunc); |
|  | D=rand(size(A)(1),size(A)(2)); |
|  | D = (D < keep\_prob) ; |
|  | # Multiply by DropoutMat |
|  | A= A .\* D; |
|  | # Divide by keep\_prob to keep expected value same |
|  | A = A ./ keep\_prob; |
|  | # Store D |
|  | dropoutMat{l}=D; |
|  | forward\_caches{l}=forward\_cache; |
|  | activation\_caches{l} = activation\_cache; |
|  | endfor |
|  | # Since this is binary classification use the sigmoid activation function in |
|  | # last layer |
|  | W = weights{L}; |
|  | b = biases{L}; |
|  | [AL, forward\_cache activation\_cache] = layerActivationForward(A, W,b, activationFunc = outputActivationFunc); |
|  | forward\_caches{L}=forward\_cache; |
|  | activation\_caches{L} = activation\_cache; |
|  |  |
|  | end |
|  |  |
|  | # Pick columns where Y==1 |
|  | function [a] = pickColumns(AL,Y,numClasses) |
|  | if(numClasses==3) |
|  | a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)]; |
|  | elseif (numClasses==10) |
|  | a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3);AL(Y==3,4);AL(Y==4,5); |
|  | AL(Y==5,6); AL(Y==6,7);AL(Y==7,8);AL(Y==8,9);AL(Y==9,10)]; |
|  | endif |
|  | end |
|  |  |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # : outputActivationFunc- sigmoid/softmax |
|  | # : numClasses |
|  | # Output: cost |
|  | function [cost]= computeCost(AL, Y, outputActivationFunc="sigmoid",numClasses) |
|  | if(strcmp(outputActivationFunc,"sigmoid")) |
|  | numTraining= size(Y)(2); |
|  | # Element wise multiply for logprobs |
|  | cost = -1/numTraining \* sum((Y .\* log(AL)) + (1-Y) .\* log(1-AL)); |
|  |  |
|  |  |
|  | elseif(strcmp(outputActivationFunc,'softmax')) |
|  | numTraining = size(Y)(2); |
|  | Y=Y'; |
|  | # Select rows where Y=0,1,and 2 and concatenate to a long vector |
|  | #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)]; |
|  | a =pickColumns(AL,Y,numClasses); |
|  |  |
|  | #Select the correct column for log prob |
|  | correct\_probs = -log(a); |
|  | #Compute log loss |
|  | cost= sum(correct\_probs)/numTraining; |
|  | endif |
|  | end |
|  |  |
|  | # Compute the cost with regularization |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # : outputActivationFunc- sigmoid/softmax |
|  | # : numClasses |
|  | # Output: cost |
|  | function [cost]= computeCostWithReg(weights, AL, Y, lambd, outputActivationFunc="sigmoid",numClasses) |
|  |  |
|  | if(strcmp(outputActivationFunc,"sigmoid")) |
|  | numTraining= size(Y)(2); |
|  | # Element wise multiply for logprobs |
|  | cost = -1/numTraining \* sum((Y .\* log(AL)) + (1-Y) .\* log(1-AL)); |
|  |  |
|  | # Regularization cost |
|  | L = size(weights)(2); |
|  | L2RegularizationCost=0; |
|  | for l=1:L |
|  | wtSqr = weights{l} .\* weights{l}; |
|  | #disp(sum(sum(wtSqr,1))); |
|  | L2RegularizationCost+=sum(sum(wtSqr,1)); |
|  | endfor |
|  | L2RegularizationCost = (lambd/(2\*numTraining))\*L2RegularizationCost; |
|  | cost = cost + L2RegularizationCost ; |
|  |  |
|  |  |
|  | elseif(strcmp(outputActivationFunc,'softmax')) |
|  | numTraining = size(Y)(2); |
|  | Y=Y'; |
|  | # Select rows where Y=0,1,and 2 and concatenate to a long vector |
|  | #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)]; |
|  | a =pickColumns(AL,Y,numClasses); |
|  |  |
|  | #Select the correct column for log prob |
|  | correct\_probs = -log(a); |
|  | #Compute log loss |
|  | cost= sum(correct\_probs)/numTraining; |
|  | # Regularization cost |
|  | L = size(weights)(2); |
|  | L2RegularizationCost=0; |
|  | for l=1:L |
|  | # Compute L2 Norm |
|  | wtSqr = weights{l} .\* weights{l}; |
|  | #disp(sum(sum(wtSqr,1))); |
|  | L2RegularizationCost+=sum(sum(wtSqr,1)); |
|  | endfor |
|  | L2RegularizationCost = (lambd/(2\*numTraining))\*L2RegularizationCost; |
|  | cost = cost + L2RegularizationCost ; |
|  | endif |
|  | end |
|  |  |
|  |  |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # # outputActivationFunc- sigmoid/softmax |
|  | # # numClasses |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  | function [dA\_prev dW db] = layerActivationBackward(dA, forward\_cache, activation\_cache, Y, activationFunc,numClasses) |
|  |  |
|  | A\_prev = forward\_cache{1}; |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | numTraining = size(A\_prev)(2); |
|  | if (strcmp(activationFunc,"relu")) |
|  | dZ = reluDerivative(dA, activation\_cache); |
|  | elseif (strcmp(activationFunc,"sigmoid")) |
|  | dZ = sigmoidDerivative(dA, activation\_cache); |
|  | elseif(strcmp(activationFunc, "tanh")) |
|  | dZ = tanhDerivative(dA, activation\_cache); |
|  | elseif(strcmp(activationFunc, "softmax")) |
|  | #dZ = softmaxDerivative(dA, activation\_cache,Y,numClasses); |
|  | dZ = stableSoftmaxDerivative(dA, activation\_cache,Y,numClasses); |
|  | endif |
|  |  |
|  |  |
|  | if (strcmp(activationFunc,"softmax")) |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | # Add the regularization factor |
|  | dW = 1/numTraining \* A\_prev \* dZ; |
|  | db = 1/numTraining \* sum(dZ,1); |
|  | dA\_prev = dZ\*W; |
|  | else |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | # Add the regularization factor |
|  | dW = 1/numTraining \* dZ \* A\_prev'; |
|  | db = 1/numTraining \* sum(dZ,2); |
|  | dA\_prev = W'\*dZ; |
|  | endif |
|  |  |
|  | end |
|  |  |
|  | # Compute the backpropoagation with regularization for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # # outputActivationFunc- sigmoid/softmax |
|  | # # numClasses |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  | function [dA\_prev dW db] = layerActivationBackwardWithReg(dA, forward\_cache, activation\_cache, Y, lambd=0, activationFunc,numClasses) |
|  |  |
|  | A\_prev = forward\_cache{1}; |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | numTraining = size(A\_prev)(2); |
|  | if (strcmp(activationFunc,"relu")) |
|  | dZ = reluDerivative(dA, activation\_cache); |
|  | elseif (strcmp(activationFunc,"sigmoid")) |
|  | dZ = sigmoidDerivative(dA, activation\_cache); |
|  | elseif(strcmp(activationFunc, "tanh")) |
|  | dZ = tanhDerivative(dA, activation\_cache); |
|  | elseif(strcmp(activationFunc, "softmax")) |
|  | #dZ = softmaxDerivative(dA, activation\_cache,Y,numClasses); |
|  | dZ = stableSoftmaxDerivative(dA, activation\_cache,Y,numClasses); |
|  | endif |
|  |  |
|  | if (strcmp(activationFunc,"softmax")) |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | # Add the regularization factor |
|  | dW = 1/numTraining \* A\_prev \* dZ + (lambd/numTraining) \* W'; |
|  | db = 1/numTraining \* sum(dZ,1); |
|  | dA\_prev = dZ\*W; |
|  | else |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | # Add the regularization factor |
|  | dW = 1/numTraining \* dZ \* A\_prev' + (lambd/numTraining) \* W; |
|  | db = 1/numTraining \* sum(dZ,2); |
|  | dA\_prev = W'\*dZ; |
|  | endif |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : AL: Output of L layer Network - weights |
|  | # # Y Real output |
|  | # # caches -- list of caches containing: |
|  | # every cache of layerActivationForward() with "relu"/"tanh" |
|  | # #(it's caches[l], for l in range(L-1) i.e l = 0...L-2) |
|  | # #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1]) |
|  | # hiddenActivationFunc - Activation function at hidden layers |
|  | # # outputActivationFunc- sigmoid/softmax |
|  | # # numClasses |
|  | # |
|  | # Returns: |
|  | # gradients -- A dictionary with the gradients |
|  | # gradients["dA" + str(l)] = ... |
|  | # gradients["dW" + str(l)] = ... |
|  |  |
|  | function [gradsDA gradsDW gradsDB]= backwardPropagationDeep(AL, Y, activation\_caches,forward\_caches, |
|  | dropoutMat, lambd=0, keep\_prob=1, hiddenActivationFunc='relu',outputActivationFunc="sigmoid",numClasses) |
|  |  |
|  |  |
|  | # Set the number of layers |
|  | L = length(activation\_caches); |
|  | m = size(AL)(2); |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | # Initializing the backpropagation |
|  | # dl/dAL= -(y/a + (1-y)/(1-a)) - At the output layer |
|  | dAL = -((Y ./ AL) - (1 - Y) ./ ( 1 - AL)); |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | dAL=0; |
|  | Y=Y'; |
|  | endif |
|  |  |
|  |  |
|  | # Since this is a binary classification the activation at output is sigmoid |
|  | # Get the gradients at the last layer |
|  | # Inputs: "AL, Y, caches". |
|  | # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"] |
|  | activation\_cache = activation\_caches{L}; |
|  | forward\_cache = forward\_caches(L); |
|  | # Note the cell array includes an array of forward caches. To get to this we need to include the index {1} |
|  | if (lambd==0) |
|  | [dA dW db] = layerActivationBackward(dAL, forward\_cache{1}, activation\_cache, Y, activationFunc = outputActivationFunc,numClasses); |
|  | else |
|  | [dA dW db] = layerActivationBackwardWithReg(dAL, forward\_cache{1}, activation\_cache, Y, lambd, activationFunc = outputActivationFunc,numClasses); |
|  | endif |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | gradsDA{L}= dA; |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | gradsDA{L}= dA';#Note the transpose |
|  | endif |
|  | gradsDW{L}= dW; |
|  | gradsDB{L}= db; |
|  |  |
|  | # Traverse in the reverse direction |
|  | for l =(L-1):-1:1 |
|  | # Compute the gradients for L-1 to 1 for Relu/tanh |
|  | # Inputs: "gradients["dA" + str(l + 2)], caches". |
|  | # Outputs: "gradients["dA" + str(l + 1)] , gradients["dW" + str(l + 1)] , gradients["db" + str(l + 1)] |
|  | activation\_cache = activation\_caches{l}; |
|  | forward\_cache = forward\_caches(l); |
|  |  |
|  | #dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackward(gradients['dA'+str(l+1)], current\_cache, activationFunc = "relu") |
|  | # dAl the dervative of the activation of the lth layer,is the first element |
|  | dAl= gradsDA{l+1}; |
|  | if(lambd == 0) |
|  | # Get the dropout mat |
|  | D = dropoutMat{l}; |
|  | #Multiply by the dropoutMat |
|  | dAl= dAl .\* D; |
|  | # Divide by keep\_prob to keep expected value same |
|  | dAl = dAl ./ keep\_prob; |
|  | [dA\_prev\_temp, dW\_temp, db\_temp] = layerActivationBackward(dAl, forward\_cache{1}, activation\_cache, Y, activationFunc = hiddenActivationFunc,numClasses); |
|  | else |
|  | [dA\_prev\_temp, dW\_temp, db\_temp] = layerActivationBackwardWithReg(dAl, forward\_cache{1}, activation\_cache, Y, lambd, activationFunc = hiddenActivationFunc,numClasses); |
|  | endif |
|  | gradsDA{l}= dA\_prev\_temp; |
|  | gradsDW{l}= dW\_temp; |
|  | gradsDB{l}= db\_temp; |
|  |  |
|  | endfor |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc |
|  | #output : Updated weights after 1 iteration |
|  | function [weights biases] = gradientDescent(weights, biases,gradsW,gradsB, learningRate,outputActivationFunc="sigmoid") |
|  |  |
|  | L = size(weights)(2); # number of layers in the neural network |
|  | # Update rule for each parameter. |
|  | for l=1:(L-1) |
|  | weights{l} = weights{l} -learningRate\* gradsW{l}; |
|  | biases{l} = biases{l} -learningRate\* gradsB{l}; |
|  | endfor |
|  |  |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | weights{L} = weights{L} -learningRate\* gradsW{L}; |
|  | biases{L} = biases{L} -learningRate\* gradsB{L}; |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | weights{L} = weights{L} -learningRate\* gradsW{L}'; |
|  | biases{L} = biases{L} -learningRate\* gradsB{L}'; |
|  | endif |
|  |  |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Update parameters with momentum |
|  | # Input : parameters |
|  | # : gradients |
|  | # : v |
|  | # : beta |
|  | # : learningRate |
|  | # : |
|  | #output : Updated parameters and velocity |
|  | function [weights biases] = gradientDescentWithMomentum(weights, biases,gradsDW,gradsDB, vdW, vdB, beta, learningRate,outputActivationFunc="sigmoid") |
|  | L = size(weights)(2); # number of layers in the neural network |
|  | # Update rule for each parameter. |
|  | for l=1:(L-1) |
|  | # Compute velocities |
|  | # v['dWk'] = beta \*v['dWk'] + (1-beta)\*dWk |
|  | vdW{l} = beta\*vdW{l} + (1 -beta) \* gradsDW{l}; |
|  | vdB{l} = beta\*vdB{l} + (1 -beta) \* gradsDB{l}; |
|  | weights{l} = weights{l} -learningRate\* vdW{l}; |
|  | biases{l} = biases{l} -learningRate\* vdB{l}; |
|  | endfor |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | vdW{L} = beta\*vdW{L} + (1 -beta) \* gradsDW{L}; |
|  | vdB{L} = beta\*vdB{L} + (1 -beta) \* gradsDB{L}; |
|  | weights{L} = weights{L} -learningRate\* vdW{L}; |
|  | biases{L} = biases{L} -learningRate\* vdB{L}; |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | vdW{L} = beta\*vdW{L} + (1 -beta) \* gradsDW{L}'; |
|  | vdB{L} = beta\*vdB{L} + (1 -beta) \* gradsDB{L}'; |
|  | weights{L} = weights{L} -learningRate\* vdW{L}; |
|  | biases{L} = biases{L} -learningRate\* vdB{L}; |
|  | endif |
|  |  |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Update parameters with RMSProp |
|  | # Input : parameters |
|  | # : gradients |
|  | # : s |
|  | # : beta |
|  | # : learningRate |
|  | # : |
|  | #output : Updated parameters RMSProp |
|  | function [weights biases] = gradientDescentWithRMSProp(weights, biases,gradsDW,gradsDB, sdW, sdB, beta1, epsilon, learningRate,outputActivationFunc="sigmoid") |
|  | L = size(weights)(2); # number of layers in the neural network |
|  | # Update rule for each parameter. |
|  | for l=1:(L-1) |
|  | sdW{l} = beta1\*sdW{l} + (1 -beta1) \* gradsDW{l} .\* gradsDW{l}; |
|  | sdB{l} = beta1\*sdB{l} + (1 -beta1) \* gradsDB{l} .\* gradsDB{l}; |
|  | weights{l} = weights{l} - learningRate\* gradsDW{l} ./ sqrt(sdW{l} + epsilon); |
|  | biases{l} = biases{l} - learningRate\* gradsDB{l} ./ sqrt(sdB{l} + epsilon); |
|  | endfor |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | sdW{L} = beta1\*sdW{L} + (1 -beta1) \* gradsDW{L} .\* gradsDW{L}; |
|  | sdB{L} = beta1\*sdB{L} + (1 -beta1) \* gradsDB{L} .\* gradsDB{L}; |
|  | weights{L} = weights{L} -learningRate\* gradsDW{L} ./ sqrt(sdW{L} +epsilon); |
|  | biases{L} = biases{L} -learningRate\* gradsDB{L} ./ sqrt(sdB{L} + epsilon); |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | sdW{L} = beta1\*sdW{L} + (1 -beta1) \* gradsDW{L}' .\* gradsDW{L}'; |
|  | sdB{L} = beta1\*sdB{L} + (1 -beta1) \* gradsDB{L}' .\* gradsDB{L}'; |
|  | weights{L} = weights{L} -learningRate\* gradsDW{L}' ./ sqrt(sdW{L} +epsilon); |
|  | biases{L} = biases{L} -learningRate\* gradsDB{L}' ./ sqrt(sdB{L} + epsilon); |
|  | endif |
|  |  |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Update parameters with Adam |
|  | # Input : parameters |
|  | # : gradients |
|  | # : v |
|  | # : beta |
|  | # : learningRate |
|  | # : |
|  | #output : Updated parameters and velocity |
|  | function [weights biases] = gradientDescentWithAdam(weights, biases,gradsDW,gradsDB, |
|  | vdW, vdB, sdW, sdB, t, beta1, beta2, epsilon, learningRate,outputActivationFunc="sigmoid") |
|  | vdW\_corrected = {}; |
|  | vdB\_corrected = {}; |
|  | sdW\_corrected = {}; |
|  | sdB\_corrected = {}; |
|  | L = size(weights)(2); # number of layers in the neural network |
|  | # Update rule for each parameter. |
|  | for l=1:(L-1) |
|  | vdW{l} = beta1\*vdW{l} + (1 -beta1) \* gradsDW{l}; |
|  | vdB{l} = beta1\*vdB{l} + (1 -beta1) \* gradsDB{l}; |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | vdW\_corrected{l} = vdW{l}/(1-beta1^t); |
|  | vdB\_corrected{l} = vdB{l}/(1-beta1^t); |
|  |  |
|  | sdW{l} = beta2\*sdW{l} + (1 -beta2) \* gradsDW{l} .\* gradsDW{l}; |
|  | sdB{l} = beta2\*sdB{l} + (1 -beta2) \* gradsDB{l} .\* gradsDB{l}; |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | sdW\_corrected{l} = sdW{l}/(1-beta2^t); |
|  | sdB\_corrected{l} = sdB{l}/(1-beta2^t); |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(sdW\_corrected{l}+epsilon); |
|  | d2=sqrt(sdB\_corrected{l}+epsilon); |
|  |  |
|  | weights{l} = weights{l} - learningRate\* vdW\_corrected{l} ./ d1; |
|  | biases{l} = biases{l} -learningRate\* vdB\_corrected{l} ./ d2; |
|  | endfor |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | vdW{L} = beta1\*vdW{L} + (1 -beta1) \* gradsDW{L}; |
|  | vdB{L} = beta1\*vdB{L} + (1 -beta1) \* gradsDB{L}; |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | vdW\_corrected{L} = v{L}/(1-beta1^t); |
|  | vdB\_corrected{L} = v{L}/(1-beta1^t); |
|  |  |
|  | sdW{L} = beta2\*sdW{L} + (1 -beta2) \* gradsDW{L} .\* gradsDW{L}; |
|  | sdB{L} = beta2\*sdB{L} + (1 -beta2) \* gradsDB{L} .\* gradsDB{L}; |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | sdW\_corrected{L} = s{L}/(1-beta2^t); |
|  | sdB\_corrected{L} = s{L}/(1-beta2^t); |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(sdW\_corrected{L}+epsilon); |
|  | d2=sqrt(sdB\_corrected{L}+epsilon); |
|  |  |
|  | weights{L} = weights{L} - learningRate\* vdW\_corrected{L} ./ d1; |
|  | biases{L} = biases{L} -learningRate\* vdB\_corrected{L} ./ d2; |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | vdW{L} = beta1\*vdW{L} + (1 -beta1) \* gradsDW{L}'; |
|  | vdB{L} = beta1\*vdB{L} + (1 -beta1) \* gradsDB{L}'; |
|  |  |
|  | # Compute bias-corrected first moment estimate. |
|  | vdW\_corrected{L} = vdW{L}/(1-beta1^t); |
|  | vdB\_corrected{L} = vdB{L}/(1-beta1^t); |
|  |  |
|  | sdW{L} = beta2\*sdW{L} + (1 -beta2) \* gradsDW{L}' .\* gradsDW{L}'; |
|  | sdB{L} = beta2\*sdB{L} + (1 -beta2) \* gradsDB{L}' .\* gradsDB{L}'; |
|  |  |
|  | # Compute bias-corrected second moment estimate. |
|  | sdW\_corrected{L} = sdW{L}/(1-beta2^t); |
|  | sdB\_corrected{L} = sdB{L}/(1-beta2^t); |
|  |  |
|  | # Update parameters. |
|  | d1=sqrt(sdW\_corrected{L}+epsilon); |
|  | d2=sqrt(sdB\_corrected{L}+epsilon); |
|  |  |
|  | weights{L} = weights{L} - learningRate\* vdW\_corrected{L} ./ d1; |
|  | biases{L} = biases{L} -learningRate\* vdB\_corrected{L} ./ d2; |
|  | endif |
|  |  |
|  |  |
|  | end |
|  |  |
|  | # Execute a L layer Deep learning model |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : num of iterations |
|  | #output : Updated weights and biases after each iteration |
|  | function [weights biases costs] = L\_Layer\_DeepModel(X, Y, layersDimensions, hiddenActivationFunc='relu', |
|  | outputActivationFunc="sigmoid",learning\_rate = .3, lambd=0, keep\_prob=1, num\_iterations = 10000,initType="default")#lr was 0.009 |
|  |  |
|  | rand ("seed", 1); |
|  | costs = [] ; |
|  | if (strcmp(initType,"He")) |
|  | # He Initialization |
|  | [weights biases] = HeInitializeDeepModel(layersDimensions); |
|  | elseif (strcmp(initType,"Xav")) |
|  | # Xavier Initialization |
|  | [weights biases] = XavInitializeDeepModel(layersDimensions); |
|  | else |
|  | # Default initialization. |
|  | [weights biases] = initializeDeepModel(layersDimensions); |
|  | endif |
|  |  |
|  | # Loop (gradient descent) |
|  | for i = 0:num\_iterations |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID. |
|  | [AL forward\_caches activation\_caches droputMat] = forwardPropagationDeep(X, weights, biases,keep\_prob, hiddenActivationFunc, outputActivationFunc=outputActivationFunc); |
|  |  |
|  | # Regularization parameter is 0 |
|  | if (lambd==0) |
|  | # Compute cost. |
|  | cost = computeCost(AL, Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | else |
|  | # Compute cost with regularization |
|  | cost = computeCostWithReg(weights, AL, Y, lambd, outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | endif |
|  | # Backward propagation. |
|  | [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y, activation\_caches,forward\_caches, droputMat, lambd, keep\_prob, hiddenActivationFunc, outputActivationFunc=outputActivationFunc, |
|  | numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | # Update parameters. |
|  | [weights biases] = gradientDescent(weights,biases, gradsDW,gradsDB,learning\_rate,outputActivationFunc=outputActivationFunc); |
|  |  |
|  |  |
|  | # Print the cost every 1000 iterations |
|  | if ( mod(i,1000) == 0) |
|  | costs =[costs cost]; |
|  | #disp ("Cost after iteration"), L2RegularizationCost(i),disp(cost); |
|  | printf("Cost after iteration i=%i cost=%d\n",i,cost); |
|  | endif |
|  | endfor |
|  |  |
|  | end |
|  |  |
|  | # Execute a L layer Deep learning model with Stochastic Gradient descent |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : mini\_batch\_size |
|  | # : num of epochs |
|  | #output : Updated weights and biases after each iteration |
|  | function [weights biases costs] = L\_Layer\_DeepModel\_SGD(X, Y, layersDimensions, hiddenActivationFunc='relu', |
|  | outputActivationFunc="sigmoid",learningRate = .3, |
|  | lrDecay=false,decayRate=1, |
|  | lambd=0, keep\_prob=1, |
|  | optimizer="gd", beta=0.9, beta1=0.9, beta2=0.999,epsilon=10^-8, |
|  | mini\_batch\_size = 64, num\_epochs = 2500) |
|  |  |
|  | disp("here"); |
|  | printf("learningRate=%f ",learningRate); |
|  | printf("lrDecay=%d ",lrDecay); |
|  | printf("decayRate=%f ",decayRate); |
|  | printf("lamd=%d ",lambd); |
|  | printf("keep\_prob=%f ",keep\_prob); |
|  | printf("optimizer=%s ",optimizer); |
|  | printf("beta=%f ",beta); |
|  | printf("beta1=%f ",beta1); |
|  | printf("beta2=%f ",beta2); |
|  | printf("epsilon=%f ",epsilon); |
|  | printf("mini\_batch\_size=%d ",mini\_batch\_size); |
|  | printf("num\_epochs=%d ",num\_epochs); |
|  | t=0; |
|  | rand ("seed", 1); |
|  | costs = [] ; |
|  | # Parameters initialization. |
|  | [weights biases] = initializeDeepModel(layersDimensions); |
|  |  |
|  | if (strcmp(optimizer,"momentum")) |
|  | [vdW vdB] = initializeVelocity(weights, biases); |
|  |  |
|  | elseif(strcmp(optimizer,"rmsprop")) |
|  | [sdW sdB] = initializeRMSProp(weights, biases); |
|  |  |
|  | elseif(strcmp(optimizer,"adam")) |
|  | [vdW vdB sdW sdB] = initializeAdam(weights, biases); |
|  | endif |
|  | seed=10; |
|  | # Loop (gradient descent) |
|  | for i = 0:num\_epochs |
|  | seed = seed + 1; |
|  | [mini\_batches\_X mini\_batches\_Y] = random\_mini\_batches(X, Y, mini\_batch\_size, seed); |
|  |  |
|  | minibatches=length(mini\_batches\_X); |
|  | for batch=1:minibatches |
|  | X=mini\_batches\_X{batch}; |
|  | Y=mini\_batches\_Y{batch}; |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID/SOFTMAX. |
|  | [AL forward\_caches activation\_caches droputMat] = forwardPropagationDeep(X, weights, biases, keep\_prob,hiddenActivationFunc, outputActivationFunc=outputActivationFunc); |
|  | #disp(batch); |
|  | #disp(size(X)); |
|  | #disp(size(Y)); |
|  | if (lambd==0) |
|  | # Compute cost. |
|  | cost = computeCost(AL, Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | else |
|  | # Compute cost with regularization |
|  | cost = computeCostWithReg(weights, AL, Y, lambd, outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | endif |
|  | #disp(cost); |
|  | # Backward propagation. |
|  | [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y, activation\_caches,forward\_caches, droputMat, lambd, keep\_prob, hiddenActivationFunc, outputActivationFunc=outputActivationFunc, |
|  | numClasses=layersDimensions(size(layersDimensions)(2))); |
|  |  |
|  | if (strcmp(optimizer,"gd")) |
|  | # Update parameters. |
|  | [weights biases] = gradientDescent(weights,biases, gradsDW,gradsDB,learningRate,outputActivationFunc=outputActivationFunc); |
|  | elseif (strcmp(optimizer,"momentum")) |
|  | [weights biases] = gradientDescentWithMomentum(weights, biases,gradsDW,gradsDB, vdW, vdB, beta, learningRate,outputActivationFunc); |
|  | elseif (strcmp(optimizer,"rmsprop")) |
|  | [weights biases] = gradientDescentWithRMSProp(weights, biases,gradsDW,gradsDB, sdW, sdB, beta1, epsilon, learningRate,outputActivationFunc); |
|  |  |
|  | elseif (strcmp(optimizer,"adam")) |
|  | t=t+1; |
|  | [weights biases] = gradientDescentWithAdam(weights, biases,gradsDW,gradsDB,vdW, vdB, sdW, sdB, t, beta1, beta2, epsilon, learningRate,outputActivationFunc); |
|  | endif |
|  | endfor |
|  | # Print the cost every 1000 iterations |
|  | if ( mod(i,1000) == 0) |
|  | costs =[costs cost]; |
|  | #disp ("Cost after iteration"), disp(i),disp(cost); |
|  | printf("Cost after iteration i=%i cost=%d\n",i,cost); |
|  | endif |
|  | if(lrDecay==true) |
|  | learningRate=decayRate^(num\_epochs/1000)\*learningRate; |
|  | endif |
|  | endfor |
|  |  |
|  | end |
|  |  |
|  |  |
|  | function plotCostVsIterations(maxIterations,costs,fig1) |
|  | iterations=[0:1000:maxIterations]; |
|  | plot(iterations,costs); |
|  | title ("Cost vs no of iterations "); |
|  | xlabel("No of iterations"); |
|  | ylabel("Cost"); |
|  | print -dpng figReg2-o |
|  | end; |
|  |  |
|  | function plotCostVsEpochs(maxEpochs,costs,fig1) |
|  | epochs=[0:1000:maxEpochs]; |
|  | plot(epochs,costs); |
|  | title ("Cost vs no of epochs "); |
|  | xlabel("No of epochs"); |
|  | ylabel("Cost"); |
|  | print -dpng fig5-o |
|  | end; |
|  |  |
|  | # Compute the predicted value for a given input |
|  | # Input : Neural Network parameters |
|  | # : Input data |
|  | function [predictions]= predict(weights, biases, X,keep\_prob=1,hiddenActivationFunc="relu") |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights, biases,keep\_prob,hiddenActivationFunc); |
|  | predictions = (AL>0.5); |
|  | end |
|  |  |
|  | # Plot the decision boundary |
|  | function plotDecisionBoundary(data,weights, biases,keep\_prob=1,hiddenActivationFunc="relu",fig2) |
|  | %Plot a non-linear decision boundary learned by the SVM |
|  | colormap ("summer"); |
|  |  |
|  | % Make classification predictions over a grid of values |
|  | x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)'; |
|  | x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)'; |
|  | [X1, X2] = meshgrid(x1plot, x2plot); |
|  | vals = zeros(size(X1)); |
|  | # Plot the prediction for the grid |
|  | for i = 1:size(X1, 2) |
|  | gridPoints = [X1(:, i), X2(:, i)]; |
|  | vals(:, i)=predict(weights, biases,gridPoints',keep\_prob, hiddenActivationFunc=hiddenActivationFunc); |
|  | endfor |
|  |  |
|  | scatter(data(:,1),data(:,2),8,c=data(:,3),"filled"); |
|  | % Plot the boundary |
|  | hold on |
|  | #contour(X1, X2, vals, [0 0], 'LineWidth', 2); |
|  | contour(X1, X2, vals,"linewidth",4); |
|  | title ({"3 layer Neural Network decision boundary"}); |
|  | hold off; |
|  | print -dpng figReg22-o |
|  |  |
|  | end |
|  |  |
|  | function [AL]= scores(weights, biases, X,hiddenActivationFunc="relu") |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights, biases,hiddenActivationFunc); |
|  | end |
|  |  |
|  | # Create Random mini batches. Return cell arrays with the mini batches |
|  | # Input : X, Y |
|  | # : Size of minibatch |
|  | #Output : mini batches X & Y |
|  | function [mini\_batches\_X mini\_batches\_Y]= random\_mini\_batches(X, Y, miniBatchSize = 64, seed = 0) |
|  |  |
|  | rand ("seed", seed); |
|  | # Get number of training samples |
|  | m = size(X)(2); |
|  |  |
|  |  |
|  | # Create a list of random numbers < m |
|  | permutation = randperm(m); |
|  | # Randomly shuffle the training data |
|  | shuffled\_X = X(:, permutation); |
|  | shuffled\_Y = Y(:, permutation); |
|  |  |
|  | # Compute number of mini batches |
|  | numCompleteMinibatches = floor(m/miniBatchSize); |
|  | batch=0; |
|  | for k = 0:(numCompleteMinibatches-1) |
|  | #Set the start and end of each mini batch |
|  | batch=batch+1; |
|  | lower=(k\*miniBatchSize)+1; |
|  | upper=(k+1) \* miniBatchSize; |
|  | mini\_batch\_X = shuffled\_X(:, lower:upper); |
|  | mini\_batch\_Y = shuffled\_Y(:, lower:upper); |
|  |  |
|  | # Create cell arrays |
|  | mini\_batches\_X{batch} = mini\_batch\_X; |
|  | mini\_batches\_Y{batch} = mini\_batch\_Y; |
|  | endfor |
|  |  |
|  | # If the batc size does not cleanly divide with number of mini batches |
|  | if mod(m ,miniBatchSize) != 0 |
|  | # Set the start and end of the last mini batch |
|  | l=floor(m/miniBatchSize)\*miniBatchSize; |
|  | m=l+ mod(m,miniBatchSize); |
|  | mini\_batch\_X = shuffled\_X(:,(l+1):m); |
|  | mini\_batch\_Y = shuffled\_Y(:,(l+1):m); |
|  |  |
|  | batch=batch+1; |
|  | mini\_batches\_X{batch} = mini\_batch\_X; |
|  | mini\_batches\_Y{batch} = mini\_batch\_Y; |
|  | endif |
|  | end |
|  |  |
|  | function plotDecisionBoundary1( data,weights, biases,keep\_prob=1, hiddenActivationFunc="relu") |
|  | % Make classification predictions over a grid of values |
|  | x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)'; |
|  | x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)'; |
|  | [X1, X2] = meshgrid(x1plot, x2plot); |
|  | vals = zeros(size(X1)); |
|  | for i = 1:size(X1, 2) |
|  | gridPoints = [X1(:, i), X2(:, i)]; |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(gridPoints', weights, biases,keep\_prob,hiddenActivationFunc, outputActivationFunc="softmax"); |
|  | [l m] = max(AL, [ ], 2); |
|  | vals(:, i)= m; |
|  | endfor |
|  |  |
|  | scatter(data(:,1),data(:,2),8,c=data(:,3),"filled"); |
|  | % Plot the boundary |
|  | hold on |
|  | contour(X1, X2, vals,"linewidth",4); |
|  | print -dpng "fig-o1.png" |
|  | end |

**Conclusion**: In this post I discuss and implement several Stochastic Gradient Descent optimization methods. The implementation of these methods enhance my already existing generic L-Layer Deep Learning Network implementation in vectorized Python, R and Octave, which I had discussed in the previous post in this series on Deep Learning from first principles in Python, R and Octave. Check it out, if you haven’t already.