**1. Initialization techniques**

The usual initialization technique is to generate Gaussian or uniform random numbers and multiply it by a small value like 0.01. Two techniques which are used to speed up convergence is the [He](https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf) initialization or [Xavier](http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf). These initialization techniques enable gradient descent to converge faster.

**1.1 a Default initialization – Python**

This technique just initializes the weights to small random values based on Gaussian or uniform distribution

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("DLfunctions61.py").read())

#Load the data

train\_X, train\_Y, test\_X, test\_Y = load\_dataset()

# Set the layers dimensions

layersDimensions = [2,7,1]

# Train a deep learning network with random initialization

parameters = L\_Layer\_DeepModel(train\_X, train\_Y, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid",learningRate = 0.6, num\_iterations = 9000, initType="default", print\_cost = True,figure="fig1.png")

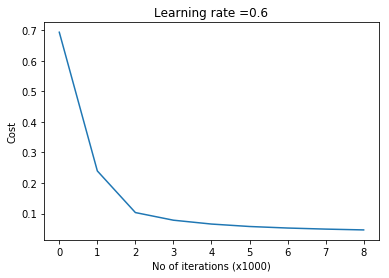
# Clear the plot

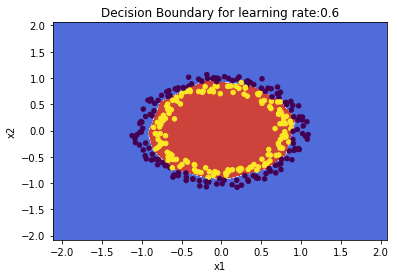
plt.clf()

plt.close()

# Plot the decision boundary

plot\_decision\_boundary(lambda x: predict(parameters, x.T), train\_X, train\_Y,str(0.6),figure1="fig2.png")





**1.1 b He initialization – Python**

‘He’ initialization attributed to [He et al](https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf), multiplies the random weights by  
\sqrt{\frac{2}{dimension\ of\ previous\ layer}}

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("DLfunctions61.py").read())

#Load the data

train\_X, train\_Y, test\_X, test\_Y = load\_dataset()

# Set the layers dimensions

layersDimensions = [2,7,1]

# Train a deep learning network with He initialization

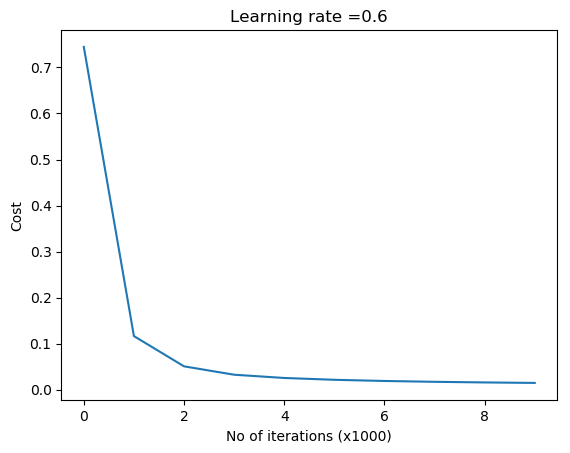
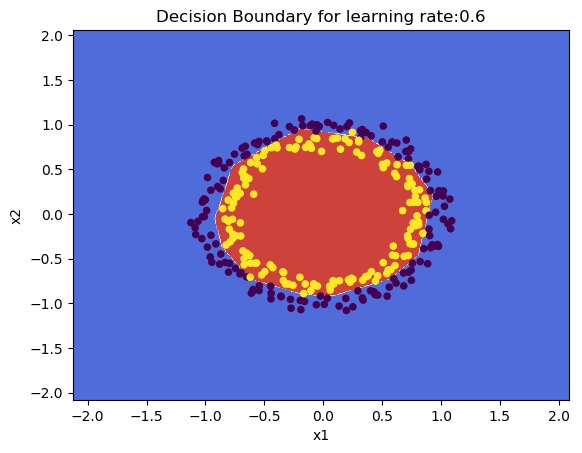
parameters = L\_Layer\_DeepModel(train\_X, train\_Y, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid", learningRate =0.6, num\_iterations = 10000,initType="He",print\_cost = True, figure="fig3.png")

plt.clf()

plt.close()

# Plot the decision boundary

plot\_decision\_boundary(lambda x: predict(parameters, x.T), train\_X, train\_Y,str(0.6),figure1="fig4.png")

**1.1 c Xavier initialization – Python**

[Xavier](http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf) initialization multiply the random weights by  
\sqrt{\frac{1}{dimension\ of\ previous\ layer}}

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("DLfunctions61.py").read())

#Load the data

train\_X, train\_Y, test\_X, test\_Y = load\_dataset()

# Set the layers dimensions

layersDimensions = [2,7,1]

# Train a L layer Deep Learning network

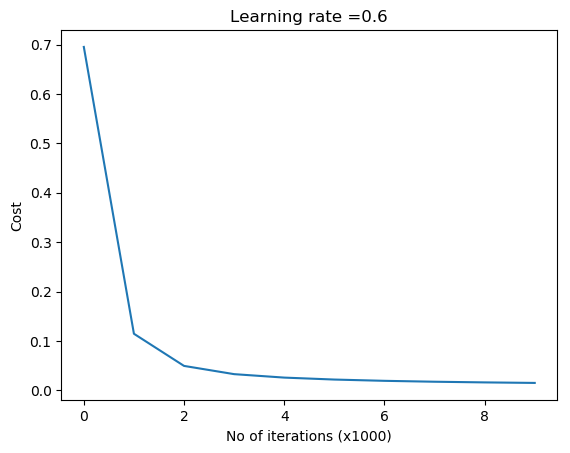
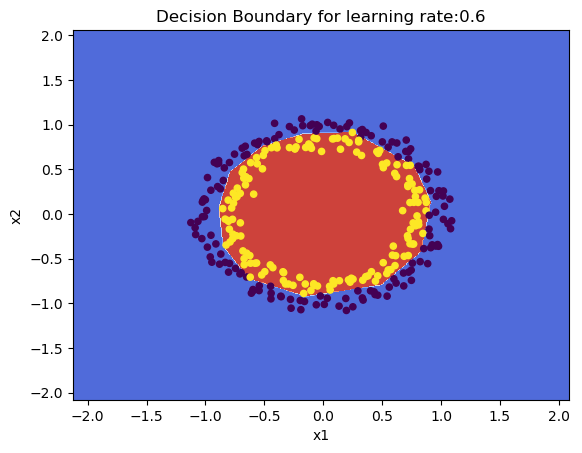
parameters = L\_Layer\_DeepModel(train\_X, train\_Y, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid",

learningRate = 0.6,num\_iterations = 10000, initType="Xavier",print\_cost = True,

figure="fig5.png")

# Plot the decision boundary

plot\_decision\_boundary(lambda x: predict(parameters, x.T), train\_X, train\_Y,str(0.6),figure1="fig6.png")

**1.2a Default initialization – R**

source("DLfunctions61.R")

#Load the data

z <- as.matrix(read.csv("circles.csv",header=FALSE))

x <- z[,1:2]

y <- z[,3]

X <- t(x)

Y <- t(y)

#Set the layer dimensions

layersDimensions = c(2,11,1)

# Train a deep learning network

retvals = L\_Layer\_DeepModel(X, Y, layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",

learningRate = 0.5,

numIterations = 8000,

initType="default",

print\_cost = True)

#Plot the cost vs iterations

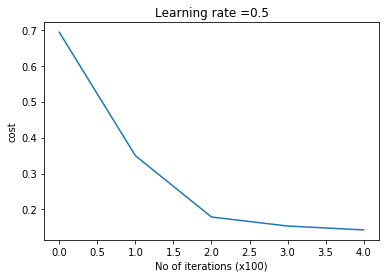
iterations <- seq(0,8000,1000)

costs=retvals$costs

df=data.frame(iterations,costs)

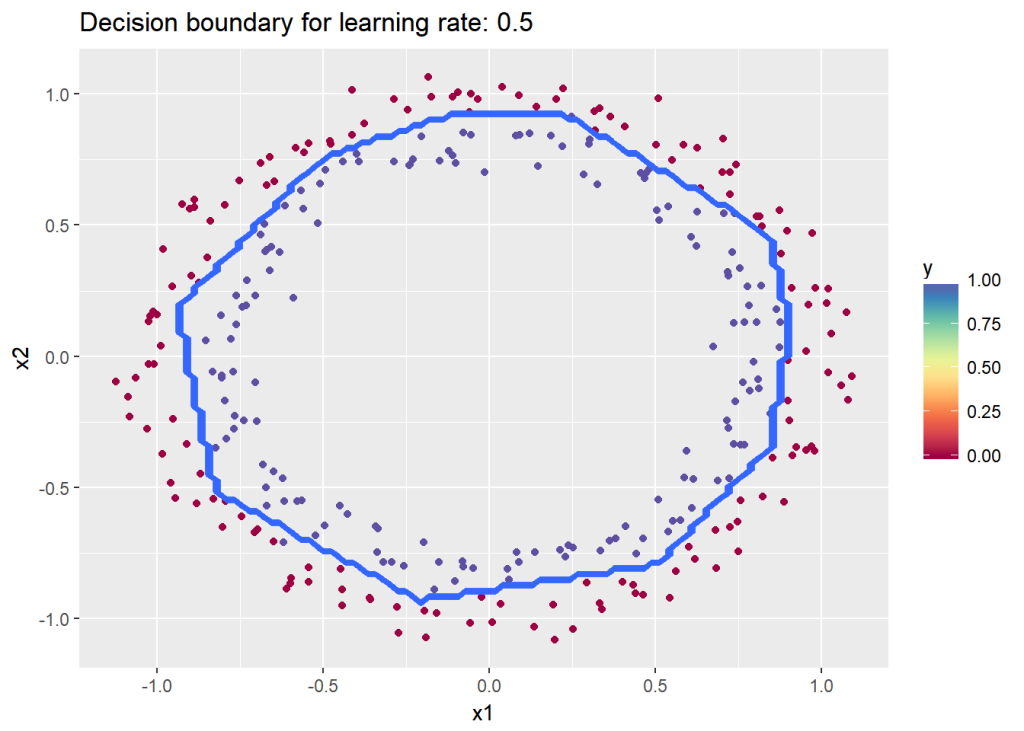
ggplot(df,aes(x=iterations,y=costs)) + geom\_point() + geom\_line(color="blue") +

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



# Plot the decision boundary

plotDecisionBoundary(z,retvals,hiddenActivationFunc="relu",lr=0.5)



**1.2b He initialization – R**

The code for ‘He’ initilaization in R is included below

# 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)

}

The code in R below uses He initialization to learn the data

source("DLfunctions61.R")

# Load the data

z <- as.matrix(read.csv("circles.csv",header=FALSE))

x <- z[,1:2]

y <- z[,3]

X <- t(x)

Y <- t(y)

# Set the layer dimensions

layersDimensions = c(2,11,1)

# Train a deep learning network

retvals = L\_Layer\_DeepModel(X, Y, layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",

learningRate = 0.5,

numIterations = 9000,

initType="He",

print\_cost = True)

#Plot the cost vs iterations

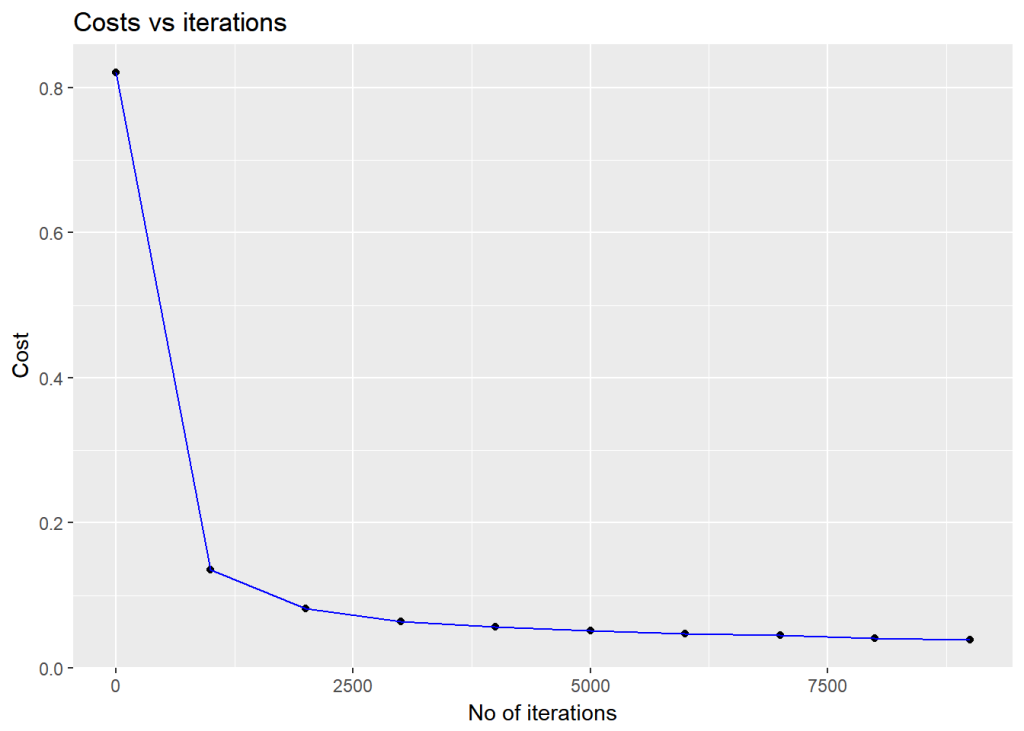
iterations <- seq(0,9000,1000)

costs=retvals$costs

df=data.frame(iterations,costs)

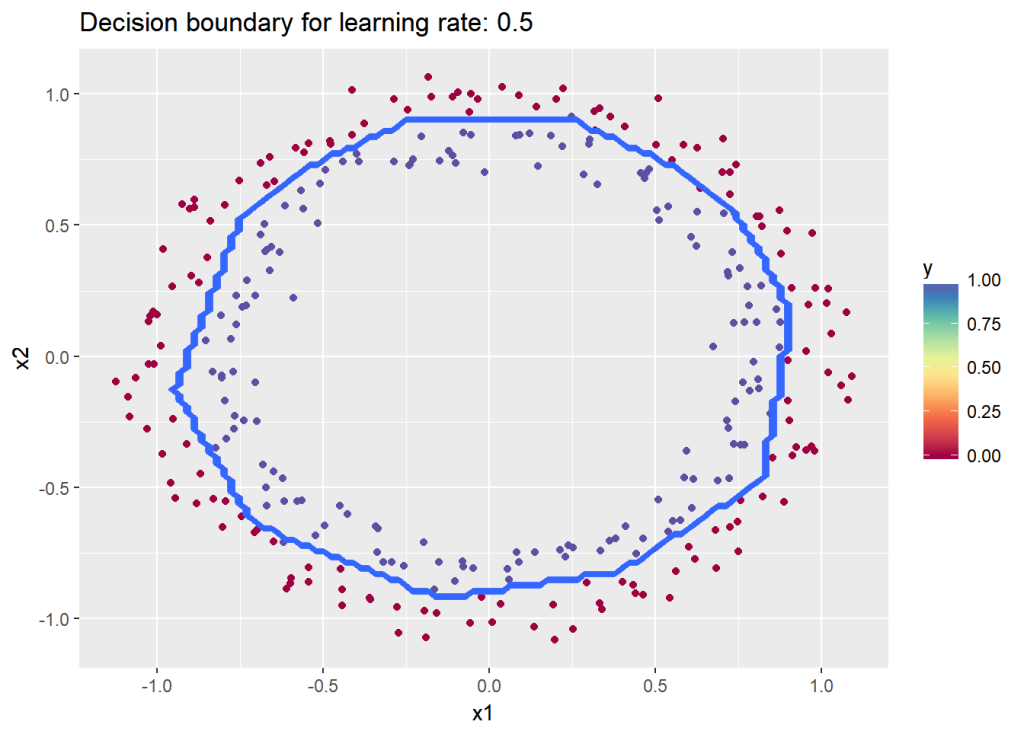
ggplot(df,aes(x=iterations,y=costs)) + geom\_point() + geom\_line(color="blue") +

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



# Plot the decision boundary

plotDecisionBoundary(z,retvals,hiddenActivationFunc="relu",0.5,lr=0.5)



**1.2c Xavier initialization – R**

## Xav initialization

# Set the layer dimensions

layersDimensions = c(2,11,1)

# Train a deep learning network

retvals = L\_Layer\_DeepModel(X, Y, layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",

learningRate = 0.5,

numIterations = 9000,

initType="Xav",

print\_cost = True)

#Plot the cost vs iterations

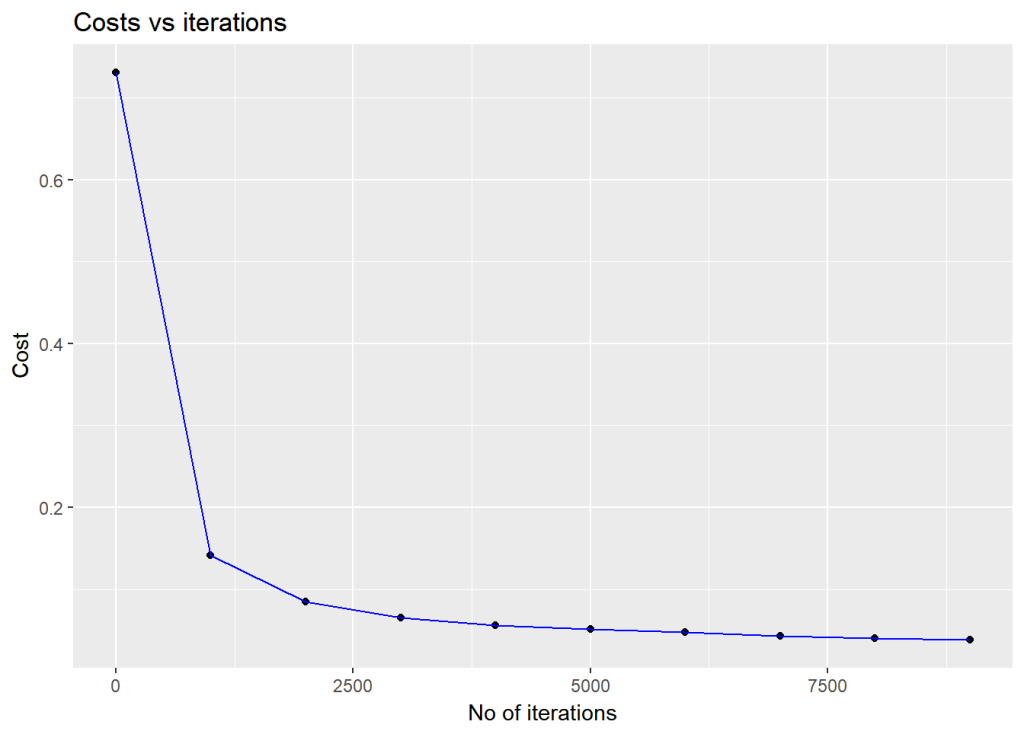
iterations <- seq(0,9000,1000)

costs=retvals$costs

df=data.frame(iterations,costs)

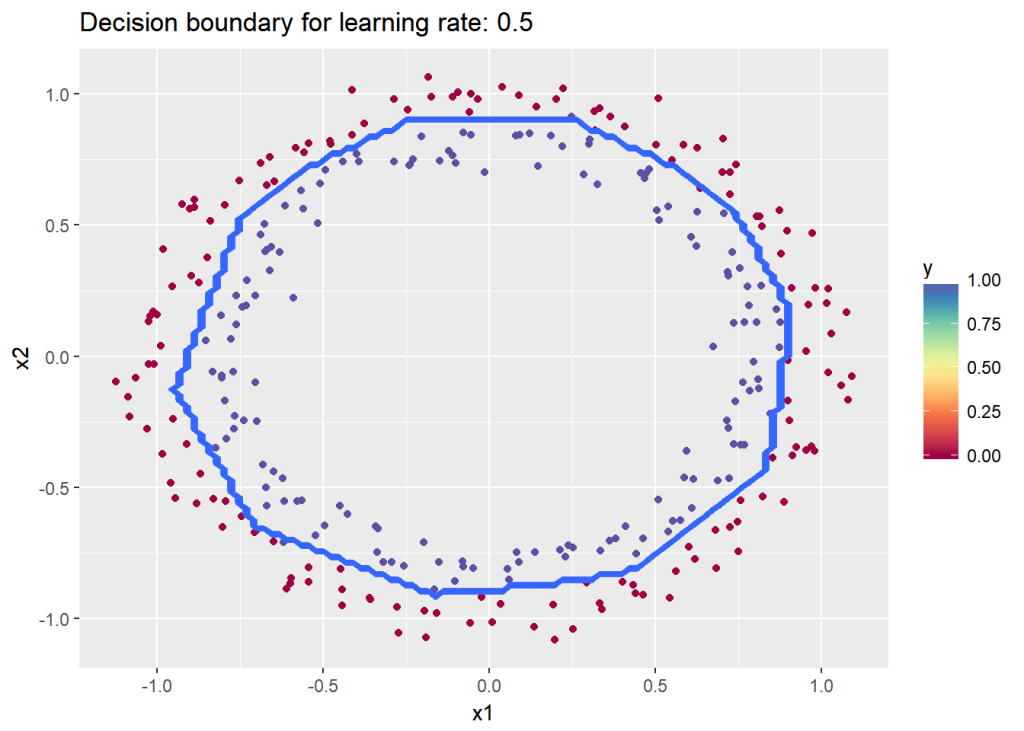
ggplot(df,aes(x=iterations,y=costs)) + geom\_point() + geom\_line(color="blue") +

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



# Plot the decision boundary

plotDecisionBoundary(z,retvals,hiddenActivationFunc="relu",0.5)



**1.3a Default initialization – Octave**

source("DL61functions.m")

# Read the data

data=csvread("circles.csv");

X=data(:,1:2);

Y=data(:,3);

# Set the layer dimensions

layersDimensions = [2 11 1]; #tanh=-0.5(ok), #relu=0.1 best!

# Train a deep learning network

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

hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",

learningRate = 0.5,

lambd=0,

keep\_prob=1,

numIterations = 10000,

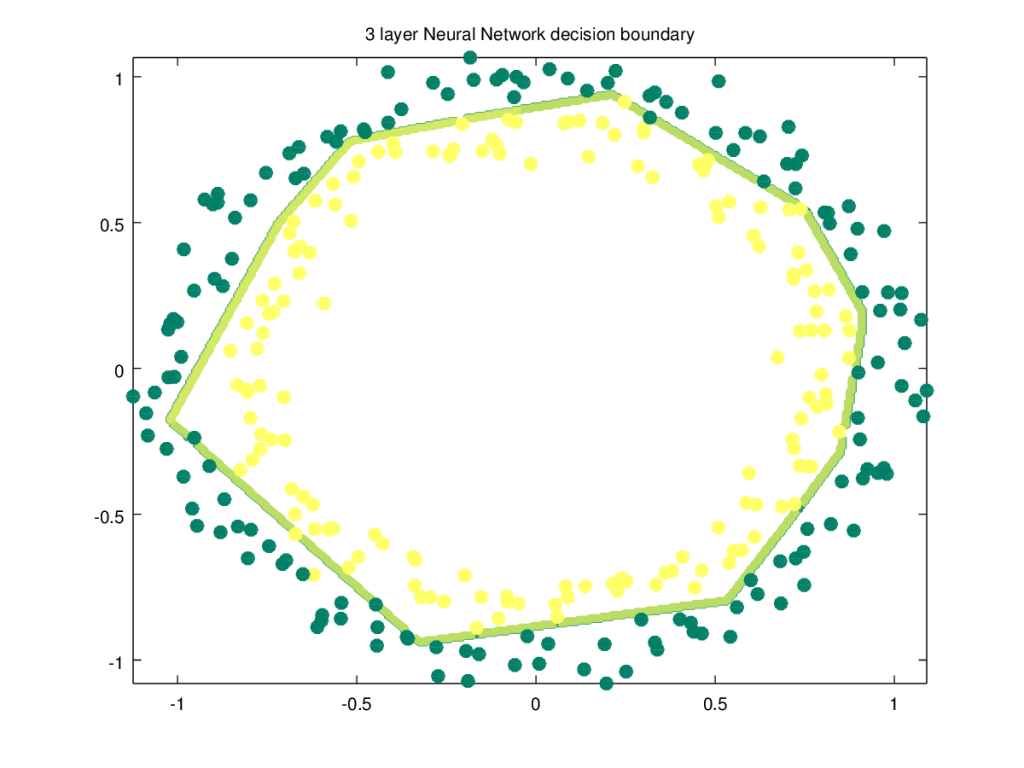
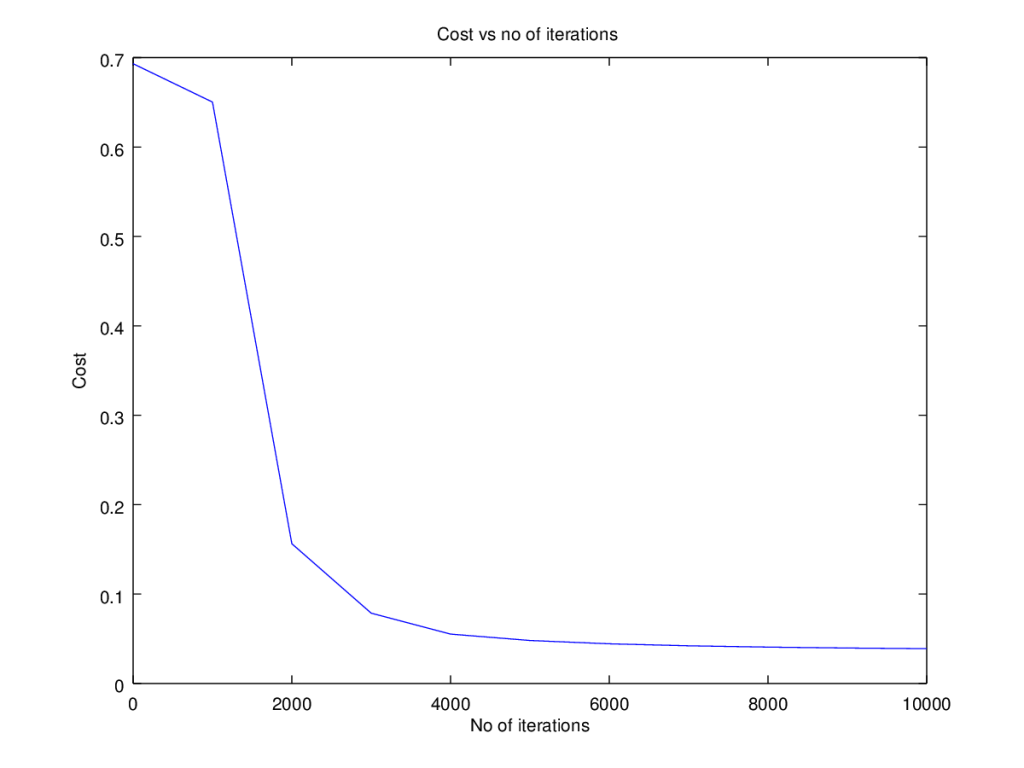
initType="default");

# Plot cost vs iterations

plotCostVsIterations(10000,costs)

#Plot decision boundary

plotDecisionBoundary(data,weights, biases,keep\_prob=1, hiddenActivationFunc="relu")

****

**1.3b He initialization – Octave**

source("DL61functions.m")

#Load data

data=csvread("circles.csv");

X=data(:,1:2);

Y=data(:,3);

# Set the layer dimensions

layersDimensions = [2 11 1]; #tanh=-0.5(ok), #relu=0.1 best!

# Train a deep learning network

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

hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",

learningRate = 0.5,

lambd=0,

keep\_prob=1,

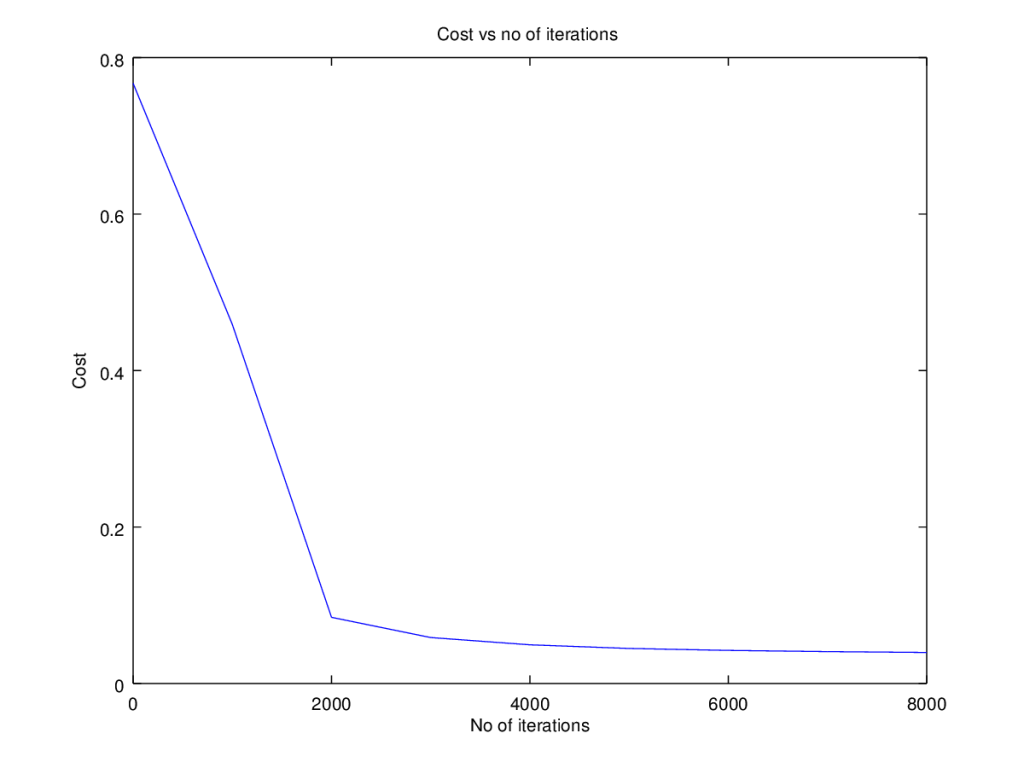
numIterations = 8000,

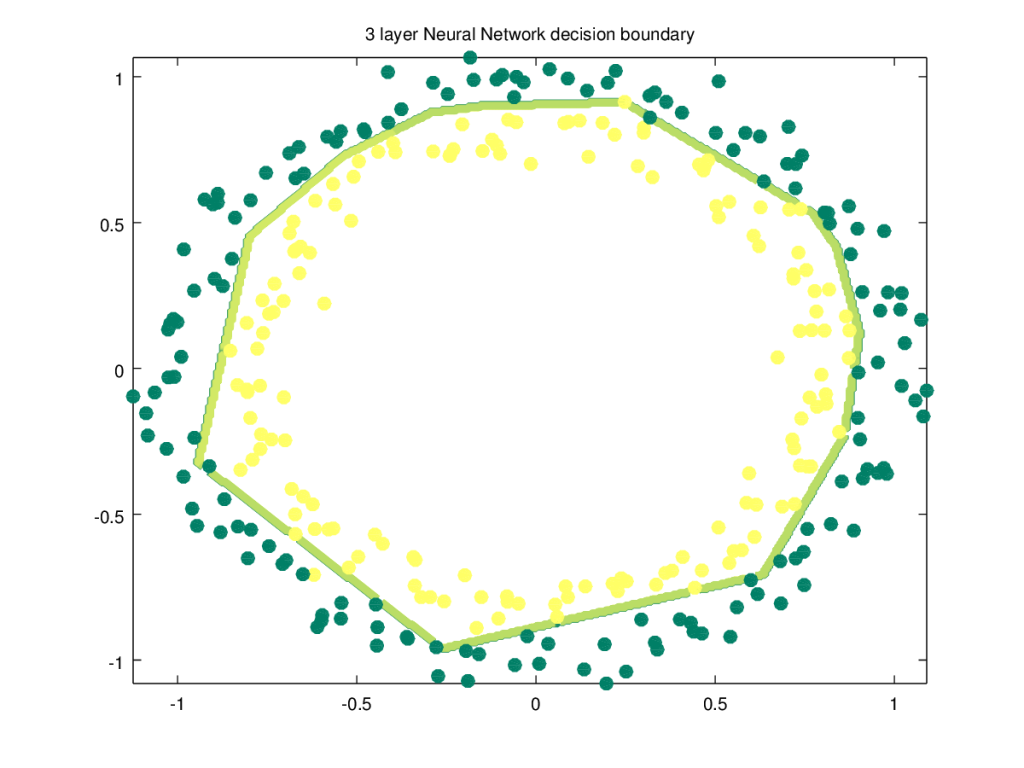
initType="He");

plotCostVsIterations(8000,costs)

#Plot decision boundary

plotDecisionBoundary(data,weights, biases,keep\_prob=1,hiddenActivationFunc="relu")





**1.3c Xavier initialization – Octave**

The code snippet for Xavier initialization in Octave is shown below

source("DL61functions.m")

# 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

The Octave code below uses Xavier initialization

source("DL61functions.m")

#Load data

data=csvread("circles.csv");

X=data(:,1:2);

Y=data(:,3);

#Set layer dimensions

layersDimensions = [2 11 1]; #tanh=-0.5(ok), #relu=0.1 best!

# Train a deep learning network

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

hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",

learningRate = 0.5,

lambd=0,

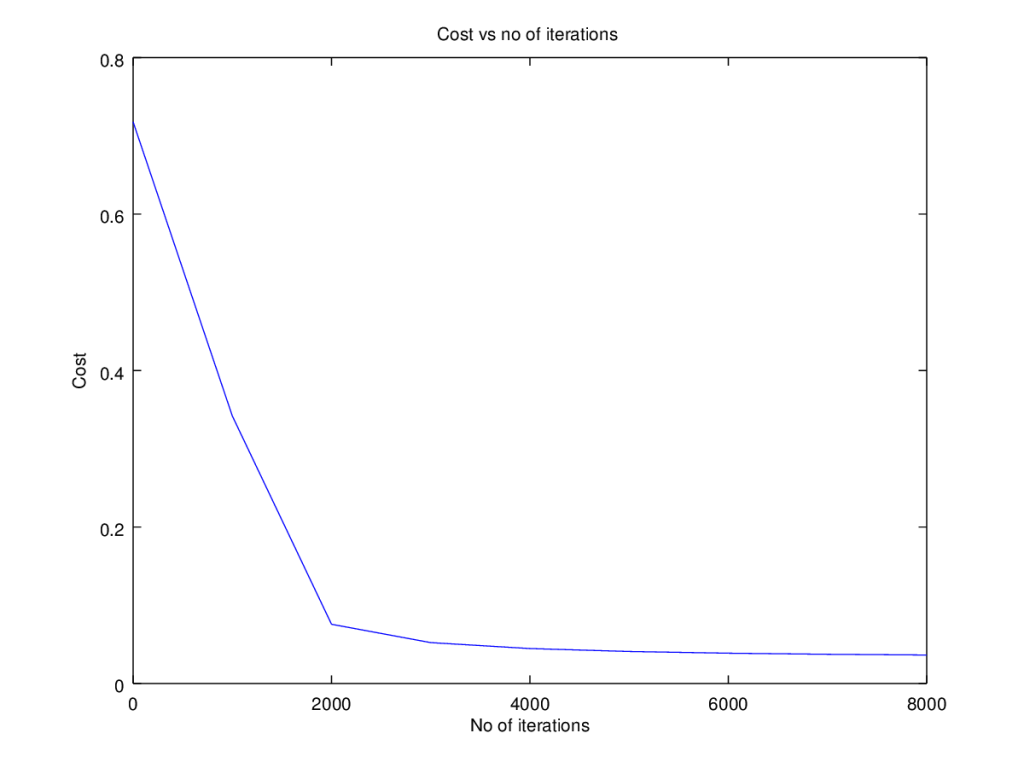
keep\_prob=1,

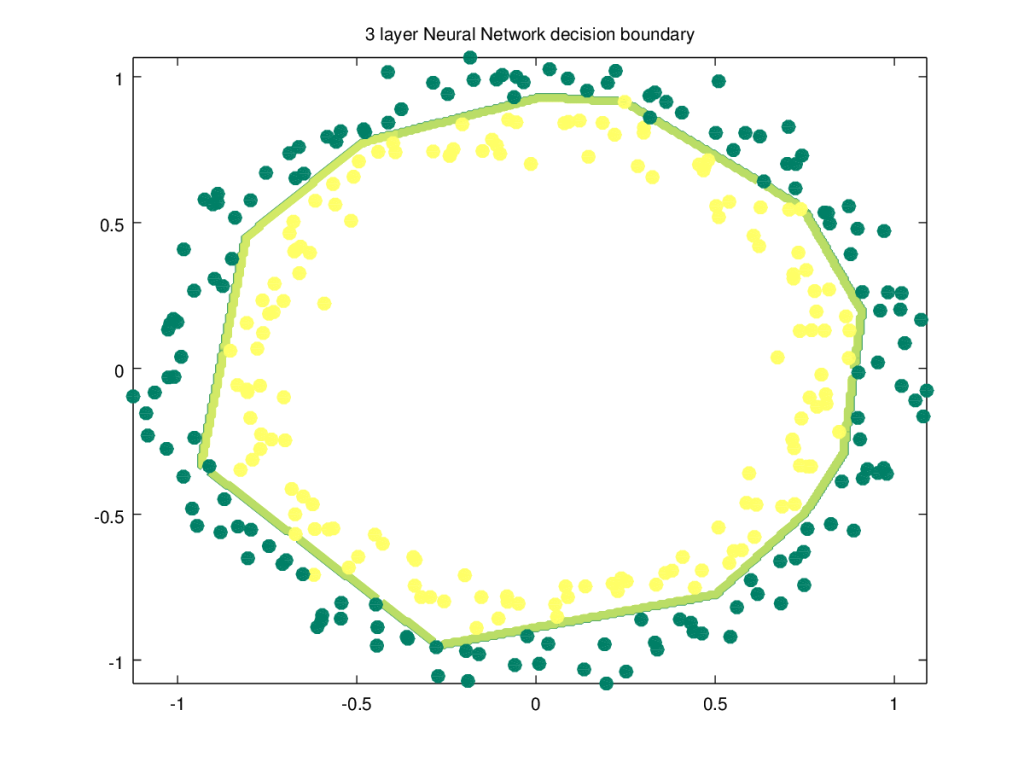
numIterations = 8000,

initType="Xav");

plotCostVsIterations(8000,costs)

plotDecisionBoundary(data,weights, biases,keep\_prob=1,hiddenActivationFunc="relu")





**2.1a Regularization : Circles data – Python**

The cross entropy cost for Logistic classification is given as J = \frac{1}{m}\sum_{i=1}^{m}y^{i}log((a^{L})^{(i)}) - (1-y^{i})log((a^{L})^{(i)})The regularized L2 cost is given by J = \frac{1}{m}\sum_{i=1}^{m}y^{i}log((a^{L})^{(i)}) - (1-y^{i})log((a^{L})^{(i)}) + \frac{\lambda}{2m}\sum \sum \sum W_{kj}^{l}

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("DLfunctions61.py").read())

#Load the data

train\_X, train\_Y, test\_X, test\_Y = load\_dataset()

# Set the layers dimensions

layersDimensions = [2,7,1]

# Train a deep learning network

parameters = L\_Layer\_DeepModel(train\_X, train\_Y, layersDimensions, hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",learningRate = 0.6, lambd=0.1, num\_iterations = 9000,

initType="default", print\_cost = True,figure="fig7.png")

# Clear the plot

plt.clf()

plt.close()

# Plot the decision boundary

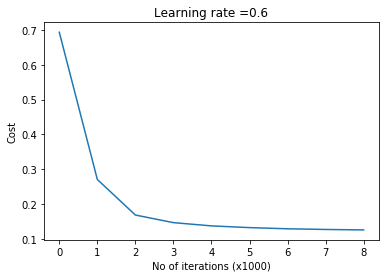
plot\_decision\_boundary(lambda x: predict(parameters, x.T), train\_X, train\_Y,str(0.6),figure1="fig8.png")

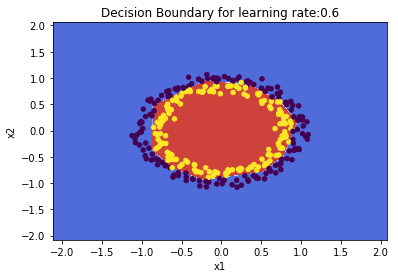
plt.clf()

plt.close()

#Plot the decision boundary

plot\_decision\_boundary(lambda x: predict(parameters, x.T,keep\_prob=0.9), train\_X, train\_Y,str(2.2),"fig8.png",)





**2.1 b Regularization: Spiral data  – 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("DLfunctions61.py").read())

N = 100 # number of points per class

D = 2 # dimensionality

K = 3 # number of classes

X = np.zeros((N\*K,D)) # data matrix (each row = single example)

y = np.zeros(N\*K, dtype='uint8') # class labels

for j in range(K):

ix = range(N\*j,N\*(j+1))

r = np.linspace(0.0,1,N) # radius

t = np.linspace(j\*4,(j+1)\*4,N) + np.random.randn(N)\*0.2 # theta

X[ix] = np.c\_[r\*np.sin(t), r\*np.cos(t)]

y[ix] = j

# Plot the data

plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)

plt.clf()

plt.close()

#Set layer dimensions

layersDimensions = [2,100,3]

y1=y.reshape(-1,1).T

# Train a deep learning network

parameters = L\_Layer\_DeepModel(X.T, y1, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="softmax",

learningRate = 1,lambd=1e-3, num\_iterations = 5000, print\_cost = True,figure="fig9.png")

plt.clf()

plt.close()

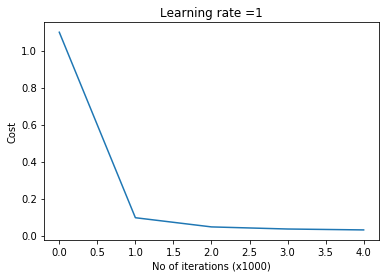
W1=parameters['W1']

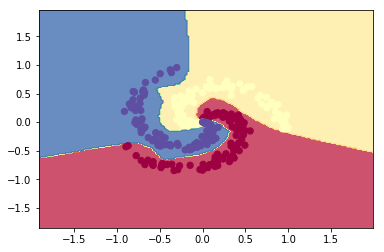
b1=parameters['b1']

W2=parameters['W2']

b2=parameters['b2']

plot\_decision\_boundary1(X, y1,W1,b1,W2,b2,figure2="fig10.png")





**2.2a Regularization: Circles data  – R**

source("DLfunctions61.R")

#Load data

df=read.csv("circles.csv",header=FALSE)

z <- as.matrix(read.csv("circles.csv",header=FALSE))

x <- z[,1:2]

y <- z[,3]

X <- t(x)

Y <- t(y)

#Set layer dimensions

layersDimensions = c(2,11,1)

# Train a deep learning network

retvals = L\_Layer\_DeepModel(X, Y, layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",

learningRate = 0.5,

lambd=0.1,

numIterations = 9000,

initType="default",

print\_cost = True)

#Plot the cost vs iterations

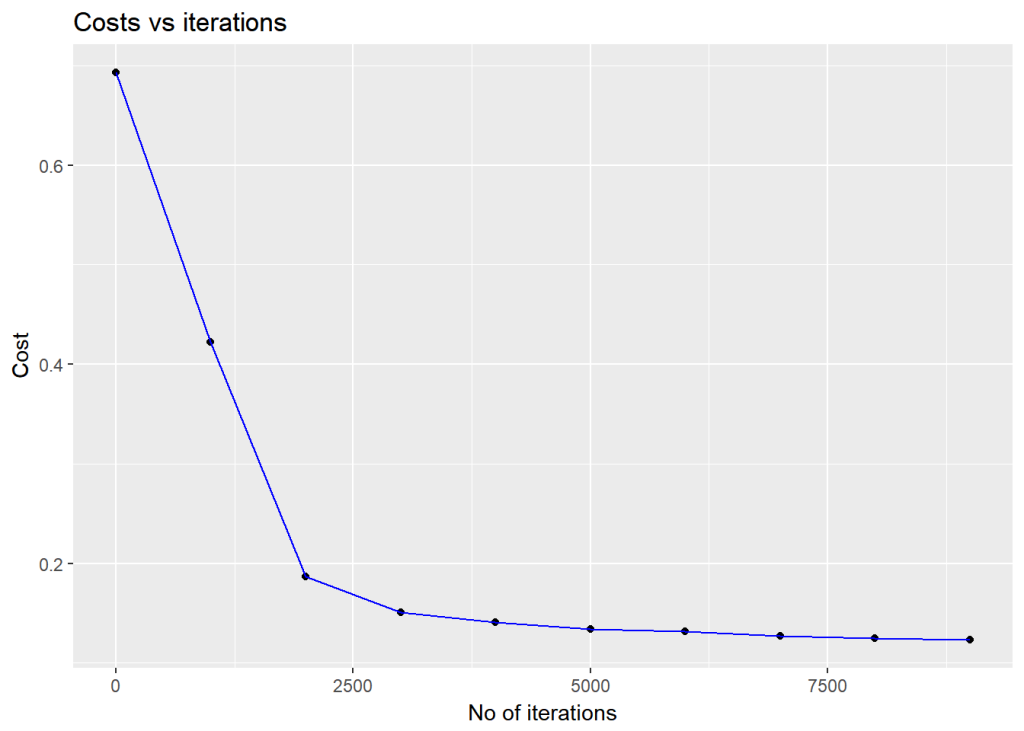
iterations <- seq(0,9000,1000)

costs=retvals$costs

df=data.frame(iterations,costs)

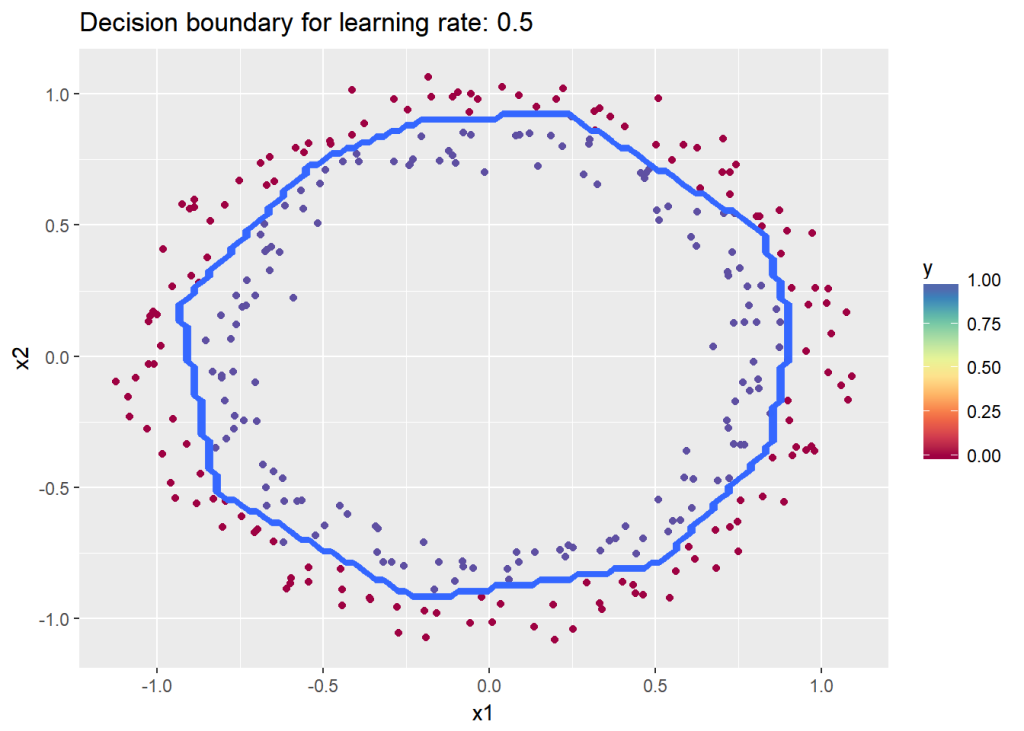
ggplot(df,aes(x=iterations,y=costs)) + geom\_point() + geom\_line(color="blue") +

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



# Plot the decision boundary

plotDecisionBoundary(z,retvals,hiddenActivationFunc="relu",0.5)



**2.2b Regularization:Spiral data – R**

# Read the spiral dataset

#Load the data

source("DLfunctions61.R")

Z <- as.matrix(read.csv("spiral.csv",header=FALSE))

# Setup the data

X <- Z[,1:2]

y <- Z[,3]

X <- t(X)

Y <- t(y)

layersDimensions = c(2, 100, 3)

# Train a deep learning network

retvals = L\_Layer\_DeepModel(X, Y, layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.5,

lambd=0.01,

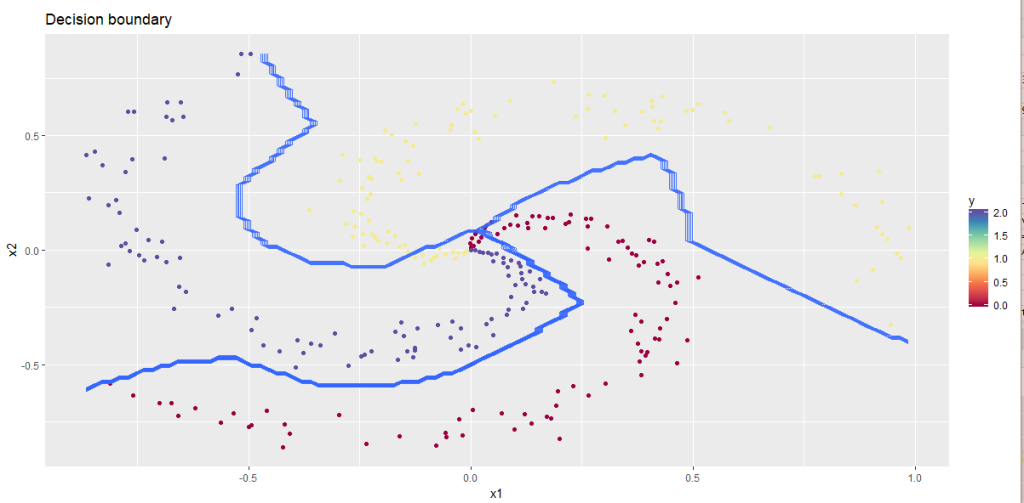
numIterations = 9000,

print\_cost = True)

print\_cost = True)

parameters<-retvals$parameters

plotDecisionBoundary1(Z,parameters)

  
**2.3a Regularization: Circles data – Octave**

source("DL61functions.m")

#Load data

data=csvread("circles.csv");

X=data(:,1:2);

Y=data(:,3);

layersDimensions = [2 11 1]; #tanh=-0.5(ok), #relu=0.1 best!

# Train a deep learning network

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

hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",

learningRate = 0.5,

lambd=0.2,

keep\_prob=1,

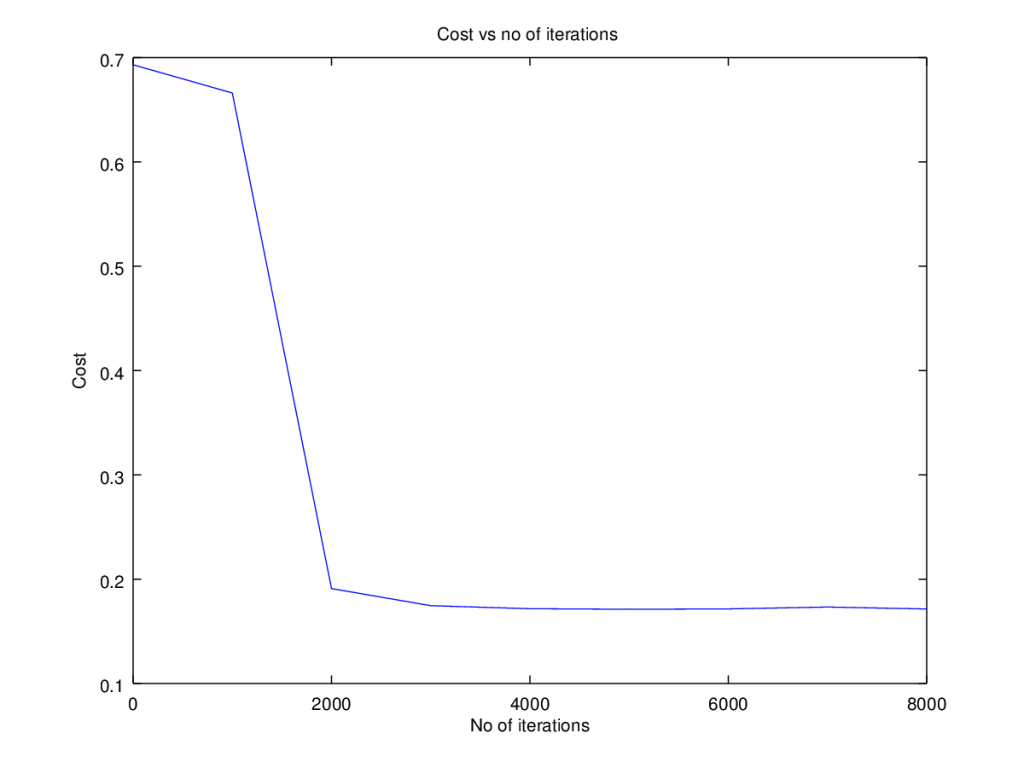
numIterations = 8000,

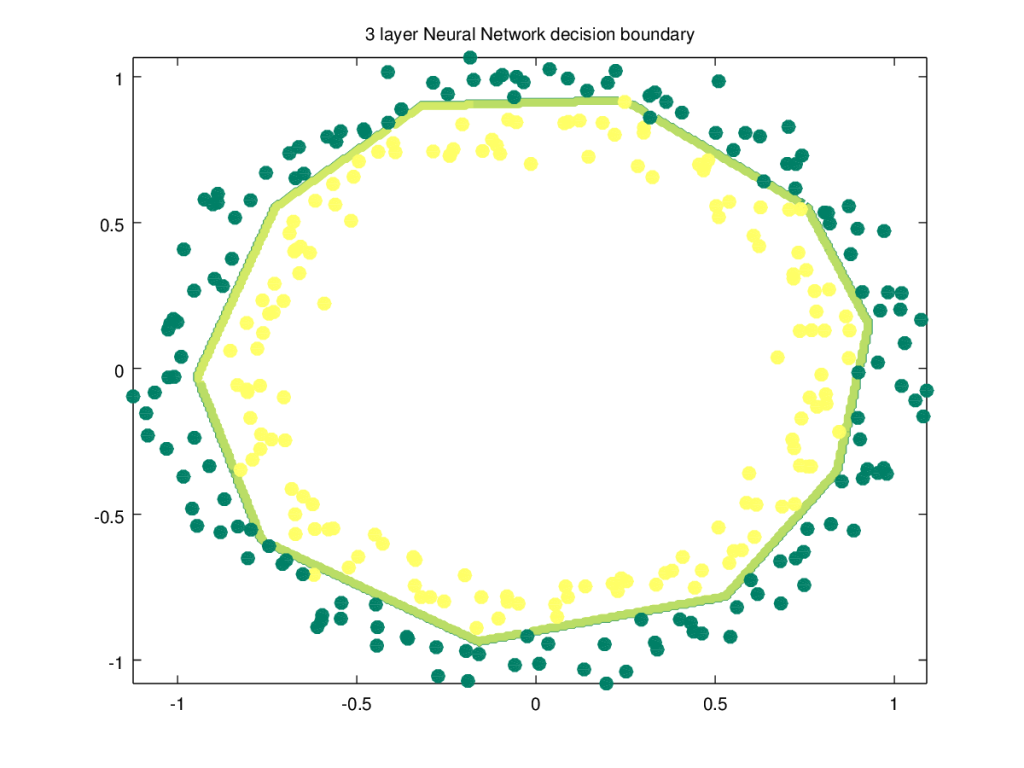
initType="default");

plotCostVsIterations(8000,costs)

#Plot decision boundary

plotDecisionBoundary(data,weights, biases,keep\_prob=1,hiddenActivationFunc="relu")





**2.3b Regularization:Spiral data  2 – Octave**

source("DL61functions.m")

data=csvread("spiral.csv");

# Setup the data

X=data(:,1:2);

Y=data(:,3);

layersDimensions = [2 100 3]

# Train a deep learning network

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

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.6,

lambd=0.2,

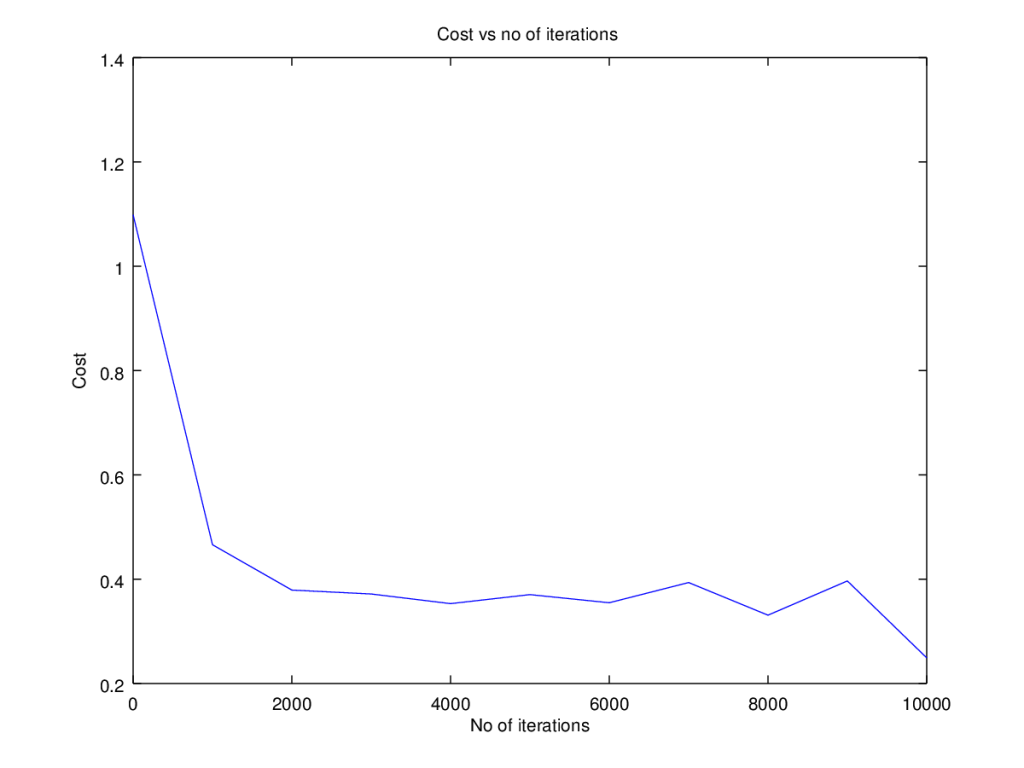
keep\_prob=1,

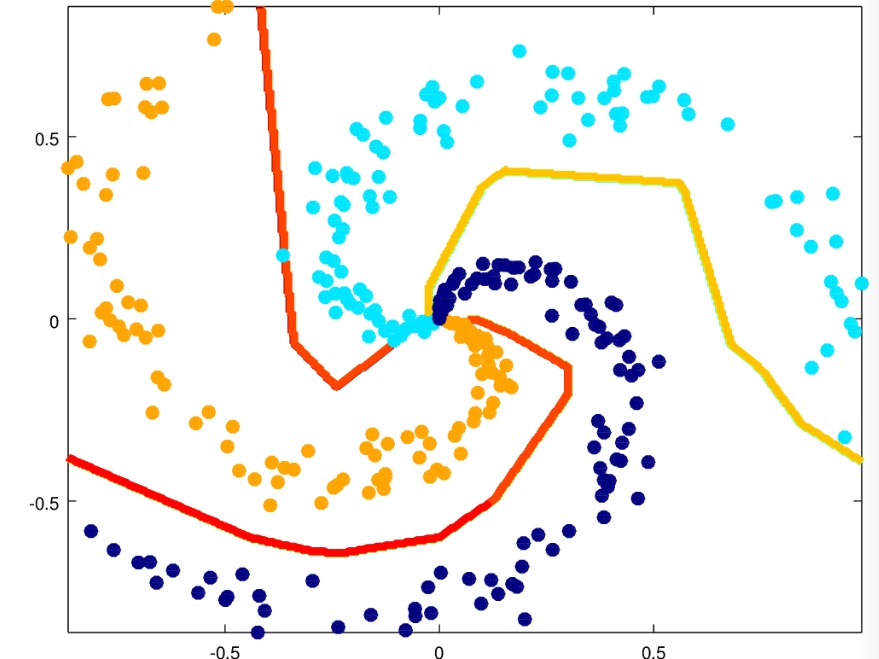
numIterations = 10000);

plotCostVsIterations(10000,costs)

#Plot decision boundary

plotDecisionBoundary1(data,weights, biases,keep\_prob=1,hiddenActivationFunc="relu")





**3.1 a Dropout: Circles data – Python**

The ‘dropout’ regularization technique was used with great effectiveness, to prevent overfitting  by Alex Krizhevsky, Ilya Sutskever and Prof Geoffrey E. Hinton in the [Imagenet classification with Deep Convolutional Neural Networks](https://www.nvidia.cn/content/tesla/pdf/machine-learning/imagenet-classification-with-deep-convolutional-nn.pdf)

The technique of dropout works by dropping a random set of activation units in each hidden layer, based on a ‘keep\_prob’ criteria in the forward propagation cycle. Here is the code for Octave. A ‘dropoutMat’ is created for each layer which specifies which units to drop **Note**: The same ‘dropoutMat has to be used which computing the gradients in the backward propagation cycle. Hence the dropout matrices are stored in a cell array.

 for l =1:L-1

...

D=rand(size(A)(1),size(A)(2));

D = (D < keep\_prob) ;

# Zero out some hidden units

A= A .\* D;

# Divide by keep\_prob to keep the expected value of A the same

A = A ./ keep\_prob;

# Store D in a dropoutMat cell array

dropoutMat{l}=D;

...

endfor

In the backward propagation cycle we have

for l =(L-1):-1:1

...

D = dropoutMat{l};

# Zero out the dAl based on same dropout matrix

dAl= dAl .\* D;

# Divide by keep\_prob to maintain the expected value

dAl = dAl ./ keep\_prob;

...

endfor

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("DLfunctions61.py").read())

#Load the data

train\_X, train\_Y, test\_X, test\_Y = load\_dataset()

# Set the layers dimensions

layersDimensions = [2,7,1]

# Train a deep learning network

parameters = L\_Layer\_DeepModel(train\_X, train\_Y, layersDimensions, hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",learningRate = 0.6, keep\_prob=0.7, num\_iterations = 9000,

initType="default", print\_cost = True,figure="fig11.png")

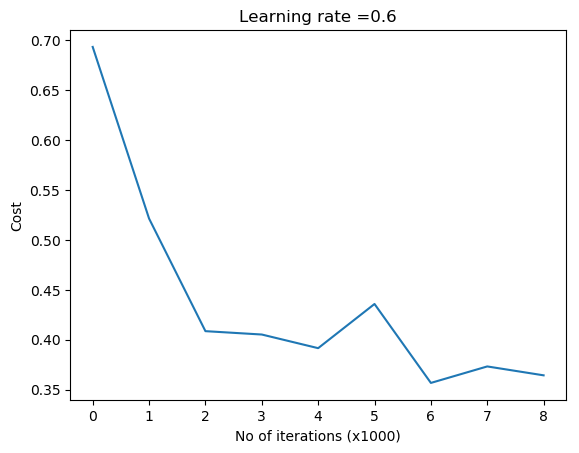
# Clear the plot

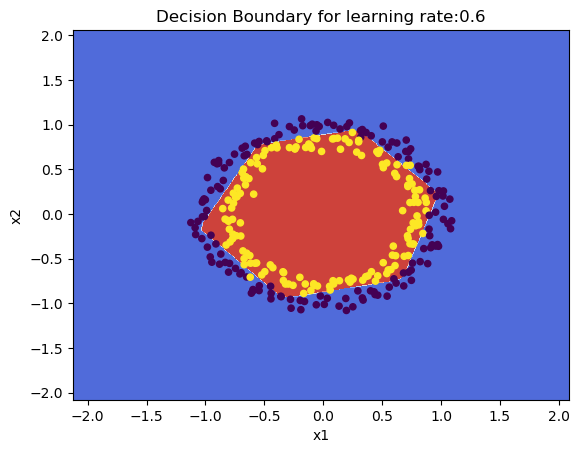
plt.clf()

plt.close()

# Plot the decision boundary

plot\_decision\_boundary(lambda x: predict(parameters, x.T,keep\_prob=0.7), train\_X, train\_Y,str(0.6),figure1="fig12.png")





**3.1b Dropout: Spiral data – 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("DLfunctions61.py").read())

# Create an input data set - Taken from CS231n Convolutional Neural networks,

# http://cs231n.github.io/neural-networks-case-study/

N = 100 # number of points per class

D = 2 # dimensionality

K = 3 # number of classes

X = np.zeros((N\*K,D)) # data matrix (each row = single example)

y = np.zeros(N\*K, dtype='uint8') # class labels

for j in range(K):

ix = range(N\*j,N\*(j+1))

r = np.linspace(0.0,1,N) # radius

t = np.linspace(j\*4,(j+1)\*4,N) + np.random.randn(N)\*0.2 # theta

X[ix] = np.c\_[r\*np.sin(t), r\*np.cos(t)]

y[ix] = j

# Plot the data

plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)

plt.clf()

plt.close()

layersDimensions = [2,100,3]

y1=y.reshape(-1,1).T

# Train a deep learning network

parameters = L\_Layer\_DeepModel(X.T, y1, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="softmax",

learningRate = 1,keep\_prob=0.9, num\_iterations = 5000, print\_cost = True,figure="fig13.png")

plt.clf()

plt.close()

W1=parameters['W1']

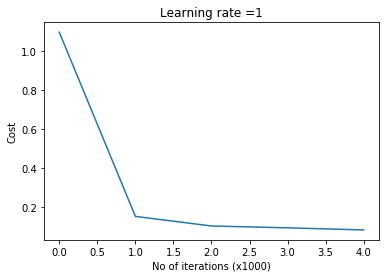
b1=parameters['b1']

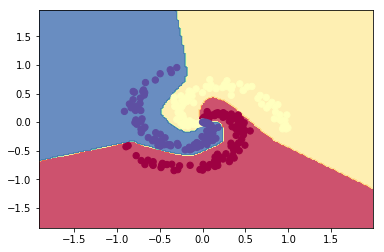
W2=parameters['W2']

b2=parameters['b2']

#Plot decision boundary

plot\_decision\_boundary1(X, y1,W1,b1,W2,b2,figure2="fig14.png")





**3.2a Dropout: Circles data – R**

source("DLfunctions61.R")

#Load data

df=read.csv("circles.csv",header=FALSE)

z <- as.matrix(read.csv("circles.csv",header=FALSE))

x <- z[,1:2]

y <- z[,3]

X <- t(x)

Y <- t(y)

layersDimensions = c(2,11,1)

# Train a deep learning network

retvals = L\_Layer\_DeepModel(X, Y, layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",

learningRate = 0.5,

keep\_prob=0.8,

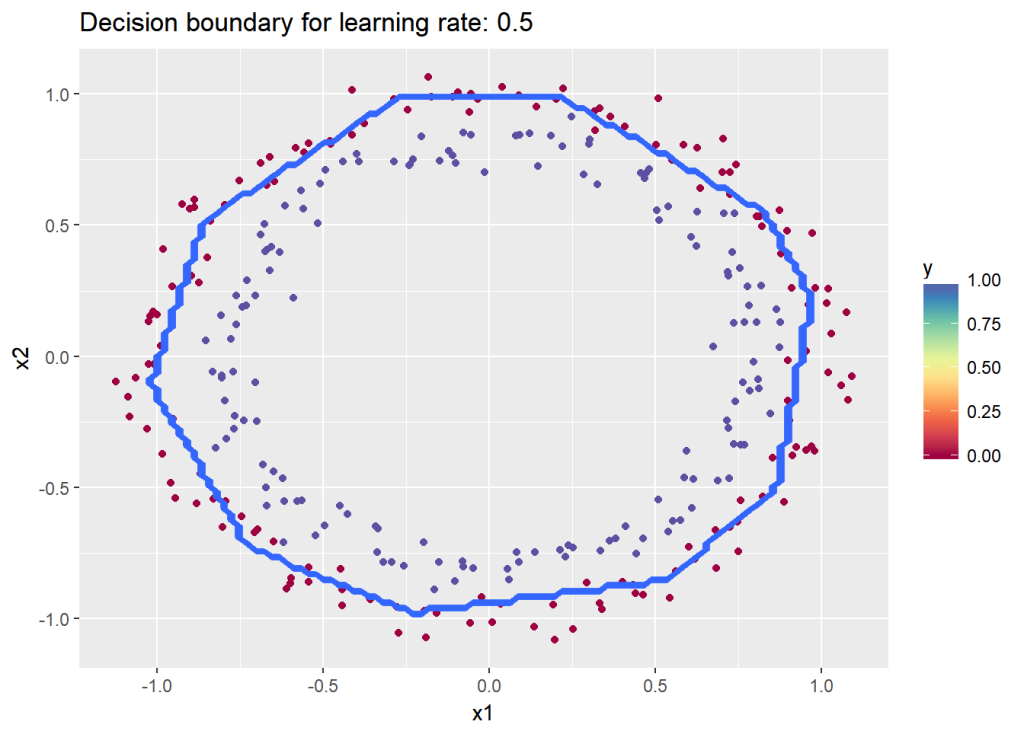
numIterations = 9000,

initType="default",

print\_cost = True)

# Plot the decision boundary

plotDecisionBoundary(z,retvals,keep\_prob=0.6, hiddenActivationFunc="relu",0.5)



**3.2b Dropout: Spiral data – R**

# Read the spiral dataset

source("DLfunctions61.R")

# Load data

Z <- as.matrix(read.csv("spiral.csv",header=FALSE))

# Setup the data

X <- Z[,1:2]

y <- Z[,3]

X <- t(X)

Y <- t(y)

# Train a deep learning network

retvals = L\_Layer\_DeepModel(X, Y, layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.1,

keep\_prob=0.90,

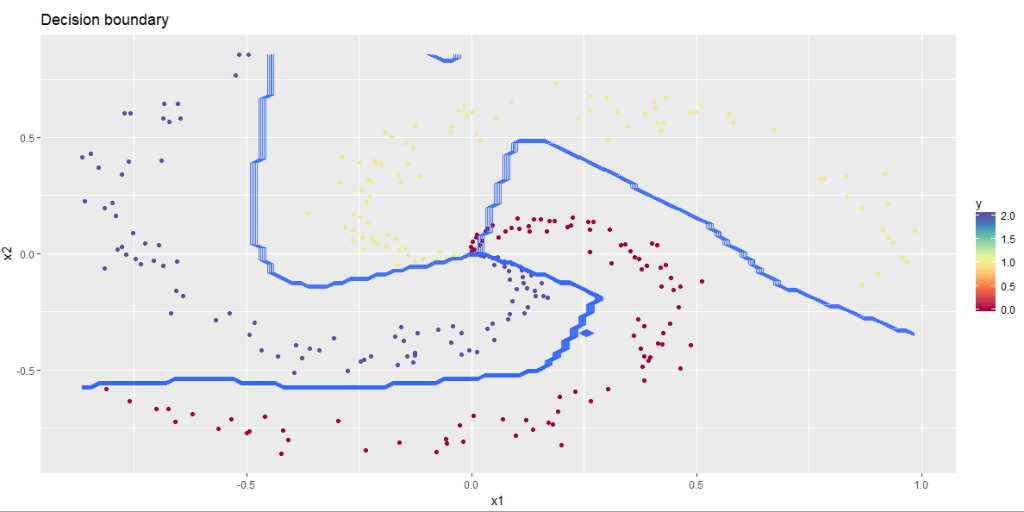
numIterations = 9000,

print\_cost = True)

parameters<-retvals$parameters

#Plot decision boundary

plotDecisionBoundary1(Z,parameters)



**3.3a Dropout: Circles data – Octave**

data=csvread("circles.csv");

X=data(:,1:2);

Y=data(:,3);

layersDimensions = [2 11 1]; #tanh=-0.5(ok), #relu=0.1 best!

# Train a deep learning network

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

hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",

learningRate = 0.5,

lambd=0,

keep\_prob=0.8,

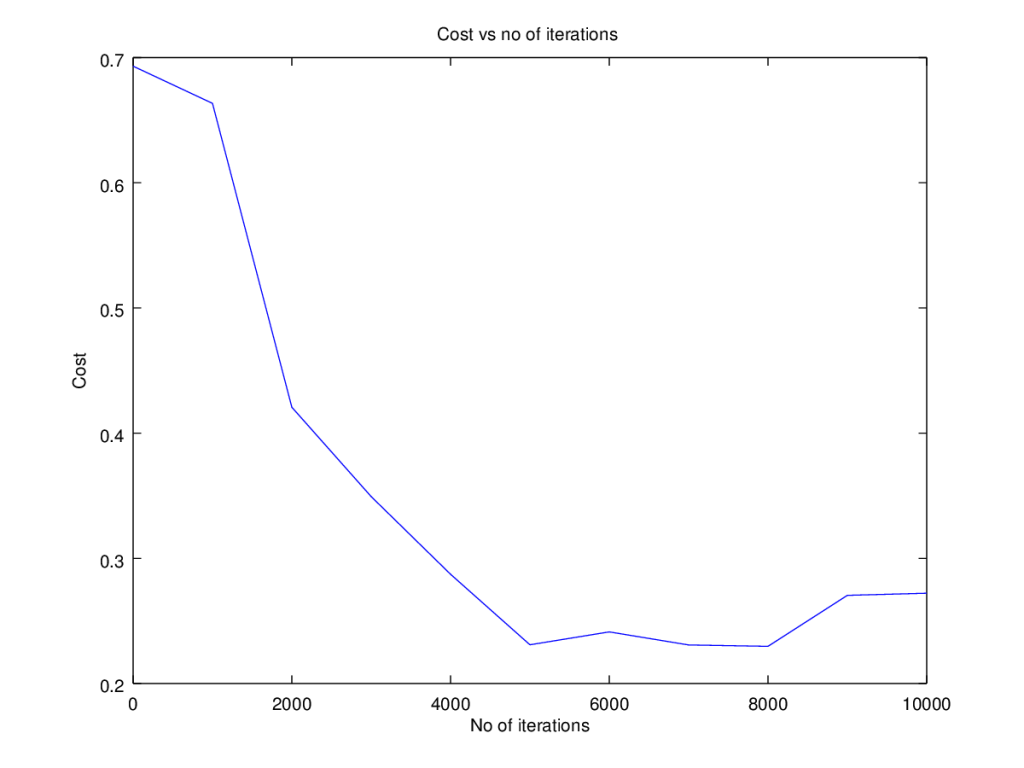
numIterations = 10000,

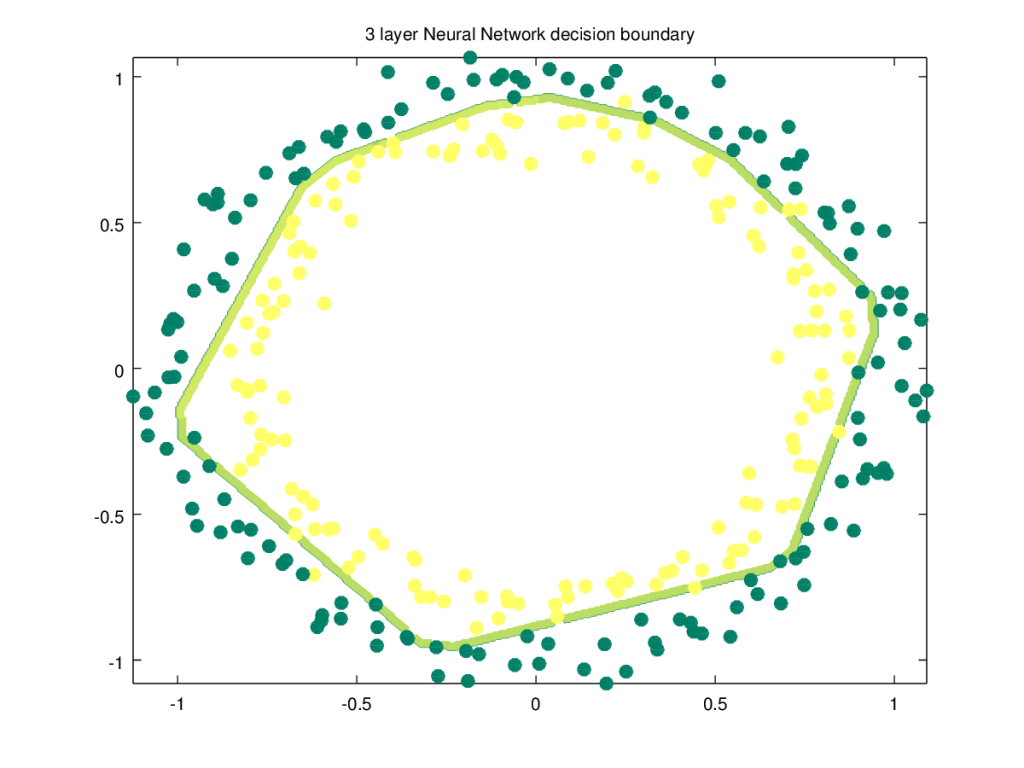
initType="default");

plotCostVsIterations(10000,costs)

#Plot decision boundary

plotDecisionBoundary1(data,weights, biases,keep\_prob=1, hiddenActivationFunc="relu")





**3.3b Dropout  Spiral data – Octave**

source("DL61functions.m")

data=csvread("spiral.csv");

# Setup the data

X=data(:,1:2);

Y=data(:,3);

layersDimensions = [numFeats numHidden numOutput];

# Train a deep learning network

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

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.1,

lambd=0,

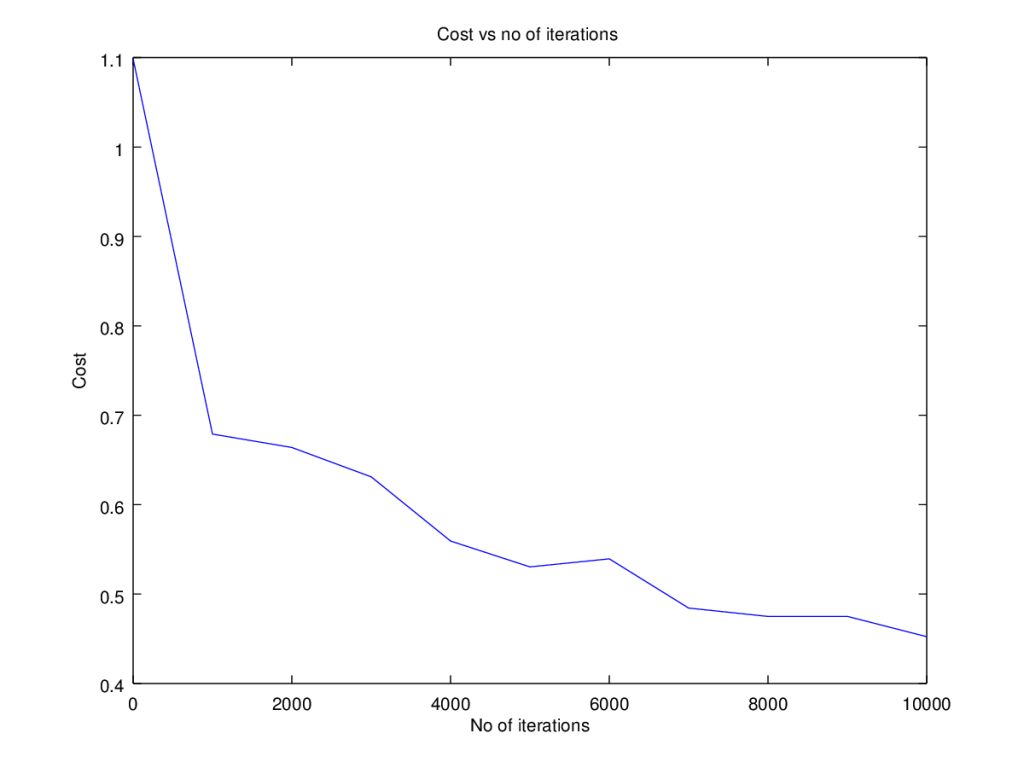
keep\_prob=0.8,

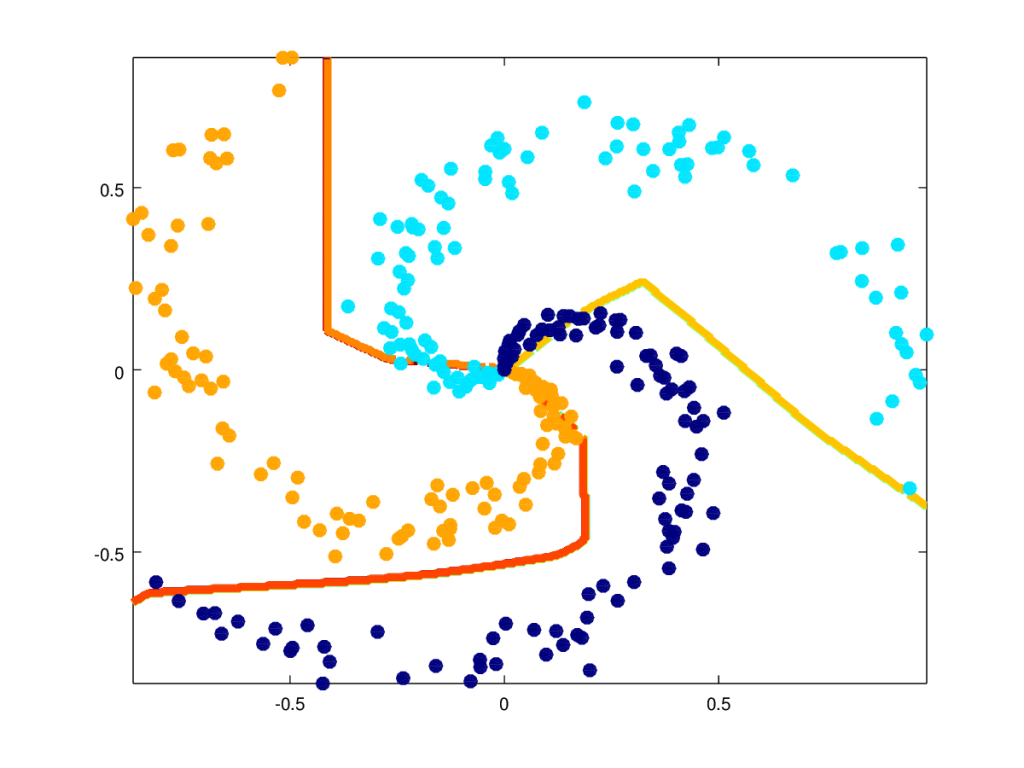
numIterations = 10000);

plotCostVsIterations(10000,costs)

#Plot decision boundary

plotDecisionBoundary1(data,weights, biases,keep\_prob=1, hiddenActivationFunc="relu")





DLFunctions61.py

|  |
| --- |
| ###################################################### |
|  | # DL functions |
|  | ###################################################### |
|  | import numpy as np |
|  | import matplotlib.pyplot as plt |
|  | import matplotlib |
|  | import matplotlib.pyplot as plt |
|  | from matplotlib import cm |
|  | import math |
|  | import sklearn |
|  | import sklearn.datasets |
|  |  |
|  | # Conmpute the sigmoid of a vector |
|  | def sigmoid(Z): |
|  | A=1/(1+np.exp(-Z)) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Conmpute the Relu of a vector |
|  | def relu(Z): |
|  | A = np.maximum(0,Z) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Conmpute the tanh of a vector |
|  | def tanh(Z): |
|  | A = np.tanh(Z) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Conmpute the softmax of a vector |
|  | def softmax(Z): |
|  | # get unnormalized probabilities |
|  | exp\_scores = np.exp(Z.T) |
|  | # normalize them for each example |
|  | A = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Conmpute the softmax of a vector |
|  | def stableSoftmax(Z): |
|  | #Compute the softmax of vector x in a numerically stable way. |
|  | shiftZ = Z.T - np.max(Z.T,axis=1).reshape(-1,1) |
|  | exp\_scores = np.exp(shiftZ) |
|  |  |
|  | # normalize them for each example |
|  | A = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Compute the detivative of Relu |
|  | def reluDerivative(dA, cache): |
|  |  |
|  | Z = cache |
|  | dZ = np.array(dA, copy=True) # just converting dz to a correct object. |
|  | # When z <= 0, you should set dz to 0 as well. |
|  | dZ[Z <= 0] = 0 |
|  | return dZ |
|  |  |
|  | # Compute the derivative of sigmoid |
|  | def sigmoidDerivative(dA, cache): |
|  | Z = cache |
|  | s = 1/(1+np.exp(-Z)) |
|  | dZ = dA \* s \* (1-s) |
|  | return dZ |
|  |  |
|  | # Compute the derivative of tanh |
|  | def tanhDerivative(dA, cache): |
|  | Z = cache |
|  | a = np.tanh(Z) |
|  | dZ = dA \* (1 - np.power(a, 2)) |
|  | return dZ |
|  |  |
|  | # Compute the derivative of softmax |
|  | def softmaxDerivative(dA, cache,y,numTraining): |
|  | # Note : dA not used. dL/dZ = dL/dA \* dA/dZ = pi-yi |
|  | Z = cache |
|  | # Compute softmax |
|  | exp\_scores = np.exp(Z.T) |
|  | # normalize them for each example |
|  | probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) |
|  |  |
|  | # compute the gradient on scores |
|  | dZ = probs |
|  |  |
|  | # dZ = pi- yi |
|  | dZ[range(int(numTraining)),y[:,0]] -= 1 |
|  | return(dZ) |
|  |  |
|  | # Compute the derivative of softmax |
|  | def stableSoftmaxDerivative(dA, cache,y,numTraining): |
|  | # Note : dA not used. dL/dZ = dL/dA \* dA/dZ = pi-yi |
|  | Z = cache |
|  | # Compute stable softmax |
|  | shiftZ = Z.T - np.max(Z.T,axis=1).reshape(-1,1) |
|  | exp\_scores = np.exp(shiftZ) |
|  | # normalize them for each example |
|  | probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) |
|  | #print(probs) |
|  | # compute the gradient on scores |
|  | dZ = probs |
|  |  |
|  | # dZ = pi- yi |
|  | dZ[range(int(numTraining)),y[:,0]] -= 1 |
|  | 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 |
|  | def initializeModel(numFeats,numHidden,numOutput): |
|  | np.random.seed(1) |
|  | W1=np.random.randn(numHidden,numFeats)\*0.01 # Multiply by .01 |
|  | b1=np.zeros((numHidden,1)) |
|  | W2=np.random.randn(numOutput,numHidden)\*0.01 |
|  | b2=np.zeros((numOutput,1)) |
|  |  |
|  | # Create a dictionary of the neural network parameters |
|  | nnParameters={'W1':W1,'b1':b1,'W2':W2,'b2':b2} |
|  | return(nnParameters) |
|  |  |
|  |  |
|  | # Initialize model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | def initializeDeepModel(layerDimensions): |
|  | np.random.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 |
|  | layerParams = {} |
|  | for l in range(1,len(layerDimensions)): |
|  | layerParams['W' + str(l)] = np.random.randn(layerDimensions[l],layerDimensions[l-1])\*0.01 # Multiply by .01 |
|  | layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1)) |
|  |  |
|  | return(layerParams) |
|  | return Z, cache |
|  |  |
|  | # 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[l-1]) |
|  | def HeInitializeDeepModel(layerDimensions): |
|  | np.random.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 |
|  | layerParams = {} |
|  | for l in range(1,len(layerDimensions)): |
|  | layerParams['W' + str(l)] = np.random.randn(layerDimensions[l], |
|  | layerDimensions[l-1])\*np.sqrt(2/layerDimensions[l-1]) |
|  | layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1)) |
|  |  |
|  | return(layerParams) |
|  | return Z, cache |
|  |  |
|  | # Xavier Initialization model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | # Xavier initilization multiplies the random numbers with sqrt(1/layerDimensions[l-1]) |
|  | def XavInitializeDeepModel(layerDimensions): |
|  | np.random.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 |
|  | layerParams = {} |
|  | for l in range(1,len(layerDimensions)): |
|  | layerParams['W' + str(l)] = np.random.randn(layerDimensions[l], |
|  | layerDimensions[l-1])\*np.sqrt(1/layerDimensions[l-1]) |
|  | layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1)) |
|  |  |
|  | return(layerParams) |
|  | return Z, cache |
|  |  |
|  | # 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) |
|  | def layerActivationForward(A\_prev, W, b, keep\_prob=1, activationFunc="relu"): |
|  |  |
|  | # Compute Z |
|  | Z = np.dot(W,A\_prev) + b |
|  | forward\_cache = (A\_prev, W, b) |
|  | # Compute the activation for sigmoid |
|  | if activationFunc == "sigmoid": |
|  | A, activation\_cache = sigmoid(Z) |
|  | # Compute the activation for Relu |
|  | elif activationFunc == "relu": |
|  | A, activation\_cache = relu(Z) |
|  | # Compute the activation for tanh |
|  | elif activationFunc == 'tanh': |
|  | A, activation\_cache = tanh(Z) |
|  | elif activationFunc == 'softmax': |
|  | A, activation\_cache = stableSoftmax(Z) |
|  |  |
|  | cache = (forward\_cache, activation\_cache) |
|  | return A, cache |
|  |  |
|  | # 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 - Activation function at output - sigmoid/softmax |
|  | # Returns : AL |
|  | # caches |
|  | # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and sigmoid actiovation at layer L |
|  | def forwardPropagationDeep(X, parameters,keep\_prob=1, hiddenActivationFunc='relu',outputActivationFunc='sigmoid'): |
|  | caches = [] |
|  | #initialize the dropout matrix |
|  | dropoutMat = {} |
|  | # Set A to X (A0) |
|  | A = X |
|  | L = int(len(parameters)/2) # number of layers in the neural network |
|  | # Loop through from layer 1 to upto layer L |
|  | for l in range(1, L): |
|  | A\_prev = A |
|  | # Zi = Wi x Ai-1 + bi and Ai = g(Zi) |
|  | A, cache = layerActivationForward(A\_prev, parameters['W'+str(l)], parameters['b'+str(l)], keep\_prob, activationFunc = hiddenActivationFunc) |
|  |  |
|  | # Randomly drop some activation units |
|  | # Create a matrix as the same shape as A |
|  | D = np.random.rand(A.shape[0],A.shape[1]) |
|  | D = (D < keep\_prob) |
|  | # We need to use the same 'dropout' matrix in backward propagation |
|  | # Save the dropout matrix for use in backprop |
|  | dropoutMat["D" + str(l)] =D |
|  | A= np.multiply(A,D) |
|  | A = np.divide(A,keep\_prob) |
|  |  |
|  | caches.append(cache) |
|  |  |
|  |  |
|  | # Since this is binary classification use the sigmoid activation function in |
|  | # last layer |
|  | AL, cache = layerActivationForward(A, parameters['W'+str(L)], parameters['b'+str(L)], activationFunc = outputActivationFunc) |
|  | caches.append(cache) |
|  |  |
|  | return AL, caches, dropoutMat |
|  |  |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # : Y |
|  | # :outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | # Output: cost |
|  | def computeCost(parameters,AL,Y,outputActivationFunc="sigmoid"): |
|  | if outputActivationFunc=="sigmoid": |
|  | m= float(Y.shape[1]) |
|  | # Element wise multiply for logprobs |
|  | cost=-1/m \*np.sum(Y\*np.log(AL) + (1-Y)\*(np.log(1-AL))) |
|  | cost = np.squeeze(cost) |
|  | elif outputActivationFunc=="softmax": |
|  | # Take transpose of Y for softmax |
|  | Y=Y.T |
|  | m= float(len(Y)) |
|  | # Compute log probs. Take the log prob of correct class based on output y |
|  | correct\_logprobs = -np.log(AL[range(int(m)),Y.T]) |
|  | # Conpute loss |
|  | cost = np.sum(correct\_logprobs)/m |
|  | return cost |
|  |  |
|  |  |
|  | # Compute the cost with regularization |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # : Y |
|  | # :outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | # Output: cost |
|  | def computeCostWithReg(parameters,AL,Y,lambd, outputActivationFunc="sigmoid"): |
|  |  |
|  |  |
|  | if outputActivationFunc=="sigmoid": |
|  | m= float(Y.shape[1]) |
|  | # Element wise multiply for logprobs |
|  | cost=-1/m \*np.sum(Y\*np.log(AL) + (1-Y)\*(np.log(1-AL))) |
|  | cost = np.squeeze(cost) |
|  |  |
|  | # Regularization cost |
|  | L= int(len(parameters)/2) |
|  | L2RegularizationCost=0 |
|  | for l in range(L): |
|  | L2RegularizationCost+=np.sum(np.square(parameters['W'+str(l+1)])) |
|  |  |
|  | L2RegularizationCost = (lambd/(2\*m))\*L2RegularizationCost |
|  | cost = cost + L2RegularizationCost |
|  |  |
|  |  |
|  | elif outputActivationFunc=="softmax": |
|  | # Take transpose of Y for softmax |
|  | Y=Y.T |
|  | m= float(len(Y)) |
|  | # Compute log probs. Take the log prob of correct class based on output y |
|  | correct\_logprobs = -np.log(AL[range(int(m)),Y.T]) |
|  | # Conpute loss |
|  | cost = np.sum(correct\_logprobs)/m |
|  |  |
|  | # Regularization cost |
|  | L= int(len(parameters)/2) |
|  | L2RegularizationCost=0 |
|  | for l in range(L): |
|  | L2RegularizationCost+=np.sum(np.square(parameters['W'+str(l+1)])) |
|  |  |
|  | L2RegularizationCost = (lambd/(2\*m))\*L2RegularizationCost |
|  | cost = cost + L2RegularizationCost |
|  |  |
|  | return cost |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  | def layerActivationBackward(dA, cache, Y, keep\_prob=1, activationFunc="relu"): |
|  | forward\_cache, activation\_cache = cache |
|  | A\_prev, W, b = forward\_cache |
|  | numtraining = float(A\_prev.shape[1]) |
|  | #print("n=",numtraining) |
|  | #print("no=",numtraining) |
|  | if activationFunc == "relu": |
|  | dZ = reluDerivative(dA, activation\_cache) |
|  | elif activationFunc == "sigmoid": |
|  | dZ = sigmoidDerivative(dA, activation\_cache) |
|  | elif activationFunc == "tanh": |
|  | dZ = tanhDerivative(dA, activation\_cache) |
|  | elif activationFunc == "softmax": |
|  | dZ = stableSoftmaxDerivative(dA, activation\_cache,Y,numtraining) |
|  |  |
|  | if activationFunc == 'softmax': |
|  | dW = 1/numtraining \* np.dot(A\_prev,dZ) |
|  | db = 1/numtraining \* np.sum(dZ, axis=0, keepdims=True) |
|  | dA\_prev = np.dot(dZ,W) |
|  |  |
|  |  |
|  | else: |
|  | #print(numtraining) |
|  | dW = 1/numtraining \*(np.dot(dZ,A\_prev.T)) |
|  | #print("dW=",dW) |
|  | db = 1/numtraining \* np.sum(dZ, axis=1, keepdims=True) |
|  | #print("db=",db) |
|  | dA\_prev = np.dot(W.T,dZ) |
|  |  |
|  | return dA\_prev, dW, db |
|  |  |
|  |  |
|  | # Compute the backpropoagation with regularization for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  | def layerActivationBackwardWithReg(dA, cache, Y, lambd, activationFunc): |
|  | forward\_cache, activation\_cache = cache |
|  | A\_prev, W, b = forward\_cache |
|  | numtraining = float(A\_prev.shape[1]) |
|  | #print("n=",numtraining) |
|  | #print("no=",numtraining) |
|  | if activationFunc == "relu": |
|  | dZ = reluDerivative(dA, activation\_cache) |
|  | elif activationFunc == "sigmoid": |
|  | dZ = sigmoidDerivative(dA, activation\_cache) |
|  | elif activationFunc == "tanh": |
|  | dZ = tanhDerivative(dA, activation\_cache) |
|  | elif activationFunc == "softmax": |
|  | dZ = stableSoftmaxDerivative(dA, activation\_cache,Y,numtraining) |
|  |  |
|  | if activationFunc == 'softmax': |
|  | # Add the regularization factor |
|  | dW = 1/numtraining \* np.dot(A\_prev,dZ) + (lambd/numtraining) \* W.T |
|  | db = 1/numtraining \* np.sum(dZ, axis=0, keepdims=True) |
|  | dA\_prev = np.dot(dZ,W) |
|  | else: |
|  | # Add the regularization factor |
|  | dW = 1/numtraining \*(np.dot(dZ,A\_prev.T)) + (lambd/numtraining) \* W |
|  | #print("dW=",dW) |
|  | db = 1/numtraining \* np.sum(dZ, axis=1, keepdims=True) |
|  | #print("db=",db) |
|  | dA\_prev = np.dot(W.T,dZ) |
|  |  |
|  |  |
|  | return dA\_prev, dW, db |
|  |  |
|  | # 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 - relu/sigmoid/tanh |
|  | # # outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | # |
|  | # Returns: |
|  | # gradients -- A dictionary with the gradients |
|  | # gradients["dA" + str(l)] = ... |
|  | # gradients["dW" + str(l)] = ... |
|  |  |
|  | def backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd=0, keep\_prob=1, hiddenActivationFunc='relu',outputActivationFunc="sigmoid"): |
|  | #initialize the gradients |
|  | gradients = {} |
|  | # Set the number of layers |
|  | L = len(caches) |
|  | m = float(AL.shape[1]) |
|  |  |
|  | if outputActivationFunc == "sigmoid": |
|  | Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL |
|  | # Initializing the backpropagation |
|  | # dl/dAL= -(y/a + (1-y)/(1-a)) - At the output layer |
|  | dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) |
|  | else: |
|  | dAL =0 |
|  | Y=Y.T |
|  |  |
|  | # 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"] |
|  | current\_cache = caches[L-1] |
|  | if lambd==0: |
|  | gradients["dA" + str(L)], gradients["dW" + str(L)], gradients["db" + str(L)] = layerActivationBackward(dAL, current\_cache, |
|  | Y, activationFunc = outputActivationFunc) |
|  | else: #Regularization |
|  | gradients["dA" + str(L)], gradients["dW" + str(L)], gradients["db" + str(L)] = layerActivationBackwardWithReg(dAL, current\_cache, |
|  | Y, lambd, activationFunc = outputActivationFunc) |
|  |  |
|  | # Note dA for softmax is the transpose |
|  | if outputActivationFunc == "softmax": |
|  | gradients["dA" + str(L)] = gradients["dA" + str(L)].T |
|  | # Traverse in the reverse direction |
|  | for l in reversed(range(L-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] |
|  |  |
|  | #dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackward(gradients['dA'+str(l+2)], current\_cache, activationFunc = "relu") |
|  | if lambd==0: |
|  |  |
|  | # In the reverse direction use the dame dropout matrix |
|  | # Random dropout |
|  | # Multiply dA'l' with the dropoutMat and divide to keep the expected value same |
|  | D = dropoutMat["D" + str(l+1)] |
|  | # Drop some dAl's |
|  | gradients['dA'+str(l+2)]= np.multiply(gradients['dA'+str(l+2)],D) |
|  | # Divide by keep\_prob to keep expected value same |
|  | gradients['dA'+str(l+2)] = np.divide(gradients['dA'+str(l+2)],keep\_prob) |
|  |  |
|  | dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackward(gradients['dA'+str(l+2)], current\_cache, Y, keep\_prob=1, activationFunc = hiddenActivationFunc) |
|  |  |
|  | else: |
|  | dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackwardWithReg(gradients['dA'+str(l+2)], current\_cache, Y, lambd, activationFunc = hiddenActivationFunc) |
|  |  |
|  | gradients["dA" + str(l + 1)] = dA\_prev\_temp |
|  | gradients["dW" + str(l + 1)] = dW\_temp |
|  | gradients["db" + str(l + 1)] = db\_temp |
|  |  |
|  |  |
|  | return gradients |
|  |  |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | def gradientDescent(parameters, gradients, learningRate,outputActivationFunc="sigmoid"): |
|  |  |
|  | L = int(len(parameters) / 2) |
|  | # Update rule for each parameter. |
|  | for l in range(L-1): |
|  | parameters["W" + str(l+1)] = parameters['W'+str(l+1)] -learningRate\* gradients['dW' + str(l+1)] |
|  | parameters["b" + str(l+1)] = parameters['b'+str(l+1)] -learningRate\* gradients['db' + str(l+1)] |
|  |  |
|  | if outputActivationFunc=="sigmoid": |
|  | parameters["W" + str(L)] = parameters['W'+str(L)] -learningRate\* gradients['dW' + str(L)] |
|  | parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate\* gradients['db' + str(L)] |
|  | elif outputActivationFunc=="softmax": |
|  | parameters["W" + str(L)] = parameters['W'+str(L)] -learningRate\* gradients['dW' + str(L)].T |
|  | parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate\* gradients['db' + str(L)].T |
|  |  |
|  |  |
|  |  |
|  | 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/sigmoid |
|  | # : learning rate |
|  | # : num of iteration |
|  | # : outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  |  |
|  | def L\_Layer\_DeepModel(X1, Y1, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid", |
|  | learningRate = .3, lambd=0, keep\_prob=1, num\_iterations = 10000,initType="default", print\_cost=False,figure="figa.png"): |
|  |  |
|  | np.random.seed(1) |
|  | costs = [] |
|  |  |
|  | # Parameters initialization. |
|  | if initType == "He": |
|  | parameters = HeInitializeDeepModel(layersDimensions) |
|  | elif initType == "Xavier" : |
|  | parameters = XavInitializeDeepModel(layersDimensions) |
|  | else: #Default |
|  | parameters = initializeDeepModel(layersDimensions) |
|  | # Loop (gradient descent) |
|  | for i in range(0, num\_iterations): |
|  |  |
|  | AL, caches, dropoutMat = forwardPropagationDeep(X1, parameters, keep\_prob, hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Regularization parameter is 0 |
|  | if lambd==0: |
|  | # Compute cost |
|  | cost = computeCost(parameters,AL, Y1, outputActivationFunc=outputActivationFunc) |
|  | # Include L2 regularization |
|  | else: |
|  | # Compute cost |
|  | cost = computeCostWithReg(parameters,AL, Y1, lambd, outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, Y1, caches, dropoutMat, lambd, keep\_prob, hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate=learningRate,outputActivationFunc=outputActivationFunc) |
|  |  |
|  |  |
|  | # Print the cost every 100 training example |
|  | if print\_cost and i % 1000 == 0: |
|  | print ("Cost after iteration %i: %f" %(i, cost)) |
|  | if print\_cost and i % 1000 == 0: |
|  | costs.append(cost) |
|  |  |
|  | # plot the cost |
|  | plt.plot(np.squeeze(costs)) |
|  | plt.ylabel('Cost') |
|  | plt.xlabel('No of iterations (x1000)') |
|  | plt.title("Learning rate =" + str(learningRate)) |
|  | plt.savefig(figure,bbox\_inches='tight') |
|  | #plt.show() |
|  | plt.clf() |
|  | plt.close() |
|  |  |
|  |  |
|  |  |
|  | return parameters |
|  |  |
|  | # Execute a L layer Deep learning model Stoachastic Gradient Descent |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh/sigmoid |
|  | # : learning rate |
|  | # : num of iteration |
|  | # : outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  |  |
|  | def L\_Layer\_DeepModel\_SGD(X1, Y1, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid",learningRate = .3, mini\_batch\_size = 64, num\_epochs = 2500, print\_cost=False):#lr was 0.009 |
|  |  |
|  | np.random.seed(1) |
|  | costs = [] |
|  |  |
|  | # Parameters initialization. |
|  | parameters = initializeDeepModel(layersDimensions) |
|  | seed=10 |
|  | # Loop for number of epochs |
|  | for i in range(num\_epochs): |
|  | # Define the random minibatches. We increment the seed to reshuffle differently the dataset after each epoch |
|  | seed = seed + 1 |
|  | minibatches = random\_mini\_batches(X1, Y1, mini\_batch\_size, seed) |
|  |  |
|  | batch=0 |
|  | # Loop through each mini batch |
|  | for minibatch in minibatches: |
|  | #print("batch=",batch) |
|  | batch=batch+1 |
|  | # Select a minibatch |
|  | (minibatch\_X, minibatch\_Y) = minibatch |
|  |  |
|  | # Perfrom forward propagation |
|  | AL, caches = forwardPropagationDeep(minibatch\_X, parameters,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Compute cost |
|  | cost = computeCost(AL, minibatch\_Y,outputActivationFunc=outputActivationFunc) |
|  | #print("minibatch\_Y=",minibatch\_Y.shape) |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, minibatch\_Y, caches,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate=learningRate,outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Print the cost every 1000 epoch |
|  | if print\_cost and i % 100 == 0: |
|  | print ("Cost after epoch %i: %f" %(i, cost)) |
|  | if print\_cost and i % 100 == 0: |
|  | costs.append(cost) |
|  |  |
|  | # plot the cost |
|  | plt.plot(np.squeeze(costs)) |
|  | plt.ylabel('cost') |
|  | plt.xlabel('No of iterations') |
|  | plt.title("Learning rate =" + str(learningRate)) |
|  | #plt.show() |
|  | plt.savefig("fig1",bbox\_inches='tight') |
|  | plt.close() |
|  | return parameters |
|  |  |
|  |  |
|  | # Create random mini batches |
|  | def random\_mini\_batches(X, Y, miniBatchSize = 64, seed = 0): |
|  |  |
|  | np.random.seed(seed) |
|  | # Get number of training samples |
|  | m = X.shape[1] |
|  | # Initialize mini batches |
|  | mini\_batches = [] |
|  |  |
|  | # Create a list of random numbers < m |
|  | permutation = list(np.random.permutation(m)) |
|  | # Randomly shuffle the training data |
|  | shuffled\_X = X[:, permutation] |
|  | shuffled\_Y = Y[:, permutation].reshape((1,m)) |
|  |  |
|  | # Compute number of mini batches |
|  | numCompleteMinibatches = math.floor(m/miniBatchSize) |
|  |  |
|  | # For the number of mini batches |
|  | for k in range(0, numCompleteMinibatches): |
|  |  |
|  | # Set the start and end of each mini batch |
|  | mini\_batch\_X = shuffled\_X[:, k\*miniBatchSize : (k+1) \* miniBatchSize] |
|  | mini\_batch\_Y = shuffled\_Y[:, k\*miniBatchSize : (k+1) \* miniBatchSize] |
|  |  |
|  | mini\_batch = (mini\_batch\_X, mini\_batch\_Y) |
|  | mini\_batches.append(mini\_batch) |
|  |  |
|  |  |
|  | #if m % miniBatchSize != 0:. The batch does not evenly divide by the mini batch |
|  | if m % miniBatchSize != 0: |
|  | l=math.floor(m/miniBatchSize)\*miniBatchSize |
|  | # Set the start and end of last mini batch |
|  | m=l+m % miniBatchSize |
|  | mini\_batch\_X = shuffled\_X[:,l:m] |
|  | mini\_batch\_Y = shuffled\_Y[:,l:m] |
|  |  |
|  | mini\_batch = (mini\_batch\_X, mini\_batch\_Y) |
|  | mini\_batches.append(mini\_batch) |
|  |  |
|  | return mini\_batches |
|  |  |
|  | # Plot a decision boundary |
|  | # Input : Input Model, |
|  | # X |
|  | # Y |
|  | # sz - Num of hiden units |
|  | # lr - Learning rate |
|  | # Fig to be saved as |
|  | # Returns Null |
|  | def plot\_decision\_boundary(model, X, y,lr,figure1="figb.png"): |
|  | print("plot") |
|  | # Set min and max values and give it some padding |
|  | x\_min, x\_max = X[0, :].min() - 1, X[0, :].max() + 1 |
|  | y\_min, y\_max = X[1, :].min() - 1, X[1, :].max() + 1 |
|  | colors=['black','gold'] |
|  | cmap = matplotlib.colors.ListedColormap(colors) |
|  | h = 0.01 |
|  | # Generate a grid of points with distance h between them |
|  | xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h)) |
|  | # Predict the function value for the whole grid |
|  | Z = model(np.c\_[xx.ravel(), yy.ravel()]) |
|  | Z = Z.reshape(xx.shape) |
|  | # Plot the contour and training examples |
|  | plt.contourf(xx, yy, Z, cmap="coolwarm") |
|  | plt.ylabel('x2') |
|  | plt.xlabel('x1') |
|  | x=X.T |
|  | y=y.T.reshape(300,) |
|  | plt.scatter(x[:, 0], x[:, 1], c=y, s=20); |
|  | print(X.shape) |
|  | plt.title("Decision Boundary for learning rate:"+lr) |
|  | plt.savefig(figure1, bbox\_inches='tight') |
|  | #plt.show() |
|  |  |
|  |  |
|  | def predict(parameters, X,keep\_prob=1,hiddenActivationFunc="relu",outputActivationFunc="sigmoid"): |
|  | A2, cache,dropoutMat = forwardPropagationDeep(X, parameters, keep\_prob=1, hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  | predictions = (A2>0.5) |
|  | return predictions |
|  |  |
|  | def predict\_proba(parameters, X,outputActivationFunc="sigmoid"): |
|  | A2, cache = forwardPropagationDeep(X, parameters) |
|  | if outputActivationFunc=="sigmoid": |
|  | proba=A2 |
|  | elif outputActivationFunc=="softmax": |
|  | proba=np.argmax(A2, axis=0).reshape(-1,1) |
|  | print("A2=",A2.shape) |
|  | return proba |
|  |  |
|  | # Plot a decision boundary |
|  | # Input : Input Model, |
|  | # X |
|  | # Y |
|  | # sz - Num of hiden units |
|  | # lr - Learning rate |
|  | # Fig to be saved as |
|  | # Returns Null |
|  | def plot\_decision\_boundary1(X, y,W1,b1,W2,b2,figure2="figc.png"): |
|  | #plot\_decision\_boundary(lambda x: predict(parameters, x.T), x1,y1.T,str(0.3),"fig2.png") |
|  | h = 0.02 |
|  | x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1 |
|  | y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1 |
|  | xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), |
|  | np.arange(y\_min, y\_max, h)) |
|  | Z = np.dot(np.maximum(0, np.dot(np.c\_[xx.ravel(), yy.ravel()], W1.T) + b1.T), W2.T) + b2.T |
|  | Z = np.argmax(Z, axis=1) |
|  | Z = Z.reshape(xx.shape) |
|  |  |
|  | fig = plt.figure() |
|  | plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8) |
|  | print(X.shape) |
|  | y1=y.reshape(300,) |
|  | plt.scatter(X[:, 0], X[:, 1], c=y1, s=40, cmap=plt.cm.Spectral) |
|  | plt.xlim(xx.min(), xx.max()) |
|  | plt.ylim(yy.min(), yy.max()) |
|  | plt.savefig(figure2, bbox\_inches='tight') |
|  |  |
|  |  |
|  | def load\_dataset(): |
|  | np.random.seed(1) |
|  | train\_X, train\_Y = sklearn.datasets.make\_circles(n\_samples=300, noise=.05) |
|  | np.random.seed(2) |
|  | test\_X, test\_Y = sklearn.datasets.make\_circles(n\_samples=100, noise=.05) |
|  | # Visualize the data |
|  | print(train\_X.shape) |
|  | print(train\_Y.shape) |
|  | print("load") |
|  | #plt.scatter(train\_X[:, 0], train\_X[:, 1], c=train\_Y, s=40, cmap=plt.cm.Spectral); |
|  | train\_X = train\_X.T |
|  | train\_Y = train\_Y.reshape((1, train\_Y.shape[0])) |
|  | test\_X = test\_X.T |
|  | test\_Y = test\_Y.reshape((1, test\_Y.shape[0])) |
|  | return train\_X, train\_Y, test\_X, test\_Y |

DLFunctions61.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) |
|  |  |
|  | } |
|  |  |
|  | # Compute the Relu(old) of a vector |
|  | 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 multiple |
|  | 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) |
|  | } |
|  |  |
|  |  |
|  | # 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="")]] |
|  | # Compute the sigmoid activation |
|  | 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 |
|  | } |
|  | #cost=-1/m\*sum(a+b) |
|  | 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) |
|  | } |
|  |  |
|  |  |
|  | # 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{ |
|  | print("Here") |
|  | 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, |
|  | mini\_batch\_size = 64, |
|  | num\_epochs = 2500, |
|  | print\_cost=False){ |
|  |  |
|  | set.seed(1) |
|  | #Initialize costs vector as NULL |
|  | costs <- NULL |
|  |  |
|  | # Parameters initialization. |
|  | parameters = initializeDeepModel(layersDimensions) |
|  | 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,hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc) |
|  | AL <- retvals[['AL']] |
|  | caches <- retvals[['caches']] |
|  |  |
|  | # Compute cost. |
|  | cost <- computeCost(AL, mini\_batch\_Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  |  |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, mini\_batch\_Y, caches,hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  | } |
|  |  |
|  | if(i%%100 == 0){ |
|  | costs=c(costs,cost) |
|  | print(cost) |
|  | } |
|  | } |
|  |  |
|  | 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 |
|  |  |
|  |  |
|  | # 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 |
|  |  |
|  |  |
|  | # 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",learning\_rate = .3, |
|  | mini\_batch\_size = 64, num\_epochs = 2500)#lr was 0.009 |
|  |  |
|  | rand ("seed", 1); |
|  | costs = [] ; |
|  |  |
|  | # Parameters initialization. |
|  | [weights biases] = initializeDeepModel(layersDimensions); |
|  | 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] = forwardPropagationDeep(X, weights, biases,hiddenActivationFunc, outputActivationFunc=outputActivationFunc); |
|  | #disp(batch); |
|  | #disp(size(X)); |
|  | #disp(size(Y)); |
|  |  |
|  | # Compute cost. |
|  | cost = computeCost(AL, Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2))); |
|  |  |
|  | #disp(cost); |
|  | # Backward propagation. |
|  | [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y, activation\_caches,forward\_caches,hiddenActivationFunc, outputActivationFunc=outputActivationFunc, |
|  | numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | # Update parameters. |
|  | [weights biases] = gradientDescent(weights,biases, gradsDW,gradsDB,learning\_rate,outputActivationFunc=outputActivationFunc); |
|  |  |
|  | 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 |
|  | 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; |
|  |  |
|  | # 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**  
This post further enhances my earlier L-Layer generic implementation of a Deep Learning network to include options for initialization techniques, L2 regularization or dropout regularization