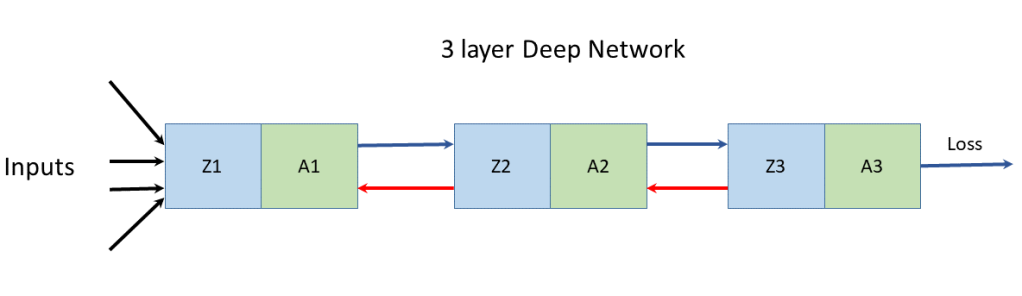
Lets take a simple 3 layer Neural network with 3 hidden layers and an output layer  
  
In the forward propagation cycle the equations are

Z_{1} = W_{1}A_{0} +b_{1} and  A_{1} = g(Z_{1})  
Z_{2} = W_{2}A_{1} +b_{2} and  A_{2} = g(Z_{2})  
Z_{3} = W_{3}A_{2} +b_{3} and A_{3} = g(Z_{3})

The loss function is given by  
L = -(ylogA3 + (1-y)log(1-A3))  
and dL/dA3 = -(Y/A_{3} + (1-Y)/(1-A_{3}))

For a binary classification the output activation function is the sigmoid function given by  
A_{3} = 1/(1+ e^{-Z3}). It can be shown that  
dA_{3}/dZ_{3} = A_{3}(1-A_3)

\partial L/\partial Z_{3} = \partial L/\partial A_{3}* \partial A_{3}/\partial Z_{3} = A3-Ysee equation (f)   
and since  
\partial L/\partial A_{2} = \partial L/\partial Z_{3} * \partial Z_{3}/\partial A_{2} = (A_{3} -Y) * W_{3}because \partial Z_{3}/\partial A_{2} = W_{3}-(1a)  
and \partial L/\partial Z_{2} =\partial L/\partial A_{2} * \partial A_{2}/\partial Z_{2} = (A_{3} -Y) * W_{3} *g'(Z_{2})-(1b)  
\partial L/\partial W_{2} = \partial L/\partial Z_{2} * A_{1}-(1c)  
since \partial Z_{2}/\partial W_{2} = A_{1}  
and  
\partial L/\partial b_{2} = \partial L/\partial Z_{2}-(1d)  
because  
\partial Z_{2}/\partial b_{2} =1

Also

\partial L/\partial A_{1} =\partial L/\partial Z_{2} * \partial Z_{2}/\partial A_{1} = \partial L/\partial Z_{2} * W_{2}     – (2a)  
\partial L/\partial Z_{1} =\partial L/\partial A_{1} * \partial A_{1}/\partial Z_{1} = \partial L/\partial A_{1} * W_{2} *g'(Z_{1})          – (2b)  
\partial L/\partial W_{1} = \partial L/\partial Z_{1} * A_{0}– (2c)  
\partial L/\partial b_{1} = \partial L/\partial Z_{1}– (2d)

Inspecting the above equations (1a – 1d & 2a-2d), our ‘Uber deep, bottomless’ brain  can easily discern the pattern in these equations. The equation for any layer ‘l’ is of the form  
Z_{l} = W_{l}A_{l-1} +b_{l}     and  A_{l} = g(Z_{l})  
The equation for the backward propagation have the general form  
\partial L/\partial A_{l} = \partial L/\partial Z_{l+1} * W^{l+1}  
\partial L/\partial Z_{l}=\partial L/\partial A_{l} *g'(Z_{l})  
\partial L/\partial W_{l} =\partial L/\partial Z_{l} *A^{l-1}  
\partial L/\partial b_{l} =\partial L/\partial Z_{l}

Some other important results The derivatives of the activation functions in the implemented Deep Learning network  
g(z) = sigmoid(z) = 1/(1+e^{-z})= a g’(z) = a(1-a) –  
g(z) = tanh(z) = a g’(z) = 1 - a^{2}  
g(z) = relu(z) = z  when z>0 and 0 when z 0 and 0 when z <= 0  
While it appears that there is a discontinuity for the derivative at 0 the small value at the discontinuity does not present a problem

The implementation of the multi layer vectorized Deep Learning Network for Python, R and Octave is included below. For all these implementations, initially I create the size and configuration of the the Deep Learning network with the layer dimennsions So for example layersDimension Vector ‘V’ of length L indicating ‘L’ layers where

V (in Python)= [v_{0}, v_{1}, v_{2}, … v_{L-1}]  
V (in R)= c(v_{1}, v_{2}, v_{3}, … v_{L})  
V (in Octave)= [ v_{1} v_{2} v_{3}… v_{L}]

In all of these implementations the first element is the number of input features to the Deep Learning network and the last element is always a ‘sigmoid’ activation function since all the problems deal with binary classification.

The number of elements between the first and the last element are the number of hidden layers and the magnitude of each v_{i}is the number of activation units in each hidden layer, which is specified while actually executing the Deep Learning network using the function L\_Layer\_DeepModel(), in all the implementations Python, R and Octave

**1a. Classification with Multi layer Deep Learning Network – Relu activation(Python)**

In the code below a 4 layer Neural Network is trained to generate a non-linear boundary between the classes. In the code below the ‘Relu’ Activation function is used. The number of activation units in each layer is 9. The cost vs iterations is plotted in addition to the decision boundary. Further the accuracy, precision, recall and F1 score are also computed

import os

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.colors

import sklearn.linear\_model

from sklearn.model\_selection import train\_test\_split

from sklearn.datasets import make\_classification, make\_blobs

from matplotlib.colors import ListedColormap

import sklearn

import sklearn.datasets

#from DLfunctions import plot\_decision\_boundary

execfile("./DLfunctions34.py") #

os.chdir("C:\\software\\DeepLearning-Posts\\part3")

# Create clusters of 2 classes

X1, Y1 = make\_blobs(n\_samples = 400, n\_features = 2, centers = 9,

cluster\_std = 1.3, random\_state = 4)

#Create 2 classes

Y1=Y1.reshape(400,1)

Y1 = Y1 % 2

X2=X1.T

Y2=Y1.T

# Set the dimensions of DL Network

# Below we have

# 2 - 2 input features

# 9,9 - 2 hidden layers with 9 activation units per layer and

# 1 - 1 sigmoid activation unit in the output layer as this is a binary classification

# The activation in the hidden layer is the 'relu' specified in L\_Layer\_DeepModel

layersDimensions = [2, 9, 9,1] # 4-layer model

parameters = L\_Layer\_DeepModel(X2, Y2, layersDimensions,hiddenActivationFunc='relu', learning\_rate = 0.3,num\_iterations = 2500, fig="fig1.png")

#Plot the decision boundary

plot\_decision\_boundary(lambda x: predict(parameters, x.T), X2,Y2,str(0.3),"fig2.png")

# Compute the confusion matrix

yhat = predict(parameters,X2)

from sklearn.metrics import confusion\_matrix

a=confusion\_matrix(Y2.T,yhat.T)

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

print('Accuracy: {:.2f}'.format(accuracy\_score(Y2.T, yhat.T)))

print('Precision: {:.2f}'.format(precision\_score(Y2.T, yhat.T)))

print('Recall: {:.2f}'.format(recall\_score(Y2.T, yhat.T)))

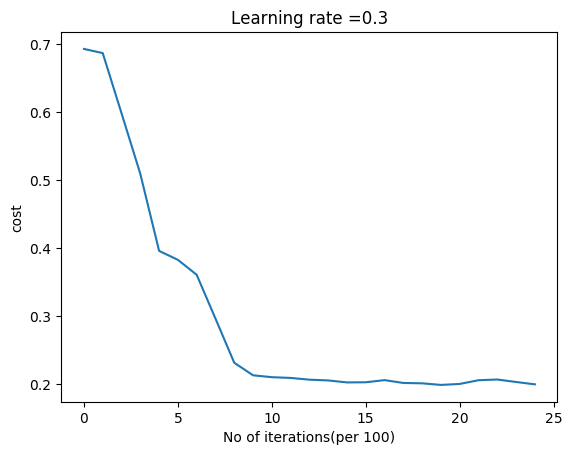
print('F1: {:.2f}'.format(f1\_score(Y2.T, yhat.T)))

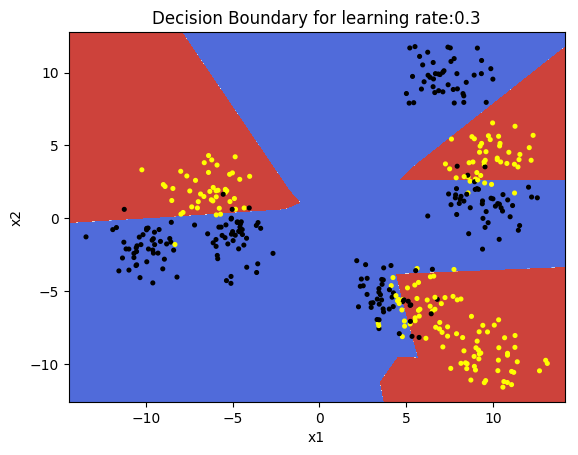
## Accuracy: 0.90

## Precision: 0.91

## Recall: 0.87

## F1: 0.89





**1b. Classification with Multi layer Deep Learning Network – Relu activation(R)**

In the code below, binary classification is performed on the same data set as above using the Relu activation function. The DL network is same as above

library(ggplot2)

# Read the data

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

x <- z[,1:2]

y <- z[,3]

X1 <- t(x)

Y1 <- t(y)

# Set the dimensions of the Deep Learning network

# No of input features =2, 2 hidden layers with 9 activation units and 1 output layer

layersDimensions = c(2, 9, 9,1)

# Execute the Deep Learning Neural Network

retvals = L\_Layer\_DeepModel(X1, Y1, layersDimensions,

hiddenActivationFunc='relu',

learningRate = 0.3,

numIterations = 5000,

print\_cost = True)

library(ggplot2)

source("DLfunctions33.R")

# Get the computed costs

costs <- retvals[['costs']]

# Create a sequence of iterations

numIterations=5000

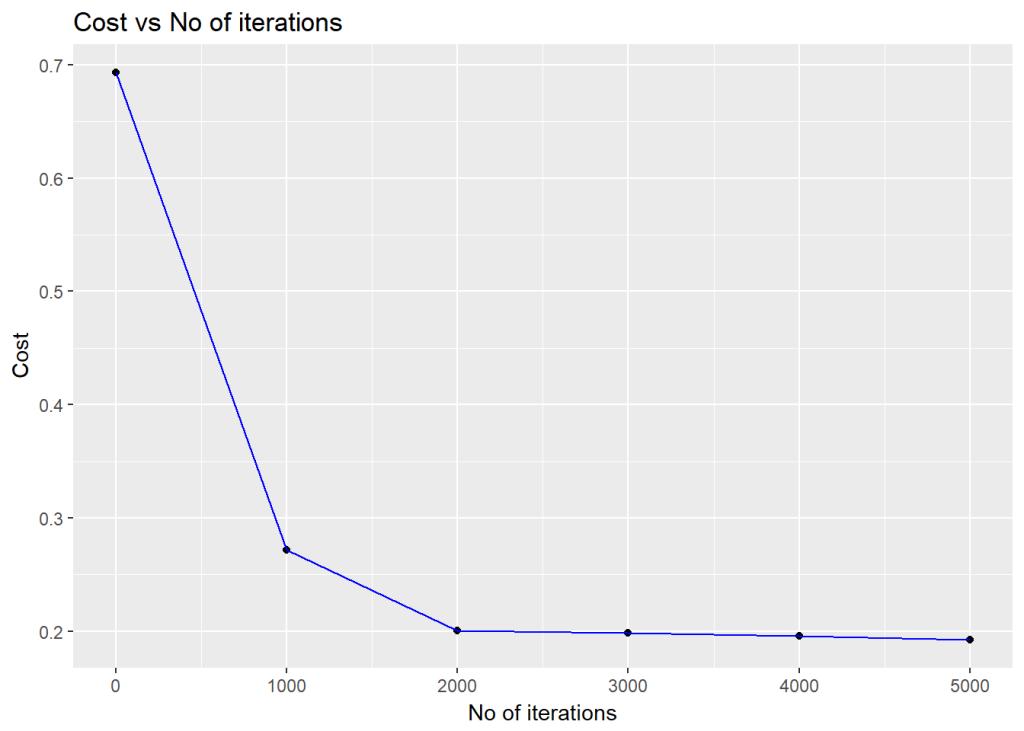
iterations <- seq(0,numIterations,by=1000)

df <-data.frame(iterations,costs)

# Plot the Costs vs number of iterations

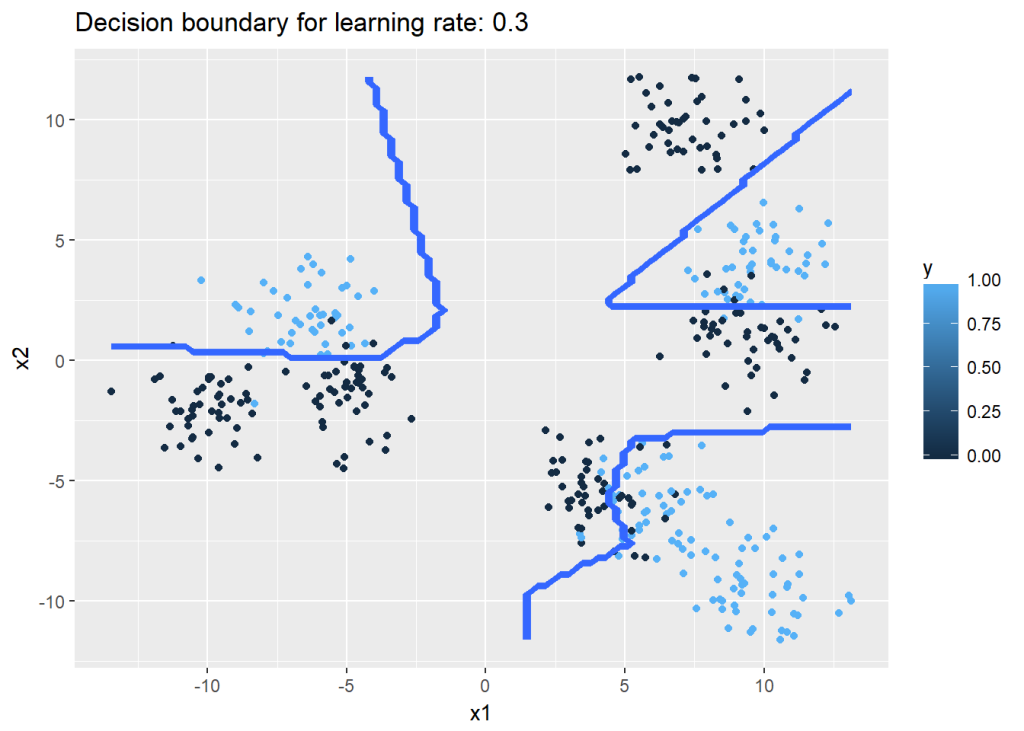
ggplot(df,aes(x=iterations,y=costs)) + geom\_point() +geom\_line(color="blue") +

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



# Plot the decision boundary

plotDecisionBoundary(z,retvals,hiddenActivationFunc="relu",0.3)



library(caret)

# Predict the output for the data values

yhat <-predict(retvals$parameters,X1,hiddenActivationFunc="relu")

yhat[yhat==FALSE]=0

yhat[yhat==TRUE]=1

# Compute the confusion matrix

confusionMatrix(yhat,Y1)

## Confusion Matrix and Statistics

##

## Reference

## Prediction 0 1

## 0 201 10

## 1 21 168

##

## Accuracy : 0.9225

## 95% CI : (0.8918, 0.9467)

## No Information Rate : 0.555

## P-Value [Acc > NIR] : < 2e-16

##

## Kappa : 0.8441

## Mcnemar's Test P-Value : 0.07249

##

## Sensitivity : 0.9054

## Specificity : 0.9438

## Pos Pred Value : 0.9526

## Neg Pred Value : 0.8889

## Prevalence : 0.5550

## Detection Rate : 0.5025

## Detection Prevalence : 0.5275

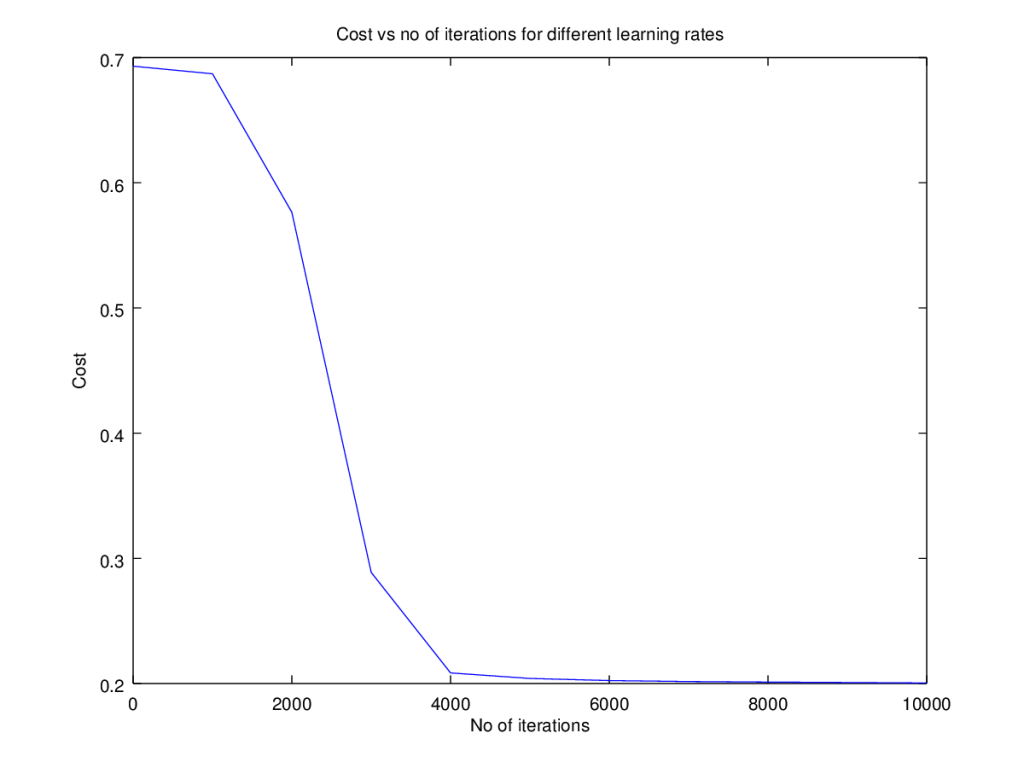
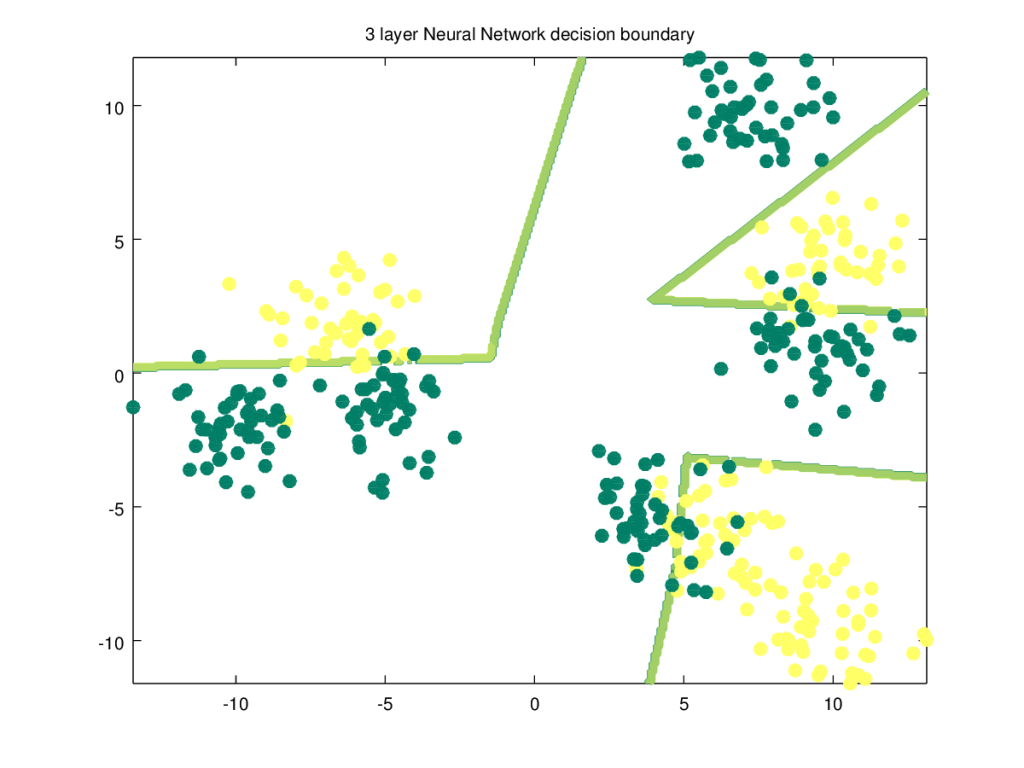
## Balanced Accuracy : 0.9246

##

## 'Positive' Class : 0

##

**1c. Classification with Multi layer Deep Learning Network – Relu activation(Octave)**

Included below is the code for performing classification. Incidentally Octave does not seem to have implemented the confusion matrix,  but confusionmat is available in Matlab.  
# Read the data  
data=csvread("data.csv");  
X=data(:,1:2);  
Y=data(:,3);  
# Set layer dimensions  
layersDimensions = [2 9 7 1] #tanh=-0.5(ok), #relu=0.1 best!  
# Execute Deep Network  
[weights biases costs]=L\_Layer\_DeepModel(X', Y', layersDimensions,  
hiddenActivationFunc='relu',  
learningRate = 0.1,  
numIterations = 10000);  
plotCostVsIterations(10000,costs);  
plotDecisionBoundary(data,weights, biases,hiddenActivationFunc="tanh")  
  
  


**2a. Classification with Multi layer Deep Learning Network – Tanh activation(Python)**

Below the Tanh activation function is used to perform the same classification. I found the Tanh activation required a simpler Neural Network of 3 layers.

# Tanh activation

import os

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.colors

import sklearn.linear\_model

from sklearn.model\_selection import train\_test\_split

from sklearn.datasets import make\_classification, make\_blobs

from matplotlib.colors import ListedColormap

import sklearn

import sklearn.datasets

#from DLfunctions import plot\_decision\_boundary

os.chdir("C:\\software\\DeepLearning-Posts\\part3")

execfile("./DLfunctions34.py")

# Create the dataset

X1, Y1 = make\_blobs(n\_samples = 400, n\_features = 2, centers = 9,

cluster\_std = 1.3, random\_state = 4)

#Create 2 classes

Y1=Y1.reshape(400,1)

Y1 = Y1 % 2

X2=X1.T

Y2=Y1.T

# Set the dimensions of the Neural Network

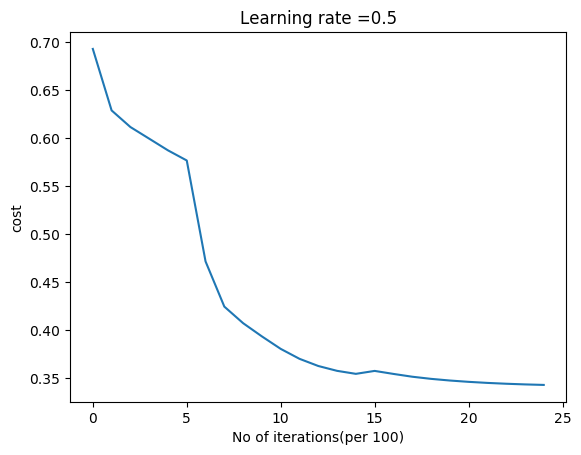
layersDimensions = [2, 4, 1] # 3-layer model

# Compute the DL network

parameters = L\_Layer\_DeepModel(X2, Y2, layersDimensions, hiddenActivationFunc='tanh', learning\_rate = .5,num\_iterations = 2500,fig="fig3.png")

#Plot the decision boundary

plot\_decision\_boundary(lambda x: predict(parameters, x.T), X2,Y2,str(0.5),"fig4.png")

****

****

**2b. Classification with Multi layer Deep Learning Network – Tanh activation(R)**

R performs better with a Tanh activation than the Relu as can be seen below

#Set the dimensions of the Neural Network

layersDimensions = c(2, 9, 9,1)

library(ggplot2)

# Read the data

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

x <- z[,1:2]

y <- z[,3]

X1 <- t(x)

Y1 <- t(y)

# Execute the Deep Model

retvals = L\_Layer\_DeepModel(X1, Y1, layersDimensions,

hiddenActivationFunc='tanh',

learningRate = 0.3,

numIterations = 5000,

print\_cost = True)

# Get the costs

costs <- retvals[['costs']]

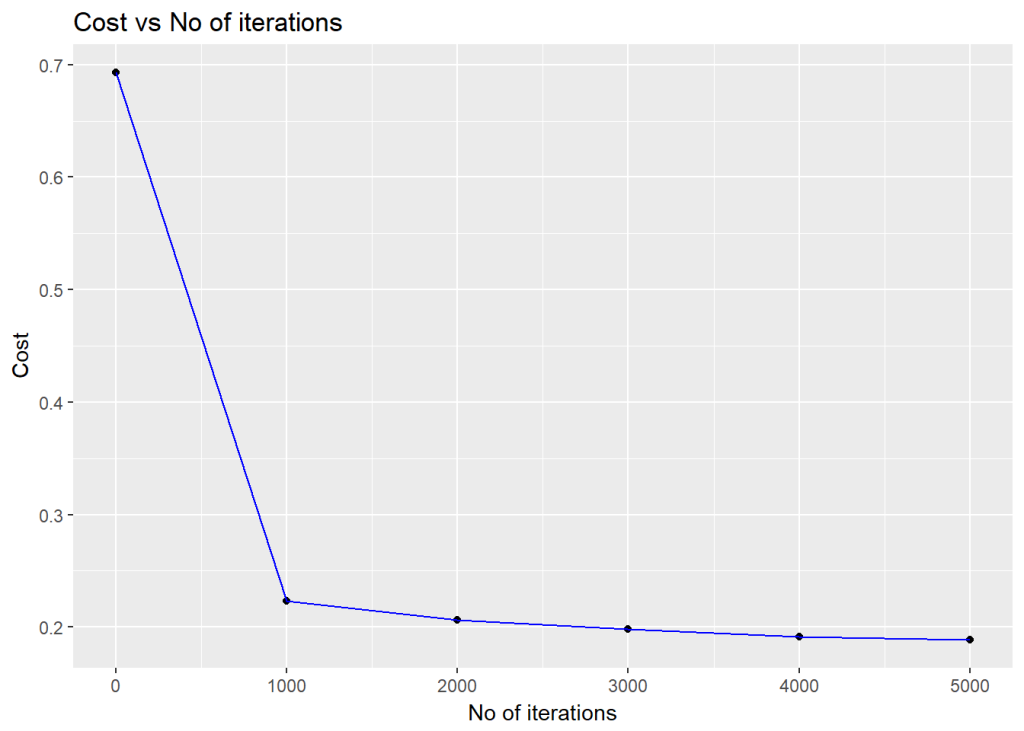
iterations <- seq(0,numIterations,by=1000)

df <-data.frame(iterations,costs)

# Plot Cost vs number of iterations

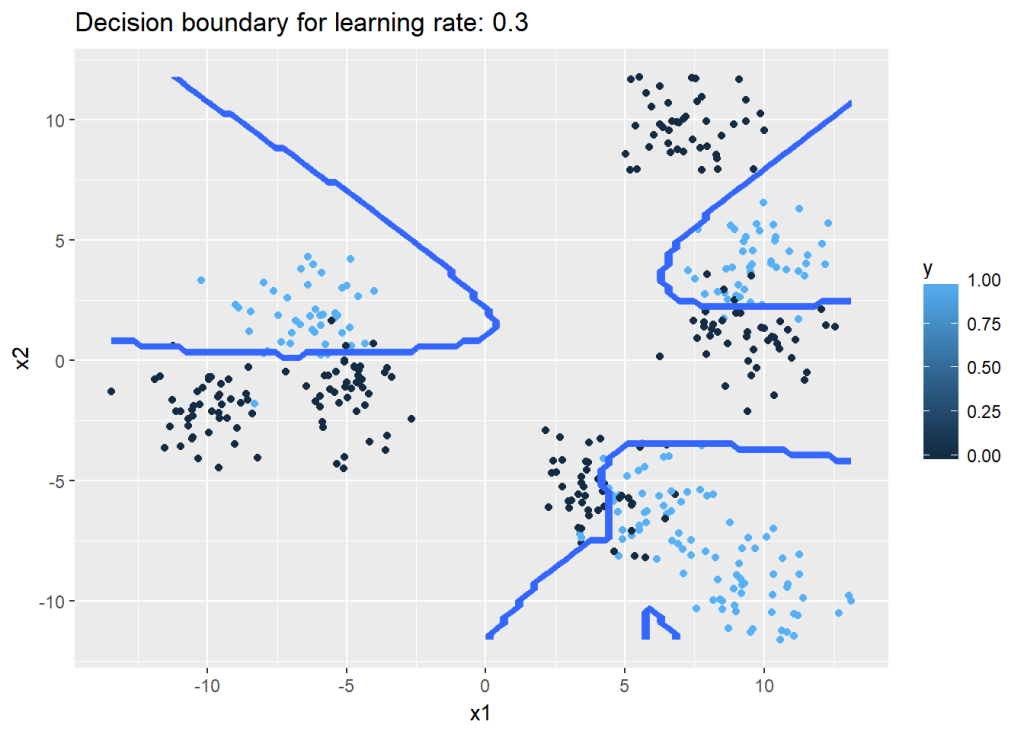
ggplot(df,aes(x=iterations,y=costs)) + geom\_point() +geom\_line(color="blue") +

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

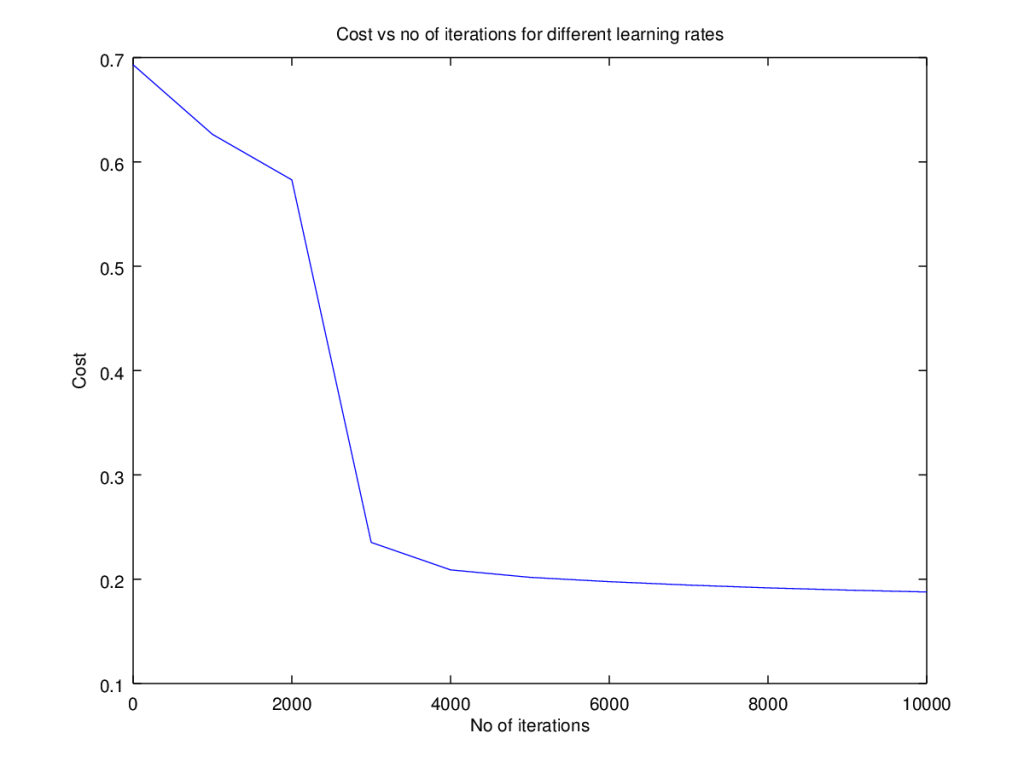
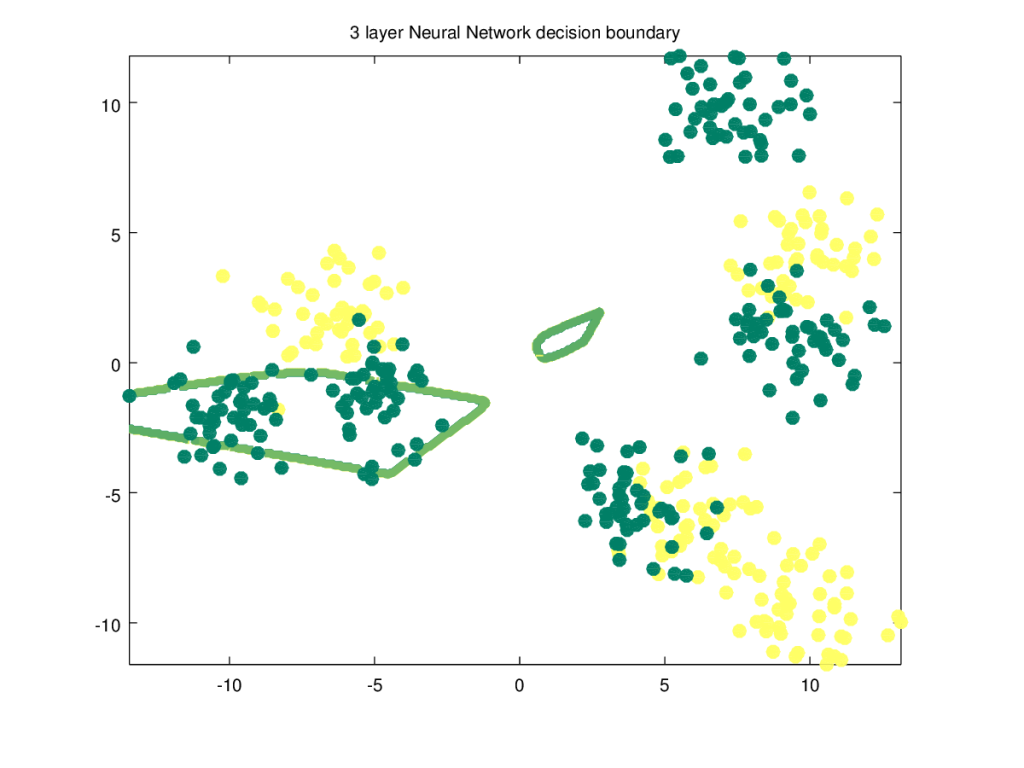


#Plot the decision boundary

plotDecisionBoundary(z,retvals,hiddenActivationFunc="tanh",0.3)

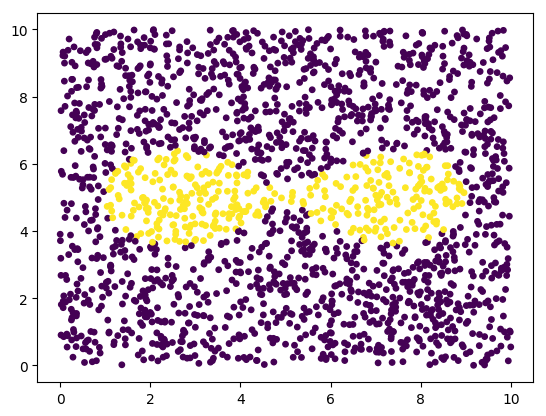


**2c. Classification with Multi layer Deep Learning Network – Tanh activation(Octave)**

The code below uses the   Tanh activation in the hidden layers for Octave  
# Read the data  
data=csvread("data.csv");  
X=data(:,1:2);  
Y=data(:,3);  
# Set layer dimensions  
layersDimensions = [2 9 7 1] #tanh=-0.5(ok), #relu=0.1 best!  
# Execute Deep Network  
[weights biases costs]=L\_Layer\_DeepModel(X', Y', layersDimensions,  
hiddenActivationFunc='tanh',  
learningRate = 0.1,  
numIterations = 10000);  
plotCostVsIterations(10000,costs);  
plotDecisionBoundary(data,weights, biases,hiddenActivationFunc="tanh")  
  
  


**3. Bernoulli’s Lemniscate**

To make things  more interesting, I create a 2D figure of the Bernoulli’s lemniscate to perform non-linear classification. The Lemniscate is given by the equation  
(x^{2} + y^{2})^{2}= 2a^{2}*(x^{2}-y^{2})



**3a. Classifying a lemniscate with Deep Learning Network – Relu activation(Python)**

import os

import numpy as np

import matplotlib.pyplot as plt

os.chdir("C:\\software\\DeepLearning-Posts\\part3")

execfile("./DLfunctions33.py")

x1=np.random.uniform(0,10,2000).reshape(2000,1)

x2=np.random.uniform(0,10,2000).reshape(2000,1)

X=np.append(x1,x2,axis=1)

X.shape

# Create a subset of values where squared is <0,4. Perform ravel() to flatten this vector

# Create the equation

# (x^{2} + y^{2})^2 - 2a^2\*(x^{2}-y^{2}) <= 0

a=np.power(np.power(X[:,0]-5,2) + np.power(X[:,1]-5,2),2)

b=np.power(X[:,0]-5,2) - np.power(X[:,1]-5,2)

c= a - (b\*np.power(4,2)) <=0

Y=c.reshape(2000,1)

# Create a scatter plot of the lemniscate

plt.scatter(X[:,0], X[:,1], c=Y, marker= 'o', s=15,cmap="viridis")

Z=np.append(X,Y,axis=1)

plt.savefig("fig50.png",bbox\_inches='tight')

plt.clf()

# Set the data for classification

X2=X.T

Y2=Y.T

# These settings work the best

# Set the Deep Learning layer dimensions for a Relu activation

layersDimensions = [2,7,4,1]

#Execute the DL network

parameters = L\_Layer\_DeepModel(X2, Y2, layersDimensions, hiddenActivationFunc='relu', learning\_rate = 0.5,num\_iterations = 10000, fig="fig5.png")

#Plot the decision boundary

plot\_decision\_boundary(lambda x: predict(parameters, x.T), X2, Y2,str(2.2),"fig6.png")

# Compute the Confusion matrix

yhat = predict(parameters,X2)

from sklearn.metrics import confusion\_matrix

a=confusion\_matrix(Y2.T,yhat.T)

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

print('Accuracy: {:.2f}'.format(accuracy\_score(Y2.T, yhat.T)))

print('Precision: {:.2f}'.format(precision\_score(Y2.T, yhat.T)))

print('Recall: {:.2f}'.format(recall\_score(Y2.T, yhat.T)))

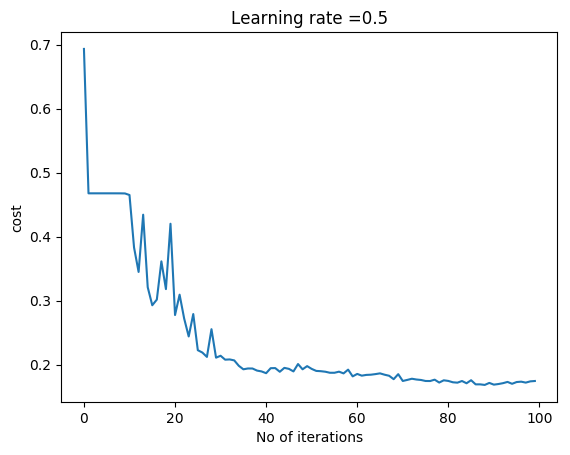
print('F1: {:.2f}'.format(f1\_score(Y2.T, yhat.T)))

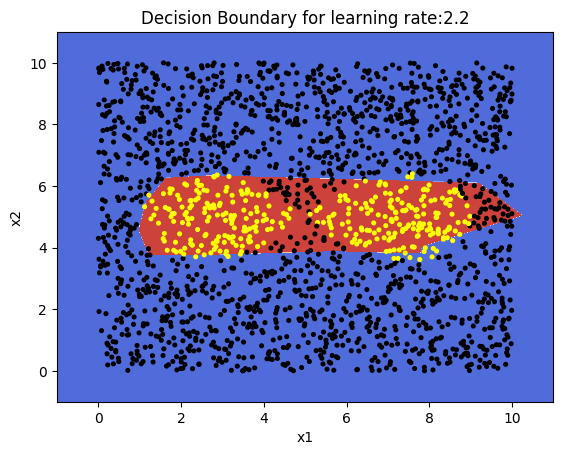
## Accuracy: 0.93

## Precision: 0.77

## Recall: 0.76

## F1: 0.76





We could get better performance by tuning further. Do play around if you fork the code.  
**Note:**: The lemniscate data is saved as a CSV and then read in R and also in Octave. I do this instead of recreating the lemniscate shape

**3b. Classifying a lemniscate with Deep Learning Network – Relu activation(R code)**

The R decision boundary for the Bernoulli’s lemniscate is shown below

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

Z1=data.frame(Z)

# Create a scatter plot of the lemniscate

ggplot(Z1,aes(x=V1,y=V2,col=V3)) +geom\_point()

#Set the data for the DL network

X=Z[,1:2]

Y=Z[,3]

X1=t(X)

Y1=t(Y)

# Set the layer dimensions for the tanh activation function

layersDimensions = c(2,5,4,1)

# Execute the Deep Learning network with Tanh activation

retvals = L\_Layer\_DeepModel(X1, Y1, layersDimensions,

hiddenActivationFunc='tanh',

learningRate = 0.3,

numIterations = 20000, print\_cost = True)

# Plot cost vs iteration

costs <- retvals[['costs']]

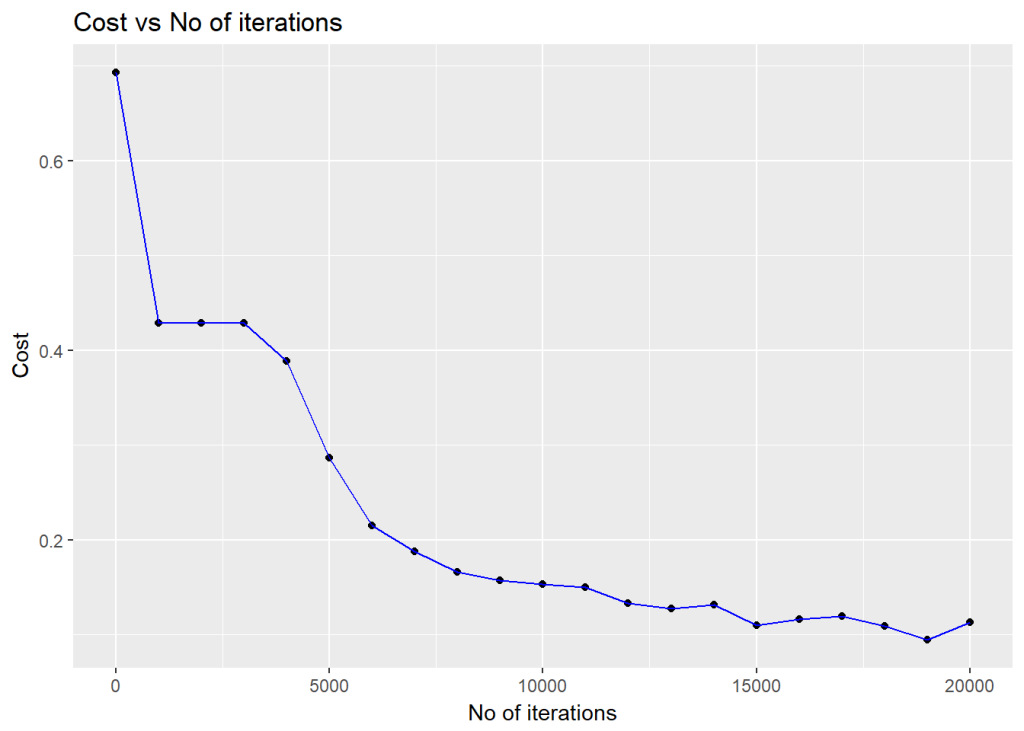
numIterations = 20000

iterations <- seq(0,numIterations,by=1000)

df <-data.frame(iterations,costs)

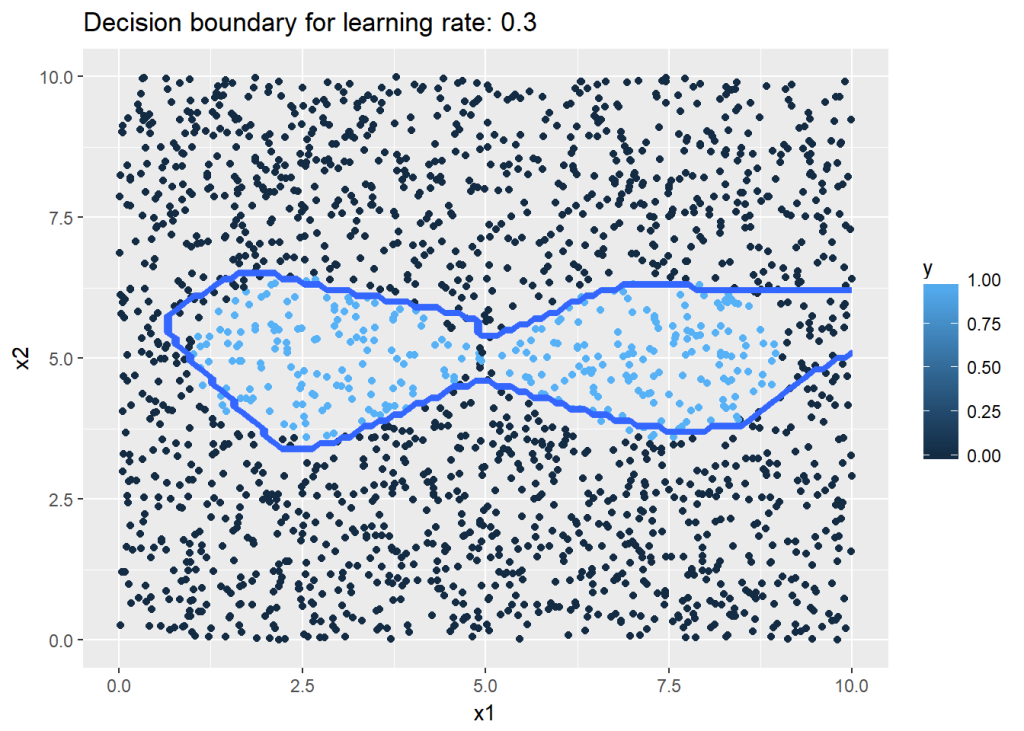
ggplot(df,aes(x=iterations,y=costs)) + geom\_point() +geom\_line(color="blue") +

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



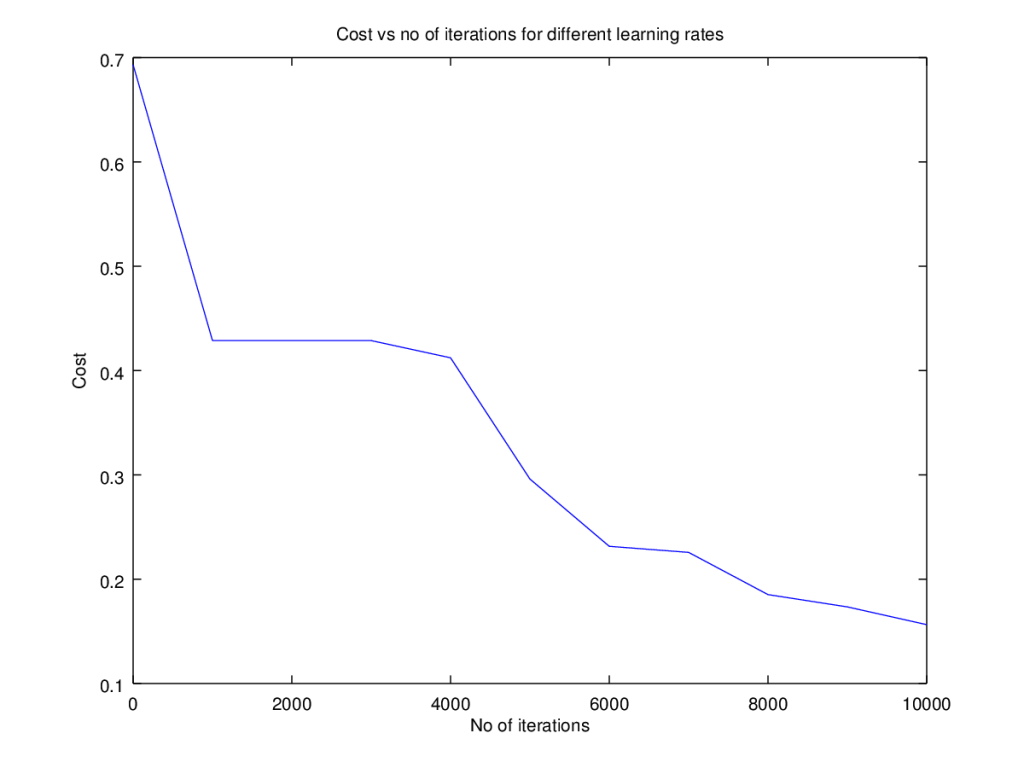
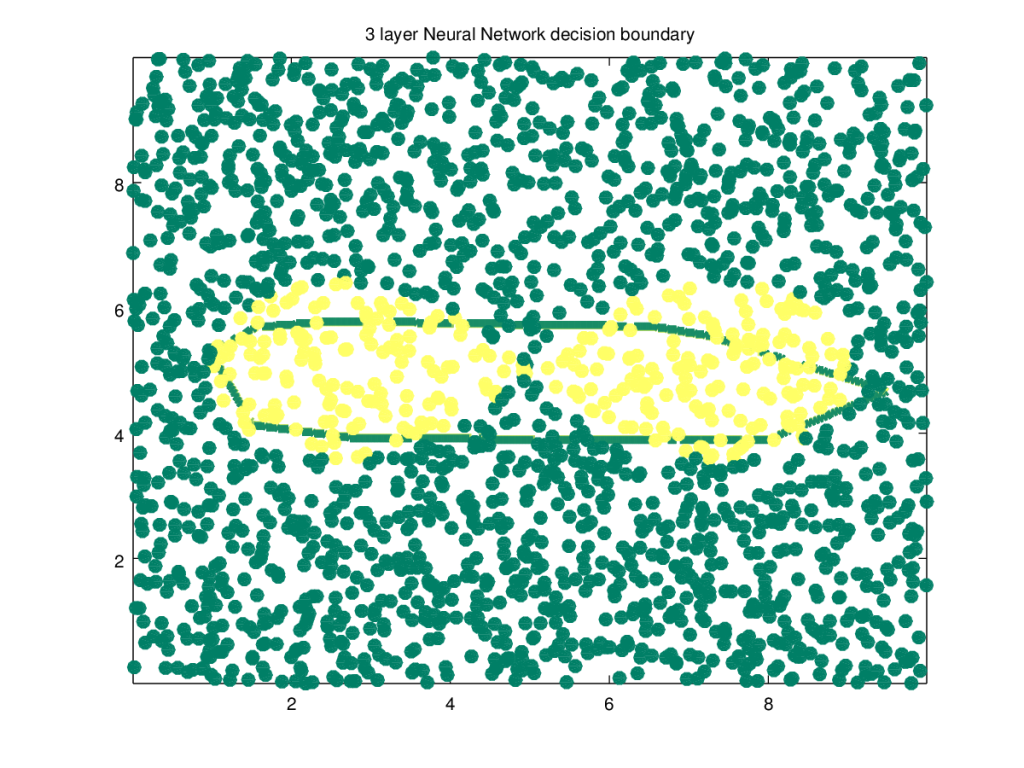
#Plot the decision boundary

plotDecisionBoundary(Z,retvals,hiddenActivationFunc="tanh",0.3)

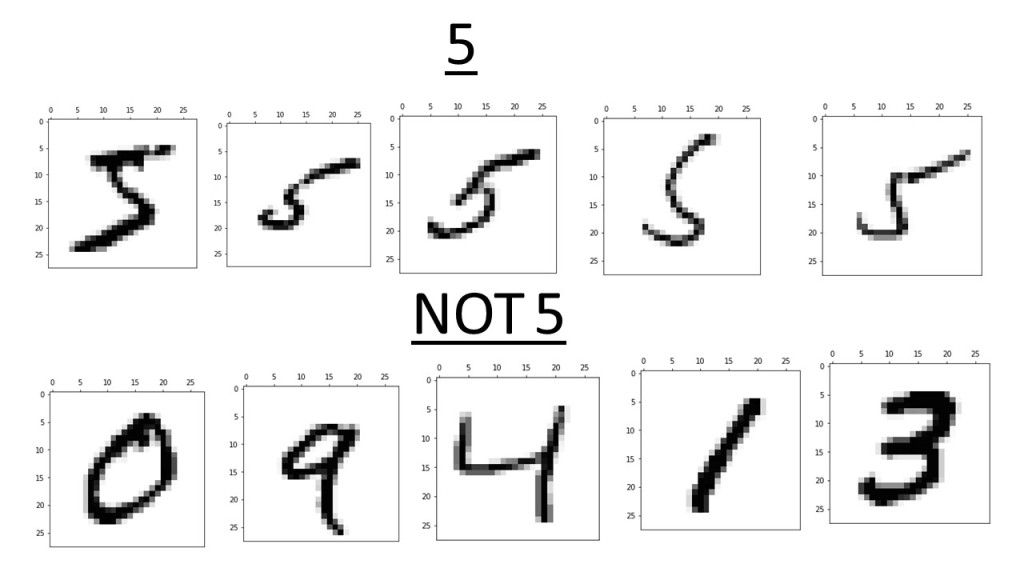


**3c. Classifying a lemniscate with Deep Learning Network – Relu activation(Octave code)**

Octave is used to generate the non-linear lemniscate boundary.  
  
# Read the data  
data=csvread("lemniscate.csv");  
X=data(:,1:2);  
Y=data(:,3);  
# Set the dimensions of the layers  
layersDimensions = [2 9 7 1]  
# Compute the DL network  
[weights biases costs]=L\_Layer\_DeepModel(X', Y', layersDimensions,  
hiddenActivationFunc='relu',  
learningRate = 0.20,  
numIterations = 10000);  
plotCostVsIterations(10000,costs);  
plotDecisionBoundary(data,weights, biases,hiddenActivationFunc="relu")

**4a. Binary Classification using MNIST – Python code**

Finally I perform a simple classification using the MNIST handwritten digits, which according to Prof Geoffrey Hinton is “the Drosophila of Deep Learning”.  


In the Python code below, I perform a simple binary classification between the handwritten digit ‘5’ and ‘not 5’ which is all other digits. I will perform the proper classification of all digits using the  Softmax classifier some time later.

import os

import numpy as np

import matplotlib.pyplot as plt

os.chdir("C:\\software\\DeepLearning-Posts\\part3")

execfile("./DLfunctions34.py")

execfile("./load\_mnist.py")

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

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

lbls=[]

pxls=[]

print(len(training))

# Select the first 10000 training data and the labels

for i in range(10000):

l,p=training[i]

lbls.append(l)

pxls.append(p)

labels= np.array(lbls)

pixels=np.array(pxls)

# Sey y=1 when labels == 5 and 0 otherwise

y=(labels==5).reshape(-1,1)

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

# Create the necessary feature and target variable

X1=X.T

Y1=y.T

# Create the layer dimensions. The number of features are 28 x 28 = 784 since the 28 x 28

# pixels is flattened to single vector of length 784.

layersDimensions=[784, 15,9,7,1] # Works very well

parameters = L\_Layer\_DeepModel(X1, Y1, layersDimensions, hiddenActivationFunc='relu', learning\_rate = 0.1,num\_iterations = 1000, fig="fig7.png")

# Test data

lbls1=[]

pxls1=[]

for i in range(800):

l,p=test[i]

lbls1.append(l)

pxls1.append(p)

testLabels=np.array(lbls1)

testData=np.array(pxls1)

ytest=(testLabels==5).reshape(-1,1)

Xtest=testData.reshape(testData.shape[0],-1)

Xtest1=Xtest.T

Ytest1=ytest.T

yhat = predict(parameters,Xtest1)

from sklearn.metrics import confusion\_matrix

a=confusion\_matrix(Ytest1.T,yhat.T)

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

print('Accuracy: {:.2f}'.format(accuracy\_score(Ytest1.T, yhat.T)))

print('Precision: {:.2f}'.format(precision\_score(Ytest1.T, yhat.T)))

print('Recall: {:.2f}'.format(recall\_score(Ytest1.T, yhat.T)))

print('F1: {:.2f}'.format(f1\_score(Ytest1.T, yhat.T)))

probs=predict\_proba(parameters,Xtest1)

from sklearn.metrics import precision\_recall\_curve

precision, recall, thresholds = precision\_recall\_curve(Ytest1.T, probs.T)

closest\_zero = np.argmin(np.abs(thresholds))

closest\_zero\_p = precision[closest\_zero]

closest\_zero\_r = recall[closest\_zero]

plt.xlim([0.0, 1.01])

plt.ylim([0.0, 1.01])

plt.plot(precision, recall, label='Precision-Recall Curve')

plt.plot(closest\_zero\_p, closest\_zero\_r, 'o', markersize = 12, fillstyle = 'none', c='r', mew=3)

plt.xlabel('Precision', fontsize=16)

plt.ylabel('Recall', fontsize=16)

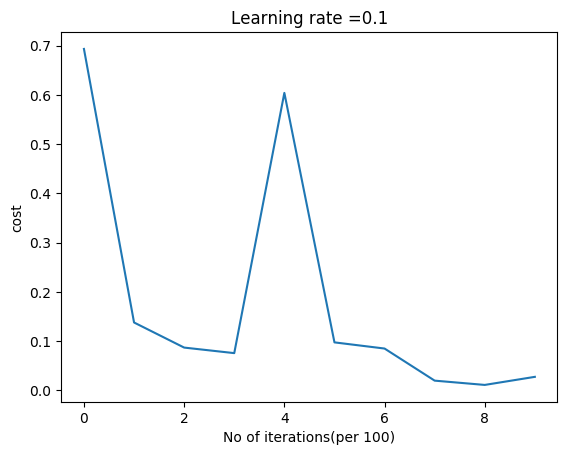
plt.savefig("fig8.png",bbox\_inches='tight')

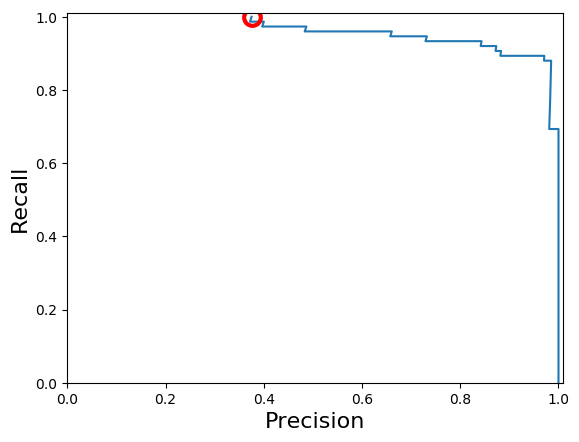
## Accuracy: 0.99

## Precision: 0.96

## Recall: 0.89

## F1: 0.92





In addition to plotting the Cost vs Iterations, I also plot the Precision-Recall curve to show how the Precision and Recall, which are complementary to each other vary with respect to the other. To know more about Precision-Recall.

**4b. Binary Classification using MNIST – R code**

In the R code below the same binary classification of the digit ‘5’ and the ‘not 5’ is performed.

source("mnist.R")

load\_mnist()

#show\_digit(train$x[2,]

layersDimensions=c(784, 7,7,3,1) # Works at 1500

x <- t(train$x)

# Choose only 5000 training data

x2 <- x[,1:5000]

y <-train$y

# Set labels for all digits that are 'not 5' to 0

y[y!=5] <- 0

# Set labels of digit 5 as 1

y[y==5] <- 1

# Set the data

y1 <- as.matrix(y)

y2 <- t(y1)

# Choose the 1st 5000 data

y3 <- y2[,1:5000]

#Execute the Deep Learning Model

retvals = L\_Layer\_DeepModel(x2, y3, layersDimensions,

hiddenActivationFunc='tanh',

learningRate = 0.3,

numIterations = 3000, print\_cost = True)

# Plot cost vs iteration

costs <- retvals[['costs']]

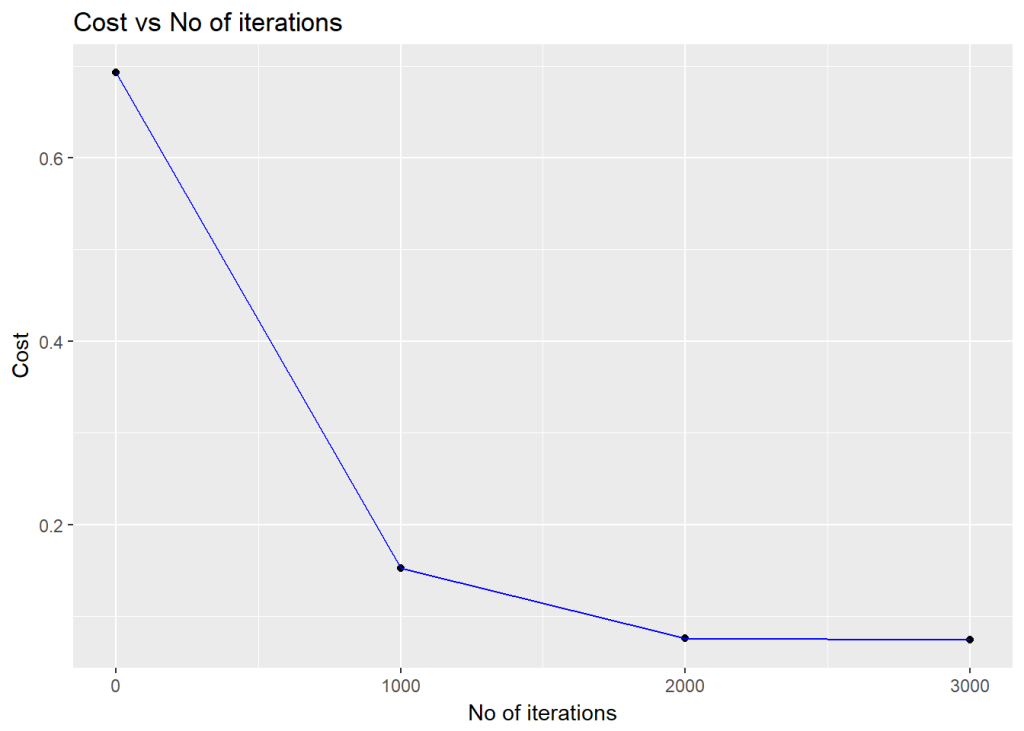
numIterations = 3000

iterations <- seq(0,numIterations,by=1000)

df <-data.frame(iterations,costs)

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

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



# Compute probability scores

scores <- computeScores(retvals$parameters, x2,hiddenActivationFunc='relu')

a=y3==1

b=y3==0

# Compute probabilities of class 0 and class 1

class1=scores[a]

class0=scores[b]

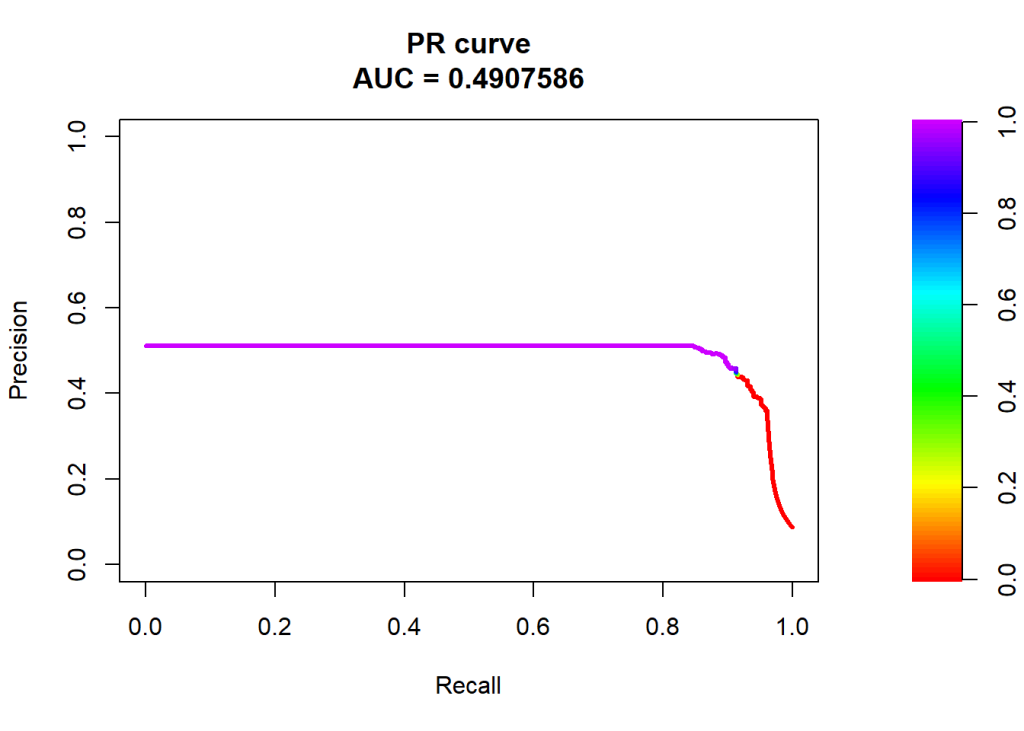
# Plot ROC curve

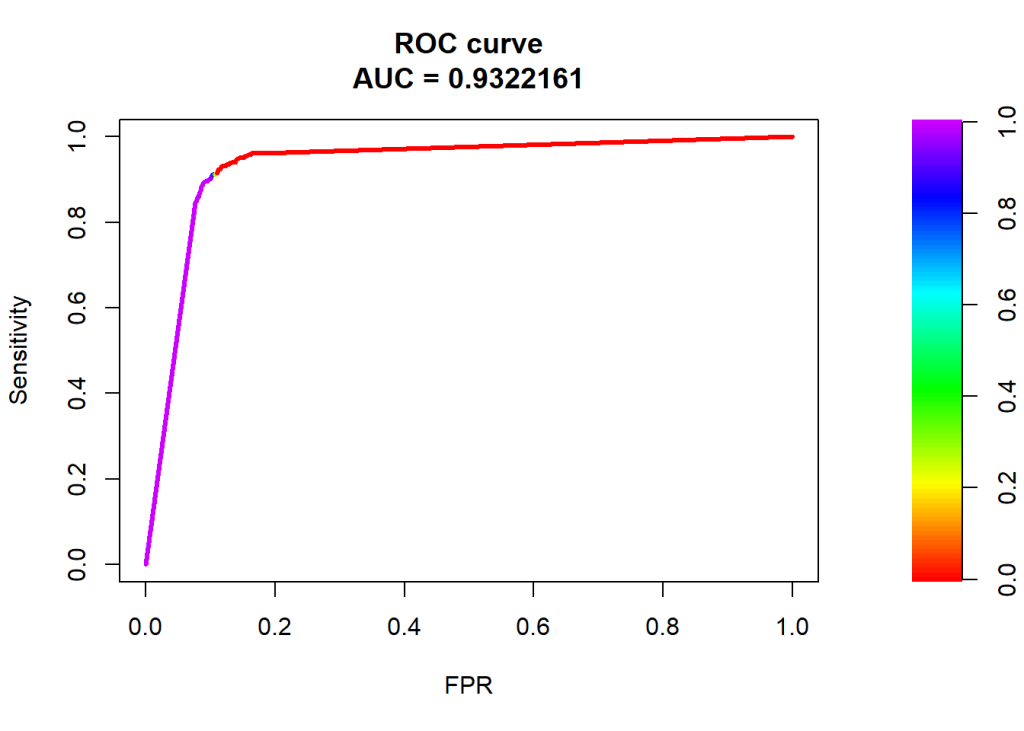
pr <-pr.curve(scores.class0=class1,

scores.class1=class0,

curve=T)

plot(pr)

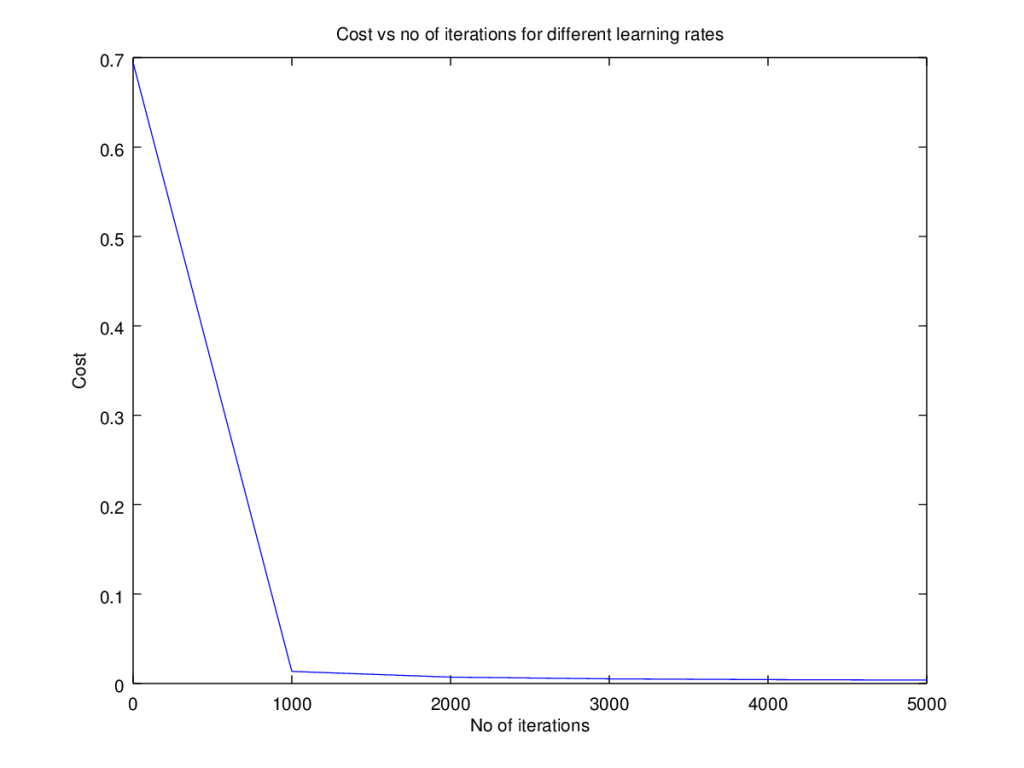




The AUC curve hugs the top left corner and hence the performance of the classifier is quite good.

**4c. Binary Classification using MNIST – Octave code**

Precision recall curves are available in Matlab but are yet to be implemented in Octave’s statistics package.  
  
load('./mnist/mnist.txt.gz'); % load the dataset  
# Subset the 'not 5' digits  
a=(trainY != 5);  
# Subset '5'  
b=(trainY == 5);  
#make a copy of trainY  
#Set 'not 5' as 0 and '5' as 1  
y=trainY;  
y(a)=0;  
y(b)=1;  
X=trainX(1:5000,:);  
Y=y(1:5000);  
# Set the dimensions of layer  
layersDimensions=[784, 7,7,3,1];  
# Compute the DL network  
[weights biases costs]=L\_Layer\_DeepModel(X', Y', layersDimensions,  
hiddenActivationFunc='relu',  
learningRate = 0.1,  
numIterations = 5000);



**DL3Functions.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 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 |
|  |  |
|  | # 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 |
|  |  |
|  | # 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); |
|  | 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 |
|  | # 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] = forwardPropagationDeep(X, weights,biases, hiddenActivationFunc='relu') |
|  | # Create an empty cell array |
|  | forward\_caches = {}; |
|  | activation\_caches = {}; |
|  | # 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); |
|  | 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 = "sigmoid"); |
|  | forward\_caches{L}=forward\_cache; |
|  | activation\_caches{L} = activation\_cache; |
|  |  |
|  | end |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # Output: cost |
|  | function [cost]= computeCost(AL,Y) |
|  | numTraining= size(Y)(2); |
|  | # Element wise multiply for logprobs |
|  | cost = -1/numTraining \* sum((Y .\* log(AL)) + (1-Y) .\* log(1-AL)); |
|  | end |
|  |  |
|  | # 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 |
|  | function [dA\_prev dW db] = layerActivationBackward(dA, forward\_cache, activation\_cache, activationFunc) |
|  |  |
|  | 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); |
|  | endif |
|  | A\_prev = forward\_cache{1}; |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | numTraining = size(A\_prev)(2); |
|  | dW = 1/numTraining \* dZ \* A\_prev'; |
|  | db = 1/numTraining \* sum(dZ,2); |
|  | dA\_prev = W'\*dZ; |
|  |  |
|  | 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 |
|  | # |
|  | # 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,hiddenActivationFunc='relu') |
|  |  |
|  |  |
|  | # Set the number of layers |
|  | L = length(activation\_caches); |
|  | m = size(AL)(2); |
|  |  |
|  |  |
|  | # Initializing the backpropagation |
|  | # dl/dAL= -(y/a + (1-y)/(1-a)) - At the output layer |
|  | dAL = -((Y ./ AL) - (1 - Y) ./ ( 1 - AL)); |
|  |  |
|  | # 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} |
|  | [dA dW db] = layerActivationBackward(dAL, forward\_cache{1}, activation\_cache, activationFunc = "sigmoid"); |
|  | gradsDA{L}= dA; |
|  | 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}; |
|  | [dA\_prev\_temp, dW\_temp, db\_temp] = layerActivationBackward(dAl, forward\_cache{1}, activation\_cache, activationFunc = hiddenActivationFunc); |
|  | 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 |
|  | #output : Updated weights after 1 iteration |
|  | function [weights biases] = gradientDescent(weights, biases,gradsW,gradsB, learningRate) |
|  |  |
|  | L = size(weights)(2); # number of layers in the neural network |
|  |  |
|  | # Update rule for each parameter. |
|  | for l=1:L |
|  | weights{l} = weights{l} -learningRate\* gradsW{l}; |
|  | biases{l} = biases{l} -learningRate\* gradsB{l}; |
|  | endfor |
|  | end |
|  |  |
|  |  |
|  |  |
|  | function [weights biases costs] = L\_Layer\_DeepModel(X, Y, layersDimensions, hiddenActivationFunc='relu', learning\_rate = .3, num\_iterations = 10000)#lr was 0.009 |
|  |  |
|  | rand ("seed", 1); |
|  | costs = [] ; |
|  |  |
|  | # Parameters initialization. |
|  | [weights biases] = initializeDeepModel(layersDimensions); |
|  |  |
|  | # Loop (gradient descent) |
|  | for i = 0:num\_iterations |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID. |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights, biases,hiddenActivationFunc); |
|  |  |
|  | # Compute cost. |
|  | cost = computeCost(AL, Y); |
|  |  |
|  | # Backward propagation. |
|  | [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y, activation\_caches,forward\_caches,hiddenActivationFunc); |
|  |  |
|  | # Update parameters. |
|  | [weights biases] = gradientDescent(weights,biases, gradsDW,gradsDB,learning\_rate); |
|  |  |
|  |  |
|  | # 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) |
|  | iterations=[0:1000:maxIterations]; |
|  | plot(iterations,costs); |
|  | title ("Cost vs no of iterations for different learning rates"); |
|  | xlabel("No of iterations"); |
|  | ylabel("Cost"); |
|  | end; |
|  |  |
|  | # Compute the predicted value for a given input |
|  | # Input : Neural Network parameters |
|  | # : Input data |
|  | function [predictions]= predict(weights, biases, X,hiddenActivationFunc="relu") |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights, biases,hiddenActivationFunc); |
|  | predictions = (AL>0.5); |
|  | end |
|  |  |
|  | # Plot the decision boundary |
|  | function plotDecisionBoundary(data,weights, biases,hiddenActivationFunc="relu") |
|  | %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',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; |
|  |  |
|  | end |
|  |  |
|  | function [AL]= scores(weights, biases, X,hiddenActivationFunc="relu") |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights, biases,hiddenActivationFunc); |
|  | end |
|  |  |

**DLFunctions33.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 of a vector |
|  | relu <-function(Z){ |
|  | A <- apply(Z, 1:2, function(x) max(0,x)) |
|  | 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) |
|  | } |
|  |  |
|  | # 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) |
|  | } |
|  |  |
|  | # 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) |
|  | } |
|  |  |
|  |  |
|  | # 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) |
|  | } |
|  | # 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 |
|  | # 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,hiddenActivationFunc='relu'){ |
|  | caches <- 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 |
|  | } |
|  |  |
|  | # 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 = "sigmoid") |
|  | 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) |
|  | return(fwdPropDeep) |
|  |  |
|  | } |
|  |  |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # Output: cost |
|  | computeCost <- function(AL,Y){ |
|  | # Element wise multiply for logprobs |
|  | m= length(Y) |
|  | cost=-1/m\*sum(Y\*log(AL) + (1-Y)\*log(1-AL)) |
|  | #cost=-1/m\*sum(a+b) |
|  | return(cost) |
|  | } |
|  |  |
|  |  |
|  | # Compute the backpropagation through a layer |
|  | # 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 |
|  | layerActivationBackward <- function(dA, cache, activationFunc){ |
|  | # Get A\_prev,W,b |
|  | forward\_cache <-cache[['forward\_cache']] |
|  | # 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) |
|  | } |
|  | A\_prev <- forward\_cache[['A\_prev']] |
|  | 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 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 |
|  | # |
|  | # Returns: |
|  | # gradients -- A dictionary with the gradients |
|  | # gradients["dA" + str(l)] = ... |
|  | # |
|  | backwardPropagationDeep <- function(AL, Y, caches,hiddenActivationFunc='relu'){ |
|  | #initialize the gradients |
|  | gradients = list() |
|  | # Set the number of layers |
|  | L = length(caches) |
|  | numTraining = dim(AL)[2] |
|  |  |
|  | # Initializing the backpropagation |
|  | # dl/dAL= -(y/a) - ((1-y)/(1-a)) - At the output layer |
|  | dAL = -( (Y/AL) -(1 - Y)/(1 - AL)) |
|  |  |
|  | # 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"] |
|  | # Start with Layer L |
|  | # Get the current cache |
|  | current\_cache = caches[[L]]$cache |
|  | #gradients["dA" + str(L)], gradients["dW" + str(L)], gradients["db" + str(L)] = layerActivationBackward(dAL, current\_cache, activationFunc = "sigmoid") |
|  | retvals <- layerActivationBackward(dAL, current\_cache, activationFunc = "sigmoid") |
|  | # Create gradients as lists |
|  | gradients[[paste("dA",L,sep="")]] <- retvals[['dA\_Prev']] |
|  | gradients[[paste("dW",L,sep="")]] <- retvals[['dW']] |
|  | gradients[[paste("db",L,sep="")]] <- retvals[['db']] |
|  |  |
|  |  |
|  | # Traverse in the reverse direction |
|  | for(l in (L-1):1){ |
|  | # Compute the gradients for L-1 to 1 for Relu/tanh |
|  | # Inputs: "gradients["dA" + str(l + 2)], caches". |
|  | # Outputs: "gradients["dA" + str(l + 1)] , gradients["dW" + str(l + 1)] , gradients["db" + str(l + 1)] |
|  | current\_cache = caches[[l]]$cache |
|  | retvals = layerActivationBackward(gradients[[paste('dA',l+1,sep="")]], |
|  | current\_cache, |
|  | 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 |
|  | #output : Updated weights after 1 iteration |
|  | gradientDescent <- function(parameters, gradients, learningRate){ |
|  |  |
|  | 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){ |
|  | 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="")]] |
|  | } |
|  | 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 |
|  | # : learning rate |
|  | # : num of iterations |
|  | #output : Updated weights after each iteration |
|  |  |
|  | L\_Layer\_DeepModel <- function(X, Y, layersDimensions, |
|  | hiddenActivationFunc='relu', |
|  | learningRate = .3, |
|  | numIterations = 10000, print\_cost=False){ |
|  | #Initialize costs vector as NULL |
|  | costs <- NULL |
|  |  |
|  | # Parameters initialization. |
|  | parameters = initializeDeepModel(layersDimensions) |
|  |  |
|  | # Loop (gradient descent) |
|  | for( i in 0:numIterations){ |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID. |
|  | retvals = forwardPropagationDeep(X, parameters,hiddenActivationFunc) |
|  | AL <- retvals[['AL']] |
|  | caches <- retvals[['caches']] |
|  |  |
|  | # Compute cost. |
|  | cost <- computeCost(AL, Y) |
|  |  |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, Y, caches,hiddenActivationFunc) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate) |
|  |  |
|  |  |
|  | if(i%%1000 == 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,hiddenActivationFunc='relu'){ |
|  |  |
|  | fwdProp <- forwardPropagationDeep(X, parameters,hiddenActivationFunc) |
|  | predictions <- fwdProp$AL>0.5 |
|  |  |
|  | return (predictions) |
|  | } |
|  |  |
|  | # Plot a decision boundary |
|  | # This function uses ggplot2 |
|  | plotDecisionBoundary <- function(z,retvals,hiddenActivationFunc,lr){ |
|  | # Find the minimum and maximum for the data |
|  | xmin<-min(z[,1]) |
|  | xmax<-max(z[,1]) |
|  | ymin<-min(z[,2]) |
|  | ymax<-max(z[,2]) |
|  |  |
|  | # Create a grid of values |
|  | a=seq(xmin,xmax,length=100) |
|  | b=seq(ymin,ymax,length=100) |
|  | grid <- expand.grid(x=a, y=b) |
|  | colnames(grid) <- c('x1', 'x2') |
|  | grid1 <-t(grid) |
|  | # Predict the output for this grid |
|  | q <-predict(retvals$parameters,grid1,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) |
|  | } |
|  |  |
|  | # 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) |
|  | } |

**DLFunctions34.py**

|  |
| --- |
| import numpy as np |
|  | import matplotlib.pyplot as plt |
|  | import matplotlib |
|  | import matplotlib.pyplot as plt |
|  | from matplotlib import cm |
|  |  |
|  |  |
|  | # Conmpute the sigmoid of a vector. Also return Z |
|  | 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 |
|  |  |
|  | # Compute the detivative of Relu |
|  | # g'(z) = 1 if z >0 and 0 otherwise |
|  | 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 |
|  | # Derivative g'(z) = a\* (1-a) |
|  | def sigmoidDerivative(dA, cache): |
|  | Z = cache |
|  | s = 1/(1+np.exp(-Z)) |
|  | dZ = dA \* s \* (1-s) |
|  | return dZ |
|  |  |
|  | # Compute the derivative of tanh |
|  | # Derivative g'(z) = 1- a^2 |
|  | def tanhDerivative(dA, cache): |
|  | Z = cache |
|  | a = np.tanh(Z) |
|  | dZ = dA \* (1 - np.power(a, 2)) |
|  | 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. Initialize an empty dictionary |
|  | layerParams = {} |
|  | for l in range(1,len(layerDimensions)): |
|  | # Append to dictionary |
|  | 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) |
|  |  |
|  |  |
|  | # 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, activationFunc): |
|  |  |
|  | # 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) |
|  | 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 |
|  | # 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,hiddenActivationFunc='relu'): |
|  | caches = [] |
|  | # 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 |
|  | A, cache = layerActivationForward(A\_prev, parameters['W'+str(l)], parameters['b'+str(l)], activationFunc = hiddenActivationFunc) |
|  | 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 = "sigmoid") |
|  | caches.append(cache) |
|  |  |
|  | return AL, caches |
|  |  |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # Output: cost |
|  | def computeCost(AL,Y): |
|  | 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) |
|  | return cost |
|  |  |
|  | # Compute the backpropoagation for tnrough 1 layer |
|  | # 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, activationFunc): |
|  | forward\_cache, activation\_cache = cache |
|  | if activationFunc == "relu": |
|  | dZ = reluDerivative(dA, activation\_cache) |
|  | elif activationFunc == "sigmoid": |
|  | dZ = sigmoidDerivative(dA, activation\_cache) |
|  | elif activationFunc == "tanh": |
|  | dZ = tanhDerivative(dA, activation\_cache) |
|  |  |
|  | A\_prev, W, b = forward\_cache |
|  | numtraining = float(A\_prev.shape[1]) |
|  | dW = 1/numtraining \*(np.dot(dZ,A\_prev.T)) |
|  | db = 1/numtraining \* np.sum(dZ, axis=1, keepdims=True) |
|  | dA\_prev = np.dot(W.T,dZ) |
|  | return dA\_prev, dW, db |
|  |  |
|  | # Compute the backpropoagation 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 |
|  | # |
|  | # Returns: |
|  | # gradients -- A dictionary with the gradients |
|  | # gradients["dA" + str(l)] = ... |
|  | # gradients["dW" + str(l)] = ... |
|  |  |
|  | def backwardPropagationDeep(AL, Y, caches,hiddenActivationFunc='relu'): |
|  | #initialize the gradients |
|  | gradients = {} |
|  | # Set the number of layers |
|  | L = len(caches) |
|  | m = float(AL.shape[1]) |
|  | 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)) |
|  |  |
|  | # Since this is a binary classification the activation at output is sigmoid |
|  | # Get the gradients at the last layer |
|  | current\_cache = caches[L-1] |
|  | gradients["dA" + str(L)], gradients["dW" + str(L)], gradients["db" + str(L)] = layerActivationBackward(dAL, current\_cache, activationFunc = "sigmoid") |
|  |  |
|  | # Traverse in the reverse direction |
|  | for l in reversed(range(L-1)): |
|  | # Compute the gradients for L-1 to 1 for Relu/tanh |
|  | current\_cache = caches[l] |
|  | dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackward(gradients['dA'+str(l+2)], current\_cache, 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 |
|  | #output : Updated weights after 1 iteration |
|  | def gradientDescent(parameters, gradients, learningRate): |
|  |  |
|  | L = len(parameters) / 2 |
|  |  |
|  | # Update rule for each parameter. |
|  | for l in range(L): |
|  | 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)] |
|  |  |
|  | 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 |
|  | # : learning rate |
|  | # : num of iterations |
|  | #output : Updated weights after 1 iteration |
|  |  |
|  | def L\_Layer\_DeepModel(X, Y, layersDimensions, hiddenActivationFunc='relu', learning\_rate = .3, num\_iterations = 10000, fig="figx.png"):#lr was 0.009 |
|  |  |
|  | np.random.seed(1) |
|  | costs = [] |
|  |  |
|  | #Initialize paramaters |
|  | parameters = initializeDeepModel(layersDimensions) |
|  |  |
|  | # Perform gradient descent |
|  | for i in range(0, num\_iterations): |
|  | # Perform one cycle of forward propagation |
|  | AL, caches = forwardPropagationDeep(X, parameters,hiddenActivationFunc) |
|  |  |
|  | # Compute cost. |
|  | cost = computeCost(AL, Y) |
|  |  |
|  | # Compute gradients through 1 cycle of backprop |
|  | gradients = backwardPropagationDeep(AL, Y, caches,hiddenActivationFunc) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learning\_rate) |
|  |  |
|  | # Store the costs |
|  | if i % 100 == 0: |
|  | print ("Cost after iteration %i: %f" %(i, cost)) |
|  | if i % 100 == 0: |
|  | costs.append(cost) |
|  |  |
|  | # plot the cost |
|  | fig1=plt.plot(np.squeeze(costs)) |
|  | fig1=plt.ylabel('cost') |
|  | fig1=plt.xlabel('No of iterations(per 100)') |
|  | fig1=plt.title("Learning rate =" + str(learning\_rate)) |
|  | #plt.show() |
|  | fig1.figure.savefig(fig,bbox\_inches='tight') |
|  | plt.clf() |
|  |  |
|  | return parameters |
|  |  |
|  |  |
|  | # 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,fig): |
|  | # 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','yellow'] |
|  | 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 |
|  | fig2=plt.contourf(xx, yy, Z, cmap="coolwarm") |
|  | fig2=plt.ylabel('x2') |
|  | fig2=plt.xlabel('x1') |
|  | fig2=plt.scatter(X[0, :], X[1, :], c=y, s=7,cmap=cmap) |
|  | fig2=plt.title("Decision Boundary for learning rate:"+lr) |
|  | fig2.figure.savefig(fig, bbox\_inches='tight') |
|  | plt.clf() |
|  |  |
|  | # Predict the output for given input |
|  | # Input : parameters |
|  | # : X |
|  | # Output: predictions |
|  | def predict(parameters, X): |
|  | A2, cache = forwardPropagationDeep(X, parameters) |
|  | predictions = (A2>0.5) |
|  | return predictions |
|  |  |
|  | # Predict the probability scores for given data |
|  | # Input : parameters |
|  | # : X |
|  | # Output: probability of output |
|  | def predict\_proba(parameters, X): |
|  | A2, cache = forwardPropagationDeep(X, parameters) |
|  | proba=A2 |
|  | 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\_surface(model, X, y,sz,lr,fig): |
|  | # 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 |
|  | z\_min, z\_max = X[2, :].min() - 1, X[2, :].max() + 1 |
|  | colors=['black','gold'] |
|  | cmap = matplotlib.colors.ListedColormap(colors) |
|  | h = 3 |
|  | # Generate a grid of points with distance h between them |
|  | xx, yy, zz = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h), np.arange(z\_min, z\_max, h)) |
|  | # Predict the function value for the whole grid |
|  | a=np.c\_[xx.ravel(), yy.ravel(), zz.ravel()] |
|  |  |
|  | Z = predict(parameters,a.T) |
|  | Z = Z.reshape(xx.shape) |
|  | # Plot the contour and training examples |
|  | #plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral) |
|  | ax = plt.axes(projection='3d') |
|  | ax.contour3D(xx, yy, Z, 50, cmap='binary') |
|  | #plt.ylabel('x2') |
|  | #plt.xlabel('x1') |
|  | plt.scatter(X[0, :], X[1, :], c=y, cmap=cmap) |
|  | plt.title("Decision Boundary for hidden layer size:" + sz +" and learning rate:"+lr) |
|  | plt.show() |
|  |  |
|  | def plotSurface(X,parameters): |
|  |  |
|  | #xx, yy, zz = np.meshgrid(np.arange(10), np.arange(10), np.arange(10)) |
|  | x\_min, x\_max = X[0, :].min() - 1, X[0, :].max() + 1 |
|  | y\_min, y\_max = X[1, :].min() - 1, X[1, :].max() + 1 |
|  | z\_min, z\_max = X[2, :].min() - 1, X[2, :].max() + 1 |
|  | colors=['red'] |
|  | cmap = matplotlib.colors.ListedColormap(colors) |
|  | h = 1 |
|  | xx, yy, zz = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h), |
|  | np.arange(z\_min, z\_max, h)) |
|  | # For the meh grid values predict a model |
|  | a=np.c\_[xx.ravel(), yy.ravel(), zz.ravel()] |
|  | Z = predict(parameters,a.T) |
|  | r=Z.T |
|  | r1=r.reshape(xx.shape) |
|  | # Find teh values for which the repdiction is 1 |
|  | xx1=xx[r1] |
|  | yy1=yy[r1] |
|  | zz1=zz[r1] |
|  | # Plot these values |
|  | ax = plt.axes(projection='3d') |
|  | #ax.plot\_trisurf(xx1, yy1, zz1, cmap='bone', edgecolor='none'); |
|  | ax.scatter3D(xx1, yy1,zz1, c=zz1,s=10,cmap=cmap) |
|  | #ax.plot\_surface(xx1, yy1, zz1, 'gray') |

**Load\_msist.py**

|  |
| --- |
| import os |
|  | import struct | |
|  | import numpy as np | |
| def read(dataset = "training", path="./mnist"): | |
|  | | """ | |
|  | | Python function for importing the MNIST data set. It returns an iterator | |
|  | | of 2-tuples with the first element being the label and the second element | |
|  | | being a numpy.uint8 2D array of pixel data for the given image. | |
|  | | """ | |
|  | |  | |
|  | | if dataset is "training": | |
|  | | fname\_img = os.path.join(path, 'train-images.idx3-ubyte') | |
|  | | fname\_lbl = os.path.join(path, 'train-labels.idx1-ubyte') | |
|  | | elif dataset is "testing": | |
|  | | fname\_img = os.path.join(path, 't10k-images.idx3-ubyte') | |
|  | | fname\_lbl = os.path.join(path, 't10k-labels.idx1-ubyte') | |
|  | | else: | |
|  | | raise ValueError, "dataset must be 'testing' or 'training'" | |
|  | |  | |
|  | | # Load everything in some numpy arrays | |
|  | | with open(fname\_lbl, 'rb') as flbl: | |
|  | | magic, num = struct.unpack(">II", flbl.read(8)) | |
|  | | lbl = np.fromfile(flbl, dtype=np.int8) | |
|  | |  | |
|  | | with open(fname\_img, 'rb') as fimg: | |
|  | | magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16)) | |
|  | | img = np.fromfile(fimg, dtype=np.uint8).reshape(len(lbl), rows, cols) | |
|  | |  | |
|  | | get\_img = lambda idx: (lbl[idx], img[idx]) | |
|  | |  | |
|  | | # Create an iterator which returns each image in turn | |
|  | | for i in xrange(len(lbl)): | |
|  | | yield get\_img(i) | |
|  | |  | |
|  | | flbl.close() | |
|  | | fimg.close() | |
|  | |  | |
|  | | def show(image): | |
|  | | """ | |
|  | | Render a given numpy.uint8 2D array of pixel data. | |
|  | | """ | |
|  | | from matplotlib import pyplot | |
|  | | import matplotlib as mpl | |
|  | | fig = pyplot.figure() | |
|  | | ax = fig.add\_subplot(1,1,1) | |
|  | | imgplot = ax.imshow(image, cmap=mpl.cm.Greys) | |
|  | | imgplot.set\_interpolation('nearest') | |
|  | | ax.xaxis.set\_ticks\_position('top') | |
|  | | ax.yaxis.set\_ticks\_position('left') | |
|  | | pyplot.show() | |
|  | |  | |
|  | | ''' | |
|  | | training\_data=list(read(dataset='training',path="./mnist")) | |
|  | |  | |
|  | | print(len(training\_data)) | |
|  | | for i in range(10): | |
|  | | labels,pixels=training\_data[i] | |
|  | | print(labels) | |
|  | | show(pixels) | |
|  | |  | |
|  | |  | |
|  | | test\_data=list(read(dataset='testing',path="./mnist")) | |
|  | | print(len(test\_data)) | |
|  | | ''' | |

**MSIST.R**

|  |
| --- |
| load\_mnist <- function() { |
|  | load\_image\_file <- function(filename) { |
|  | ret = list() |
|  | f = file(filename,'rb') |
|  | readBin(f,'integer',n=1,size=4,endian='big') |
|  | ret$n = readBin(f,'integer',n=1,size=4,endian='big') |
|  | nrow = readBin(f,'integer',n=1,size=4,endian='big') |
|  | ncol = readBin(f,'integer',n=1,size=4,endian='big') |
|  | x = readBin(f,'integer',n=ret$n\*nrow\*ncol,size=1,signed=F) |
|  | ret$x = matrix(x, ncol=nrow\*ncol, byrow=T) |
|  | close(f) |
|  | ret |
|  | } |
|  | load\_label\_file <- function(filename) { |
|  | f = file(filename,'rb') |
|  | readBin(f,'integer',n=1,size=4,endian='big') |
|  | n = readBin(f,'integer',n=1,size=4,endian='big') |
|  | y = readBin(f,'integer',n=n,size=1,signed=F) |
|  | close(f) |
|  | y |
|  | } |
|  | train <<- load\_image\_file('./mnist/train-images.idx3-ubyte') |
|  | test <<- load\_image\_file('./mnist/t10k-images.idx3-ubyte') |
|  |  |
|  | train$y <<- load\_label\_file('./mnist/train-labels.idx1-ubyte') |
|  | test$y <<- load\_label\_file('./mnist/t10k-labels.idx1-ubyte') |
|  | } |
|  |  |
|  |  |
|  | show\_digit <- function(arr784, col=gray(12:1/12), ...) { |
|  | image(matrix(arr784, nrow=28)[,28:1], col=col, ...) |
|  | } |

**Conclusion**

It was quite a challenge coding a Deep Learning Network in Python, R and Octave. The Deep Learning network implementation, in this post,is the base Deep Learning network, without any of the regularization methods included. Here are some key learning that I got while playing with different multi-layer networks on different problems

a. Deep Learning Networks come with many levers, the hyper-parameters,  
– learning rate  
– activation unit  
– number of hidden layers  
– number of units per hidden layer  
– number of iterations while performing gradient descent  
b. Deep Networks are very sensitive. A change in any of the hyper-parameter makes it perform very differently  
c. Initially I thought adding more hidden layers, or more units per hidden layer will make the DL network better at learning. On the contrary, there is a performance degradation after the optimal DL configuration  
d. At a sub-optimal number of hidden layers or number of hidden units, gradient descent seems to get stuck at a local minima  
e. There were occasions when the cost came down, only to increase slowly as the number of iterations were increased. Probably early stopping would have helped.