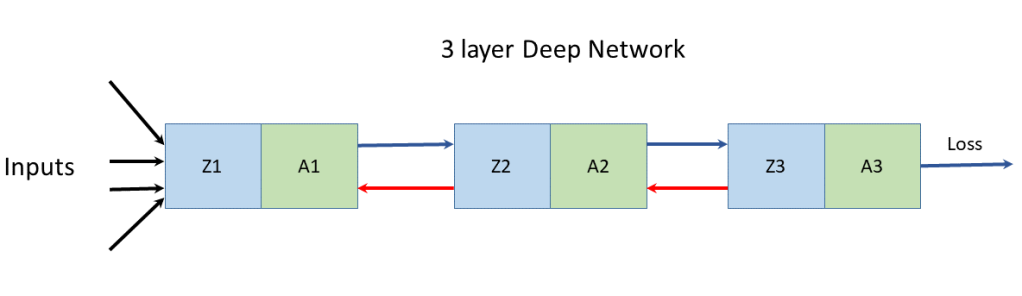
Lets take a simple 3 layer Neural network with 3 hidden layers and an output layer  
  
In the forward propagation cycle the equations are

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)see equation 2 in [Part 1](https://gigadom.wordpress.com/2018/01/04/deep-learning-from-basic-principles-in-python-r-and-octave-part-1/)

\partial L/\partial Z_{3} = \partial L/\partial A_{3}* \partial A_{3}/\partial Z_{3} = A3-Ysee equation (f) in  [Part 1](https://gigadom.wordpress.com/2018/01/04/deep-learning-from-basic-principles-in-python-r-and-octave-part-1/)  
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) – See [Part 1](https://gigadom.wordpress.com/2018/01/04/deep-learning-from-basic-principles-in-python-r-and-octave-part-1/)  
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)))

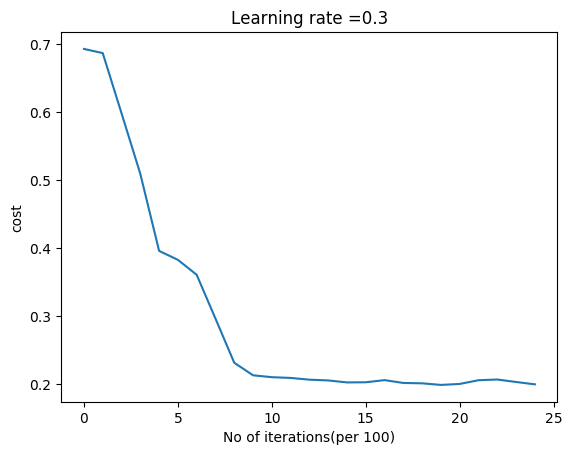
print('F1: {:.2f}'.format(f1\_score(Y2.T, yhat.T)))

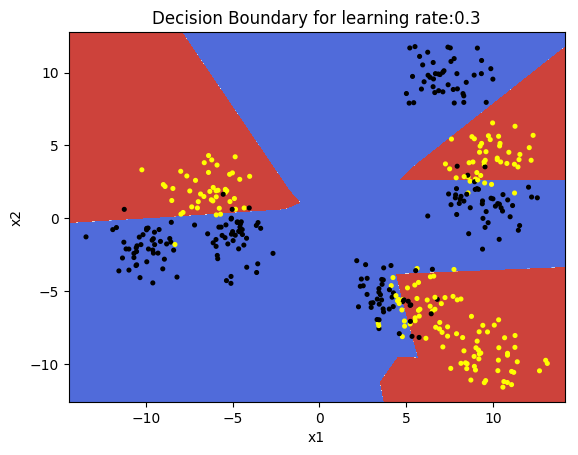
## Accuracy: 0.90

## Precision: 0.91

## Recall: 0.87

## F1: 0.89





For more details on metrics like Accuracy, Recall, Precision etc. used in classification take a look at my post [Practical Machine Learning with R and Python – Part 2](https://gigadom.wordpress.com/2017/10/13/practical-machine-learning-with-r-and-python-part-2/). More details about these and other metrics besides implementation of the most common machine learning algorithms are available in my book [My book ‘Practical Machine Learning with R and Python’ on Amazon](https://gigadom.wordpress.com/2017/12/05/my-book-practical-machine-learning-with-r-and-python-on-amazon/)

**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

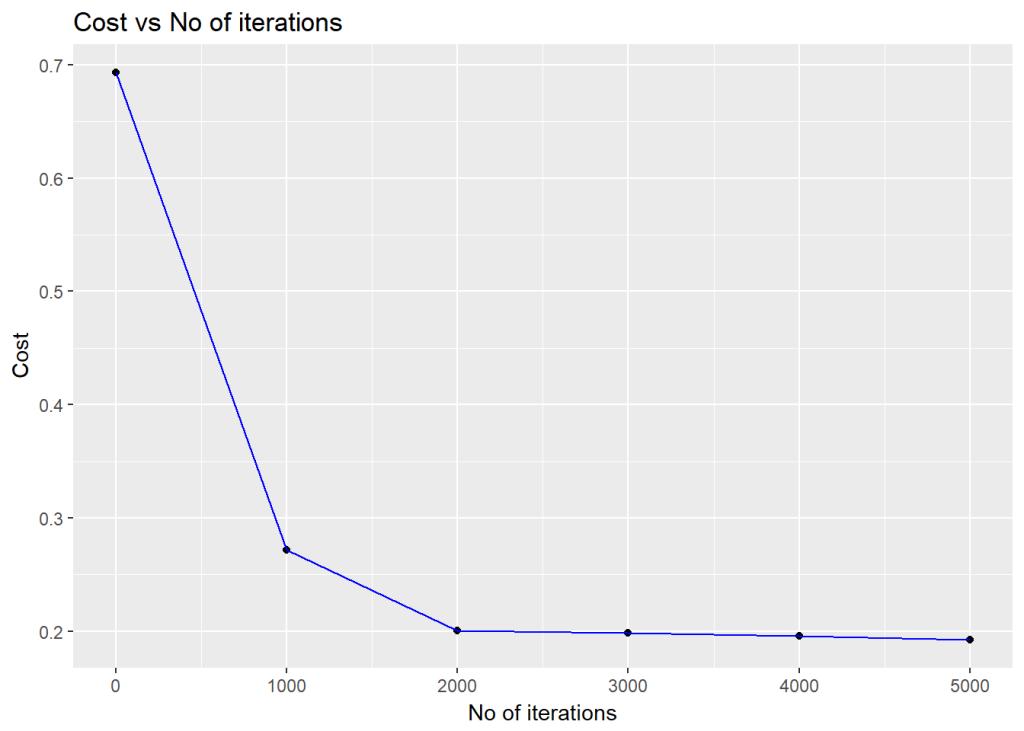
iterations <- seq(0,numIterations,by=1000)

df <-data.frame(iterations,costs)

# Plot the Costs vs number of iterations

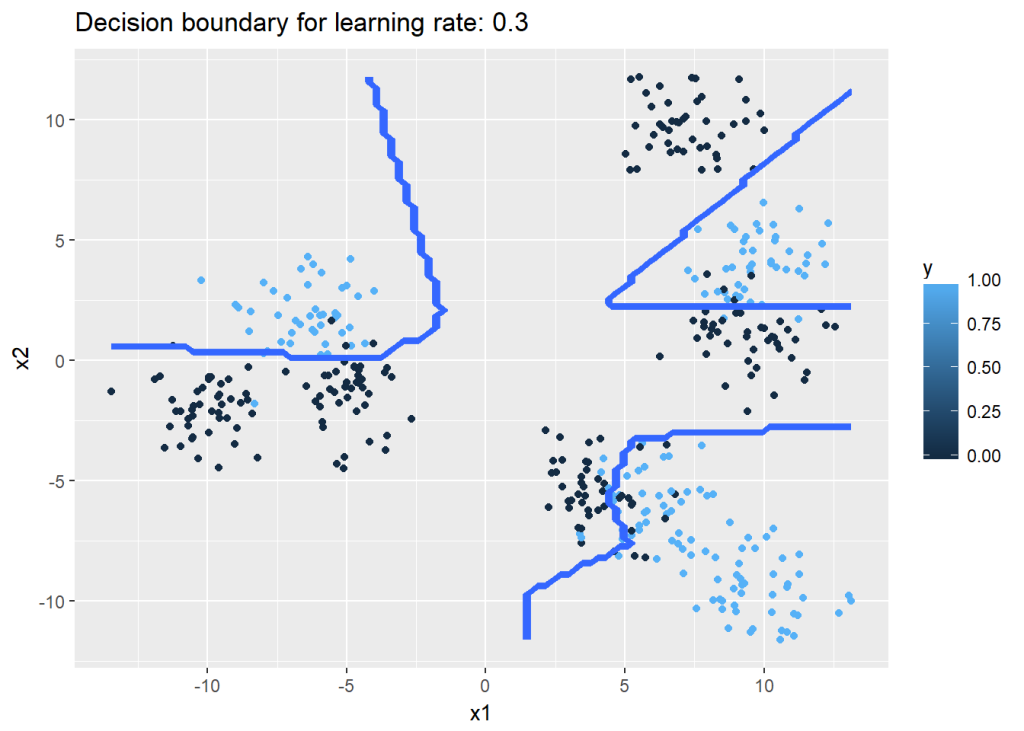
ggplot(df,aes(x=iterations,y=costs)) + geom\_point() +geom\_line(color="blue") +

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

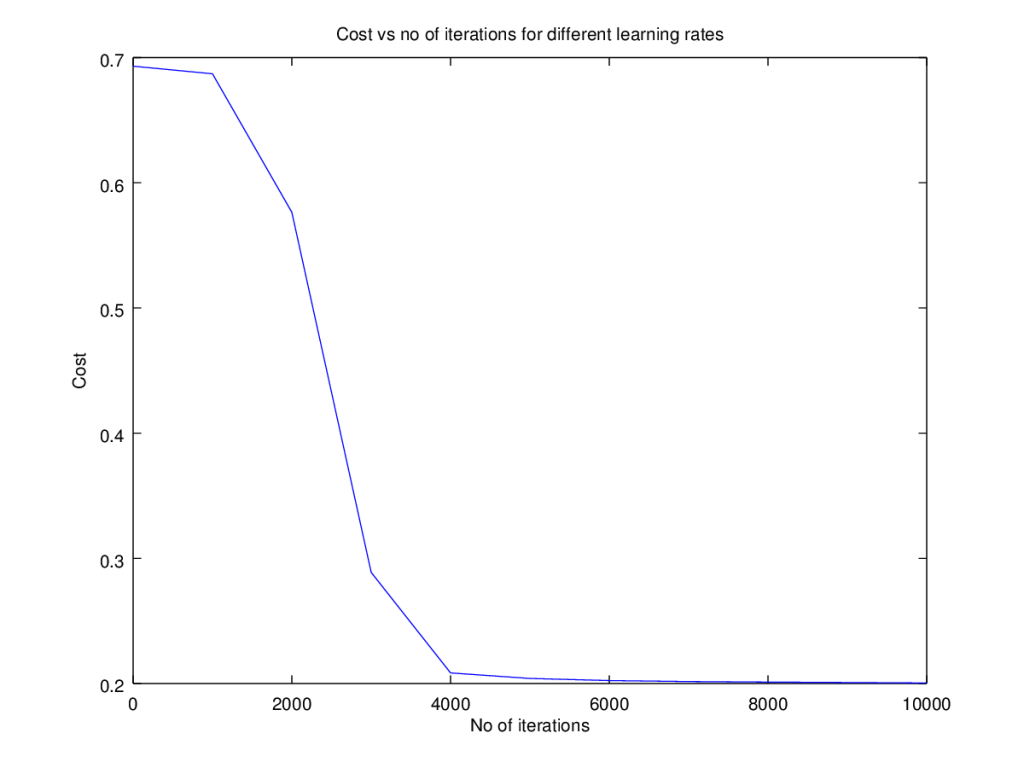
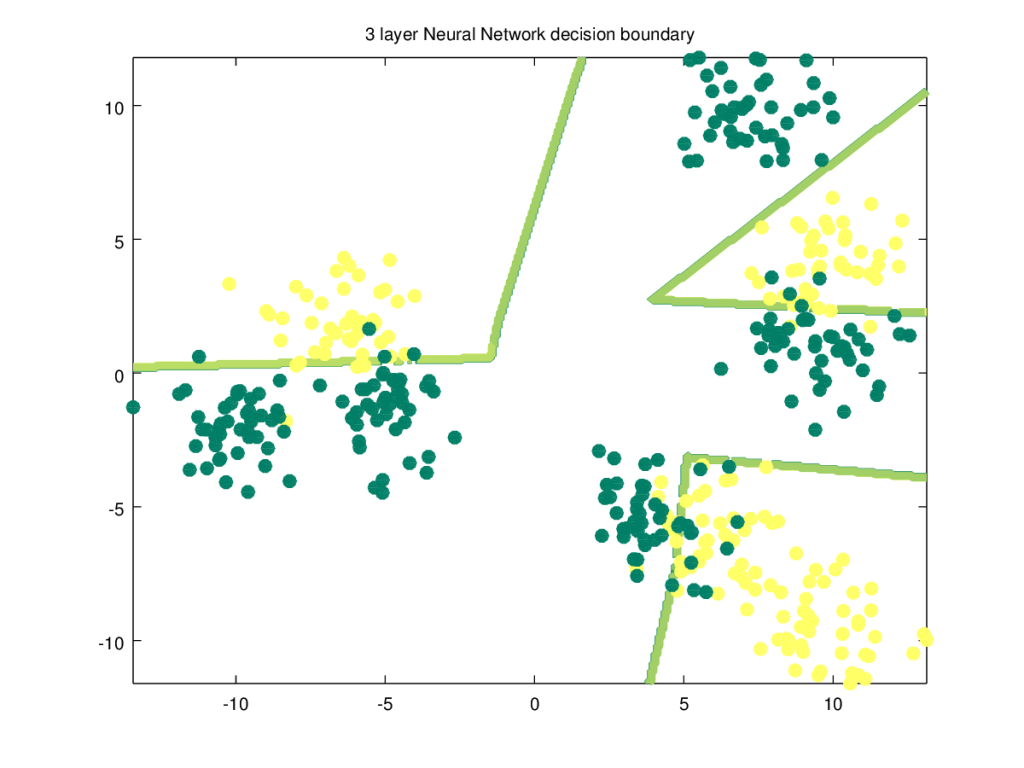
## Balanced Accuracy : 0.9246

##

## '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

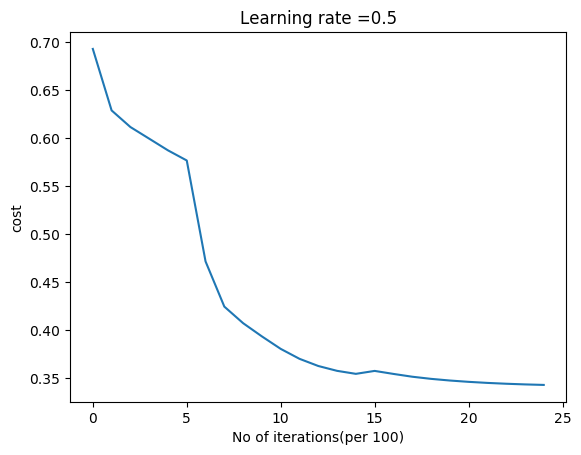
layersDimensions = [2, 4, 1] # 3-layer model

# 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']]

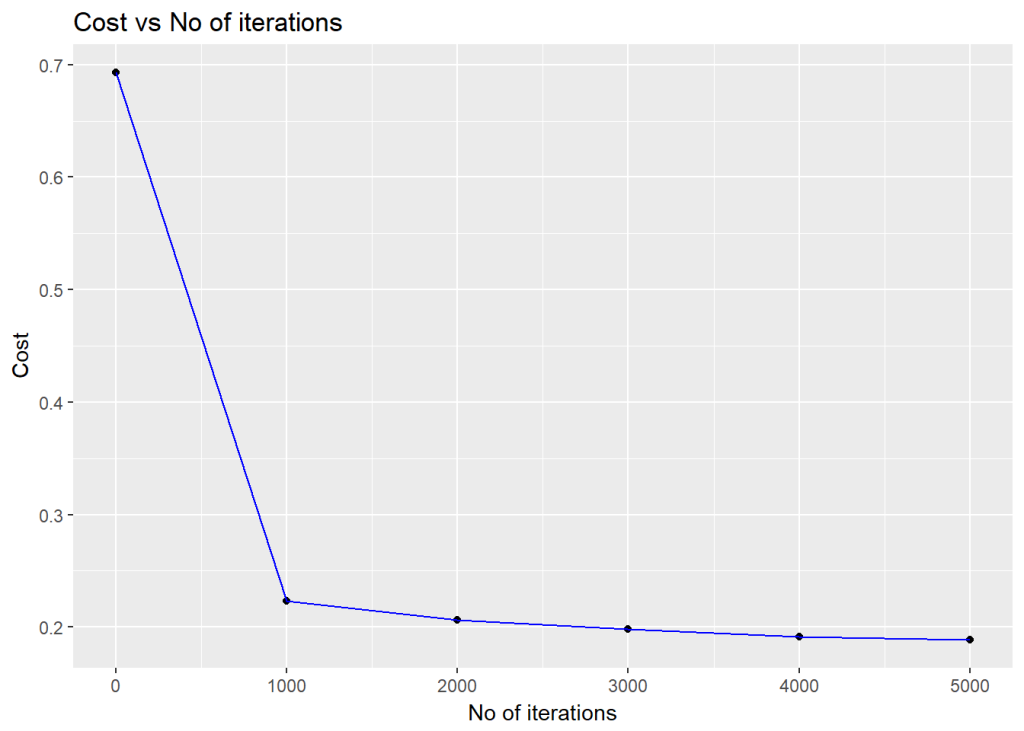
iterations <- seq(0,numIterations,by=1000)

df <-data.frame(iterations,costs)

# Plot Cost vs number of iterations

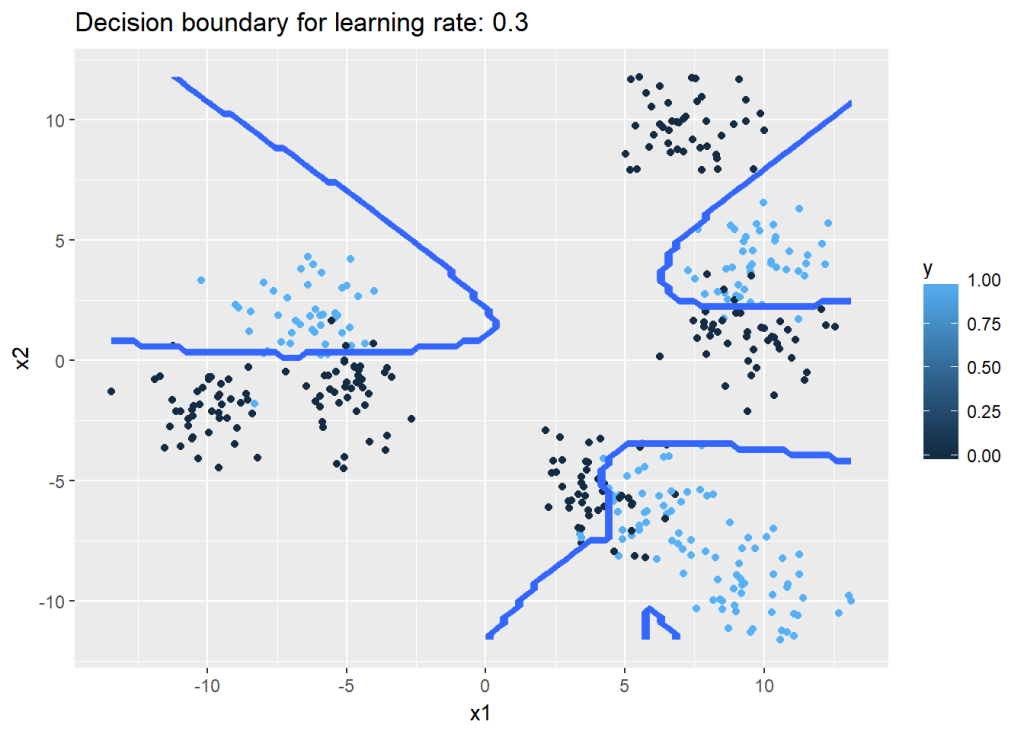
ggplot(df,aes(x=iterations,y=costs)) + geom\_point() +geom\_line(color="blue") +

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

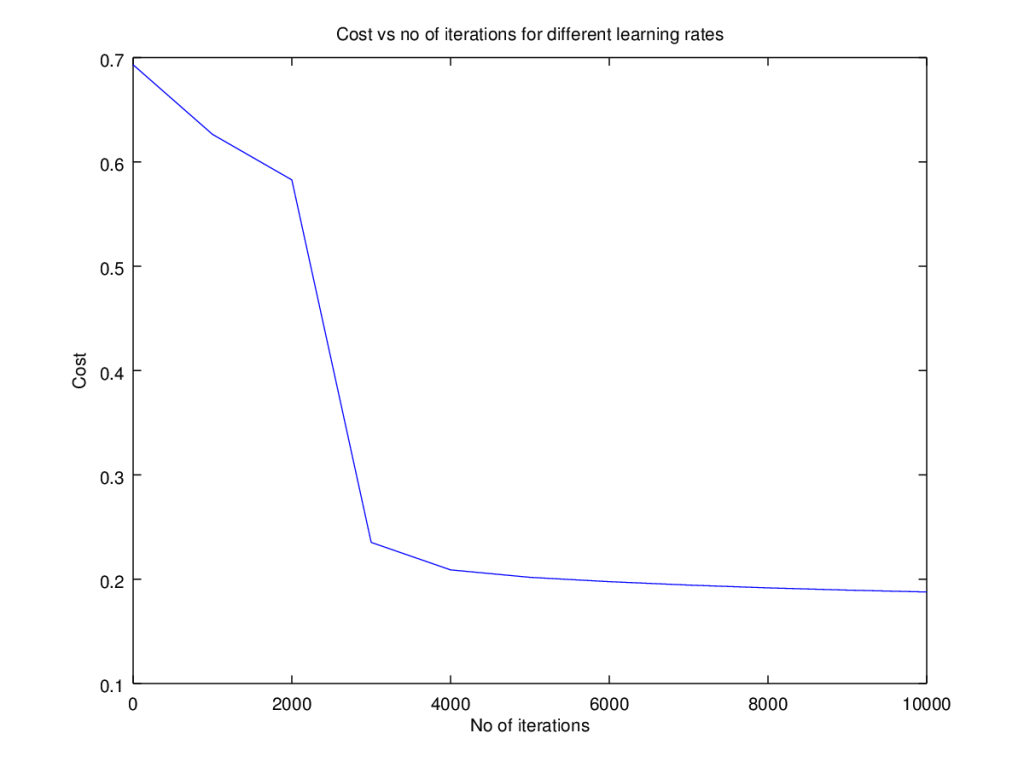
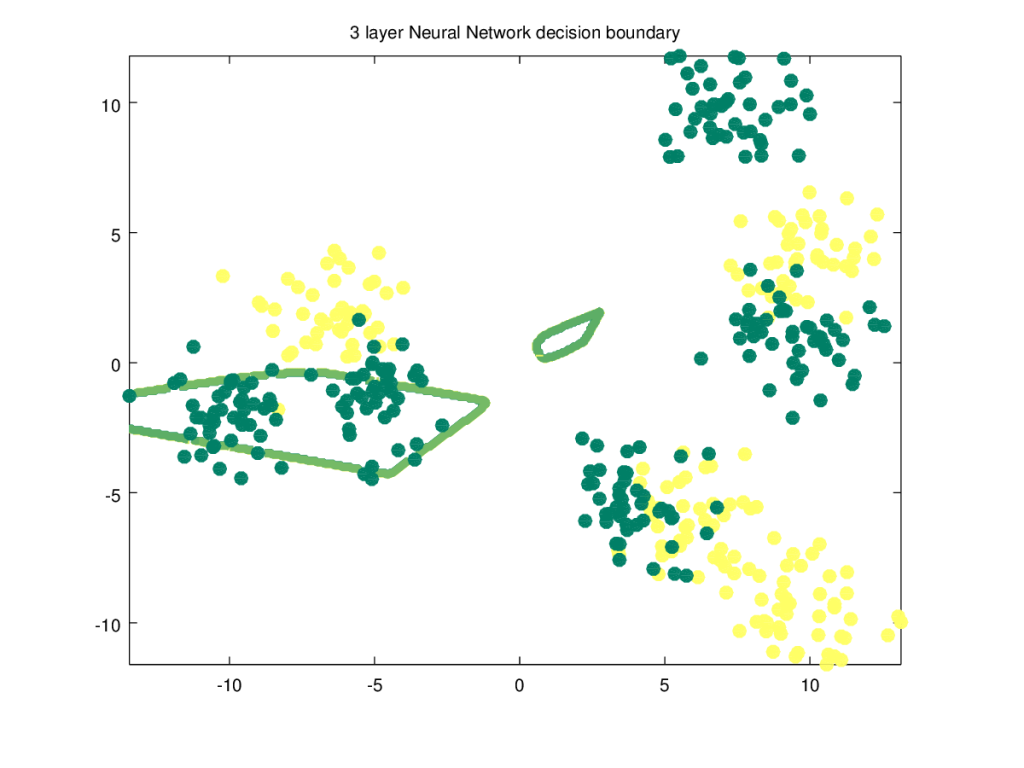


#Plot the decision boundary

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

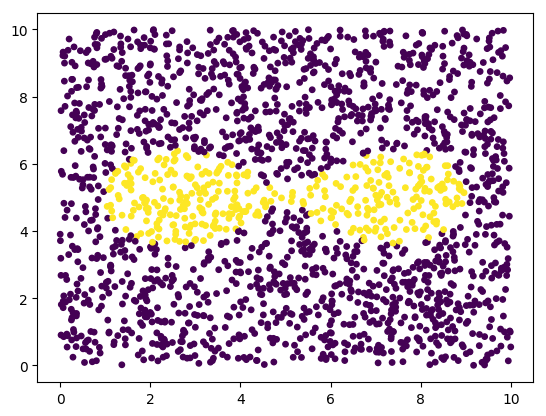


**2c. Classification with Multi layer Deep Learning Network – Tanh activation(Octave)**

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)))

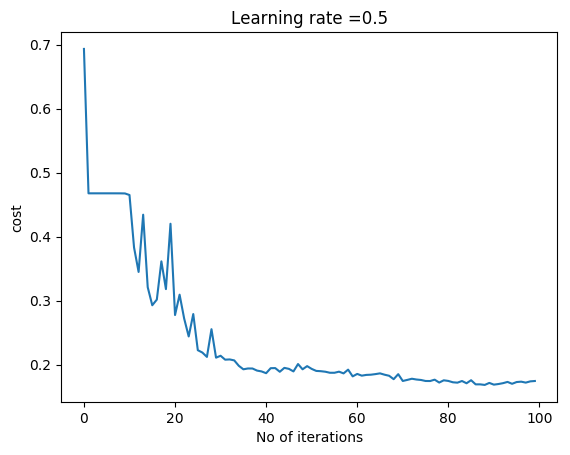
print('F1: {:.2f}'.format(f1\_score(Y2.T, yhat.T)))

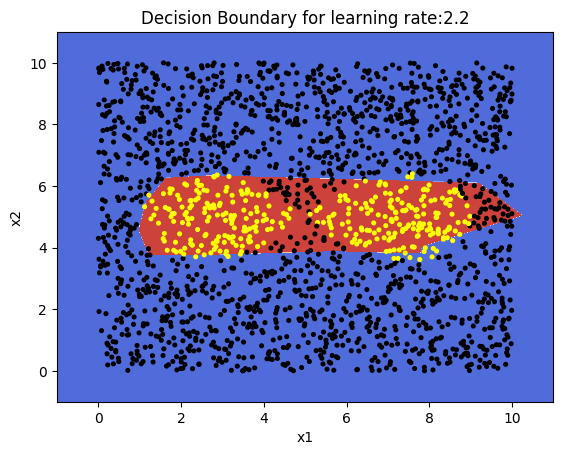
## Accuracy: 0.93

## 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']]

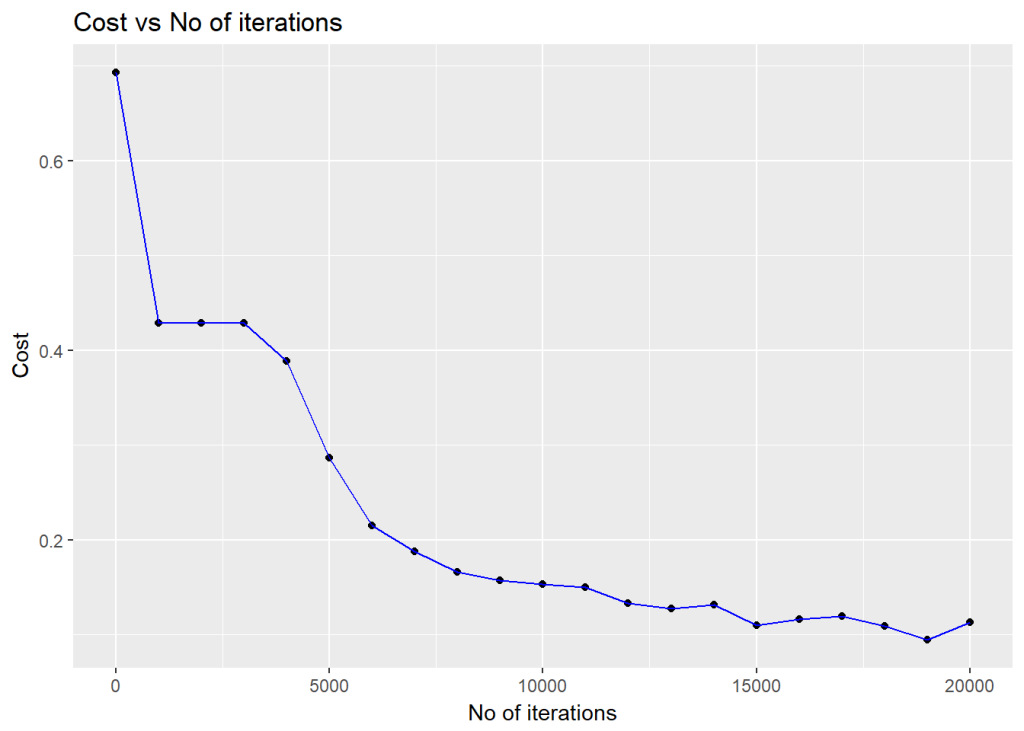
numIterations = 20000

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

df <-data.frame(iterations,costs)

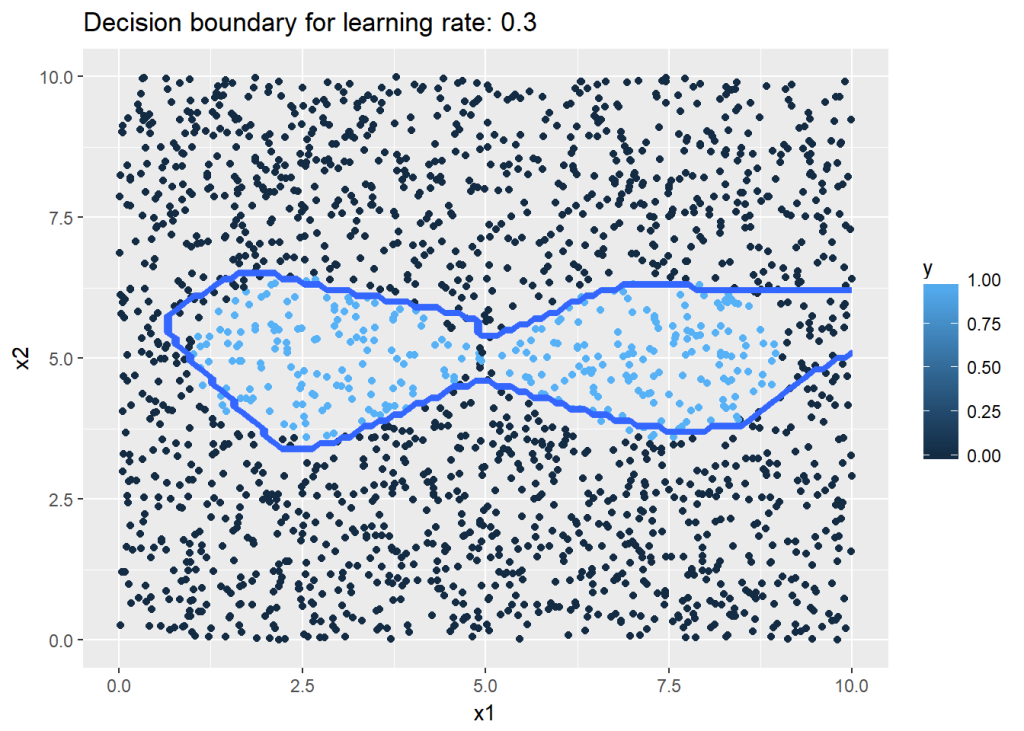
ggplot(df,aes(x=iterations,y=costs)) + geom\_point() +geom\_line(color="blue") +

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



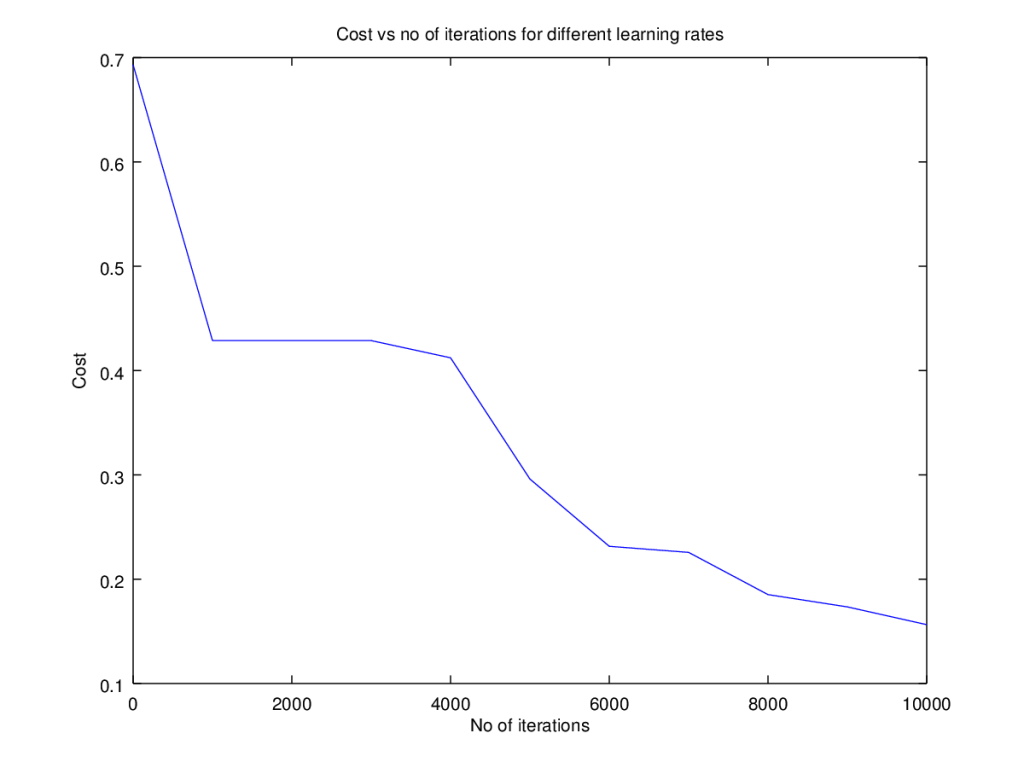
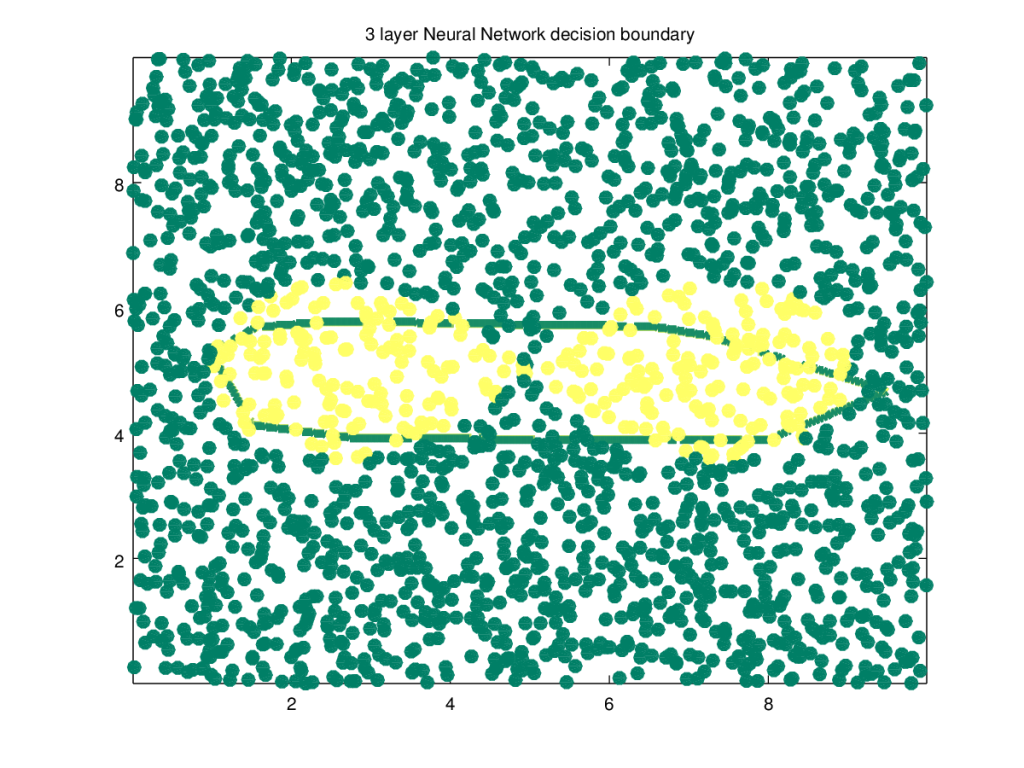
#Plot the decision boundary

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

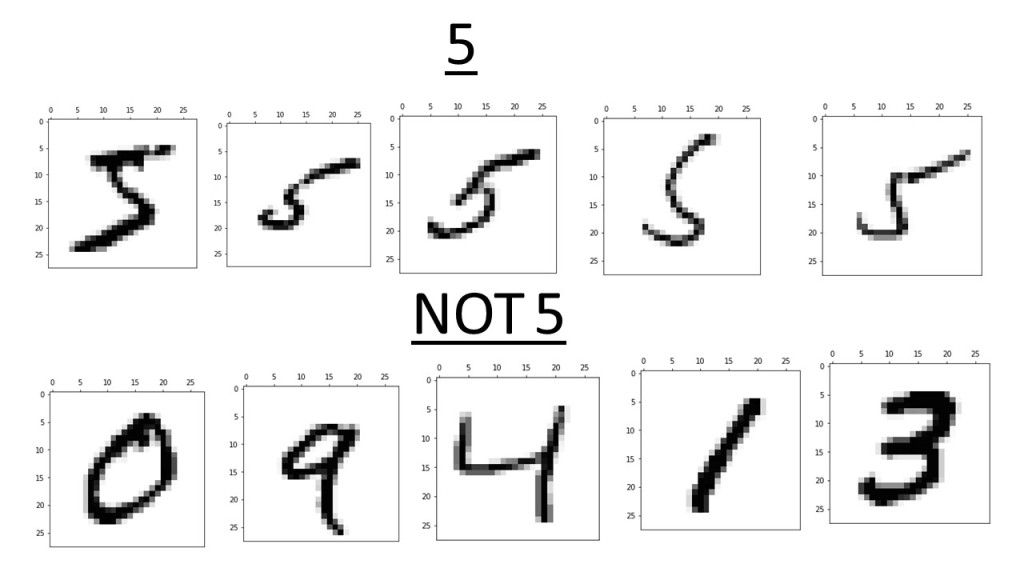


**3c. Classifying a lemniscate with Deep Learning Network – Relu activation(Octave code)**

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”.  
  
The Python code for reading the MNIST data is taken from Alex Kesling’s github link [MNIST](https://gist.github.com/akesling/5358964).

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)

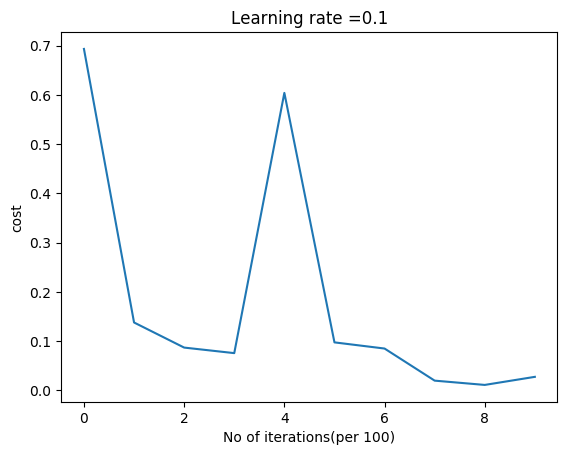
plt.savefig("fig8.png",bbox\_inches='tight')

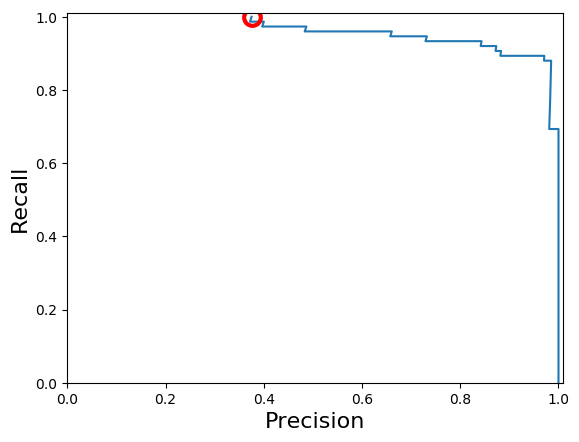
## Accuracy: 0.99

## Precision: 0.96

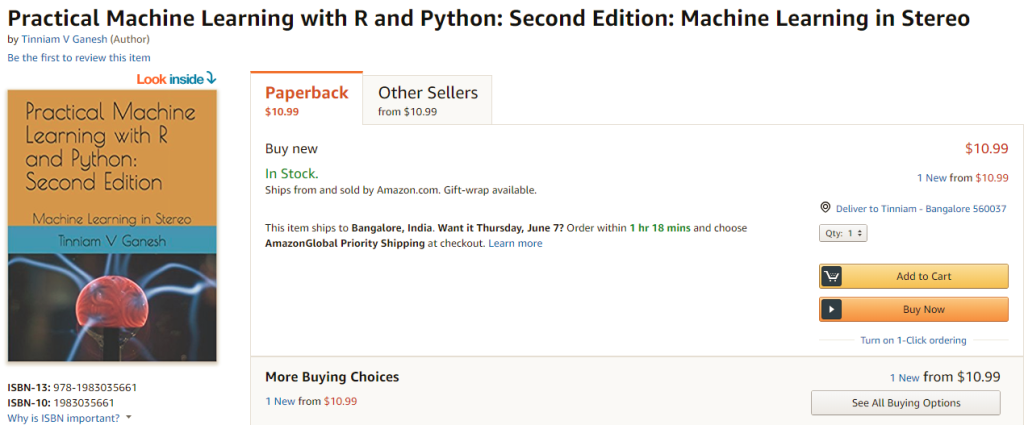
## Recall: 0.89

## 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, please check my post [Practical Machine Learning with R and Python – Part 4](https://gigadom.wordpress.com/2017/10/29/practical-machine-learning-with-r-and-python-part-4/).

Check out my compact and minimal book  “Practical Machine Learning with R and Python:Second edition- Machine Learning in stereo”  available in Amazon in [paperback](https://www.amazon.com/dp/1983035661)($10.99) and [kindle](https://www.amazon.com/dp/B07DFKSCWZ)($7.99) versions. My book includes implementations of key ML algorithms and associated measures and metrics. The book is ideal for anybody who is familiar with the concepts and would like a quick reference to the different ML algorithms that can be applied to problems and how to select the best model. Pick your copy today!!  


A physical copy of the book is much better than scrolling down a webpage. Personally, I tend to use my own book quite frequently to refer to R, Python constructs,  subsetting, machine Learning function calls and the necessary parameters etc. It is useless to commit any of this to memory, and a physical copy of a book is much easier to thumb through for the relevant code snippet. Pick up your copy today!

**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. The code to read and display the MNIST data is taken from Brendan O’ Connor’s github link at [MNIST](https://gist.github.com/brendano/39760)

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']]

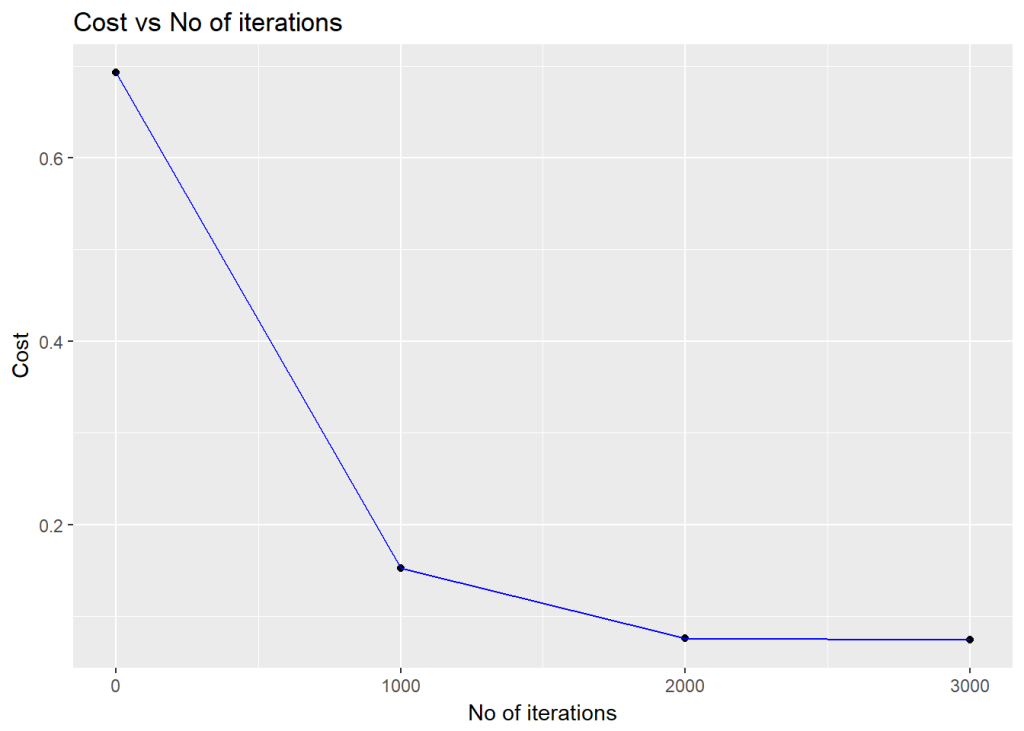
numIterations = 3000

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]

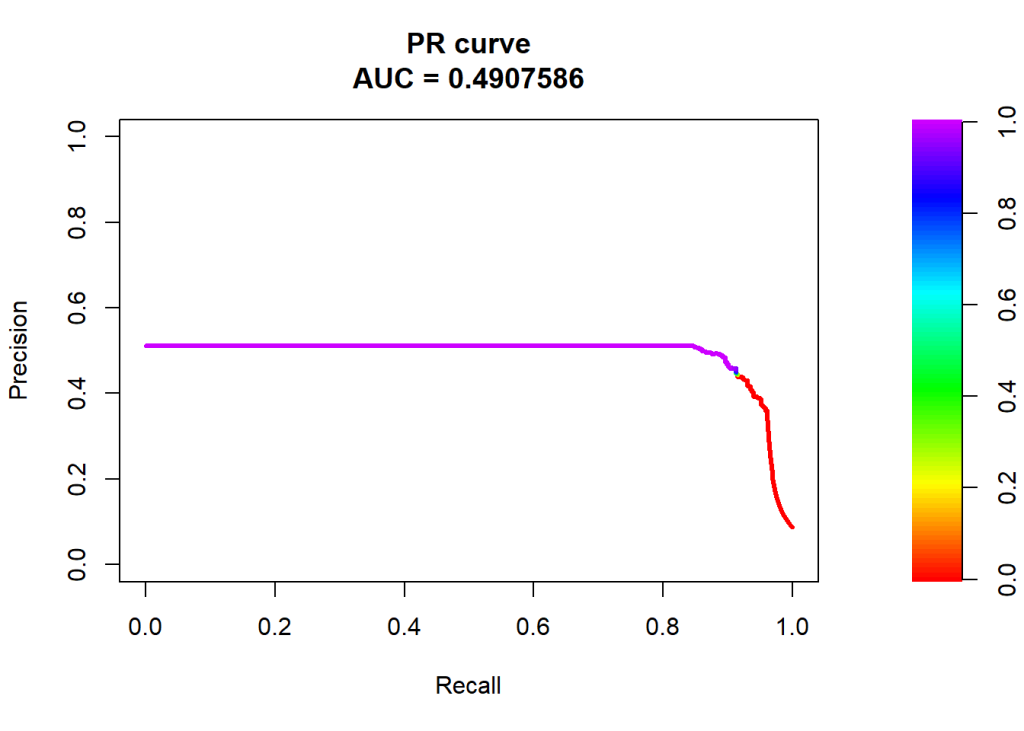
# Plot ROC curve

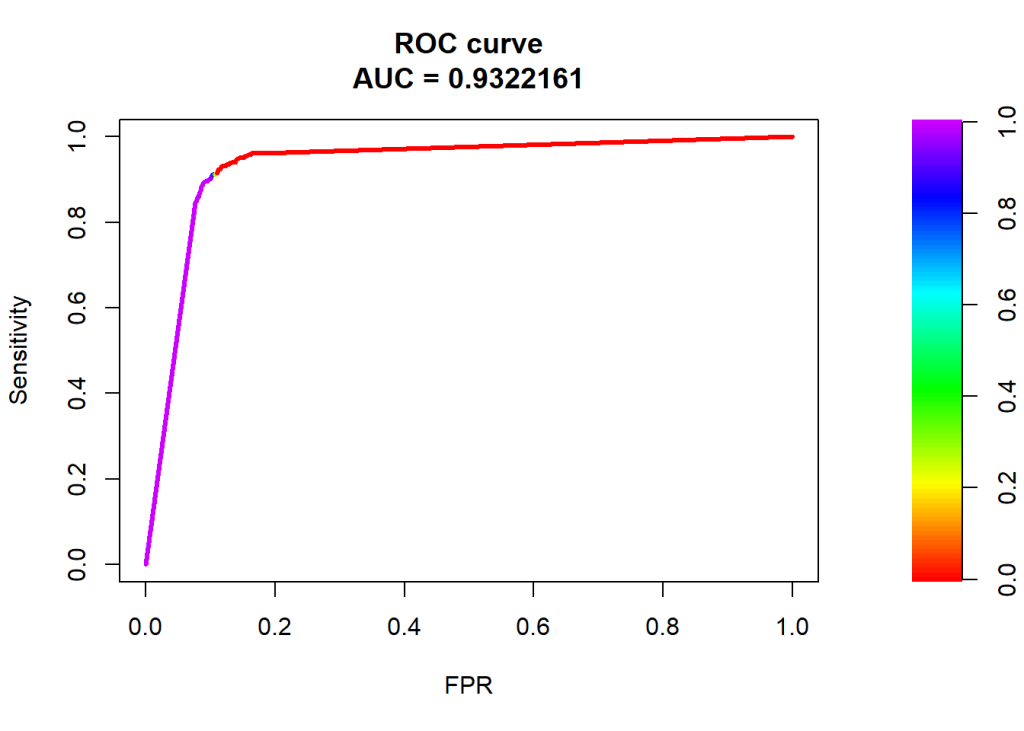
pr <-pr.curve(scores.class0=class1,

scores.class1=class0,

curve=T)

plot(pr)

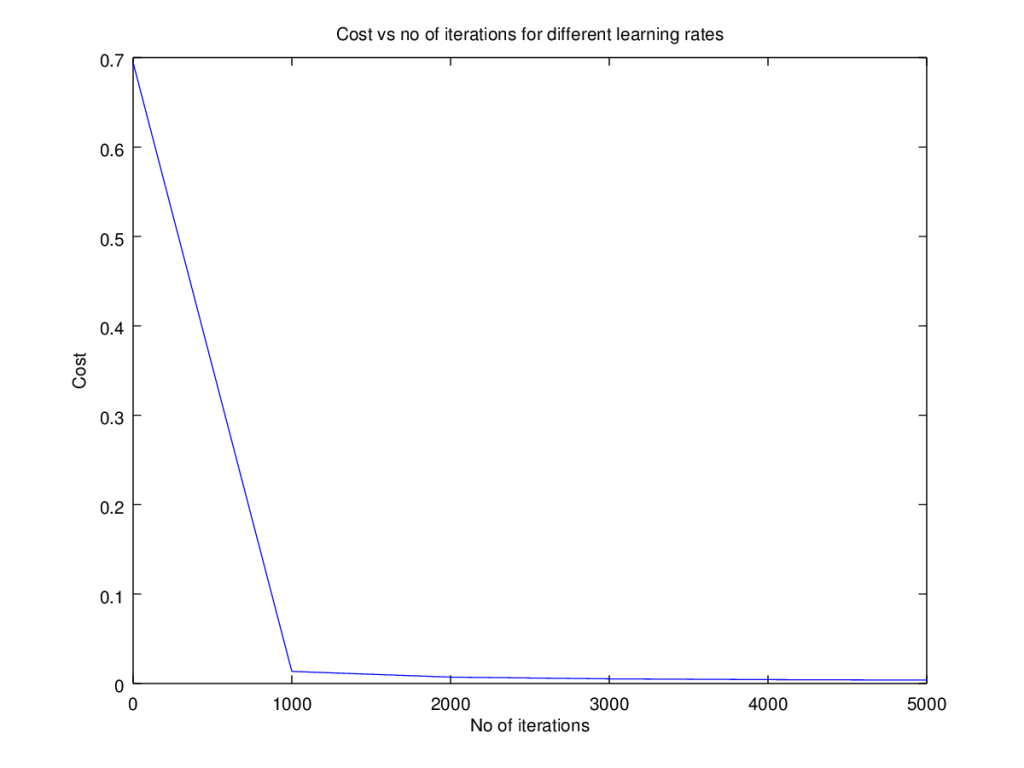




The AUC curve hugs the top left corner and hence the performance of the classifier is quite good.

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

This code to load MNIST data was taken from [Daniel E blog.](http://daniel-e.github.io/2017-10-20-loading-mnist-handwritten-digits-with-octave-or-matlab/)   
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);



**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.