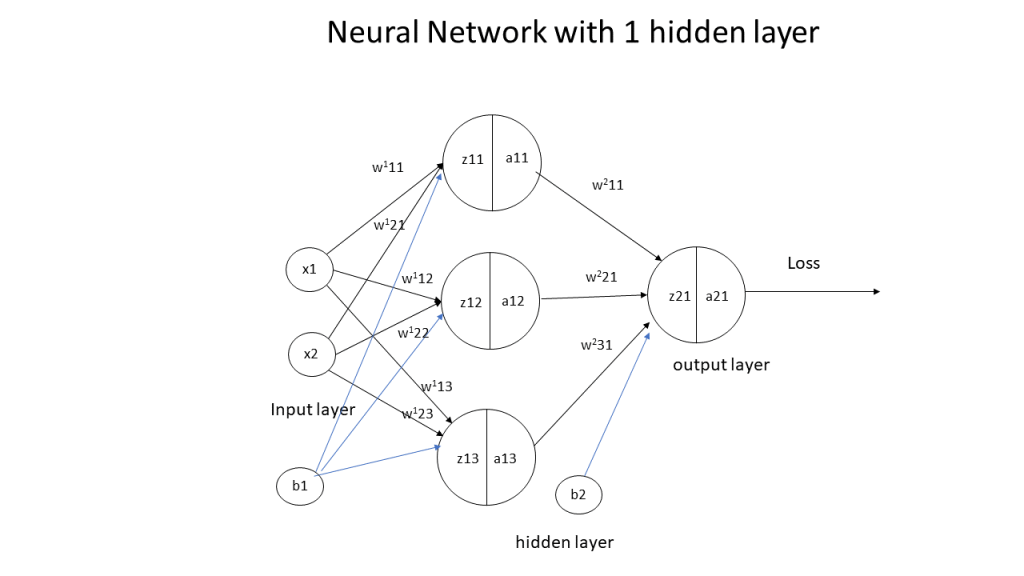
**The 3 layer Neural Network**

A simple representation of a 3 layer Neural Network (NN) with 1 hidden layer is shown below.  
  
In the above Neural Network, there are 2 input features at the input layer, 3 hidden units at the hidden layer and 1 output layer as it deals with binary classification. The activation unit at the hidden layer can be a tanh, sigmoid, relu etc. At the output layer the activation is a sigmoid to handle binary classification

# Superscript indicates layer 1  
z_{11} = w_{11}^{1}x_{1} + w_{21}^{1}x_{2} + b_{1}  
z_{12} = w_{12}^{1}x_{1} + w_{22}^{1}x_{2} + b_{1}  
z_{13} = w_{13}^{1}x_{1} + w_{23}^{1}x_{2} + b_{1}

Also a_{11} = tanh(z_{11})  
a_{12} = tanh(z_{12})  
a_{13} = tanh(z_{13})

# Superscript indicates layer 2  
z_{21} = w_{11}^{2}a_{11} + w_{21}^{2}a_{12} + w_{31}^{2}a_{13} + b_{2}  
a_{21} = sigmoid(z21)

Hence  
Z1= \begin{pmatrix}  z11\\  z12\\  z13  \end{pmatrix} =\begin{pmatrix}  w_{11}^{1} & w_{21}^{1} \\  w_{12}^{1} & w_{22}^{1} \\  w_{13}^{1} & w_{23}^{1}  \end{pmatrix} * \begin{pmatrix}  x1\\  x2  \end{pmatrix} + b_{1}  
And  
A1= \begin{pmatrix}  a11\\  a12\\  a13  \end{pmatrix} = \begin{pmatrix}  tanh(z11)\\  tanh(z12)\\  tanh(z13)  \end{pmatrix}

Similarly  
Z2= z_{21}  = \begin{pmatrix}  w_{11}^{2} & w_{21}^{2} & w_{31}^{2}  \end{pmatrix} *\begin{pmatrix}  z_{11}\\  z_{12}\\  z_{13}  \end{pmatrix} +b_{2}  
and A2 = a_{21} = sigmoid(z_{21})

These equations can be written as  
Z1 = W1 * X + b1  
A1 = tanh(Z1)  
Z2 = W2 * A1 + b2  
A2 = sigmoid(Z2)

**I) Some important results** (a memory refresher!)  
d/dx(e^{x}) = e^{x}and d/dx(e^{-x}) = -e^{-x}-(a) and  
sinhx = (e^{x} - e^{-x})/2and coshx = (e^{x} + e^{-x})/2  
Using (a) we can shown that d/dx(sinhx) = coshxand d/dx(coshx) = sinhx(b)  
Now d/dx(f(x)/g(x)) = (g(x)*d/dx(f(x)) - f(x)*d/dx(g(x)))/g(x)^{2}-(c)

Since tanhx =z= sinhx/coshxand using (b) we get  
tanhx = (coshx*d/dx(sinhx) - sinhx*d/dx(coshx))/(cosh^{2})  
Using the values of the derivatives of sinhx and coshx from (b) above we get  
d/dx(tanhx) = (coshx^{2} - sinhx{2})/coshx{2} = 1 - tanhx^{2}  
Since tanhx =z  
d/dx(tanhx) = 1 - tanhx^{2}= 1 - z^{2}-(d)

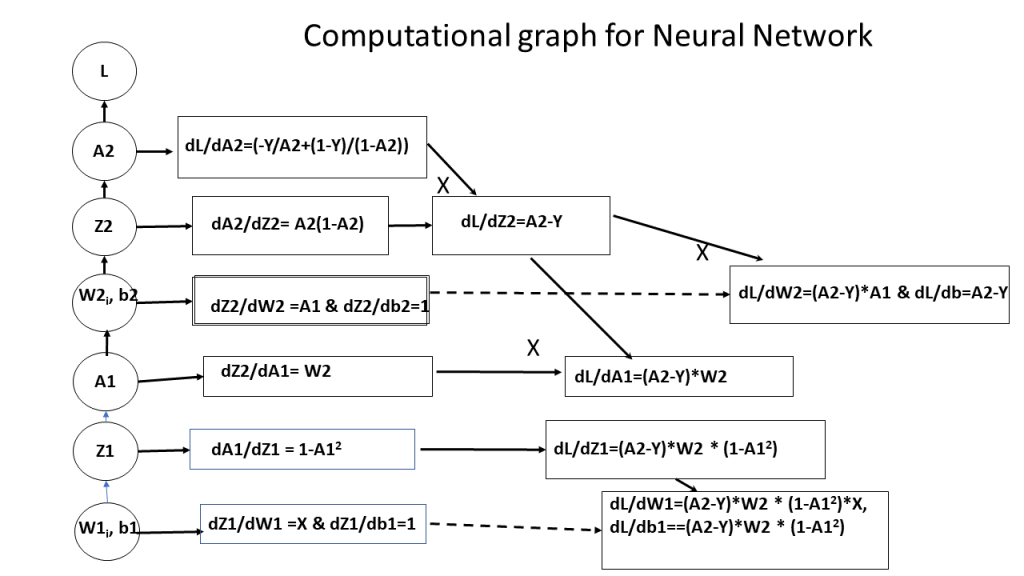
**II) Derivatives**  
L=-(Ylog(A2) + (1-Y)log(1-A2))  
dL/dA2 = -(Y/A2 + (1-Y)/(1-A2))  
Since A2 = sigmoid(Z2)therefore dA2/dZ2 = A2(1-A2)   
Z2 = W2A1 +b2  
dZ2/dW2 = A1  
dZ2/db2 = 1  
A1 = tanh(Z1)and dA1/dZ1 = 1 - A1^{2}  
Z1 = W1X + b1  
dZ1/dW1 = X  
dZ1/db1 = 1

**III) Back propagation**  
Using the derivatives from II) we can derive the following results using Chain Rule  
\partial L/\partial Z2 = \partial L/\partial A2 * \partial A2/\partial Z2   
= -(Y/A2 + (1-Y)/(1-A2)) * A2(1-A2) = A2 - Y  
\partial L/\partial W2 = \partial L/\partial A2 * \partial A2/\partial Z2 * \partial Z2/\partial W2  
= (A2-Y) *A1-(A)  
\partial L/\partial b2 = \partial L/\partial A2 * \partial A2/\partial Z2 * \partial Z2/\partial b2 = (A2-Y)-(B)

\partial L/\partial Z1 = \partial L/\partial A2 * \partial A2/\partial Z2 * \partial Z2/\partial A1 *\partial A1/\partial Z1 = (A2-Y) * W2 * (1-A1^{2})  
\partial L/\partial W1 = \partial L/\partial A2 * \partial A2/\partial Z2 * \partial Z2/\partial A1 *\partial A1/\partial Z1 *\partial Z1/\partial W1   
=(A2-Y) * W2 * (1-A1^{2}) * X-(C)  
\partial L/\partial b1 = \partial L/\partial A2 * \partial A2/\partial Z2 * \partial Z2/\partial A1 *dA1/dZ1 *dZ1/db1  
= (A2-Y) * W2 * (1-A1^{2})-(D)

**IV) Gradient Descent**  
The key computations in the backward cycle are  
W1 = W1-learningRate * \partial L/\partial W1– From (C)  
b1 = b1-learningRate * \partial L/\partial b1– From (D)  
W2 = W2-learningRate * \partial L/\partial W2– From (A)  
b2 = b2-learningRate * \partial L/\partial b2– From (B)

The weights and biases (W1,b1,W2,b2) are updated for each iteration thus minimizing the loss/cost.

These derivations can be represented pictorially using the computation graph (from the book Deep Learning by Ian Goodfellow, Joshua Bengio and Aaron Courville)  


**3. Manually create a data set that is not lineary separable**

Initially I create a dataset with 2 classes which has around 9 clusters that cannot be separated by linear boundaries. **Note**: *This data set is saved as data.csv and is used for the R and Octave Neural networks to see how they perform on the same dataset.*

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

colors=['black','gold']

cmap = matplotlib.colors.ListedColormap(colors)

X, y = make\_blobs(n\_samples = 400, n\_features = 2, centers = 7,

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

#Create 2 classes

y=y.reshape(400,1)

y = y % 2

#Plot the figure

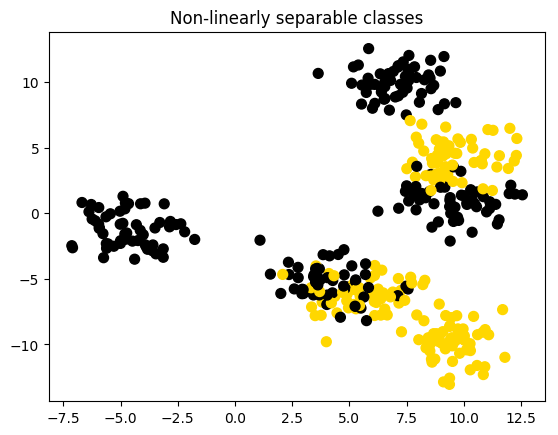
plt.figure()

plt.title('Non-linearly separable classes')

plt.scatter(X[:,0], X[:,1], c=y,

marker= 'o', s=50,cmap=cmap)

plt.savefig('fig1.png', bbox\_inches='tight')

****

**4. Logistic Regression**

On the above created dataset, classification with logistic regression is performed, and the decision boundary is plotted. It can be seen that logistic regression performs quite poorly

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("./DLfunctions.py") # Since import does not work in Rmd!!!

colors=['black','gold']

cmap = matplotlib.colors.ListedColormap(colors)

X, y = make\_blobs(n\_samples = 400, n\_features = 2, centers = 7,

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

#Create 2 classes

y=y.reshape(400,1)

y = y % 2

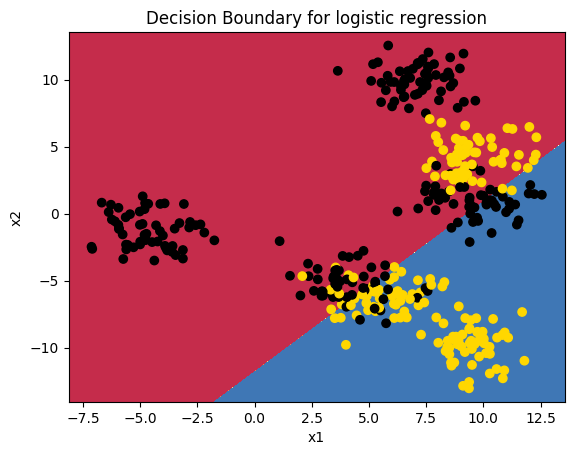
# Train the logistic regression classifier

clf = sklearn.linear\_model.LogisticRegressionCV();

clf.fit(X, y);

# Plot the decision boundary for logistic regression

plot\_decision\_boundary\_n(lambda x: clf.predict(x), X.T, y.T,"fig2.png")

****

**5. The 3 layer Neural Network in Python (vectorized)**

The vectorized implementation is included below. Note that in the case of Python a learning rate of 0.5 and 3 hidden units performs very well.

## Random data set with 9 clusters

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import sklearn.linear\_model

import pandas as pd

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

execfile("./DLfunctions.py") # Since import does not work in Rmd!!!

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

#Perform gradient descent

parameters,costs = computeNN(X2, Y2, numHidden = 4, learningRate=0.5, numIterations = 10000)

plot\_decision\_boundary(lambda x: predict(parameters, x.T), X2, Y2,str(4),str(0.5),"fig3.png")

## Cost after iteration 0: 0.692669

## Cost after iteration 1000: 0.246650

## Cost after iteration 2000: 0.227801

## Cost after iteration 3000: 0.226809

## Cost after iteration 4000: 0.226518

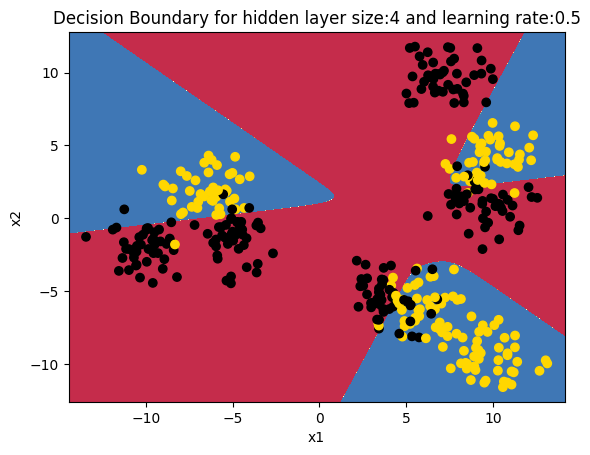
## Cost after iteration 5000: 0.226331

## Cost after iteration 6000: 0.226194

## Cost after iteration 7000: 0.226085

## Cost after iteration 8000: 0.225994

## Cost after iteration 9000: 0.225915

****

**6. The 3 layer Neural Network in R (vectorized)**

For this the dataset created by Python is saved  to see how R performs on the same dataset. The vectorized implementation of a Neural Network was just a little more interesting as R does not have a similar package like ‘numpy’. While numpy handles broadcasting implicitly, in R I had to use the ‘sweep’ command to broadcast. The implementaion is included below. Note that since the initialization with random weights is slightly different, R performs best with a learning rate of 0.1 and with 6 hidden units

source("DLfunctions2\_1.R")

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

x <- z[,1:2]

y <- z[,3]

x1 <- t(x)

y1 <- t(y)

#Perform gradient descent

nn <-computeNN(x1, y1, 6, learningRate=0.1,numIterations=10000) # Good

## [1] 0.7075341

## [1] 0.2606695

## [1] 0.2198039

## [1] 0.2091238

## [1] 0.211146

## [1] 0.2108461

## [1] 0.2105351

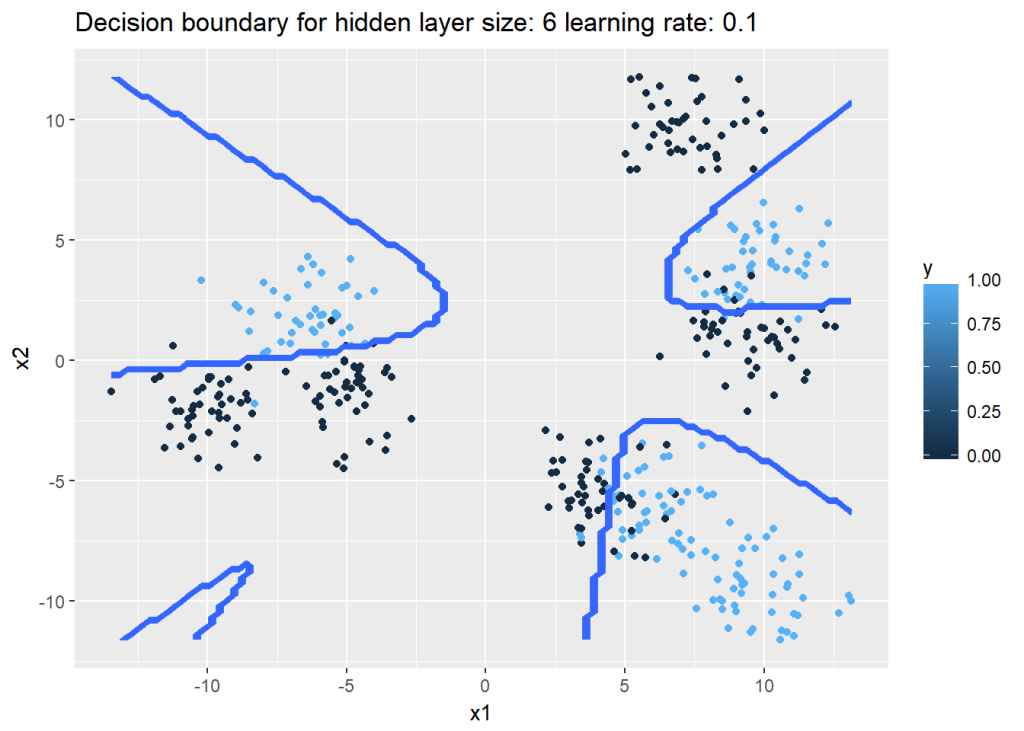
## [1] 0.210211

## [1] 0.2099104

## [1] 0.2096437

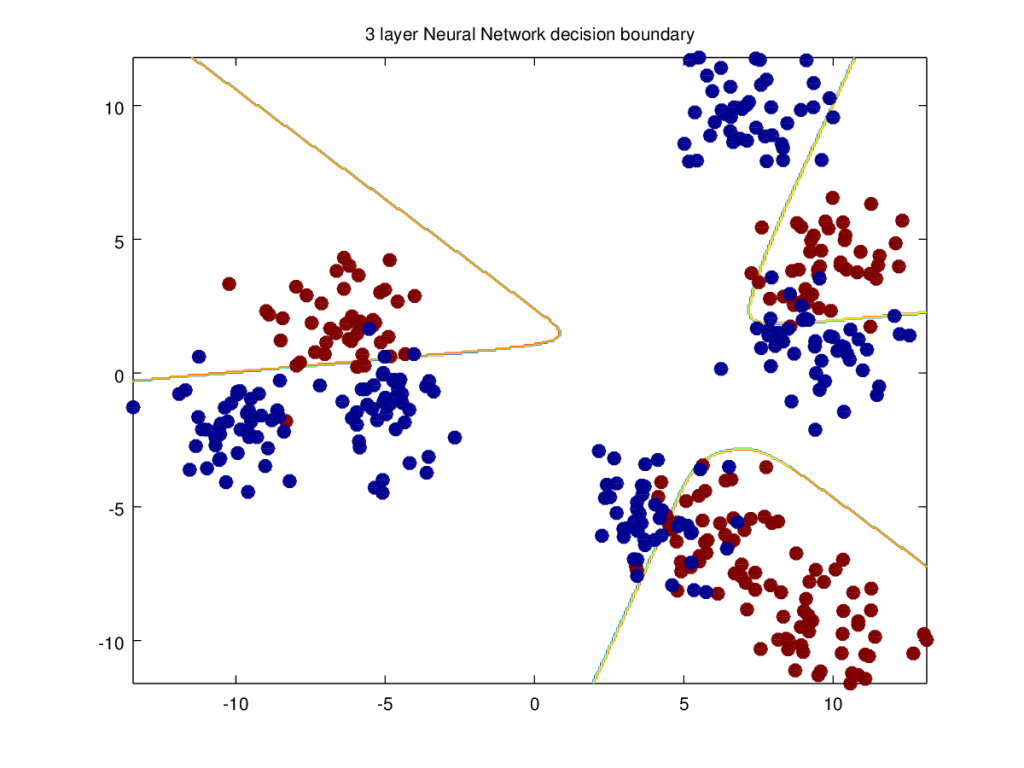
## [1] 0.209409

plotDecisionBoundary(z,nn,6,0.1)



**7.  The 3 layer Neural Network in Octave (vectorized)**

This uses the same dataset that was generated using Python code.  
source("DL-function2.m")  
data=csvread("data.csv");  
X=data(:,1:2);  
Y=data(:,3);  
# Make sure that the model parameters are correct. Take the transpose of X & Y  
#Perform gradient descent  
[W1,b1,W2,b2,costs]= computeNN(X', Y',4, learningRate=0.5, numIterations = 10000);

****

**8a. Performance  for different learning rates (Python)**

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import sklearn.linear\_model

import pandas as pd

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

execfile("./DLfunctions.py") # Since import does not work in Rmd!!!

# Create data

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

# Create a list of learning rates

learningRate=[0.5,1.2,3.0]

df=pd.DataFrame()

#Compute costs for each learning rate

for lr in learningRate:

parameters,costs = computeNN(X2, Y2, numHidden = 4, learningRate=lr, numIterations = 10000)

print(costs)

df1=pd.DataFrame(costs)

df=pd.concat([df,df1],axis=1)

#Set the iterations

iterations=[0,1000,2000,3000,4000,5000,6000,7000,8000,9000]

#Create data frame

#Set index

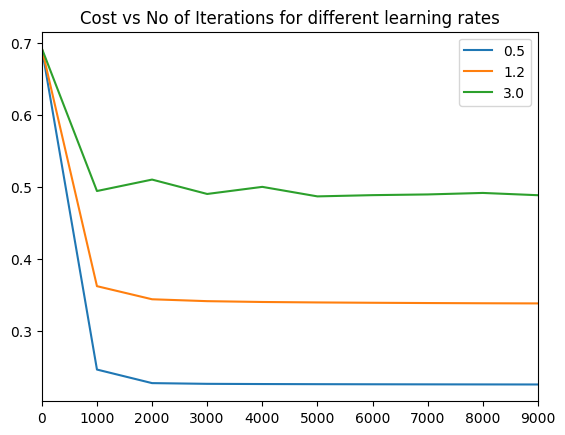
df1=df.set\_index([iterations])

df1.columns=[0.5,1.2,3.0]

fig=df1.plot()

fig=plt.title("Cost vs No of Iterations for different learning rates")

plt.savefig('fig4.png', bbox\_inches='tight')

****

**8b. Performance  for different hidden units (Python)**

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import sklearn.linear\_model

import pandas as pd

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

execfile("./DLfunctions.py") # Since import does not work in Rmd!!!

#Create data set

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

# Make a list of hidden unis

numHidden=[3,5,7]

df=pd.DataFrame()

#Compute costs for different hidden units

for numHid in numHidden:

parameters,costs = computeNN(X2, Y2, numHidden = numHid, learningRate=1.2, numIterations = 10000)

print(costs)

df1=pd.DataFrame(costs)

df=pd.concat([df,df1],axis=1)

#Set the iterations

iterations=[0,1000,2000,3000,4000,5000,6000,7000,8000,9000]

#Set index

df1=df.set\_index([iterations])

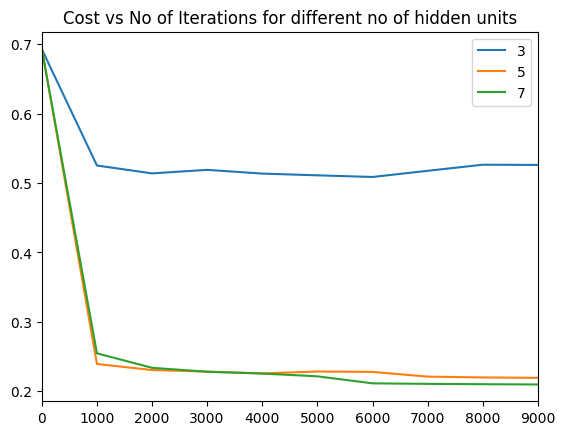
df1.columns=[3,5,7]

#Plot

fig=df1.plot()

fig=plt.title("Cost vs No of Iterations for different no of hidden units")

plt.savefig('fig5.png', bbox\_inches='tight')

****

**9a. Performance  for different learning rates (R)**

source("DLfunctions2\_1.R")

# Read data

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

x <- z[,1:2]

y <- z[,3]

x1 <- t(x)

y1 <- t(y)

#Loop through learning rates and compute costs

learningRate <-c(0.1,1.2,3.0)

df <- NULL

for(i in seq\_along(learningRate)){

nn <- computeNN(x1, y1, 6, learningRate=learningRate[i],numIterations=10000)

cost <- nn$costs

df <- cbind(df,cost)

}

#Create dataframe

df <- data.frame(df)

iterations=seq(0,10000,by=1000)

df <- cbind(iterations,df)

names(df) <- c("iterations","0.5","1.2","3.0")

library(reshape2)

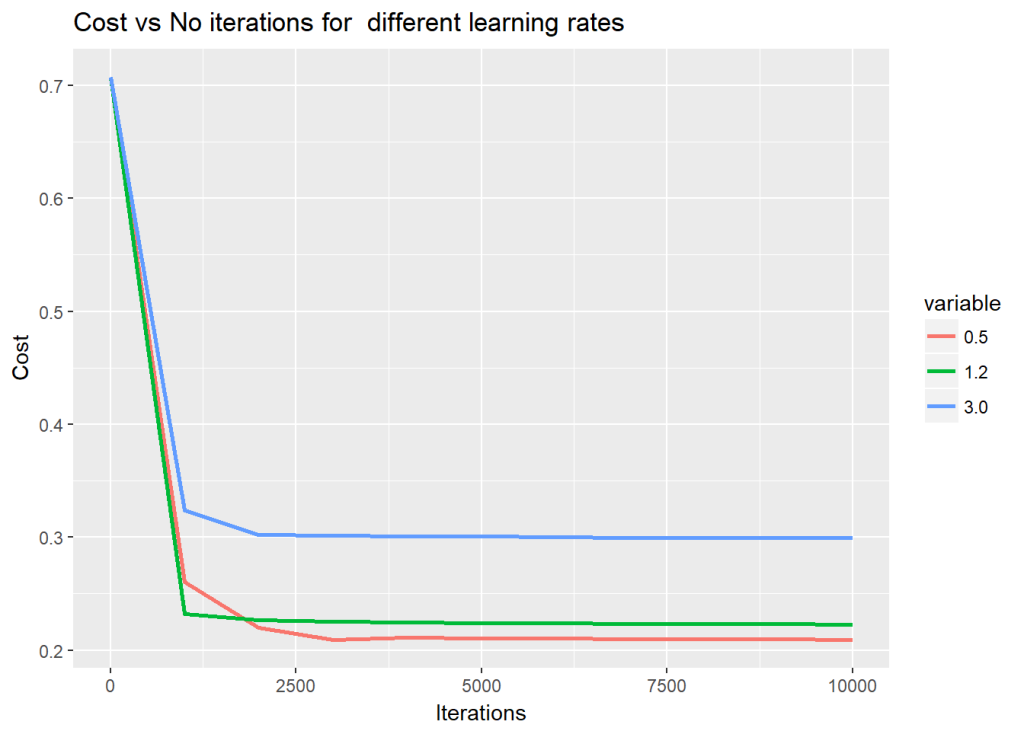
df1 <- melt(df,id="iterations") # Melt the data

#Plot

ggplot(df1) + geom\_line(aes(x=iterations,y=value,colour=variable),size=1) +

xlab("Iterations") +

ylab('Cost') + ggtitle("Cost vs No iterations for different learning rates")



**9b. Performance  for different hidden units (R)**

source("DLfunctions2\_1.R")

# Loop through Num hidden units

numHidden <-c(4,6,9)

df <- NULL

for(i in seq\_along(numHidden)){

nn <- computeNN(x1, y1, numHidden[i], learningRate=0.1,numIterations=10000)

cost <- nn$costs

df <- cbind(df,cost)

}

df <- data.frame(df)

iterations=seq(0,10000,by=1000)

df <- cbind(iterations,df)

names(df) <- c("iterations","4","6","9")

library(reshape2)

# Melt

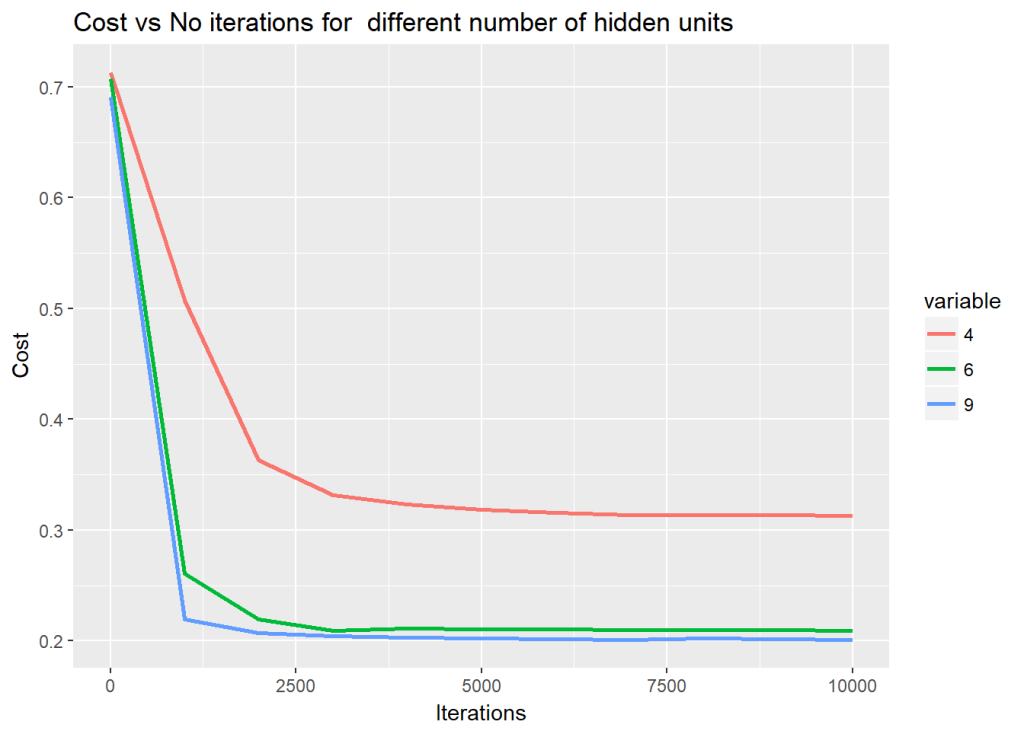
df1 <- melt(df,id="iterations")

# Plot

ggplot(df1) + geom\_line(aes(x=iterations,y=value,colour=variable),size=1) +

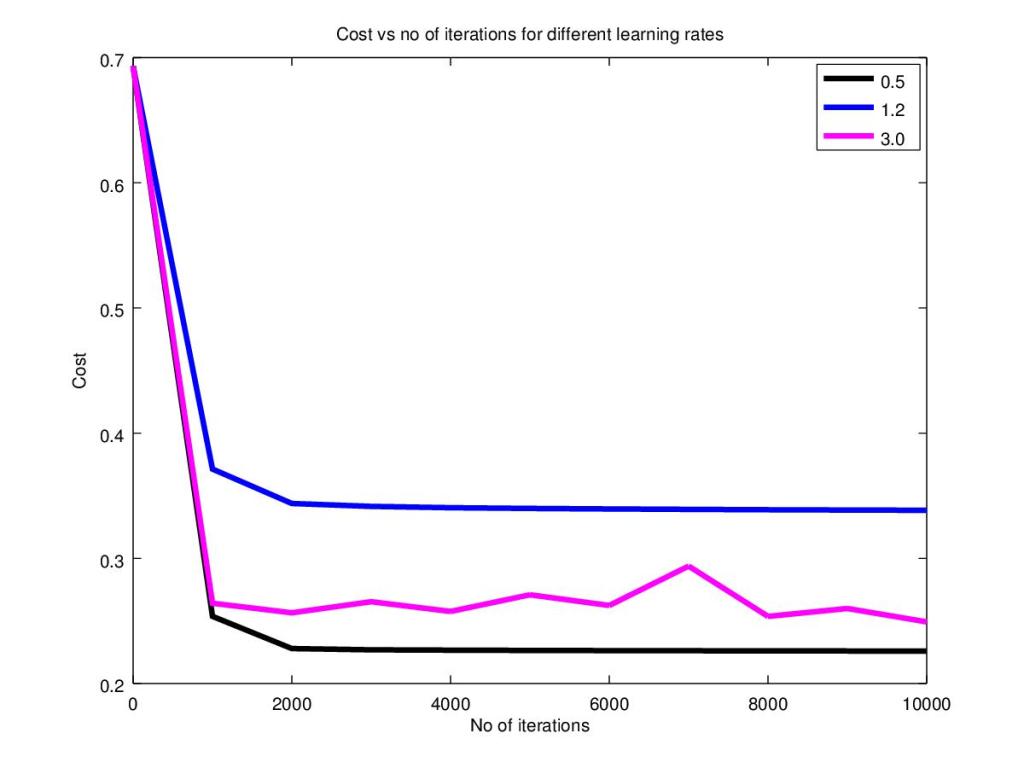
xlab("Iterations") +

ylab('Cost') + ggtitle("Cost vs No iterations for different number of hidden units")

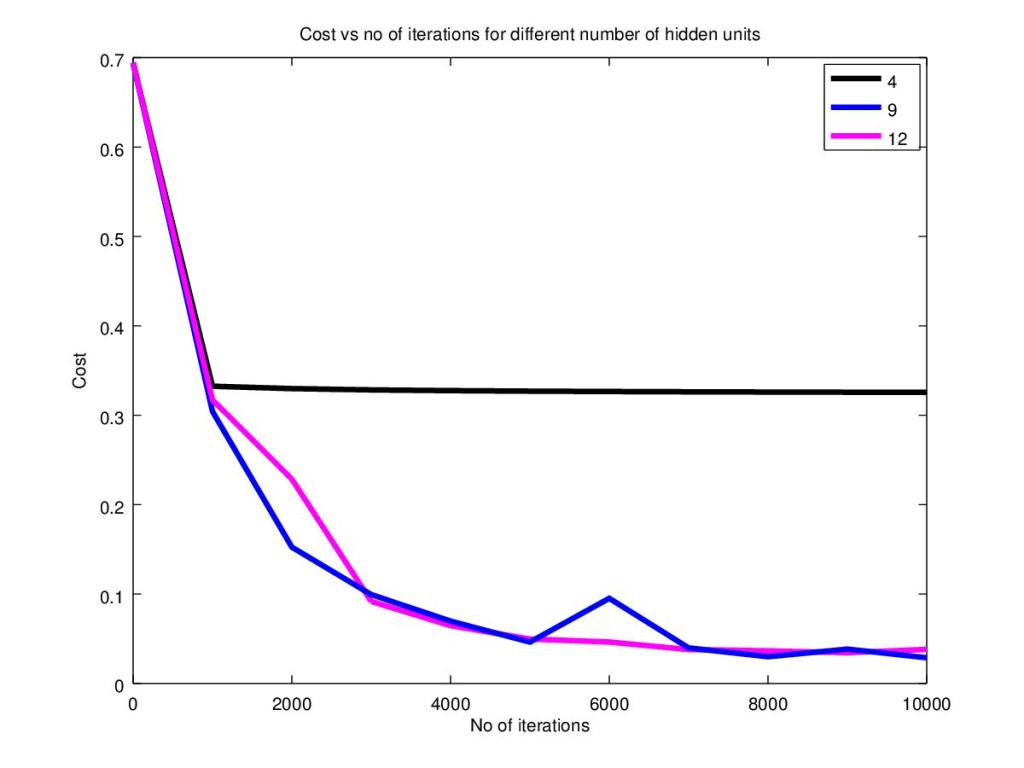


**10a. Performance of the Neural Network for different learning rates (Octave)**

source("DL-function2.m")  
plotLRCostVsIterations()  
print -djph figa.jpg



**10b. Performance of the Neural Network for different number of hidden units (Octave)**

source("DL-function2.m")  
plotHiddenCostVsIterations()  
print -djph figa.jpg  


**11. Turning the heat on the Neural Network**

In this 2nd part I create a a central region of positives and and the outside region as negatives. The points are generated using the equation of a circle (x – a)^{2} + (y -b) ^{2} = R^{2} . How does the 3 layer Neural Network perform on this?  Here’s a look! **Note**: *The same dataset is also used for R and Octave Neural Network constructions*

**12. Manually creating a circular central region**

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

colors=['black','gold']

cmap = matplotlib.colors.ListedColormap(colors)

x1=np.random.uniform(0,10,800).reshape(800,1)

x2=np.random.uniform(0,10,800).reshape(800,1)

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

X.shape

# Create (x-a)^2 + (y-b)^2 = R^2

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

a=(np.power(X[:,0]-5,2) + np.power(X[:,1]-5,2) <= 6).ravel()

Y=a.reshape(800,1)

cmap = matplotlib.colors.ListedColormap(colors)

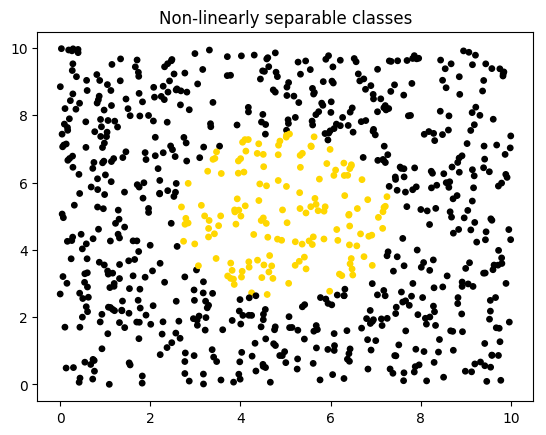
plt.figure()

plt.title('Non-linearly separable classes')

plt.scatter(X[:,0], X[:,1], c=Y,

marker= 'o', s=15,cmap=cmap)

plt.savefig('fig6.png', bbox\_inches='tight')

****

**13a. Decision boundary with hidden units=4 and learning rate = 2.2 (Python)**

With the above hyper parameters the decision boundary is triangular

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.colors

import sklearn.linear\_model

execfile("./DLfunctions.py")

x1=np.random.uniform(0,10,800).reshape(800,1)

x2=np.random.uniform(0,10,800).reshape(800,1)

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

a=(np.power(X[:,0]-5,2) + np.power(X[:,1]-5,2) <= 6).ravel()

Y=a.reshape(800,1)

X2=X.T

Y2=Y.T

parameters,costs = computeNN(X2, Y2, numHidden = 4, learningRate=2.2, numIterations = 10000)

plot\_decision\_boundary(lambda x: predict(parameters, x.T), X2, Y2,str(4),str(2.2),"fig7.png")

## Cost after iteration 0: 0.692836

## Cost after iteration 1000: 0.331052

## Cost after iteration 2000: 0.326428

## Cost after iteration 3000: 0.474887

## Cost after iteration 4000: 0.247989

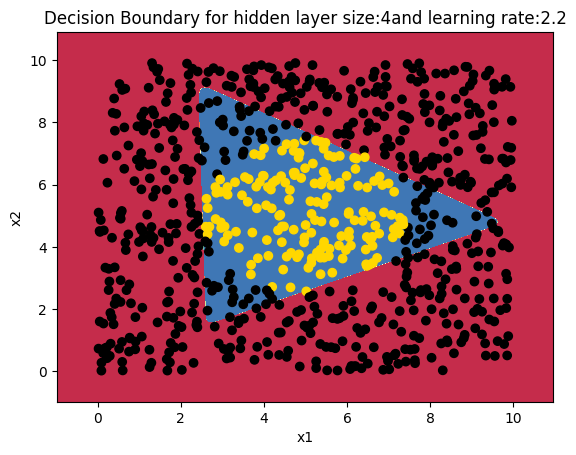
## Cost after iteration 5000: 0.218009

## Cost after iteration 6000: 0.201034

## Cost after iteration 7000: 0.197030

## Cost after iteration 8000: 0.193507

## Cost after iteration 9000: 0.191949

****

**13b. Decision boundary with hidden units=12 and learning rate = 2.2 (Python)**

With the above hyper parameters the decision boundary is triangular

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.colors

import sklearn.linear\_model

execfile("./DLfunctions.py")

x1=np.random.uniform(0,10,800).reshape(800,1)

x2=np.random.uniform(0,10,800).reshape(800,1)

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

a=(np.power(X[:,0]-5,2) + np.power(X[:,1]-5,2) <= 6).ravel()

Y=a.reshape(800,1)

X2=X.T

Y2=Y.T

parameters,costs = computeNN(X2, Y2, numHidden = 12, learningRate=2.2, numIterations = 10000)

plot\_decision\_boundary(lambda x: predict(parameters, x.T), X2, Y2,str(12),str(2.2),"fig8.png")

## Cost after iteration 0: 0.693291

## Cost after iteration 1000: 0.383318

## Cost after iteration 2000: 0.298807

## Cost after iteration 3000: 0.251735

## Cost after iteration 4000: 0.177843

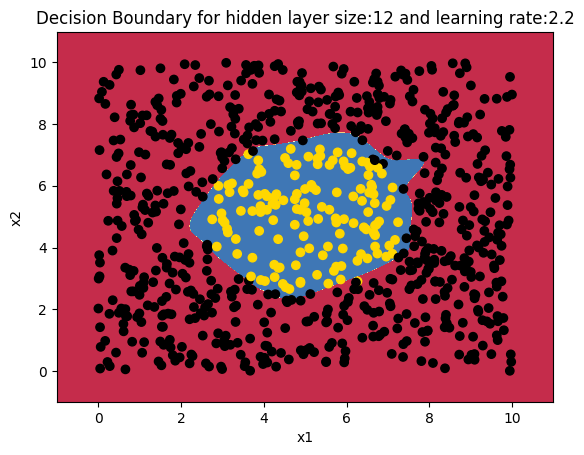
## Cost after iteration 5000: 0.130414

## Cost after iteration 6000: 0.152400

## Cost after iteration 7000: 0.065359

## Cost after iteration 8000: 0.050921

## Cost after iteration 9000: 0.039719

****

**14a. Decision boundary with hidden units=9 and learning rate = 0.5 (R)**

When the number of hidden units is 6 and the learning rate is 0,1, is also a triangular shape in R

source("DLfunctions2\_1.R")

z <- as.matrix(read.csv("data1.csv",header=FALSE)) # N

x <- z[,1:2]

y <- z[,3]

x1 <- t(x)

y1 <- t(y)

nn <-computeNN(x1, y1, 9, learningRate=0.5,numIterations=10000) # Triangular

## [1] 0.8398838

## [1] 0.3303621

## [1] 0.3127731

## [1] 0.3012791

## [1] 0.3305543

## [1] 0.3303964

## [1] 0.2334615

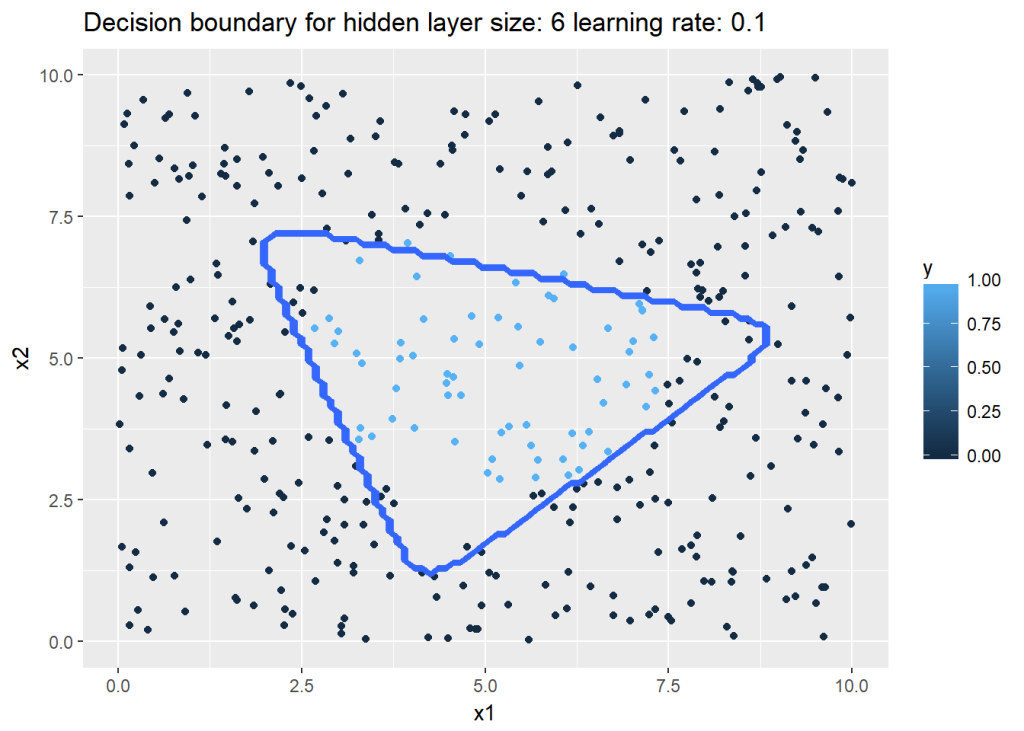
## [1] 0.1920771

## [1] 0.2341225

## [1] 0.2188118

## [1] 0.2082687

plotDecisionBoundary(z,nn,6,0.1)



**14b. Decision boundary with hidden units=8 and learning rate = 0.1 (R)**

source("DLfunctions2\_1.R")

z <- as.matrix(read.csv("data1.csv",header=FALSE)) # N

x <- z[,1:2]

y <- z[,3]

x1 <- t(x)

y1 <- t(y)

nn <-computeNN(x1, y1, 8, learningRate=0.1,numIterations=10000) # Hemisphere

## [1] 0.7273279

## [1] 0.3169335

## [1] 0.2378464

## [1] 0.1688635

## [1] 0.1368466

## [1] 0.120664

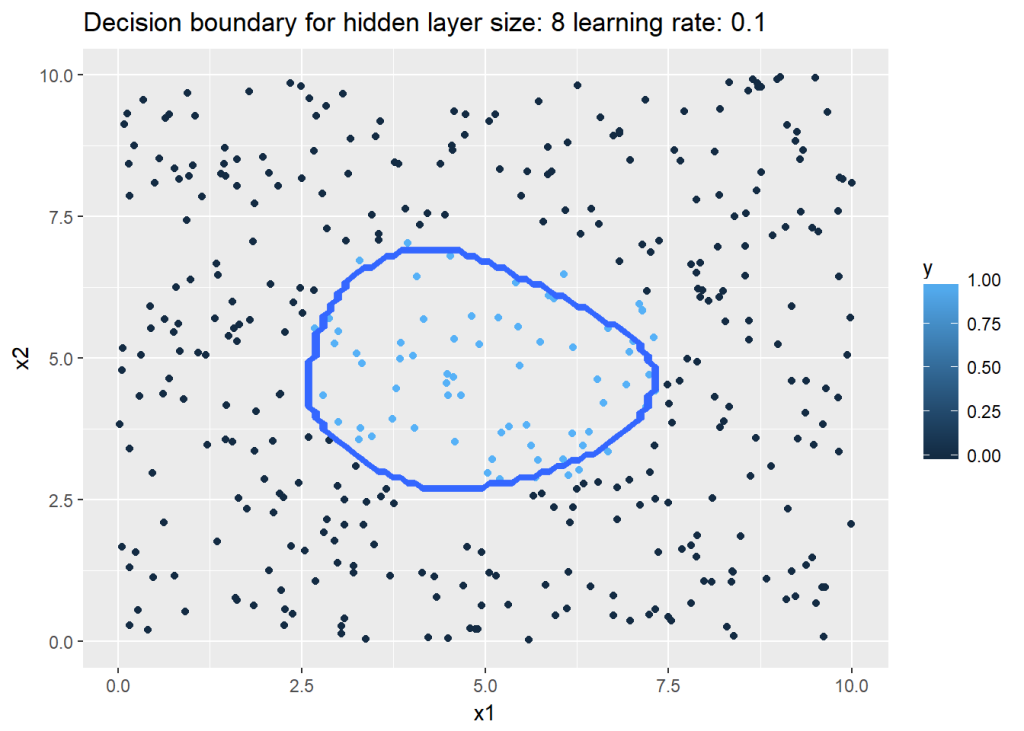
## [1] 0.111211

## [1] 0.1043362

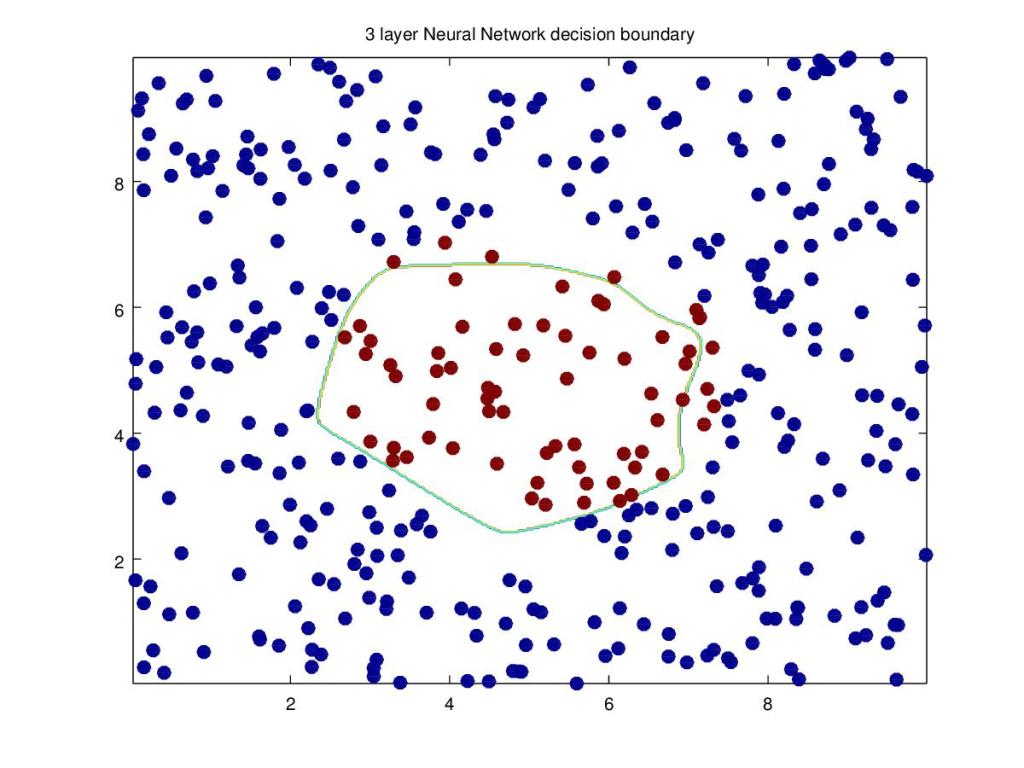
## [1] 0.09800573

## [1] 0.09126161

## [1] 0.0840379

plotDecisionBoundary(z,nn,8,0.1)

**15a. Decision boundary with hidden units=12 and learning rate = 1.5 (Octave)**

source("DL-function2.m")  
data=csvread("data1.csv");  
X=data(:,1:2);  
Y=data(:,3);  
# Make sure that the model parameters are correct. Take the transpose of X & Y  
[W1,b1,W2,b2,costs]= computeNN(X', Y',12, learningRate=1.5, numIterations = 10000);  
plotDecisionBoundary(data, W1,b1,W2,b2)  
print -djpg fige.jpg  


**DL-Function2.1.m**

|  |
| --- |
| 1; |
|  | # Define sigmoid function |
|  | function a = sigmoid(z) |
|  | a = 1 ./ (1+ exp(-z)); |
|  | end |
|  |  |
|  | # Compute the loss |
|  | function loss=computeLoss(numtraining,Y,A) |
|  | loss = -1/numtraining \* sum((Y .\* log(A)) + (1-Y) .\* log(1-A)); |
|  | end |
|  |  |
|  | # Compute the model shape given the dataset |
|  | function [n\_x,m,n\_h,n\_y] = getModelShape(X,Y) |
|  | m= size(X)(2); |
|  | n\_x=size(X)(1); |
|  | n\_h=4; |
|  | n\_y=size(Y)(1); |
|  | end |
|  |  |
|  | function [W1,b1,W2,b2] = modelInit(n\_x,n\_h,n\_y) |
|  | rand ("seed", 2); |
|  | W1=rand(n\_h,n\_x)\*0.01; # Set the initial values to a small number |
|  | b1=zeros(n\_h,1); |
|  | W2=rand(n\_y,n\_h)\*0.01; |
|  | b2=zeros(n\_y,1); |
|  |  |
|  | end |
|  |  |
|  | # Compute the forward propoagation through the neural network |
|  | # Input : Features |
|  | # Weight and bias matrices and vectors |
|  | # Returns : The Activation of 2nd layer |
|  | # : Output and activation of layer 1 & 2 |
|  | function [Z1,A1,Z2,A2]= forwardPropagation(X,W1,b1,W2,b2) |
|  | # Get the parameters |
|  |  |
|  | # Determine the number of training samples |
|  | m=size(X)(2); |
|  | # Compute Z1 of the input layer |
|  | # Octave also handles broadcasting like Python!! |
|  | Z1=W1 \* X +b1; |
|  | # Compute the output A1 with the tanh activation function. The tanh activation function |
|  | # performs better than the sigmoid function |
|  | A1=tanh(Z1); |
|  |  |
|  | # Compute Z2 of the input layer |
|  | Z2=W2 \* A1+b2; |
|  | # Compute the output A1 with the tanh activation function. The tanh activation function |
|  | # performs better than the sigmoid function |
|  | A2=sigmoid(Z2); |
|  |  |
|  | end |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of 2nd layer |
|  | # : Output from data |
|  | # Output: cost |
|  | function [cost] = computeCost(A,Y) |
|  | numTraining= size(Y)(2); |
|  | # Element wise multiply for logprobs |
|  | cost = -1/numTraining \* sum((Y .\* log(A)) + (1-Y) .\* log(1-A)); |
|  | end |
|  |  |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : Neural Network parameters - weights and biases |
|  | # # Z and Activations of 2 layers |
|  | # # Input features |
|  | # # Output values Y |
|  | # Returns: Gradients |
|  | function [dW1,db1,dW2,db2]= backPropagation(W1,W2,A1,A2, X, Y) |
|  | numTraining=size(X)(2); |
|  |  |
|  | dZ2 = A2 - Y; |
|  | dW2 = 1/numTraining \* dZ2 \* A1'; |
|  | db2 = 1/numTraining \* sum(dZ2); |
|  |  |
|  | dZ1 = W2' \* dZ2 .\* (1 - power(A1, 2)); |
|  | dW1 = 1/numTraining \* dZ1 \* X'; |
|  | # Note the '2' in the next statement indicates that a row sum has to done , 2nd dimension |
|  | db1 = 1/numTraining \* sum(dZ1,2); |
|  |  |
|  | end |
|  |  |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | #output : Updated weights after 1 iteration |
|  | function [W1,b1,W2,b2]= gradientDescent(W1,b1,W2,b2, dW1,db1,dW2,db2, learningRate) |
|  | W1 = W1-learningRate\*dW1; |
|  | b1 = b1-learningRate\*db1; |
|  | W2 = W2-learningRate\*dW2; |
|  | b2 = b2-learningRate\*db2; |
|  | end |
|  |  |
|  |  |
|  | # Compute the Neural Network by minimizing the cost |
|  | # Input : Input data X, |
|  | # Output Y |
|  | # No of hidden units in hidden layer |
|  | # No of iterations |
|  | # Returns Updated weight and bias vectors of the neural network |
|  | function [W1,b1,W2,b2,costs]= computeNN(X, Y,numHidden, learningRate, numIterations = 10000) |
|  |  |
|  | [numFeats,numTraining,n\_h,numOutput] = getModelShape(X, Y); |
|  |  |
|  | costs=[]; |
|  |  |
|  | [W1,b1,W2,b2] = modelInit(numFeats,numHidden,numOutput) ; |
|  | #W1 =[-0.00416758, -0.00056267; -0.02136196, 0.01640271; -0.01793436, -0.00841747; 0.00502881 -0.01245288]; |
|  | #W2=[-0.01057952, -0.00909008, 0.00551454, 0.02292208]; |
|  | #b1=[0;0;0;0]; |
|  | #b2=[0]; |
|  | # Perform gradient descent |
|  | for i =0:numIterations |
|  | # Evaluate forward prop to compute activation at output layer |
|  | [Z1,A1,Z2,A2] = forwardPropagation(X, W1,b1,W2,b2); |
|  | # Compute cost from Activation at output and Y |
|  | cost = computeCost(A2, Y); |
|  | # Perform backprop to compute gradients |
|  | [dW1,db1,dW2,db2] = backPropagation(W1,W2,A1,A2, X, Y); |
|  | # Use gradients to update the weights for each iteration. |
|  |  |
|  | [W1,b1,W2,b2] = gradientDescent(W1,b1,W2,b2, dW1,db1,dW2,db2,learningRate); |
|  | # Print the cost every 1000 iterations |
|  | if ( mod(i,1000) == 0) |
|  | costs =[costs cost]; |
|  | #disp ("Cost after iteration"), disp(i),disp(cost); |
|  | endif |
|  | endfor |
|  | end |
|  |  |
|  |  |
|  |  |
|  |  |
|  | # Compute the predicted value for a given input |
|  | # Input : Neural Network parameters |
|  | # : Input data |
|  | function [predictions]= predict(W1,b1,W2,b2, X) |
|  | [Z1,A1,Z2,A2] = forwardPropagation(X, W1,b1,W2,b2); |
|  | predictions = (A2>0.5); |
|  | end |
|  |  |
|  | # Plot the decision boundary |
|  | function plotDecisionBoundary(data,W1,b1,W2,b2) |
|  | %Plot a non-linear decision boundary learned by the SVM |
|  | colormap ("default"); |
|  |  |
|  | % 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(W1,b1,W2,b2,gridPoints'); |
|  | 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); |
|  | title ({"3 layer Neural Network decision boundary"}); |
|  | hold off; |
|  |  |
|  | end |
|  |  |
|  | # Plot the cost vs iterations |
|  | function plotLRCostVsIterations() |
|  | data=csvread("data.csv"); |
|  |  |
|  | X=data(:,1:2); |
|  | Y=data(:,3); |
|  | lr=[0.5,1.2,3] |
|  | col='kbm' |
|  | for i=1:3 |
|  | [W1,b1,W2,b2,costs]= computeNN(X', Y',4, learningRate=lr(i), numIterations = 10000); |
|  | iterations = 1000\*[0:10]; |
|  | hold on; |
|  | plot(iterations,costs,color=col(i),"linewidth", 3); |
|  | hold off; |
|  | title ("Cost vs no of iterations for different learning rates"); |
|  | xlabel("No of iterations") |
|  | ylabel("Cost") |
|  | legend('0.5','1.2','3.0') |
|  | endfor |
|  | end |
|  |  |
|  | # Plot the cost vs number of hidden units |
|  | function plotHiddenCostVsIterations() |
|  | data=csvread("data1.csv"); |
|  |  |
|  | X=data(:,1:2); |
|  | Y=data(:,3); |
|  | hidden=[4,9,12] |
|  | col='kbm' |
|  | for i=1:3 |
|  | [W1,b1,W2,b2,costs]= computeNN(X', Y',hidden(i), learningRate=1.5, numIterations = 10000); |
|  | iterations = 1000\*[0:10]; |
|  | hold on; |
|  | plot(iterations,costs,color=col(i),"linewidth", 3); |
|  | hold off; |
|  | title ("Cost vs no of iterations for different number of hidden units"); |
|  | xlabel("No of iterations") |
|  | ylabel("Cost") |
|  | legend('4','9','12') |
|  | endfor |
|  | end |

**DlFunctions.py**

|  |
| --- |
| import numpy as np |
|  | import matplotlib |
|  | import matplotlib.pyplot as plt |
|  |  |
|  | # Conmpute the sigmoid of a vector |
|  | def sigmoid(z): |
|  | a=1/(1+np.exp(-z)) |
|  | return a |
|  |  |
|  | # Compute the model shape given the dataset |
|  | def getModelShape(X,Y): |
|  | numTraining= X.shape[1] # No of training examples |
|  | numFeats=X.shape[0] # No of input features |
|  | numHidden=4 # No of units in hidden layer |
|  | numOutput=Y.shape[0] # No of output units |
|  | # Create a dcitionary of values |
|  | modelParams={"numTraining":numTraining,"numFeats":numFeats,"numHidden":numHidden,"numOutput":numOutput} |
|  | return(modelParams) |
|  |  |
|  |  |
|  | # 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(2) |
|  | 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) |
|  |  |
|  | # Compute the forward propoagation through the neural network |
|  | # Input : Features |
|  | # Weight and bias matrices and vectors |
|  | # Returns : The Activation of 2nd layer |
|  | # : Output and activation of layer 1 & 2 |
|  |  |
|  | def forwardPropagation(X,nnParameters): |
|  | # Get the parameters |
|  | W1=nnParameters["W1"] |
|  | b1=nnParameters["b1"] |
|  | W2=nnParameters["W2"] |
|  | b2=nnParameters["b2"] |
|  |  |
|  | # Compute Z1 of the input layer |
|  | Z1=np.dot(W1,X)+b1 |
|  | # Compute the output A1 with the tanh activation function. The tanh activation function |
|  | # performs better than the sigmoid function |
|  | A1=np.tanh(Z1) |
|  |  |
|  | # Compute Z2 of the 2nd layer |
|  | Z2=np.dot(W2,A1)+b2 |
|  | # Compute the output A1 with the tanh activation function. The tanh activation function |
|  | # performs better than the sigmoid function |
|  | A2=sigmoid(Z2) |
|  | cache={'Z1':Z1,'A1':A1,'Z2':Z2,'A2':A2} |
|  | return A2,cache |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of 2nd layer |
|  | # : Output from data |
|  | # Output: cost |
|  | def computeCost(A2,Y): |
|  | m= float(Y.shape[1]) |
|  | # Element wise multiply for logprobs |
|  | cost=-1/m \*np.sum(Y\*np.log(A2) + (1-Y)\*(np.log(1-A2))) |
|  | cost = np.squeeze(cost) |
|  | return cost |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : Neural Network parameters - weights and biases |
|  | # # Z and Activations of 2 layers |
|  | # # Input features |
|  | # # Output values Y |
|  | # Returns: Gradients |
|  | def backPropagation(nnParameters, cache, X, Y): |
|  | numtraining=float(X.shape[1]) |
|  | # Get parameters |
|  | W1=nnParameters["W1"] |
|  | W2=nnParameters["W2"] |
|  |  |
|  | #Get the NN cache |
|  | A1=cache["A1"] |
|  | A2=cache["A2"] |
|  |  |
|  | # Compute gradients |
|  | dZ2 = A2 - Y |
|  | dW2 = 1/numtraining \*np.dot(dZ2,A1.T) |
|  | db2 = 1/numtraining \*np.sum(dZ2,axis=1,keepdims=True) |
|  | dZ1 = np.multiply(np.dot(W2.T,dZ2), (1 - np.power(A1, 2))) |
|  | dW1 = 1/numtraining\* np.dot(dZ1,X.T) |
|  | db1 = 1/numtraining \*np.sum(dZ1,axis=1,keepdims=True) |
|  |  |
|  | # Create a dictionary |
|  | gradients = {"dW1": dW1, |
|  | "db1": db1, |
|  | "dW2": dW2, |
|  | "db2": db2} |
|  | return gradients |
|  |  |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | #output : Updated weights after 1 iteration |
|  | def gradientDescent(nnParameters, gradients, learningRate): |
|  | W1 = nnParameters['W1'] |
|  | b1 = nnParameters['b1'] |
|  | W2 = nnParameters['W2'] |
|  | b2 = nnParameters['b2'] |
|  | dW1 = gradients["dW1"] |
|  | db1 = gradients["db1"] |
|  | dW2 = gradients["dW2"] |
|  | db2 = gradients["db2"] |
|  | W1 = W1-learningRate\*dW1 |
|  | b1 = b1-learningRate\*db1 |
|  | W2 = W2-learningRate\*dW2 |
|  | b2 = b2-learningRate\*db2 |
|  | # Update the Neural Network parametrs |
|  | updatedNNParameters = {"W1": W1, |
|  | "b1": b1, |
|  | "W2": W2, |
|  | "b2": b2} |
|  |  |
|  | return updatedNNParameters |
|  |  |
|  | # Compute the Neural Network by minimizing the cost |
|  | # Input : Input data X, |
|  | # Output Y |
|  | # No of hidden units in hidden layer |
|  | # No of iterations |
|  | # Returns Updated weight and bias vectors of the neural network |
|  | def computeNN(X, Y, numHidden, learningRate, numIterations = 10000): |
|  | np.random.seed(3) |
|  | modelParams = getModelShape(X, Y) |
|  | numFeats=modelParams['numFeats'] |
|  | numOutput=modelParams['numOutput'] |
|  |  |
|  | costs=[] |
|  |  |
|  | nnParameters = initializeModel(numFeats,numHidden,numOutput) |
|  | W1 = nnParameters['W1'] |
|  | b1 = nnParameters['b1'] |
|  | W2 = nnParameters['W2'] |
|  | b2 = nnParameters['b2'] |
|  | # Perform gradient descent |
|  | for i in range(0, numIterations): |
|  | # Evaluate forward prop to compute activation at output layer |
|  | A2, cache = forwardPropagation(X, nnParameters) |
|  | # Compute cost from Activation at output and Y |
|  | cost = computeCost(A2, Y) |
|  | # Perform backprop to compute gradients |
|  | gradients = backPropagation(nnParameters, cache, X, Y) |
|  | # Use gradients to update the weights for each iteration. |
|  | nnParameters = gradientDescent(nnParameters, gradients,learningRate) |
|  | # Print the cost every 1000 iterations |
|  | if i % 1000 == 0: |
|  | costs.append(cost) |
|  | print ("Cost after iteration %i: %f" %(i, cost)) |
|  | return nnParameters,costs |
|  |  |
|  | # Compute the predicted value for a given input |
|  | # Input : Neural Network parameters |
|  | # : Input data |
|  | def predict(nnParameters, X): |
|  | A2, cache = forwardPropagation(X, nnParameters) |
|  | predictions = (A2>0.5) |
|  | return predictions |
|  |  |
|  | # Plot a decision boundary |
|  | # Input : Input Model, |
|  | # X |
|  | # Y |
|  | # Fig to be saved as |
|  | # Returns Null |
|  | def plot\_decision\_boundary\_n(model, X, y,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','gold'] |
|  | cmap = matplotlib.colors.ListedColormap(colors) |
|  | h = 0.01 |
|  | # Generate a grid of points with distance h between them |
|  | xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h)) |
|  | # Predict the function value for the whole grid |
|  | Z = model(np.c\_[xx.ravel(), yy.ravel()]) |
|  | Z = Z.reshape(xx.shape) |
|  | # Plot the contour and training examples |
|  | plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral) |
|  | plt.ylabel('x2') |
|  | plt.xlabel('x1') |
|  | plt.scatter(X[0, :], X[1, :], c=y, cmap=cmap) |
|  | plt.title("Decision Boundary for logistic regression") |
|  | plt.savefig(fig, bbox\_inches='tight') |
|  |  |
|  | # 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,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 |
|  | colors=['black','gold'] |
|  | cmap = matplotlib.colors.ListedColormap(colors) |
|  | h = 0.01 |
|  | # Generate a grid of points with distance h between them |
|  | xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h)) |
|  | # Predict the function value for the whole grid |
|  | Z = model(np.c\_[xx.ravel(), yy.ravel()]) |
|  | Z = Z.reshape(xx.shape) |
|  | # Plot the contour and training examples |
|  | plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral) |
|  | 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.savefig(fig, bbox\_inches='tight') |

**DLFunctions2\_1.R**

|  |
| --- |
| library(ggplot2) |
|  | # Sigmoid function |
|  | sigmoid <- function(z){ |
|  | a <- 1/(1+ exp(-z)) |
|  | a |
|  | } |
|  |  |
|  | # Compute the model shape given the dataset |
|  | # Input : X - features |
|  | # Y - output |
|  | # |
|  | # Returns: no of training samples, no features, no hidden, no output |
|  | getModelShape <- function(X,Y){ |
|  | numTraining <- dim(X)[2] # No of training examples |
|  | numFeats <- dim(X)[1] # No of input features |
|  | numHidden<-4 # No of units in hidden layer |
|  | # If Y is a row vector set numOutput as 1 |
|  | if(is.null(dim(Y))) |
|  | numOutput <-1 # No of output units |
|  | else |
|  | numOutput <- dim(Y)[1] |
|  | # Create a list of values |
|  | modelParams <- list("numTraining"=numTraining,"numFeats"=numFeats,"numHidden"=numHidden, |
|  | "numOutput"=numOutput) |
|  | return(modelParams) |
|  | } |
|  |  |
|  |  |
|  | # Initialize the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  | initializeModel <- function(numFeats,numHidden,numOutput){ |
|  | set.seed(2) |
|  | w= rnorm(numHidden\*numFeats)\*0.01 |
|  | W1<-matrix(w,nrow=numHidden,ncol=numFeats) # Multiply by .01 |
|  | b1<-matrix(rep(0,numHidden),nrow=numHidden,ncol=1) |
|  | w= rnorm(numOutput\*numHidden) |
|  | W2<-matrix(w,nrow=numOutput,ncol=numHidden) |
|  | b2<- matrix(rep(0,numOutput),nrow=numOutput,ncol=1) |
|  |  |
|  | # Create a list of the neural network parameters |
|  | nnParameters<- list('W1'=W1,'b1'=b1,'W2'=W2,'b2'=b2) |
|  | return(nnParameters) |
|  | } |
|  |  |
|  |  |
|  | #Compute the forward propagation through the neural network |
|  | # Input : Features |
|  | # Weight and bias matrices and vectors |
|  | # Returns : The Activation of 2nd layer |
|  | # : Output and activation of layer 1 & 2 |
|  |  |
|  | forwardPropagation <- function(X,nnParameters){ |
|  | # Get the parameters |
|  | W1<-nnParameters$W1 |
|  | b1<-nnParameters$b1 |
|  | W2<-nnParameters$W2 |
|  | b2<-nnParameters$b2 |
|  |  |
|  | z <- W1 %\*% X |
|  |  |
|  | # Broadcast the bias vector for each row. Use 'sweep' The value '1' for MARGIN |
|  | # indicates sweep each row by this value( add a column vector to each row). |
|  | # If we want to sweep by column use '2'for MARGIN. Here a row vector is added to |
|  | # column (braodcasting!) |
|  |  |
|  | Z1 <-sweep(z,1,b1,'+') |
|  | # Compute the output A1 with the tanh activation function. The tanh activation function |
|  | # performs better than the sigmoid function |
|  |  |
|  | A1<-tanh(Z1) |
|  |  |
|  | # Compute Z2 of the 2nd layer |
|  | z <- W2 %\*% A1 |
|  | # Broadcast the bias vector for each row. Use 'sweep' |
|  | Z2 <- sweep(z,1,b2,'+') |
|  | # Compute the output A1 with the tanh activation function. The tanh activation function |
|  | # performs better than the sigmoid function |
|  | A2<-sigmoid(Z2) |
|  | cache <- list('Z1'=Z1,'A1'=A1,'Z2'=Z2,'A2'=A2) |
|  | return(list('A2'=A2, 'cache'=cache)) |
|  | } |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of 2nd layer |
|  | # : Output from data |
|  | # Output: cost |
|  | computeCost <- function(A2,Y){ |
|  | m= length(Y) |
|  | cost=-1/m\*sum(Y\*log(A2) + (1-Y)\*log(1-A2)) |
|  | #cost=-1/m\*sum(a+b) |
|  | return(cost) |
|  | } |
|  |  |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : Neural Network parameters - weights and biases |
|  | # # Z and Activations of 2 layers |
|  | # # Input features |
|  | # # Output values Y |
|  | # Returns: Gradients |
|  | backPropagation <- function(nnParameters, cache, X, Y){ |
|  | numtraining<- dim(X)[2] |
|  | # Get parameters |
|  | W1<-nnParameters$W1 |
|  | W2<-nnParameters$W2 |
|  |  |
|  | #Get the NN cache |
|  | A1<-cache$A1 |
|  | A2<-cache$A2 |
|  |  |
|  |  |
|  | dZ2 <- A2 - Y |
|  | dW2 <- 1/numtraining \* dZ2 %\*% t(A1) |
|  | db2 <- 1/numtraining \* rowSums(dZ2) |
|  | dZ1 <- t(W2) %\*% dZ2 \* (1 - A1^2) |
|  | dW1 = 1/numtraining\* dZ1 %\*% t(X) |
|  | db1 = 1/numtraining \* rowSums(dZ1) |
|  |  |
|  | gradients <- list("dW1"= dW1, "db1"= db1, "dW2"= dW2, "db2"= db2) |
|  | return(gradients) |
|  | } |
|  |  |
|  |  |
|  |  |
|  | # Gradient descent |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | #output : Updated weights after 1 iteration |
|  | gradientDescent <- function(nnParameters, gradients, learningRate){ |
|  | W1 <- nnParameters$W1 |
|  | b1 <- nnParameters$b1 |
|  | W2 <- nnParameters$W2 |
|  | b2 <- nnParameters$b2 |
|  | dW1<-gradients$dW1 |
|  | db1 <- gradients$db1 |
|  | dW2 <- gradients$dW2 |
|  | db2 <-gradients$db2 |
|  | W1 <- W1-learningRate\*dW1 |
|  | b1 <- b1-learningRate\*db1 |
|  | W2 <- W2-learningRate\*dW2 |
|  | b2 <- b2-learningRate\*db2 |
|  | updatedNNParameters <- list("W1"= W1, "b1"= b1, "W2"= W2, "b2"= b2) |
|  | return(updatedNNParameters) |
|  | } |
|  |  |
|  | # Compute the Neural Network by minimizing the cost |
|  | # Input : Input data X, |
|  | # Output Y |
|  | # No of hidden units in hidden layer |
|  | # No of iterations |
|  | # Returns Updated weight and bias vectors of the neural network |
|  | computeNN <- function(X, Y, numHidden, learningRate, numIterations = 10000){ |
|  |  |
|  | modelParams <- getModelShape(X, Y) |
|  | numFeats<-modelParams$numFeats |
|  | numOutput<-modelParams$numOutput |
|  | costs=NULL |
|  | nnParameters <- initializeModel(numFeats,numHidden,numOutput) |
|  | W1 <- nnParameters$W1 |
|  | b1<-nnParameters$b1 |
|  | W2<-nnParameters$W2 |
|  | b2<-nnParameters$b2 |
|  | # Perform gradient descent |
|  | for(i in 0: numIterations){ |
|  |  |
|  | # Evaluate forward prop to compute activation at output layer |
|  | #print("Here") |
|  | fwdProp = forwardPropagation(X, nnParameters) |
|  | # Compute cost from Activation at output and Y |
|  | cost = computeCost(fwdProp$A2, Y) |
|  | # Perform backprop to compute gradients |
|  | gradients = backPropagation(nnParameters, fwdProp$cache, X, Y) |
|  | # Use gradients to update the weights for each iteration. |
|  | nnParameters = gradientDescent(nnParameters, gradients,learningRate) |
|  | # Print the cost every 1000 iterations |
|  | if(i%%1000 == 0){ |
|  | costs=c(costs,cost) |
|  | print(cost) |
|  | } |
|  | } |
|  |  |
|  | nnVals <- list("nnParameter"=nnParameters,"costs"=costs) |
|  | return(nnVals) |
|  | } |
|  | # Predict the output |
|  | predict <- function(parameters, X){ |
|  |  |
|  | fwdProp <- forwardPropagation(X, parameters) |
|  | predictions <- fwdProp$A2>0.5 |
|  |  |
|  | return (predictions) |
|  | } |
|  |  |
|  | # Plot a decision boundary |
|  | # This function uses the contour method |
|  | drawBoundary <- function(z,nn){ |
|  | # Find the minimum and maximum of the 2 fatures |
|  | xmin<-min(z[,1]) |
|  | xmax<-max(z[,1]) |
|  | ymin<-min(z[,2]) |
|  | ymax<-max(z[,2]) |
|  |  |
|  | a=seq(xmin,xmax,length=100) |
|  | b=seq(ymin,ymax,length=100) |
|  | grid <- expand.grid(x=a, y=b) |
|  | grid1 <-t(grid) |
|  | q <-predict(nn$nnParameter,grid1) |
|  | # Works |
|  | contour(a, b, z=matrix(q, nrow=100), levels=0.5, |
|  | col="black", drawlabels=FALSE, lwd=2,xlim=range(2,10)) |
|  | points(z[,1],z[,2],col=ifelse(z[,3]==1, "coral", "cornflowerblue"),pch=18) |
|  | } |
|  |  |
|  | # Plot a decision boundary |
|  | # This function uses ggplot2 |
|  | plotDecisionBoundary <- function(z,nn,sz,lr){ |
|  | xmin<-min(z[,1]) |
|  | xmax<-max(z[,1]) |
|  | ymin<-min(z[,2]) |
|  | ymax<-max(z[,2]) |
|  |  |
|  |  |
|  | 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) |
|  | q <-predict(nn$nnParameter,grid1) |
|  | q1 <- t(data.frame(q)) |
|  | q2 <- as.numeric(q1) |
|  | grid2 <- cbind(grid,q2) |
|  | colnames(grid2) <- c('x1', 'x2','q2') |
|  |  |
|  | z1 <- data.frame(z) |
|  | names(z1) <- c("x1","x2","y") |
|  | atitle=paste("Decision boundary for hidden layer size:",sz,"learning rate:",lr) |
|  | ggplot(z1) + |
|  | geom\_point(data = z1, aes(x = x1, y = x2, color = y)) + |
|  | stat\_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2), alpha = 0.9)+ |
|  | ggtitle(atitle) |
|  | } |
|  |  |
|  | # Plot a decision boundary |
|  | # This function uses ggplot2 and stat\_contour |
|  | plotBoundary <- function(z,nn){ |
|  | xmin<-min(z[,1]) |
|  | xmax<-max(z[,1]) |
|  | ymin<-min(z[,2]) |
|  | ymax<-max(z[,2]) |
|  |  |
|  |  |
|  | a=seq(xmin,xmax,length=100) |
|  | b=seq(ymin,ymax,length=100) |
|  | grid <- expand.grid(x=a, y=b) |
|  | colnames(grid) <- c('x1', 'x2') |
|  | grid1 <-t(grid) |
|  | q <-predict(nn$nnParameter,grid1) |
|  | 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") |
|  | data.plot <- ggplot() + |
|  | geom\_point(data = z1, aes(x = x1, y = x2, color = y)) + |
|  | coord\_fixed() + |
|  |  |
|  | xlab('x1') + |
|  | ylab('x2') |
|  | print(data.plot) |
|  |  |
|  | data.plot + stat\_contour(data = grid2, aes(x = x1, y = x2, z = q2), alpha = 0.9) |
|  | } |

**Conclusion**: This post implemented a 3 layer Neural Network to create non-linear boundaries while performing classification. Clearly the Neural Network performs very well when the number of hidden units and learning rate are varied.