

Deep Learning from first principles – Second Edition

In vectorized Python, R and Octave

This book is dedicated to the memory of my Mom (Late P. Janaki) and Dad (Late T.D.Venkataraman) who continue to be the force behind all my actions

This book is also dedicated to my wife Shanthi for her support and for giving me the space to work, and finally to my daughter, Shreya, for bringing joy to my life.

Table of Contents

Preface	4
Introduction.....	6
1. Logistic Regression as a Neural Network.....	8
2. Implementing a simple Neural Network	23
3. Building a L- Layer Deep Learning Network.....	48
4. Deep Learning network with the Softmax	85
5. MNIST classification with Softmax.....	103
6. Initialization, regularization in Deep Learning.....	121
7. Gradient Descent Optimization techniques.....	167
8. Gradient Check in Deep Learning.....	197
1. Appendix A.....	214
2. Appendix 1 – Logistic Regression as a Neural Network.....	220
3. Appendix 2 - Implementing a simple Neural Network.....	227
4. Appendix 3 - Building a L- Layer Deep Learning Network	240
5. Appendix 4 - Deep Learning network with the Softmax.....	259
6. Appendix 5 - MNIST classification with Softmax	269
7. Appendix 6 - Initialization, regularization in Deep Learning	302
8. Appendix 7 - Gradient Descent Optimization techniques	344
9. Appendix 8 – Gradient Check.....	405
References	475

Preface

You don't understand anything until you learn it more than one way. Marvin Minsky

The last decade and some, has witnessed some remarkable advancements in the area of Deep Learning. This area of AI has proliferated into many branches - Deep Belief Networks, Recurrent Neural Networks, Convolution Neural Networks, Adversarial Networks, Reinforcement Learning, Capsule Networks and the list goes on. These years have also resulted in Deep Learning to move from the research labs and closer to the home, thanks to progress in hardware, storage and cloud technology.

One common theme when you listen to Deep Learning pundits, is that in order to get a good grasp of the Deep Learning domain, it is essential that you learn to build such a network from scratch. It is towards that end that this book was written.

In this book, I implement Deep Learning Networks from the basics. Each successive chapter builds upon the implementations of the previous chapters so that by chapter 7, I have a full-fledged, generic L-Layer Deep Learning network, with all the bells and whistles. All the necessary derivations required for implementing a multi-layer Deep Learning network is included in the chapters. Detailed derivations for forward propagation and backward propagation cycles with relu, tanh and sigmoid hidden layer units, and sigmoid and softmax output activation units are included. These may serve to jog your memory of all those whose undergrad calculus is a little rusty.

The first chapter derives and implements logistic regression as a neural network in Python, R and Octave. The second chapter deals with the derivation and implementation of the most primitive neural network, one with just one hidden layer. The third chapter extends on the principles of the 2nd chapter and implements a L-Layer Deep Learning network with the sigmoid activation in vectorized Python, R and Octave. This implementation can include an arbitrary number of hidden units and any number of hidden layers for the sigmoid activation output layer. The fourth chapter introduces the Softmax function required for multi-class classification. The Jacobian of the Softmax and cross-entropy loss is derived and then this implemented to demonstrate multi-class classification of a simple spiral data set. The fifth chapter incorporates the softmax implementation of the fourth chapter into the L-Layer implementation in the 3rd chapter. With this enhancement, the fifth chapter classifies MNIST digits using Softmax output activation unit in a generic L-Layer implementation. The sixth chapter addresses different initialization techniques like He and Xavier. Further, this chapter also discusses and implements L2 regularization and random dropout technique. The seventh chapter looks at gradient descent optimization techniques like learning rate decay, momentum, rmsprop, and adam. The eighth chapter discusses a critical technique, that is required to ensure the correctness of the backward propagation implementation. Specifically this chapter discusses and implements 'gradient checking' and also demonstrates how to find bugs in your implementation.

All the chapters include vectorized implementations in Python, R and Octave. The implementations are identical. So, if you are conversant in any one of the languages you can look at the implementations in any other language. It should be a good way to learn the other language.

Note: The functions that are invoked in each of the chapters are included in Appendix 1-Appendix 8.

Feel free to check out the implementations by playing around with the hyper-parameters. A good way to learn is to take the code apart. You could also try to enhance the implementation to include other activation functions like the leaky relu, parametric relu etc. Maybe, you could work on other regularization or gradient descent optimization methods. There may also be opportunities to optimize my code with further vectorization of functions.

This course is largely based on Prof Andrew Ng's Deep Learning Specialization (<https://www.coursera.org/specializations/deep-learning>).

I would like to thank Prof Andrew Ng and Prof Geoffrey Hinton for making the apparent complexity of the Deep Learning subject into remarkably simple concepts through their courses.

I hope this book sets you off on a exciting and challenging journey into the Deep Learning domain

Tinniam V Ganesh

16 May 2018

Introduction

This is the second edition of my book ‘Deep Learning from first principles: Second Edition – In vectorized Python, R and Octave’. Since this book has about 70% code, I wanted to make the code more readable. Hence, in this second edition, I have changed all the code to use the fixed-width font Lucida Console. This makes the code more organized and can be more easily absorbed. I have also included line numbers for all functions and code snippets. Finally, I have corrected some of the typos in the book.

Other books by the author (available on Amazon in paperback and kindle versions)

1. Practical Machine Learning with R and Python: Second Edition – Machine Learning in stereo
2. Beaten by sheer pace: Third Edition – Cricket analytics with yorkr
3. Cricket analytics with cricketr:Third Edition

1. Logistic Regression as a Neural Network

“You don’t perceive objects as they are. You perceive them as you are.”

“Your interpretation of physical objects has everything to do with the historical trajectory of your brain – and little to do with the objects themselves.”

“The brain generates its own reality, even before it receives information coming in from the eyes and the other senses. This is known as the internal model”

David Eagleman - The Brain: The Story of You

This chapter deals with the implementation of Logistic regression as a 2-layer Neural Network i.e. a Neural Network that just has an input layer and an output layer and with no hidden layer. This 2-layer network is implemented in Python, R and Octave languages. I have included Octave, into the mix, as Octave is a close cousin of Matlab. These implementations in Python, R and Octave are equivalent vectorized implementations. Therefore, if you are familiar in any one of the languages, you should be able to look at the corresponding code in the other two. The implementations of the functions invoked in this chapter are in Appendix 1 – Logistic Regression as a Neural Network.

You can also clone/download the vectorized code in Python, R and Octave from Github at DeepLearningFromFirstPrinciples
(<https://github.com/tvganesh/DeepLearningFromFirstPrinciples/tree/master/Chap1-LogisticRegressionAsNeuralNetwork>). To start with, Logistic Regression is performed using sklearn’s logistic regression package, for the cancer data set also from sklearn. This is shown below

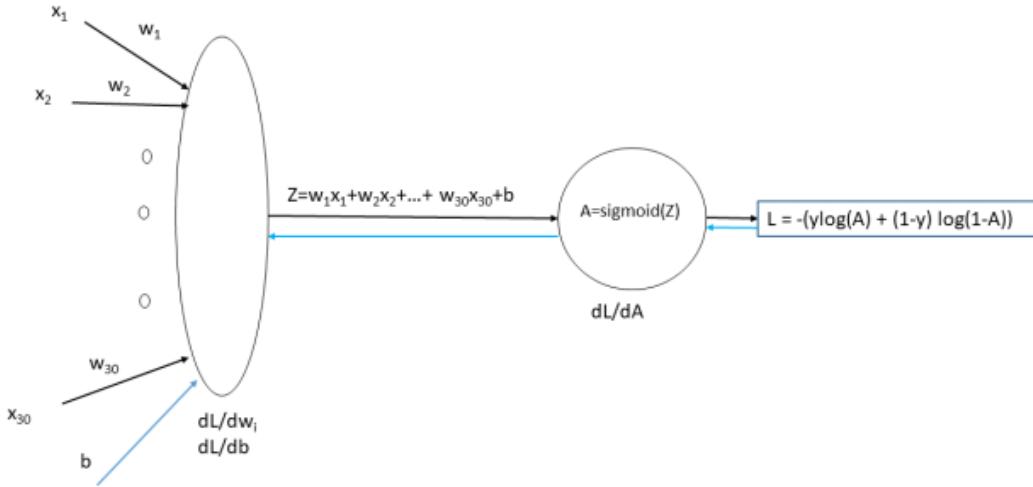
1. Logistic Regression

```
1 import numpy as np
2 import pandas as pd
3 import os
4 import matplotlib.pyplot as plt
5 from sklearn.model_selection import train_test_split
6 from sklearn.linear_model import LogisticRegression
7 from sklearn.datasets import make_classification, make_blobs
8
9 from sklearn.metrics import confusion_matrix
10 from matplotlib.colors import ListedColormap
11 from sklearn.datasets import load_breast_cancer
12
13 # Load the cancer data
14 (X_cancer, y_cancer) = load_breast_cancer(return_X_y = True)
15 X_train, X_test, y_train, y_test = train_test_split(X_cancer, y_cancer,
16 random_state = 0)
17
18 # Call the Logistic Regression function
```

```
19 clf = LogisticRegression().fit(X_train, y_train)
20
21 #Print accuracy of training and test set
22 print('Accuracy of Logistic regression classifier on training set: {:.2f}'
23     .format(clf.score(X_train, y_train)))
24 print('Accuracy of Logistic regression classifier on test set: {:.2f}'
25     .format(clf.score(X_test, y_test)))
26 ## Accuracy of Logistic regression classifier on training set: 0.96
27 ## Accuracy of Logistic regression classifier on test set: 0.96
28
```

2. Logistic Regression as a 2-layer Neural Network

In the following section, Logistic Regression is implemented as a 2-layer Neural Network in Python, R and Octave. The same cancer data set from sklearn is used to train and test the Neural Network in Python, R and Octave. This can be represented diagrammatically as below



The cancer data set has 30 input features, and the target variable ‘output’ is either 0 or 1. Hence, the sigmoid activation function will be used in the output layer for classification.

This simple 2-layer Neural Network is shown below.

At the input layer, there are 30 features and the corresponding weights of these inputs which are initialized to small random values.

$$Z = w_1x_1 + w_2x_2 + \dots + w_{30}x_{30} + b$$

where ‘ b ’ is the bias term

The Activation function is the sigmoid function which is given by $a = 1/(1 + e^{-z})$

The Loss, when the sigmoid function is used in the output layer, is given by

$$L = -(y\log(a) + (1 - y)\log(1 - a)) \quad (1)$$

3. Gradient Descent

3.1 Forward propagation

The forward propagation cycle of the Neural Network computes the output Z and the activation the sigmoid activation function. Then using the output 'y' for the given features, the 'Loss' is computed using equation (1) above.

3.2 Backward propagation

The backward propagation cycle determines how the 'Loss' is impacted for small variations from the previous layers up to the input layer. In other words, backward propagation computes the changes in the weights at the input layer, which will minimize the loss at the output layer. Several cycles of gradient descent are performed in the path of steepest descent to find the local minima. In other words, the set of weights and biases, at the input layer, which will result in the lowest loss, is computed by gradient descent. The weights at the input layer are decreased by a parameter known as the 'learning rate'. Too big a 'learning rate' can overshoot the local minima, and too small a 'learning rate' can take a long time to reach the local minima. Gradient Descent iterated through this forward propagation and backward propagation cycle until the loss is minimized. This is done for 'm' training examples.

3.3 Chain rule of differentiation

Let $y=f(u)$

and $u=g(x)$ then by chain rule

$$\partial y / \partial x = \partial y / \partial u * \partial u / \partial x$$

3.4 Derivative of sigmoid

$$\sigma = 1/(1 + e^{-z})$$

Let $x = 1 + e^{-z}$ then

$$\sigma = 1/x$$

$$\partial \sigma / \partial x = -1/x^2$$

$$\text{and } \partial x / \partial z = -e^{-z}$$

Using the chain rule of differentiation we get

$$\partial \sigma / \partial z = \partial \sigma / \partial x * \partial x / \partial z$$

$$= -1/(1 + e^{-z})^2 * -e^{-z} = e^{-z}/(1 + e^{-z})^2$$

$$\text{Therefore } \partial \sigma / \partial z = \sigma(1 - \sigma) \quad -(2)$$

The 3 equations for the 2 layer Neural Network representation of Logistic Regression are

$$L = -(y * \log(a) + (1 - y) * \log(1 - a)) \quad -(a)$$

$$a = 1/(1 + e^{-Z}) \quad -(b)$$

$$Z = w_1x_1 + w_2x_2 + \dots + w_{30}x_{30} + b = Z = \sum_i w_i * x_i + b \quad -(c)$$

Where L is the loss for the sigmoid output activation function

The back-propagation step requires the computation of dL/dw_i and dL/db_i . In the case of regression it would be dE/dw_i and dE/db_i where dE is the Mean Squared Error function.

Computing the derivatives for the Loss function we have

$$dL/da = -(y/a + (1 - y)/(1 - a)) \quad \text{-(d)}$$

because $d/dx(\log x) = 1/x$

Also from equation (2) we can write

$$da/dZ = a(1 - a) \quad \text{-(e)}$$

By chain rule

$$\partial L/\partial Z = \partial L/\partial a * \partial a/\partial Z$$

therefore substituting the results of (d) & (e) into the equation above we get

$$\partial L/\partial Z = -(y/a + (1 - y)/(1 - a)) * a(1 - a) = a - y \quad \text{-(f)}$$

Finally

$$\partial L/\partial w_i = \partial L/\partial a * \partial a/\partial Z * \partial Z/\partial w_i \quad \text{-(g)}$$

$$\partial Z/\partial w_i = x_i \quad \text{-(h)}$$

and from (f) we have $\partial L/\partial Z = a - y$

Therefore (g) reduces to

$$\partial L/\partial w_i = x_i * (a - y) \quad \text{-(i)}$$

Also

$$\partial L/\partial b = \partial L/\partial a * \partial a/\partial Z * \partial Z/\partial b \quad \text{-(j)}$$

Since

$\partial Z/\partial b = 1$ and using (f) in (j) we get

$$\partial L/\partial b = a - y$$

The gradient computes the weights at the input layer and the corresponding bias by using the values of dw_i and db

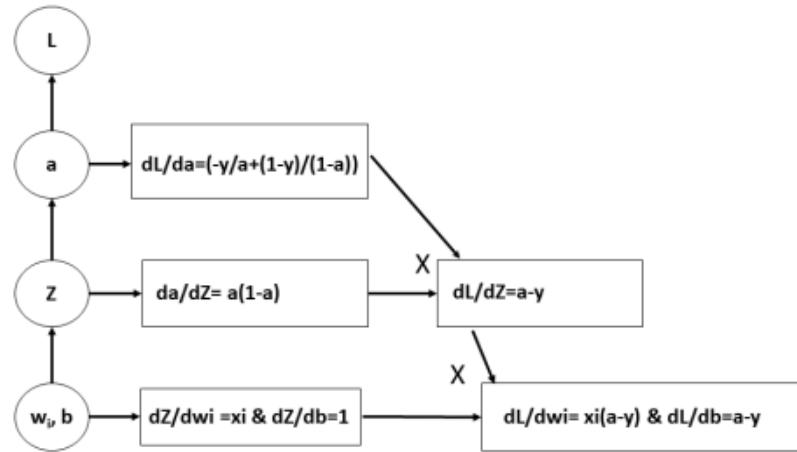
$$w_i := w_i - \alpha * dw_i$$

$$b := b - \alpha * db$$

The computation graph representation in the book Deep Learning

(<http://www.deeplearningbook.org/>) : Ian Goodfellow, Yoshua Bengio, Aaron Courville, is very useful to visualize and compute the backward propagation. For the 2-layer Neural Network of Logistic Regression the computation graph is shown below

Computational graph for Neural Network of Logistic Regression



4. Neural Network for Logistic Regression -Python code

```

1 import numpy as np
2 import pandas as pd
3 import os
4 import matplotlib.pyplot as plt
5 from sklearn.model_selection import train_test_split
6
7 # Define the sigmoid function
8 def sigmoid(z):
9     a=1/(1+np.exp(-z))
10    return a
11
12 # Initialize weights and biases
13 def initialize(dim):
14     w = np.zeros(dim).reshape(dim,1)
15     b = 0
16     return w
17
18 # Compute the loss
19 def computeLoss(numTraining,Y,A):
20     loss=-1/numTraining *np.sum(Y*np.log(A) + (1-Y)*(np.log(1-A)))
21     return(loss)
22
23 # Execute the forward propagation
24 def forwardPropagation(w,b,X,Y):
25     # Compute Z
26     Z=np.dot(w.T,X)+b
27     # Determine the number of training samples
28     numTraining=float(len(X))
29     # Compute the output of the sigmoid activation function
30     A=sigmoid(Z)
31     #Compute the loss
32     loss = computeLoss(numTraining,Y,A)
33     # Compute the gradients dz, dw and db
34     dZ=A-Y
  
```

```

35     dw=1/numTraining*np.dot(x,dz.T)
36     db=1/numTraining*np.sum(dz)
37
38     # Return the results as a dictionary
39     gradients = {"dw": dw,
40                   "db": db}
41     loss = np.squeeze(loss)
42     return gradients,loss
43
44 # Compute Gradient Descent
45 def gradientDescent(w, b, X, Y, numIterations, learningRate):
46     losses=[]
47     idx =[]
48     # Iterate
49     for i in range(numIterations):
50         gradients,loss=forwardPropagation(w,b,X,Y)
51         #Get the derivates
52         dw = gradients["dw"]
53         db = gradients["db"]
54         w = w-learningRate*dw
55         b = b-learningRate*db
56
57         # Store the loss
58         if i % 100 == 0:
59             idx.append(i)
60             losses.append(loss)
61         # Set params and grads
62         params = {"w": w,
63                   "b": b}
64         grads = {"dw": dw,
65                   "db": db}
66
67     return params, grads, losses,idx
68
69 # Predict the output for a training set
70 def predict(w,b,x):
71     size=x.shape[1]
72     yPredicted=np.zeros((1,size))
73     Z=np.dot(w.T,x)
74     # Compute the sigmoid
75     A=sigmoid(Z)
76     for i in range(A.shape[1]):
77         #If the value is > 0.5 then set as 1
78         if(A[0][i] > 0.5):
79             yPredicted[0][i]=1
80         else:
81             # Else set as 0
82             yPredicted[0][i]=0
83
84     return yPredicted
85
86 #Normalize the data
87 def normalize(x):
88     x_norm = None
89     x_norm = np.linalg.norm(x, axis=1, keepdims=True)
90     x= x/x_norm
91     return x
92
93
94 # Run the 2-layer Neural Network on the cancer data set
95 from sklearn.datasets import load_breast_cancer
96 # Load the cancer data
97 (X_cancer, y_cancer) = load_breast_cancer(return_X_y = True)
98 # Create train and test sets

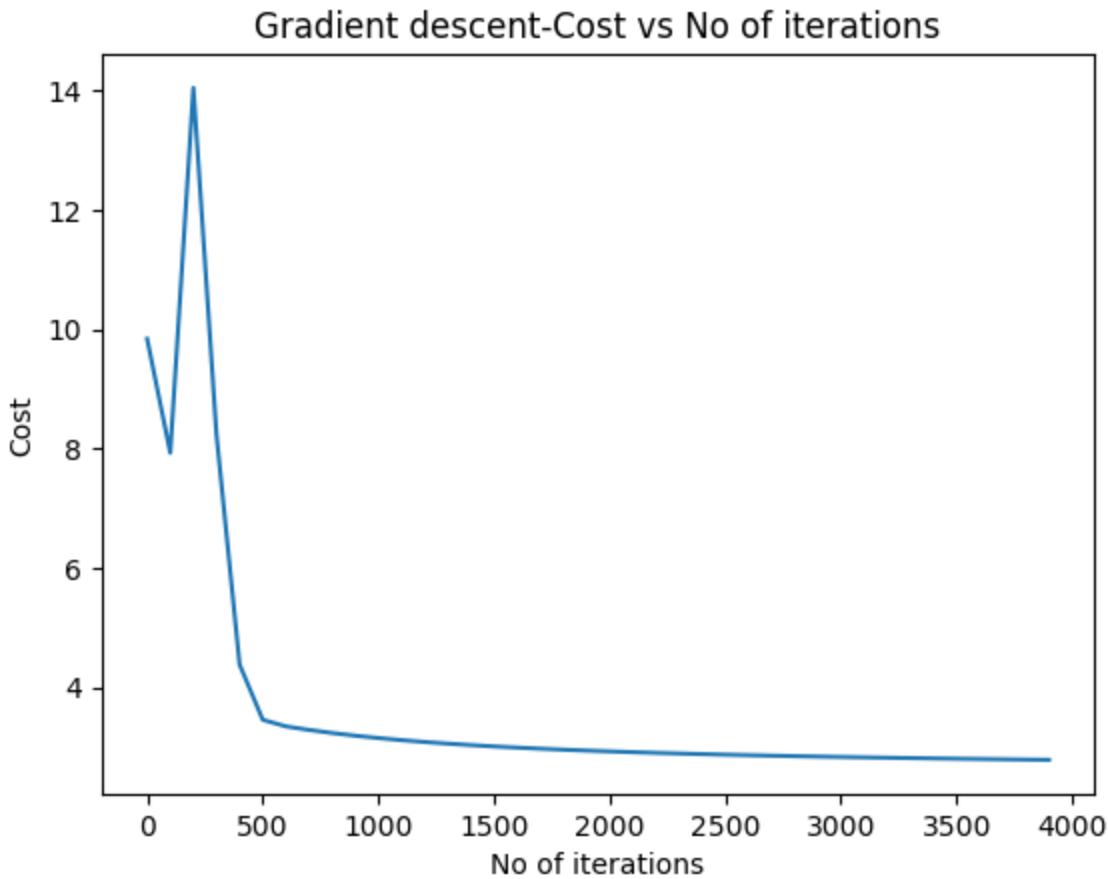
```

```

99 x_train, x_test, y_train, y_test = train_test_split(x_cancer, y_cancer,
100                                         random_state = 0)
101 # Normalize the data for better performance
102 x_train1=normalize(x_train)
103
104
105 # Create weight vectors of zeros. The size is the number of features in the
106 data set=30
107 w=np.zeros((x_train.shape[1],1))
108 #w=np.zeros((30,1))
109 b=0
110
111 #Normalize the training data so that gradient descent performs better
112 x_train1=normalize(x_train)
113 #Transpose x_train so that we have a matrix as (features, numSamples)
114 x_train2=x_train1.T
115
116 # Reshape to remove the rank 1 array and then transpose
117 y_train1=y_train.reshape(len(y_train),1)
118 y_train2=y_train1.T
119
120 # Run gradient descent for 4000 times and compute the weights
121 parameters, grads, costs, idx = gradientDescent(w, b, x_train2, y_train2,
122 numIterations=4000, learningRate=0.75)
123 w = parameters["w"]
124 b = parameters["b"]
125
126
127 # Normalize x_test
128 x_test1=normalize(x_test)
129 #Transpose x_train so that we have a matrix as (features, numSamples)
130 x_test2=x_test1.T
131
132 #Reshape y_test
133 y_test1=y_test.reshape(len(y_test),1)
134 y_test2=y_test1.T
135
136 # Predict the values for
137 yPredictionTest = predict(w, b, x_test2)
138 yPredictionTrain = predict(w, b, x_train2)
139
140 # Print the accuracy
141 print("train accuracy: {}".format(100 - np.mean(np.abs(yPredictionTrain -
142 y_train2)) * 100))
143 print("test accuracy: {}".format(100 - np.mean(np.abs(yPredictionTest -
144 y_test)) * 100))
145
146 # Plot the Costs vs the number of iterations
147 fig1=plt.plot(idx,costs)
148 fig1=plt.title("Gradient descent-Cost vs No of iterations")
149 fig1=plt.xlabel("No of iterations")
150 fig1=plt.ylabel("Cost")
151 fig1.figure.savefig("fig1", bbox_inches='tight')
152 ## train accuracy: 90.3755868545 %
153 ## test accuracy: 89.5104895105 %

```

Note: The Accuracy on the training and test set is 90.37% and 89.51%. This is comparatively poorer than the 96%, which the logistic regression of sklearn achieves! But, this is mainly because of the absence of hidden layers which is the real power of neural networks.



5. Neural Network for Logistic Regression -R code

```

1 source("RFunctions-1.R")
2 # Define the sigmoid function
3 sigmoid <- function(z){
4     a <- 1/(1+ exp(-z))
5     a
6 }
7
8 # Compute the loss
9 computeLoss <- function(numTraining,Y,A){
10    loss <- -1/numTraining* sum(Y*log(A) + (1-Y)*log(1-A))
11    return(loss)
12 }
13
14 # Compute forward propagation
15 forwardPropagation <- function(w,b,x,Y){
16     # Compute z
17     Z <- t(w) %*% x +b
18     #Set the number of samples
19     numTraining <- ncol(x)
20     # Compute the activation function
21     A=sigmoid(Z)
22
23     #Compute the loss
24     loss <- computeLoss(numTraining,Y,A)
25 }
```

```

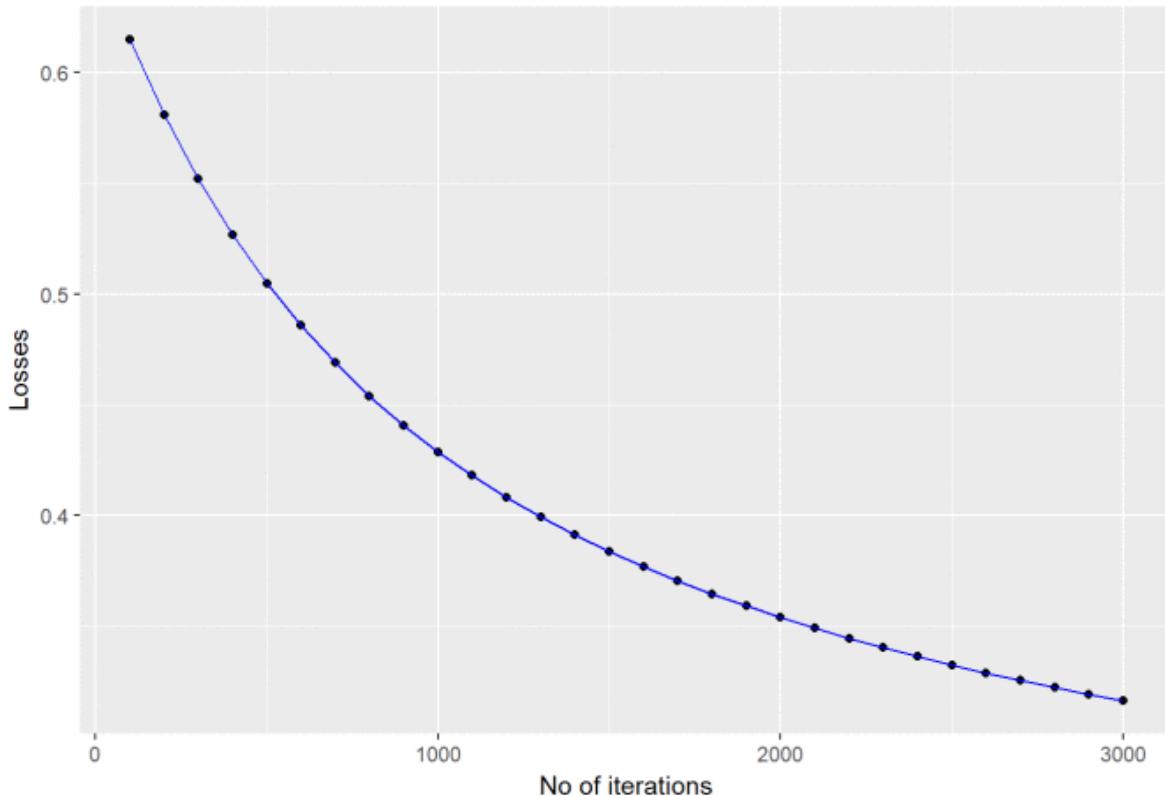
26     # Compute the gradients dz, dw and db
27     dZ<-A-Y
28     dw<-1/numTraining * x %*% t(dZ)
29     db<-1/numTraining*sum(dZ)
30
31     fwdProp <- list("loss" = loss, "dw" = dw, "db" = db)
32     return(fwdProp)
33 }
34
35 # Perform one cycle of Gradient descent
36 gradientDescent <- function(w, b, X, Y, numIterations, learningRate){
37   losses <- NULL
38   idx <- NULL
39   # Loop through the number of iterations
40   for(i in 1:numIterations){
41     fwdProp <- forwardPropagation(w,b,X,Y)
42     #Get the derivatives
43     dw <- fwdProp$dw
44     db <- fwdProp$db
45     #Perform gradient descent
46     w = w-learningRate*dw
47     b = b-learningRate*db
48     l <- fwdProp$loss
49     # Stoe the loss
50     if(i %% 100 == 0){
51       idx <- c(idx,i)
52       losses <- c(losses,l)
53     }
54   }
55
56   # Return the weights and losses
57   gradDescnt <- list("w"=w,"b"=b,"dw"=dw,"db"=db,"losses"=losses,"idx"=idx)
58   return(gradDescnt)
59 }
60
61 # Compute the predicted value for input
62 predict <- function(w,b,X){
63   m=dim(X)[2]
64   # Create a vector of 0's
65   yPredicted=matrix(rep(0,m),nrow=1,ncol=m)
66   Z <- t(w) %*% x +b
67   # Compute sigmoid
68   A=sigmoid(Z)
69   for(i in 1:dim(A)[2]){
70     # If A > 0.5 set value as 1
71     if(A[1,i] > 0.5)
72       yPredicted[1,i]=1
73     else
74       # Else set as 0
75       yPredicted[1,i]=0
76   }
77
78   return(yPredicted)
79 }
80
81 # Normalize the matrix
82 normalize <- function(x){
83   #Create the norm of the matrix.Perform the Frobenius norm of the matrix
84   n<-as.matrix(sqrt(rowSums(x^2)))
85   #Sweep by rows by norm. Note '1' in the function which performing on
86   every row
87   normalized<-sweep(x, 1, n, FUN="/")
88   return(normalized)
89 }
```

```

90
91 # Run the 2 layer Neural Network on the cancer data set
92 # Read the data (from sklearn)
93 cancer <- read.csv("cancer.csv")
94 # Rename the target variable
95 names(cancer) <- c(seq(1,30), "output")
96 # Split as training and test sets
97 train_idx <- trainTestSplit(cancer, trainPercent=75, seed=5)
98 train <- cancer[train_idx, ]
99 test <- cancer[-train_idx, ]
100
101 # Set the features
102 x_train <- train[,1:30]
103 y_train <- train[,31]
104 x_test <- test[,1:30]
105 y_test <- test[,31]
106 # Create a matrix of 0's with the number of features
107 w <- matrix(rep(0, dim(x_train)[2]))
108 b <- 0
109 x_train1 <- normalize(x_train)
110 x_train2=t(x_train1)
111
112 # Reshape then transpose
113 y_train1=as.matrix(y_train)
114 y_train2=t(y_train1)
115
116 # Perform gradient descent
117 gradDescent= gradientDescent(w, b, x_train2, y_train2, numIterations=3000,
118 learningRate=0.77)
119 # Normalize X_test
120 x_test1=normalize(x_test)
121 #Transpose X_train so that we have a matrix as (features, numSamples)
122 x_test2=t(x_test1)
123
124 #Reshape y_test and take transpose
125 y_test1=as.matrix(y_test)
126 y_test2=t(y_test1)
127
128 # Use the values of the weights generated from Gradient Descent
129 yPredictionTest = predict(gradDescent$w, gradDescent$b, x_test2)
130 yPredictionTrain = predict(gradDescent$w, gradDescent$b, x_train2)
131
132 sprintf("Train accuracy: %f", (100 - mean(abs(yPredictionTrain - y_train2)) *
133 100))
134 ## [1] "Train accuracy: 90.845070"
135 sprintf("test accuracy: %f", (100 - mean(abs(yPredictionTest - y_test)) *
136 100))
137 ## [1] "test accuracy: 87.323944"
138 df <- data.frame(gradDescent$idx, gradDescent$losses)
139 names(df) <- c("iterations", "losses")
140 ggplot(df,aes(x=iterations,y=losses)) + geom_point() + geom_line(col="blue") +
141
142   ggttitle("Gradient Descent - Losses vs No of Iterations") +
143   xlab("No of iterations") + ylab("Losses")

```

Gradient Descent - Losses vs No of Iterations



144

6. Neural Network for Logistic Regression -Octave code

```

1;
2 # Define sigmoid function
3 function a = sigmoid(z)
4     a = 1 ./ (1+ exp(-z));
5 end
6
7 # Compute the loss
8 function loss=computeLoss(numtraining,Y,A)
9     loss = -1/numtraining * sum((Y .* log(A)) + (1-Y) .* log(1-A));
10 end
11
12 # Perform forward propagation
13 function [loss,dw,db,dZ] = forwardPropagation(w,b,x,Y)
14     # Compute Z
15     Z = w' * X + b;
16     numtraining = size(X)(1,2);
17     # Compute sigmoid
18     A = sigmoid(Z);
19     #Compute loss. Note this is element wise product
20     loss =computeLoss(numtraining,Y,A);
21     # Compute the gradients dZ, dw and db
22     dZ = A-Y;
23     dw = 1/numtraining* x * dZ';
24     db =1/numtraining*sum(dZ);
25 end
26
27 # Compute Gradient Descent

```

```

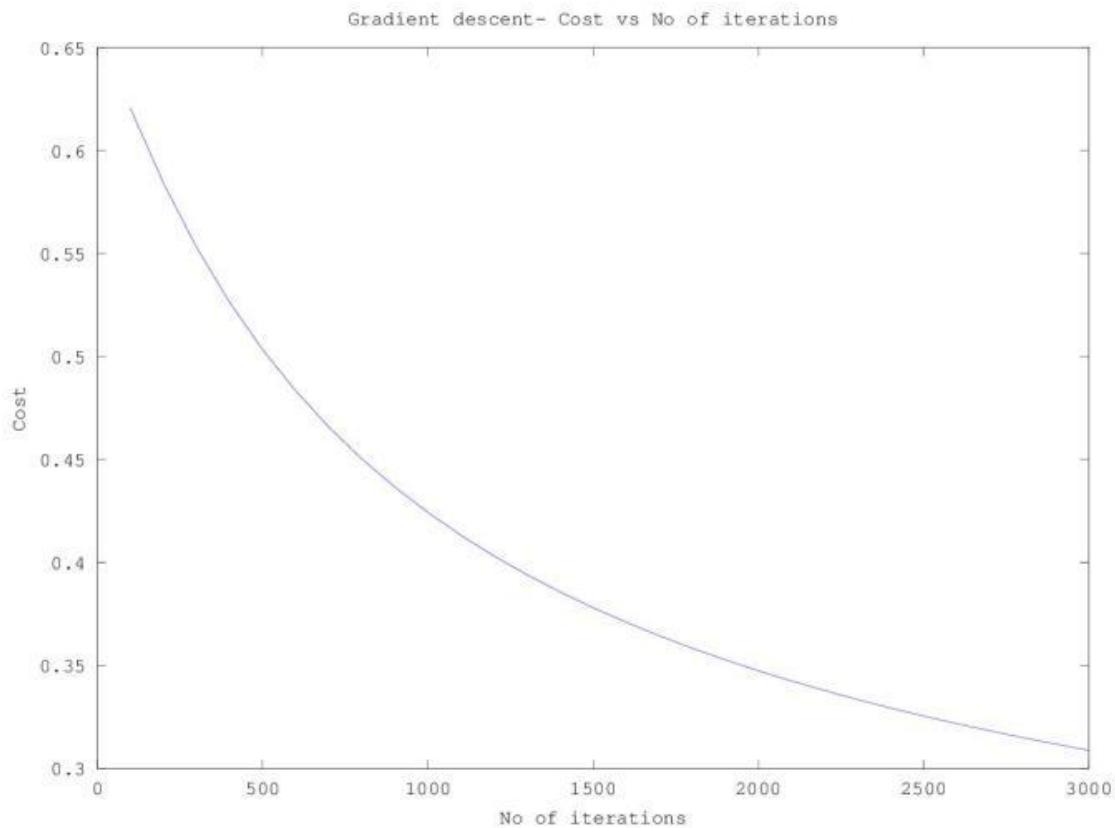
28 function [w,b,dw,db,losses,index]=gradientDescent(w, b, X, Y, numIterations,
29 learningRate)
30     #Initialize losses and idx
31     losses=[];
32     index=[];
33     # Loop through the number of iterations
34     for i=1:numIterations,
35         [loss,dw,db,dZ] = forwardPropagation(w,b,X,Y);
36         # Perform Gradient descent
37         w = w - learningRate*dw;
38         b = b - learningRate*db;
39         if(mod(i,100) ==0)
40             # Append index and loss
41             index = [index i];
42             losses = [losses loss];
43         endif
44     end
45 end
46
47 # Determine the predicted value for dataset
48 function yPredicted = predict(w,b,X)
49     m = size(X)(1,2);
50     yPredicted=zeros(1,m);
51     # Compute Z
52     Z = w' * X + b;
53     # Compute sigmoid
54     A = sigmoid(Z);
55     for i=1:size(X)(1,2),
56         # Set predicted as 1 if A > 0,5
57         if(A(1,i) >= 0.5)
58             yPredicted(1,i)=1;
59         else
60             yPredicted(1,i)=0;
61         endif
62     end
63 end
64
65 # Normalize by dividing each value by the sum of squares
66 function normalized = normalize(x)
67     # Compute Frobenius norm. Square the elements, sum rows and then find
68     square root
69     a = sqrt(sum(x .^ 2,2));
70     # Perform element wise division
71     normalized = x ./ a;
72 end
73
74 # Split into train and test sets
75 function [X_train,y_train,X_test,y_test] =
76 trainTestSplit(dataset,trainPercent)
77     # Create a random index
78     ix = randperm(length(dataset));
79     # Split into training
80     trainSize = floor(trainPercent/100 * length(dataset));
81     train=dataset(ix(1:trainSize),:);
82     # And test
83     test=dataset(ix(trainSize+1:length(dataset)),:);
84     X_train = train(:,1:30);
85     y_train = train(:,31);
86     X_test = test(:,1:30);
87     y_test = test(:,31);
88 end
89
90 # Read the data
91 cancer=csvread("cancer.csv");

```

```

92
93 # Split as train and test
94 [X_train,y_train,X_test,y_test] = trainTestSplit(cancer,75);
95
96 #Initialize w and b
97 w=zeros(size(X_train)(1,2),1);
98 b=0;
99
100 #Normalize training
101 X_train1=normalize(X_train);
102 X_train2=X_train1';
103 y_train1=y_train';
104
105 #Perform gradient descent
106 [w1,b1,dw,db,losses,idx]=gradientDescent(w, b, X_train2, y_train1,
107 numIterations=3000, learningRate=0.75);
108
109 # Normalize X_test
110 X_test1=normalize(X_test);
111 #Transpose X_train so that we have a matrix as (features, numSamples)
112 X_test2=X_test1';
113 y_test1=y_test';
114 # Use the values of the weights generated from Gradient Descent
115 yPredictionTest = predict(w1, b1, X_test2);
116 yPredictionTrain = predict(w1, b1, X_train2);
117
118 #Compute Accuracy
119 trainAccuracy=100-mean(abs(yPredictionTrain - y_train1))*100
120 testAccuracy=100- mean(abs(yPredictionTest - y_test1))*100
121 trainAccuracy = 90.845
122 testAccuracy = 89.510
123
124 graphics_toolkit('gnuplot')
125 plot(idx,losses);
126 title ('Gradient descent- Cost vs No of iterations');
127 xlabel ("No of iterations");
128 ylabel ("Cost");

```



129

Conclusion

This chapter starts with a simple 2-layer Neural Network implementation of Logistic Regression. Clearly, the performance of this simple Neural Network is comparatively poor to the highly optimized sklearn's Logistic Regression. This is because the above neural network did not have any hidden layers. Deep Learning & Neural Networks achieve extraordinary performance because of the presence of deep hidden layers

2.Implementing a simple Neural Network

“What does the world outside your head really ‘look’ like? Not only is there no color, there’s also no sound: the compression and expansion of air is picked up by the ears and turned into electrical signals. The brain then presents these signals to us as mellifluous tones and swishes and clatters and jangles. Reality is also odorless: there’s no such thing as smell outside our brains. Molecules floating through the air bind to receptors in our nose and are interpreted as different smells by our brain. The real world is not full of rich sensory events; instead, our brains light up the world with their own sensuality.”

The Brain: The Story of You” by David Eagleman

“The world is Maya, illusory. The ultimate reality, the Brahman, is all-pervading and all-permeating, which is colourless, odourless, tasteless, nameless and formless“

Bhagavad Gita

1. Introduction

In the first chapter, I implemented Logistic Regression, in vectorized Python, R and Octave, with a wannabe Neural Network (a Neural Network with no hidden layers). In this second chapter, I implement a regular, but somewhat primitive Neural Network, (a Neural Network with just 1 hidden layer). This chapter implements classification of manually created datasets, where the different clusters of the 2 classes are not linearly separable.

Neural Network perform well in learning all sorts of non-linear boundaries between classes. Initially logistic regression is used to perform the classification. A simple 3-layer Neural Network is then used on the same data set and the decision boundary plotted. Vanilla logistic regression performs quite poorly. Using SVMs with a radial basis kernel would have performed much better in creating non-linear boundaries. The implementations of the functions invoked in this chapter are in Appendix 2 - Implementing a simple Neural Network

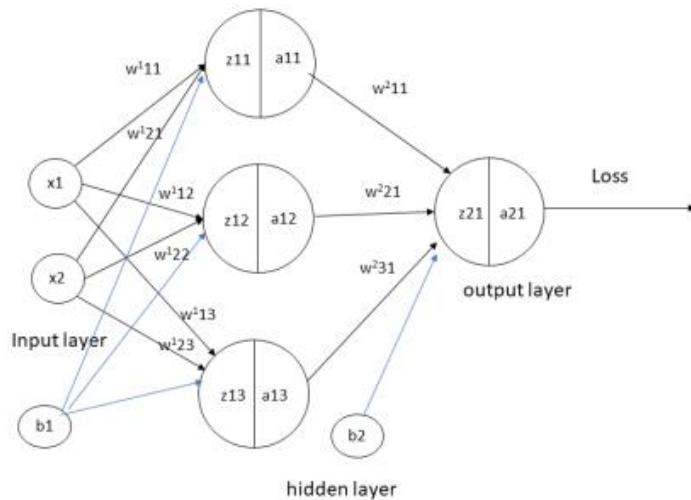
You can clone and fork the vectorized implementations of the 3 layer Neural Network for Python, R and Octave from Github at DeepLearningFromFirstPrinciples

(<https://github.com/tvganesh/DeepLearningFromFirstPrinciples/tree/master/Chap2-SimpleNeuralNetwork>)

2. The 3 layer Neural Network

A simple representation of a 3 layer Neural Network (NN) with 1 hidden layer is shown below.

Neural Network with 1 hidden layer



In the above Neural Network, there are two input features at the input layer, three hidden units at the hidden layer and one 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'

$$z_{21} = w_{11}^2 a_{11} + w_{21}^2 a_{12} + w_{31}^2 a_{13} + b_2$$

$$a_{21} = \text{sigmoid}(z_{21})$$

Hence

And

$$A1 = \begin{pmatrix} a11 \\ a12 \\ a13 \end{pmatrix} = \begin{pmatrix} \tanh(z11) \\ \tanh(z12) \\ \tanh(z13) \end{pmatrix}$$

Similarly

$$\text{and } A2 = a_{21} = \text{sigmoid}(z_{21})$$

These equations can be written as

$$Z1 = W1 * X + b1$$

$$A1 = \tanh(Z1)$$

$$Z2 = W2 * A1 + b2$$

$$A2 = \text{sigmoid}(Z2)$$

I) Some important results (a memory refresher!)

$$d/dx(e^x) = e^x \text{ and } d/dx(e^{-x}) = -e^{-x} \text{ -(a)}$$

$$\sinhx = (e^x - e^{-x})/2 \text{ and } \coshx = (e^x + e^{-x})/2$$

Using (a) we can show that $d/dx(\sinhx) = \coshx$ and $d/dx(\coshx) = \sinhx$ (b)

$$\text{Now } d/dx(f(x)/g(x)) = (g(x) * d/dx(f(x)) - f(x) * d/dx(g(x)))/g(x)^2 \text{ -(c)}$$

Since $\tanhx = z = \sinhx/\coshx$ and using (c) we get

$$\tanhx = (\coshx * d/dx(\sinhx) - \sinhx * d/dx(\coshx))/\cosh^2 x$$

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 $\tanhx = z$

$$d/dx(\tanhx) = 1 - \tanhx^2 = 1 - z^2 \text{ -(d)}$$

II) Derivatives

The log loss is given below

$$L = -(Y \log(A2) + (1 - Y) \log(1 - A2))$$

$$dL/dA2 = -(Y/A2 + (1 - Y)/(1 - A2))$$

Since $A2 = \text{sigmoid}(Z2)$ therefore $dA2/dZ2 = A2(1 - A2)$ see equation (2) Chapter 1

$$Z2 = W2A1 + b2$$

Therefore $dZ2/dW2 = A1$ and (e)

$$dZ2/db2 = 1 \text{ (f) and }$$

$$A1 = \tanh(Z1) \text{ and } dA1/dZ1 = 1 - A1^2 \text{ from (d)}$$

$$Z1 = W1X + b1 \\ dZ1/dW1 = X \text{ (g)}$$

$$dZ1/db1 = 1 \quad (h)$$

III) Back propagation

Using the derivatives from II) we can derive the following results using Chain Rule

$$\begin{aligned} \partial L / \partial Z2 &= \partial L / \partial A2 * \partial A2 / \partial Z2 \\ &= -(Y/A2 + (1 - Y)/(1 - A2)) * A2(1 - A2) = A2 - Y \quad (i) \end{aligned}$$

$$\partial L / \partial W2 = \partial L / \partial A2 * \partial A2 / \partial Z2 * \partial Z2 / \partial W2$$

Using the results of (i) and (e) we get

$$= (A2 - Y) * A1 \quad (j)$$

$$\partial L / \partial b2 = \partial L / \partial A2 * \partial A2 / \partial Z2 * \partial Z2 / \partial b2 = (A2 - Y) \quad -(k)$$

And

$$\begin{aligned} \partial L / \partial Z1 &= \partial L / \partial A2 * \partial A2 / \partial Z2 * \partial Z2 / \partial A1 * \partial A1 / \partial Z1 = (A2 - Y) * W2 * \\ &(1 - A1^2) \end{aligned}$$

$$\partial L / \partial W1 = \partial L / \partial A2 * \partial A2 / \partial Z2 * \partial Z2 / \partial A1 * \partial A1 / \partial Z1 * \partial Z1 / \partial W1$$

Simplifying we get

$$= (A2 - Y) * W2 * (1 - A1^2) * X \quad (l) \text{ and}$$

$$\partial L / \partial b1 = \partial L / \partial A2 * \partial A2 / \partial Z2 * \partial Z2 / \partial A1 * dA1 / dZ1 * dZ1 / db1$$

$$= (A2 - Y) * W2 * (1 - A1^2) \quad (m)$$

IV) Gradient Descent

The key computations in the backward cycle are based on the gradient computed above in equations (h), (i), (j) and (k)

$$W1 = W1 - learningRate * \partial L / \partial W1$$

$$b1 = b1 - learningRate * \partial L / \partial b1$$

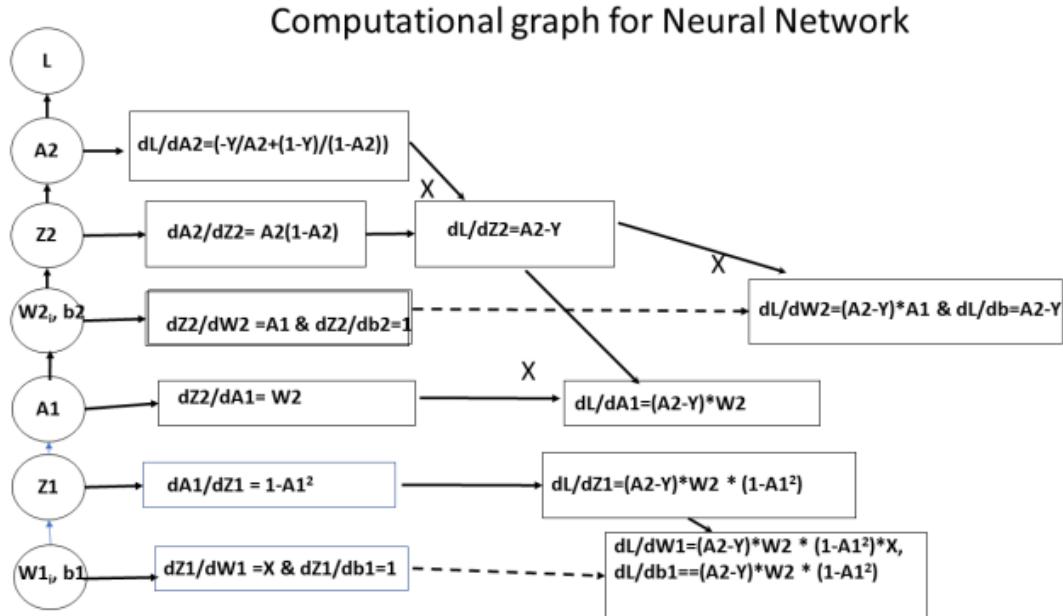
$$W2 = W2 - learningRate * \partial L / \partial W2$$

$$b2 = b2 - learningRate * \partial L / \partial b2$$

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 (<http://www.deeplearningbook.org/>) by Ian Goodfellow, Joshua Bengio and Aaron

Courville)



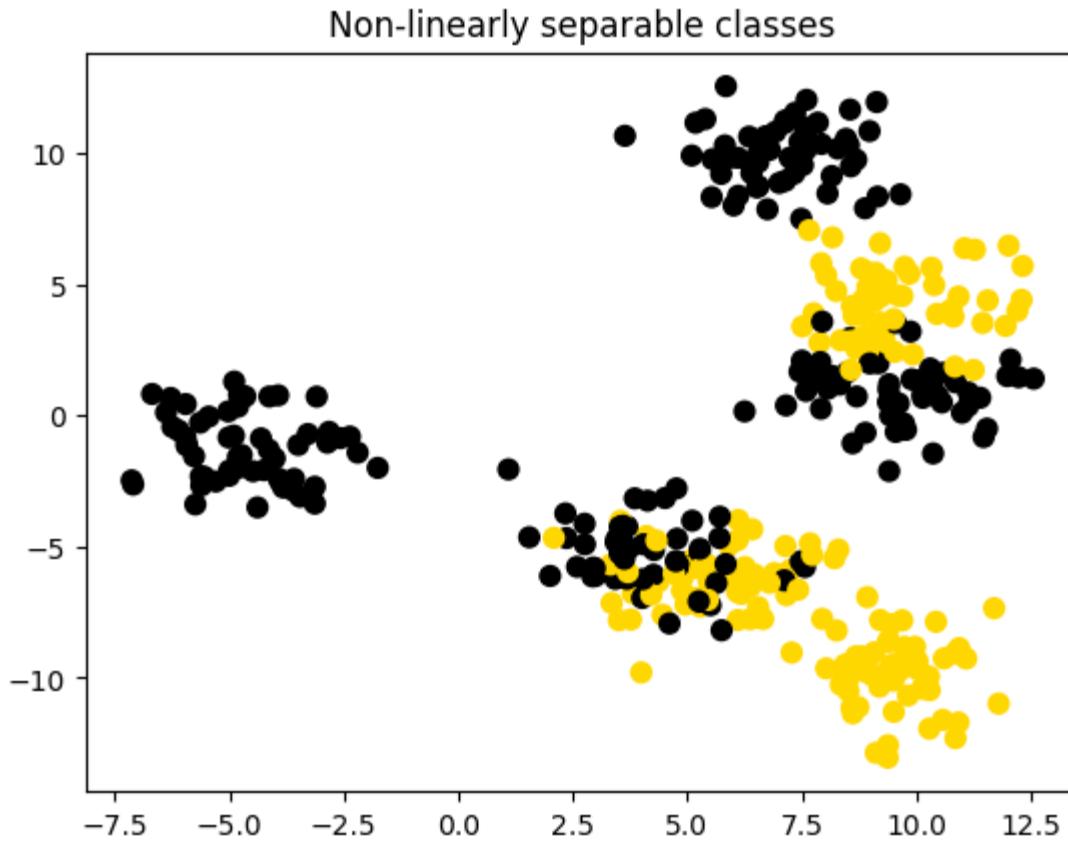
1. Manually create a data set that is not linearly separable

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 also saved as `data.csv` and is used for the R and Octave Neural networks to see how they perform on the same dataset.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib.colors
4 import sklearn.linear_model
5
6 from sklearn.model_selection import train_test_split
7 from sklearn.datasets import make_classification, make_blobs
8 from matplotlib.colors import ListedColormap
9 import sklearn
10 import sklearn.datasets
11
12 colors=['black', 'gold']
13 cmap = matplotlib.colors.ListedColormap(colors)
14 X, y = make_blobs(n_samples = 400, n_features = 2, centers = 7,
15 cluster_std = 1.3, random_state = 4)
16
17 #Create 2 classes
18 y=y.reshape(400,1)
19 y = y % 2
20
21 #Plot the figure
22 plt.figure()
23 plt.title('Non-linearly separable classes')
24
  
```

```
25 plt.scatter(X[:,0], X[:,1], c=y,
26             marker='o', s=50, cmap=cmap)
27 plt.savefig('fig1.png', bbox_inches='tight')
```



28

2. Logistic Regression

When classification with logistic regression is performed on the above data set, and the decision boundary is plotted it can be seen that logistic regression performs quite poorly

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib.colors
4 import sklearn.linear_model
5
6 from sklearn.model_selection import train_test_split
7 from sklearn.datasets import make_classification, make_blobs
8 from matplotlib.colors import ListedColormap
9 import sklearn
10 import sklearn.datasets
11
12 #from DLfunctions import plot_decision_boundary
13 execfile("./DLfunctions.py") # Since import does not work in Rmd!!!
14
15 colors=['black','gold']
16 cmap = matplotlib.colors.ListedColormap(colors)
17 X, y = make_blobs(n_samples = 400, n_features = 2, centers = 7,
18                   cluster_std = 1.3, random_state = 4)
```

```

20 #Create 2 classes
21 y=y.reshape(400,1)
22 y = y % 2
23
24 # Train the logistic regression classifier
25 clf = sklearn.linear_model.LogisticRegressionCV();
26 clf.fit(x, y);
27
28 # Plot the decision boundary for logistic regression
29 plot_decision_boundary_n(lambda x: clf.predict(x), X.T, y.T,"fig2.png")

```



30

5.1 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.

```

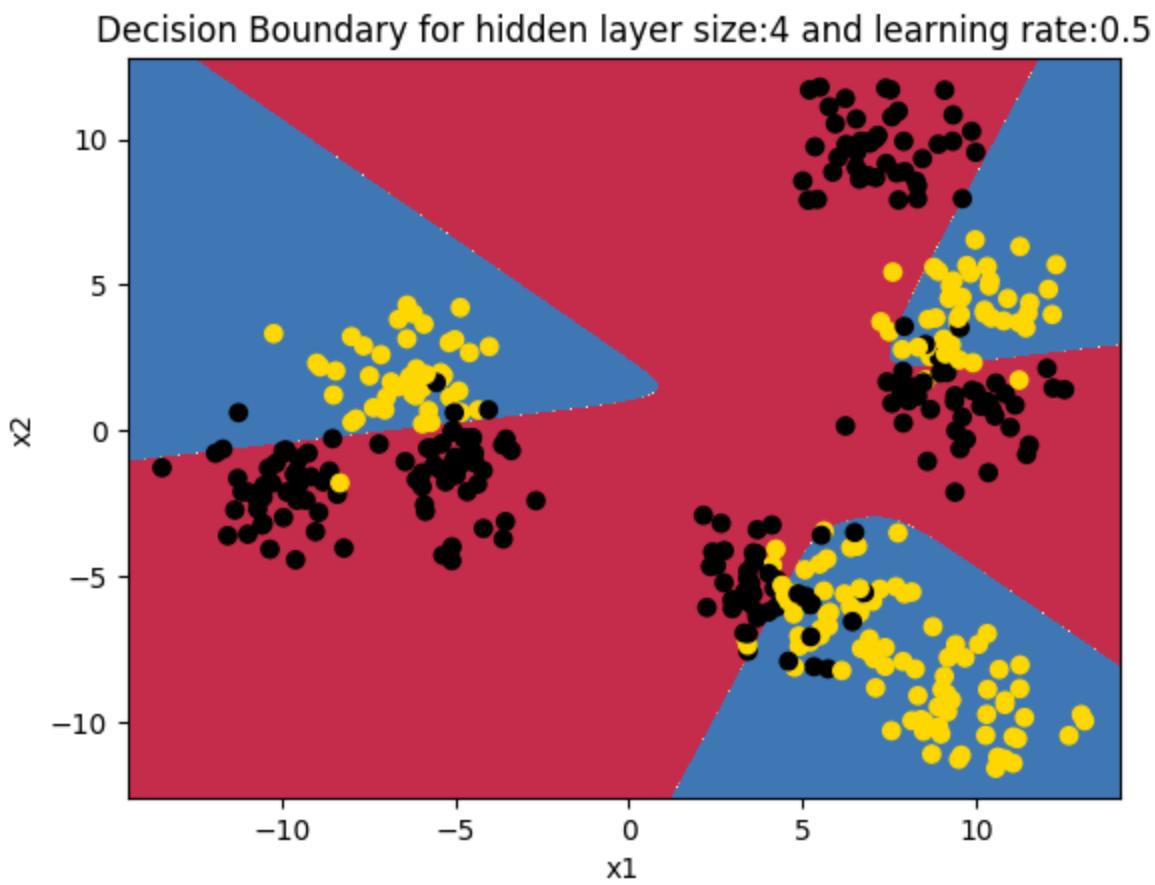
1 ## Random data set with 9 clusters
2 import numpy as np
3 import matplotlib
4 import matplotlib.pyplot as plt
5 import sklearn.linear_model
6 import pandas as pd
7
8 from sklearn.datasets import make_classification, make_blobs
9 execfile("./DLfunctions.py") # Since import does not work in Rmd!!!

```

```

10
11 X1, Y1 = make_blobs(n_samples = 400, n_features = 2, centers = 9,
12                      cluster_std = 1.3, random_state = 4)
13 #Create 2 classes
14 Y1=Y1.reshape(400,1)
15 Y1 = Y1 % 2
16 X2=X1.T
17 Y2=Y1.T
18
19 # Execute the 3 layer Neural Network
20 parameters,costs = computeNN(X2, Y2, numHidden = 4, learningRate=0.5,
21 numIterations = 10000)
22
23 #Plot the decision boundary
24 plot_decision_boundary(lambda x: predict(parameters, x.T), x2,
25 Y2,str(4),str(0.5),"fig3.png")
26 ## Cost after iteration 0: 0.692669
27 ## Cost after iteration 1000: 0.246650
28 ## Cost after iteration 2000: 0.227801
29 ## Cost after iteration 3000: 0.226809
30 ## Cost after iteration 4000: 0.226518
31 ## Cost after iteration 5000: 0.226331
32 ## Cost after iteration 6000: 0.226194
33 ## Cost after iteration 7000: 0.226085
34 ## Cost after iteration 8000: 0.225994
35 ## Cost after iteration 9000:
36 0.225915

```



37

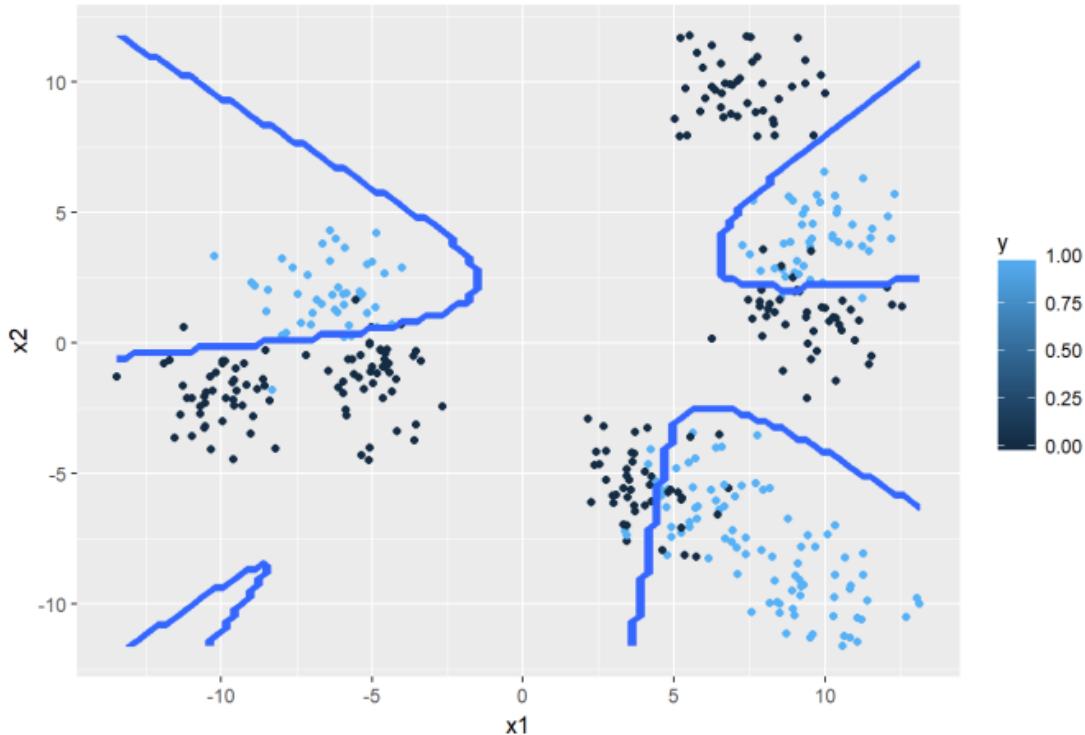
It can be seen the the 3 layer Neural Network with a single hidden layer is able to create the non-linear boundary separating the classes

5.2 The 3 layer Neural Network in R (vectorized)

This the dataset created by Python was saved as data.csv. The R code reads this data to see how R performs on the same dataset. The vectorized implementation of a Neural Network in R, 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 implementation 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

```
1 source("DLfunctions2_1.R")
2 z <- as.matrix(read.csv("data.csv", header=FALSE)) #
3 x <- z[,1:2]
4 y <- z[,3]
5 x1 <- t(x)
6 y1 <- t(y)
7
8 # Execute the 3 layer Neural Network
9 nn <- computeNN(x1, y1, 6, learningRate=0.1, numIterations=10000) # Good
10 ## [1] 0.7075341
11 ## [1] 0.2606695
12 ## [1] 0.2198039
13 ## [1] 0.2091238
14 ## [1] 0.211146
15 ## [1] 0.2108461
16 ## [1] 0.2105351
17 ## [1] 0.210211
18 ## [1] 0.2099104
19 ## [1] 0.2096437
20 ## [1] 0.209409
21
22 plotDecisionBoundary(z, nn, 6, 0.1)
```

Decision boundary for hidden layer size: 6 learning rate: 0.1

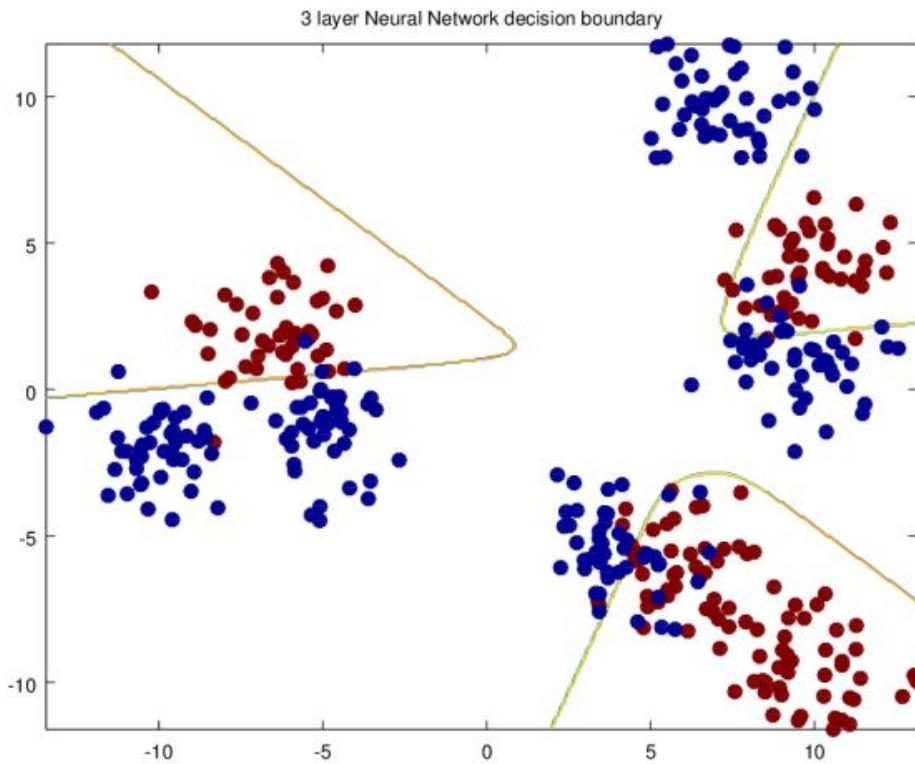


R is also able to create the non-linear boundary.

5.3 The 3 layer Neural Network in Octave (vectorized)

This uses the same dataset (data.csv) that was generated using Python code.

```
1 source("DL-function2.m")
2 # Read the data
3 data=csvread("data.csv");
4 X=data(:,1:2);
5 Y=data(:,3);
6
7 # Make sure that the model parameters are correct. Take the transpose of x &
8 Y
9
10 # Execute the 3 layer Neural Network and plot decision boundary
11 [w1,b1,w2,b2,costs]= computeNN(X', Y',4, learningRate=0.5, numIterations =
12 10000);
```



The above plot shows the non-linear boundary created by Octave.

6.1a Performance for different learning rates (Python)

```

1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6
7 from sklearn.datasets import make_classification, make_blobs
8 execfile("./DLfunctions.py") # Since import does not work in Rmd!!!
9
10 # Create data
11 X1, Y1 = make_blobs(n_samples = 400, n_features = 2, centers = 9,
12                      cluster_std = 1.3, random_state = 4)
13
14 #Create 2 classes
15 Y1=Y1.reshape(400,1)
16 Y1 = Y1 % 2
17 X2=X1.T
18 Y2=Y1.T
19
20 # Create a list of learning rates
21 learningRate=[0.5,1.2,3.0]
22 df=pd.DataFrame()
23
24 #Compute costs for each learning rate
25 for lr in learningRate:
26     # Execute the 3 layer Neural Network

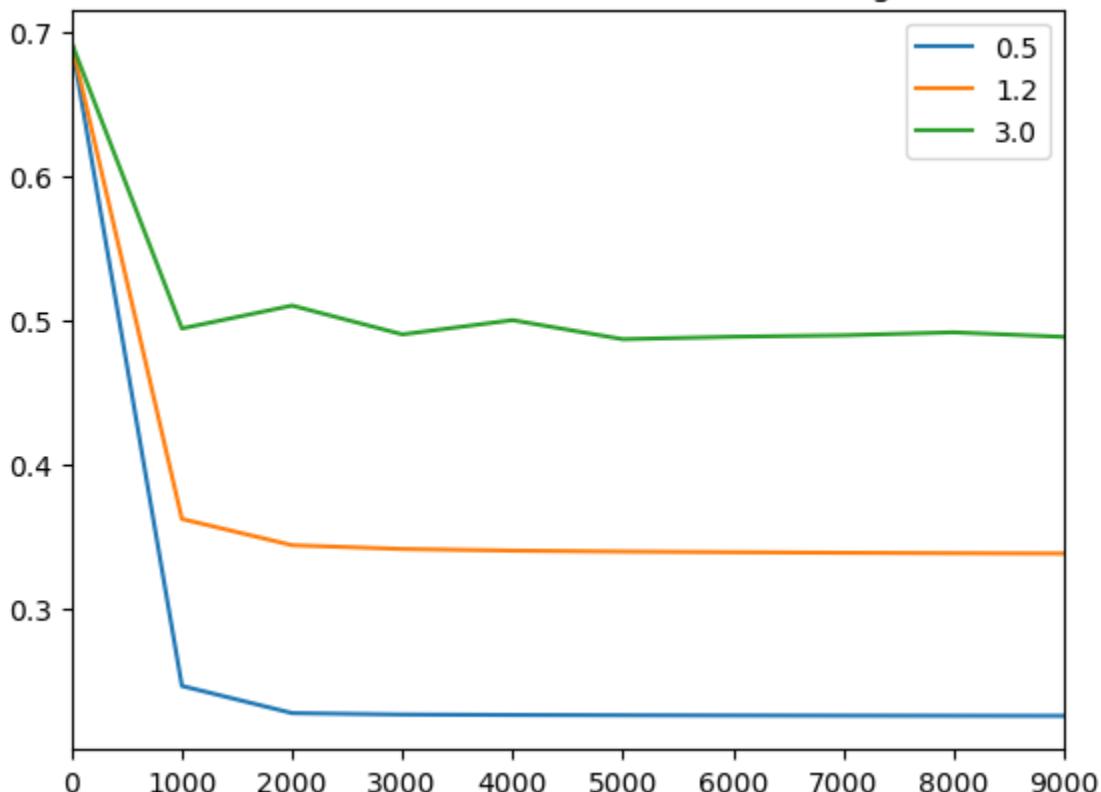
```

```

27     parameters,costs = computeNN(X2, Y2, numHidden = 4, learningRate=lr,
28     numIterations = 10000)
29     print(costs)
30     df1=pd.DataFrame(costs)
31     df=pd.concat([df,df1],axis=1)
32
33 #Set the iterations
34 iterations=[0,1000,2000,3000,4000,5000,6000,7000,8000,9000]
35
36 #Create data frame
37 #Set index
38 df1=df.set_index([iterations])
39 df1.columns=[0.5,1.2,3.0]
40
41 # Plot cost vs number of iterations for different learning rates
42 fig=df1.plot()
43 fig=plt.title("Cost vs No of Iterations for different learning rates")
44 plt.savefig('fig4.png', bbox_inches='tight')

```

Cost vs No of Iterations for different learning rates



6.1b Performance for different hidden units (Python)

```

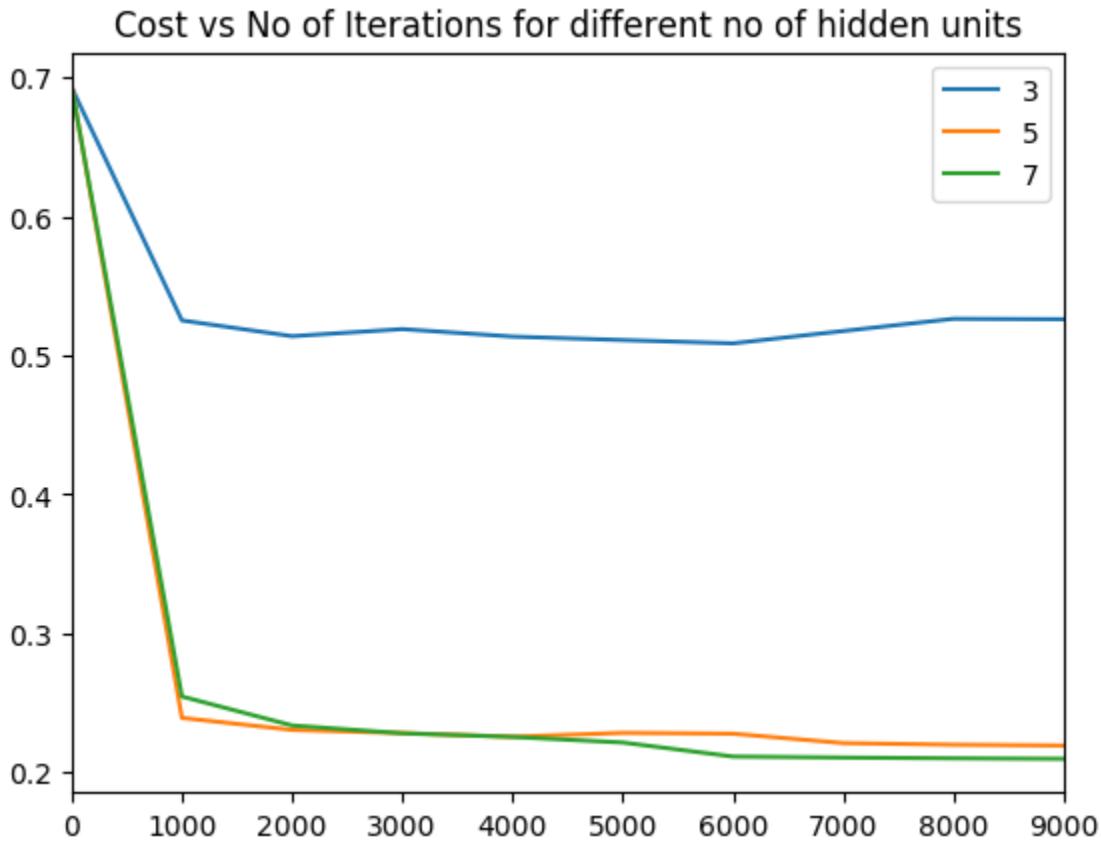
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6
7 from sklearn.datasets import make_classification, make_blobs
8 execfile("./DLfunctions.py") # Since import does not work in Rmd!!!

```

```

9 #Create data set
10 X1, Y1 = make_blobs(n_samples = 400, n_features = 2, centers = 9,
11                      cluster_std = 1.3, random_state = 4)
12
13 #Create 2 classes
14 Y1=Y1.reshape(400,1)
15 Y1 = Y1 % 2
16 X2=X1.T
17 Y2=Y1.T
18
19 # Make a list of hidden unis
20 numHidden=[3,5,7]
21 df=pd.DataFrame()
22 #Compute costs for different hidden units
23 for numHid in numHidden:
24     # Execute the 3 layer Neural Network
25     parameters,costs = computeNN(X2, Y2, numHidden = numHid, learningRate=1.2,
26     numIterations = 10000)
27     print(costs)
28     df1=pd.DataFrame(costs)
29     df=pd.concat([df,df1],axis=1)
30
31 #Set the iterations
32 iterations=[0,1000,2000,3000,4000,5000,6000,7000,8000,9000]
33 #Set index
34 df1=df.set_index([iterations])
35
36 #Plot the cost vs iterations for different number of hidden units
37 df1.columns=[3,5,7]
38 #Plot
39 fig=df1.plot()
40 fig=plt.title("Cost vs No of Iterations for different no of hidden units")
41 plt.savefig('fig5.png', bbox_inches='tight')

```



6.2a Performance for different learning rates (R)

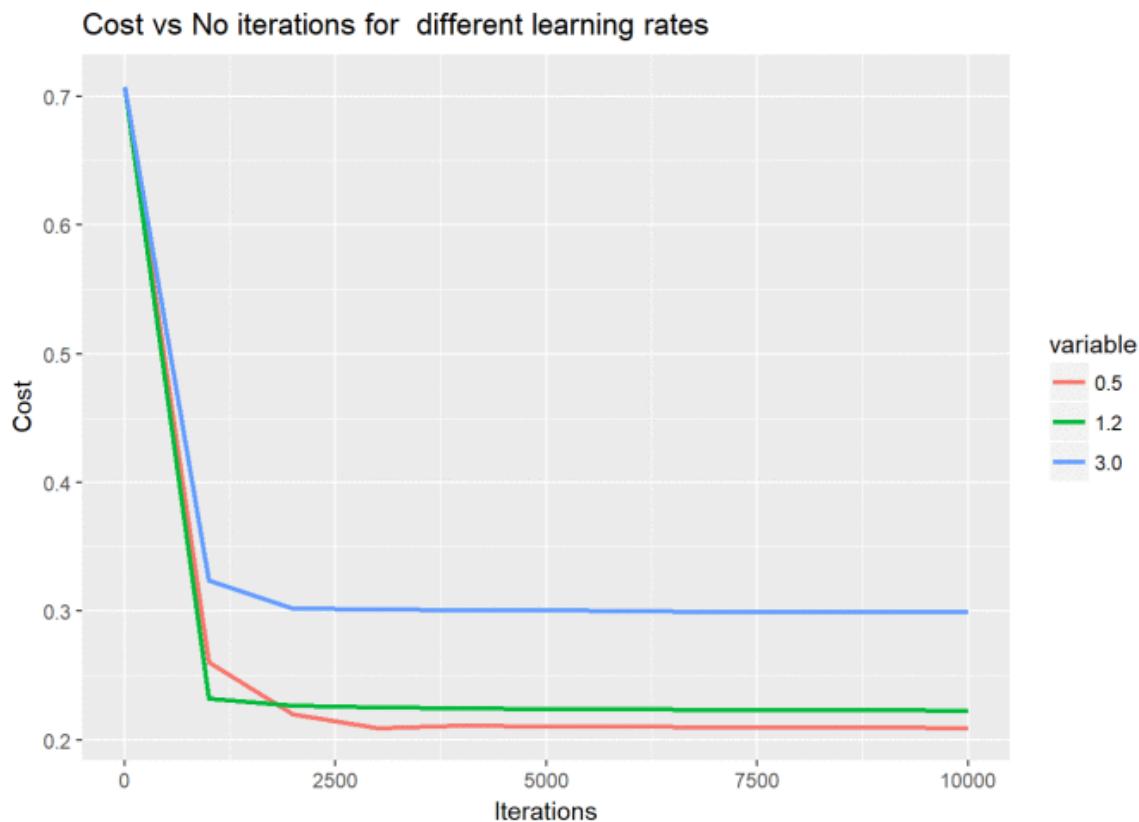
```

1 source("DLfunctions2_1.R")
2
3 # Read data
4 z <- as.matrix(read.csv("data.csv",header=FALSE)) #
5 x <- z[,1:2]
6 y <- z[,3]
7 x1 <- t(x)
8 y1 <- t(y)
9
10 #Loop through learning rates and compute costs
11 learningRate <- c(0.1,1.2,3.0)
12 df <- NULL
13 for(i in seq_along(learningRate)){
14   # Execute the 3 layer Neural Network
15   nn <- computeNN(x1, y1, 6,
16   learningRate=learningRate[i],numIterations=10000)
17   cost <- nn$costs
18   df <- cbind(df,cost)
19 }
20
21 #Create dataframe
22 df <- data.frame(df)
23 iterations=seq(0,10000,by=1000)
24 df <- cbind(iterations,df)
25 names(df) <- c("iterations","0.5","1.2","3.0")
26
27 # Reshape the data
28
```

```

29 library(reshape2)
30 df1 <- melt(df,id="iterations") # Melt the data
31
32 #Plot the cost vs iterations for different learning rates
33 ggplot(df1) + geom_line(aes(x=iterations,y=value,colour=variable),size=1) +
34   xlab("Iterations") +
35   ylab('Cost') + ggtitle("Cost vs No iterations for different learning
36 rates")

```



6.2b Performance for different hidden units (R)

```

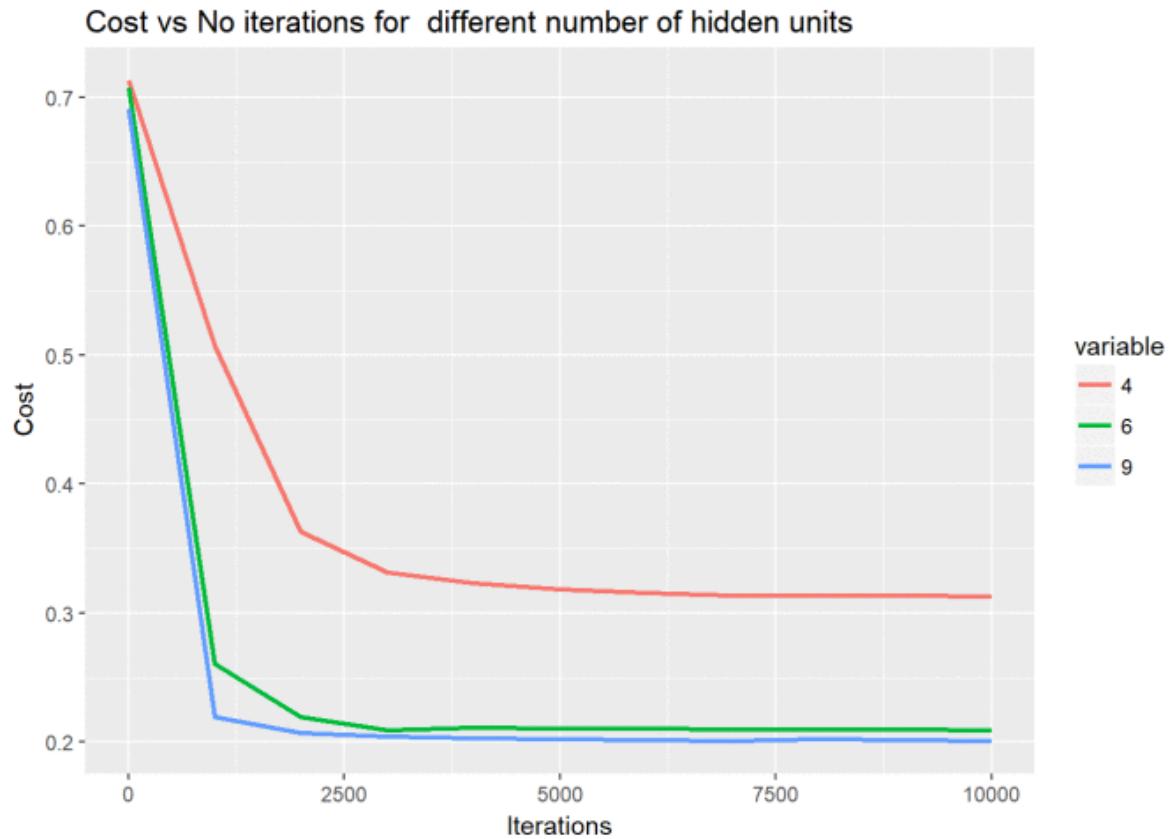
1 source("DLfunctions2_1.R")
2 # Loop through number of hidden units
3 numHidden <- c(4,6,9)
4 df <- NULL
5 for(i in seq_along(numHidden)){
6   # Execute the 3 layer Neural Network
7   nn <- computeNN(x1, y1, numHidden[i],
8   learningRate=0.1,numIterations=10000)
9   cost <- nn$costs
10  df <- cbind(df,cost)
11
12}
13
14 #Create a dataframe
15 df <- data.frame(df)
16 iterations=seq(0,10000,by=1000)
17 df <- cbind(iterations,df)
18 names(df) <- c("iterations","4","6","9")
19

```

```

20 #Reshape the dataframe
21 library(reshape2)
22 # Melt
23 df1 <- melt(df,id="iterations")
24
25 # Plot cost vs iterations for different number of hidden units
26 ggplot(df1) + geom_line(aes(x=iterations,y=value,colour=variable),size=1) +
27   xlab("Iterations") +
28   ylab('Cost') + ggtitle("Cost vs No iterations for different number of
29 hidden units")

```

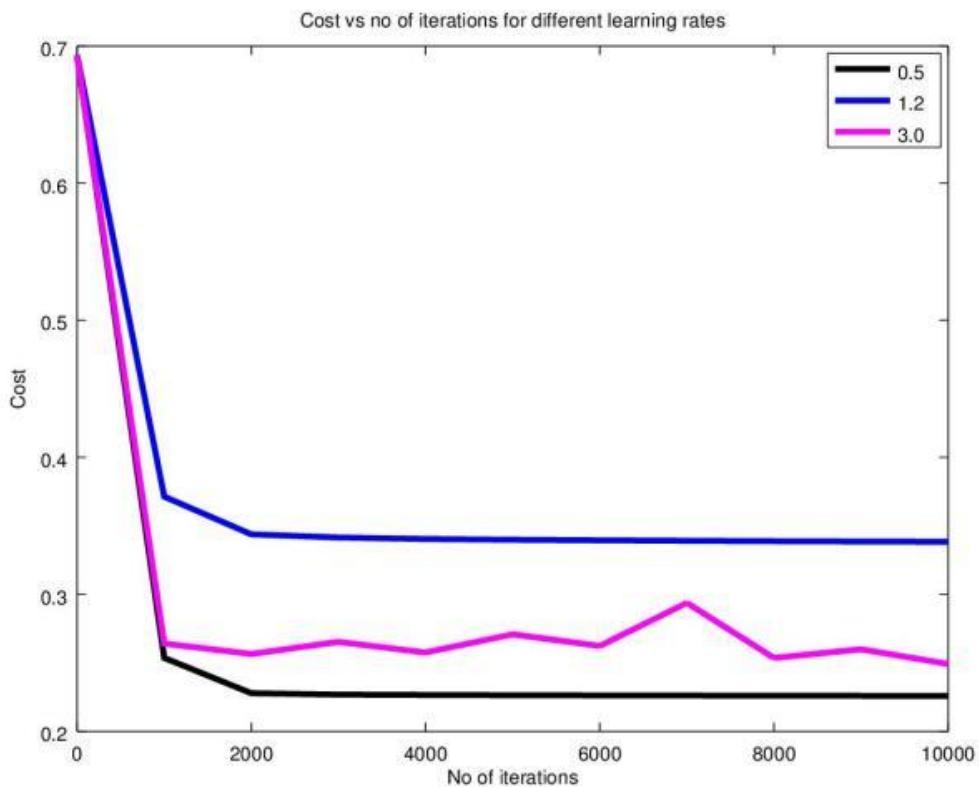


6.3a Performance of the Neural Network for different learning rates (Octave)

```

1 source("DL-function2.m")
2 #Plot cost vs iterations for different learning rates
3 plotLRCostVsIterations()
4 print -djph figa.jpg

```

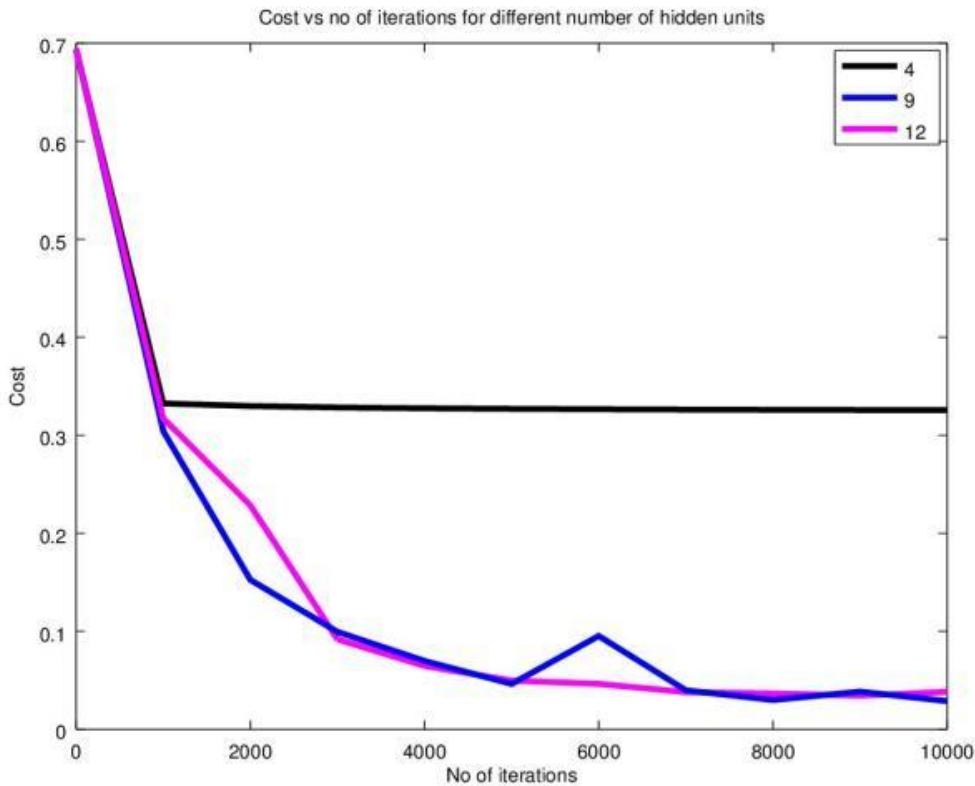


6.3b Performance of the Neural Network for different number of hidden units (Octave)

```

1 source("DL-function2.m")
2 #Plot cost vs Iterations for different number of hidden units
3 plotHiddenCostVsIterations()
4 print -djph figa.jpg

```



7. Turning the heat on the Neural Network

In this 2nd part, I create a central region of positives and an 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:** *R and Octave Neural Network constructions also use the same data set.*

8. Manually creating a circular central region

```

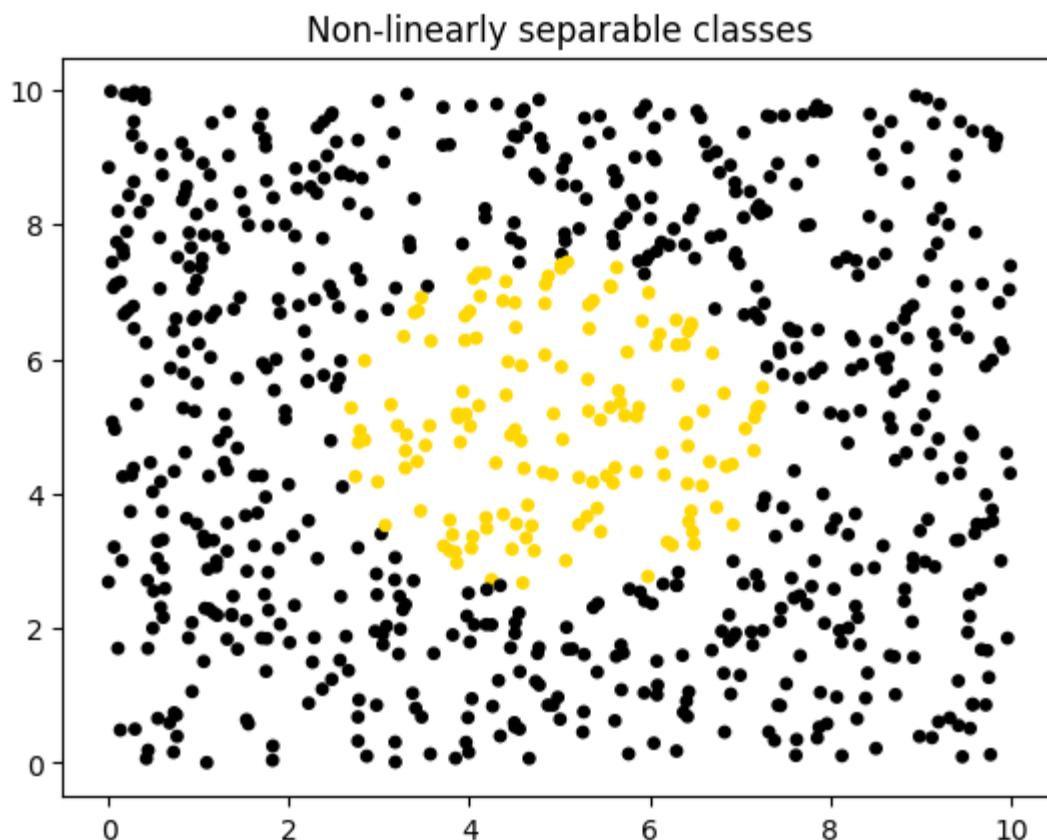
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib.colors
4 import sklearn.linear_model
5
6 from sklearn.model_selection import train_test_split
7 from sklearn.datasets import make_classification, make_blobs
8 from matplotlib.colors import ListedColormap
9 import sklearn
10 import sklearn.datasets
11
12 colors=['black','gold']
13 cmap = matplotlib.colors.ListedColormap(colors)
14 x1=np.random.uniform(0,10,800).reshape(800,1)
15 x2=np.random.uniform(0,10,800).reshape(800,1)
16 X=np.append(x1,x2,axis=1)
17 X.shape
18

```

```

19 # Create the data set with (x-a)^2 + (y-b)^2 = R^2
20 # Create a subset of values where squared is <0,4. Perform ravel() to flatten
21 this vector
22 a=(np.power(x[:,0]-5,2) + np.power(x[:,1]-5,2) <= 6).ravel()
23 Y=a.reshape(800,1)
24
25 cmap = matplotlib.colors.ListedColormap(colors)
26
27 #Plot the dataset
28 plt.figure()
29 plt.title('Non-linearly separable classes')
30 plt.scatter(x[:,0], x[:,1], c=Y,
31             marker= 'o', s=15,cmap=cmap)
32 plt.savefig('fig6.png', bbox_inches='tight')

```



8.1a Decision boundary with hidden units=4 and learning rate = 2.2 (Python)

With the above hyper-parameters, the decision boundary is triangular

```

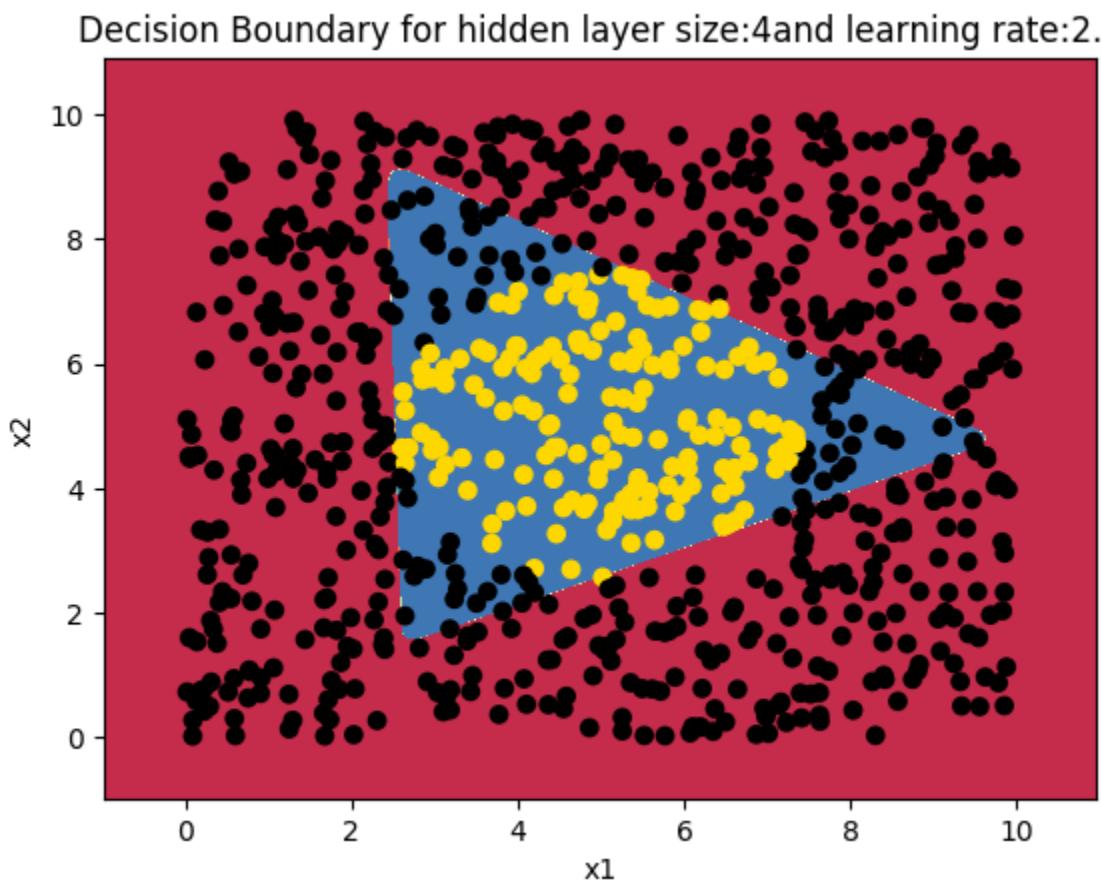
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib.colors
4 import sklearn.linear_model
5
6 execfile("./DLfunctions.py")
7 x1=np.random.uniform(0,10,800).reshape(800,1)
8 x2=np.random.uniform(0,10,800).reshape(800,1)
9 X=np.append(x1,x2,axis=1)

```

```

10 x.shape
11
12 # Create a subset of values where squared is <0,4. Perform ravel() to flatten
13 this vector
14 a=(np.power(x[:,0]-5,2) + np.power(x[:,1]-5,2) <= 6).ravel()
15 Y=a.reshape(800,1)
16
17 X2=X.T
18 Y2=Y.T
19
20 # Execute the 3 layer Neural network
21 parameters,costs = computeNN(X2, Y2, numHidden = 4, learningRate=2.2,
22 numIterations = 10000)
23
24 #Plot the decision boundary
25 plot_decision_boundary(lambda x: predict(parameters, x.T), X2,
26 Y2,str(4),str(2.2),"fig7.png")
27 ## Cost after iteration 0: 0.692836
28 ## Cost after iteration 1000: 0.331052
29 ## Cost after iteration 2000: 0.326428
30 ## Cost after iteration 3000: 0.474887
31 ## Cost after iteration 4000: 0.247989
32 ## Cost after iteration 5000: 0.218009
33 ## Cost after iteration 6000: 0.201034
34 ## Cost after iteration 7000: 0.197030
35 ## Cost after iteration 8000: 0.193507
36 ## Cost after iteration 9000: 0.191949

```

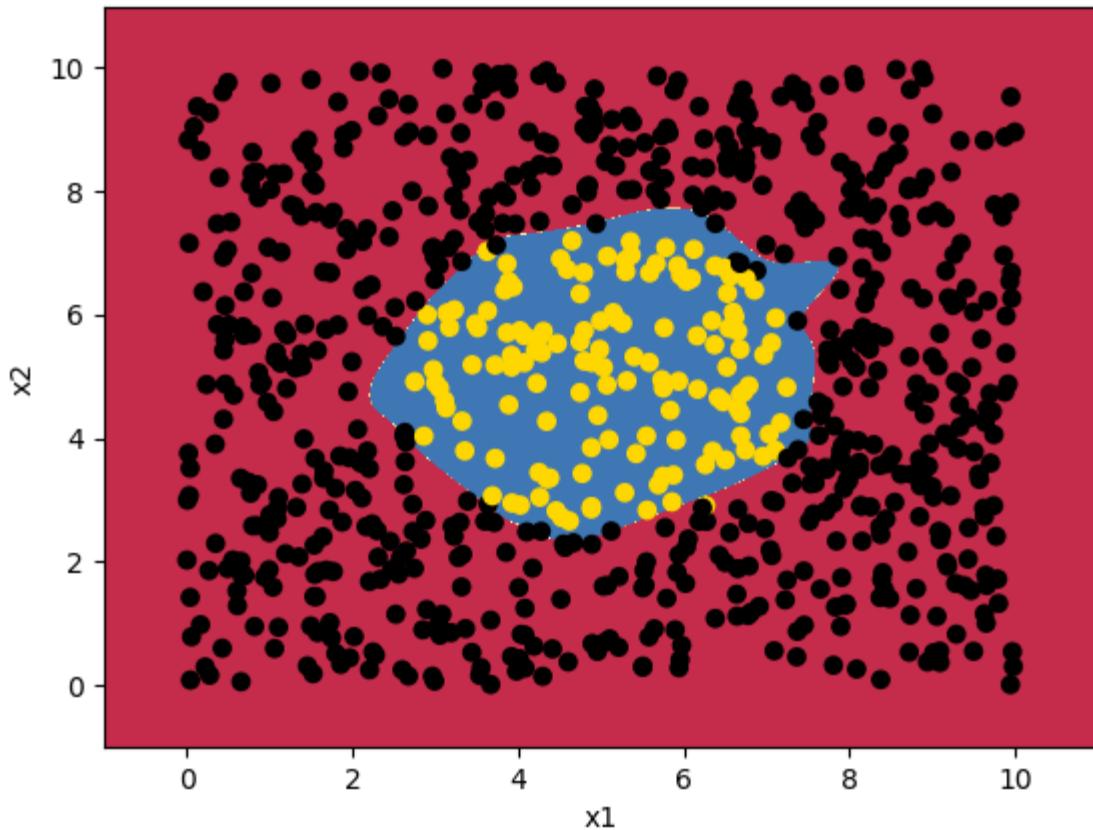


8.1b Decision boundary with hidden units=12 and learning rate = 2.2 (Python)

Increasing the number of hidden units makes the decision boundary much more circular

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import matplotlib.colors
4 import sklearn.linear_model
5 execfile("./DLfunctions.py")
6
7 x1=np.random.uniform(0,10,800).reshape(800,1)
8 x2=np.random.uniform(0,10,800).reshape(800,1)
9 X=np.append(x1,x2,axis=1)
10 X.shape
11
12 # Create a subset of values where squared is <0,4. Perform ravel() to flatten
13 # this vector
14 a=(np.power(X[:,0]-5,2) + np.power(X[:,1]-5,2) <= 6).ravel()
15 Y=a.reshape(800,1)
16 X2=X.T
17 Y2=Y.T
18
19 # Execute the 3 layer Neural network
20 parameters,costs = computeNN(X2, Y2, numHidden = 12, learningRate=2.2,
21 numIterations = 10000)
22
23 #Plot the decision boundary
24 plot_decision_boundary(lambda x: predict(parameters, x.T), X2,
25 Y2,str(12),str(2.2),"fig8.png")
26 ## Cost after iteration 0: 0.693291
27 ## Cost after iteration 1000: 0.383318
28 ## Cost after iteration 2000: 0.298807
29 ## Cost after iteration 3000: 0.251735
30 ## Cost after iteration 4000: 0.177843
31 ## Cost after iteration 5000: 0.130414
32 ## Cost after iteration 6000: 0.152400
33 ## Cost after iteration 7000: 0.065359
34 ## Cost after iteration 8000: 0.050921
35 ## Cost after iteration 9000: 0.039719
```

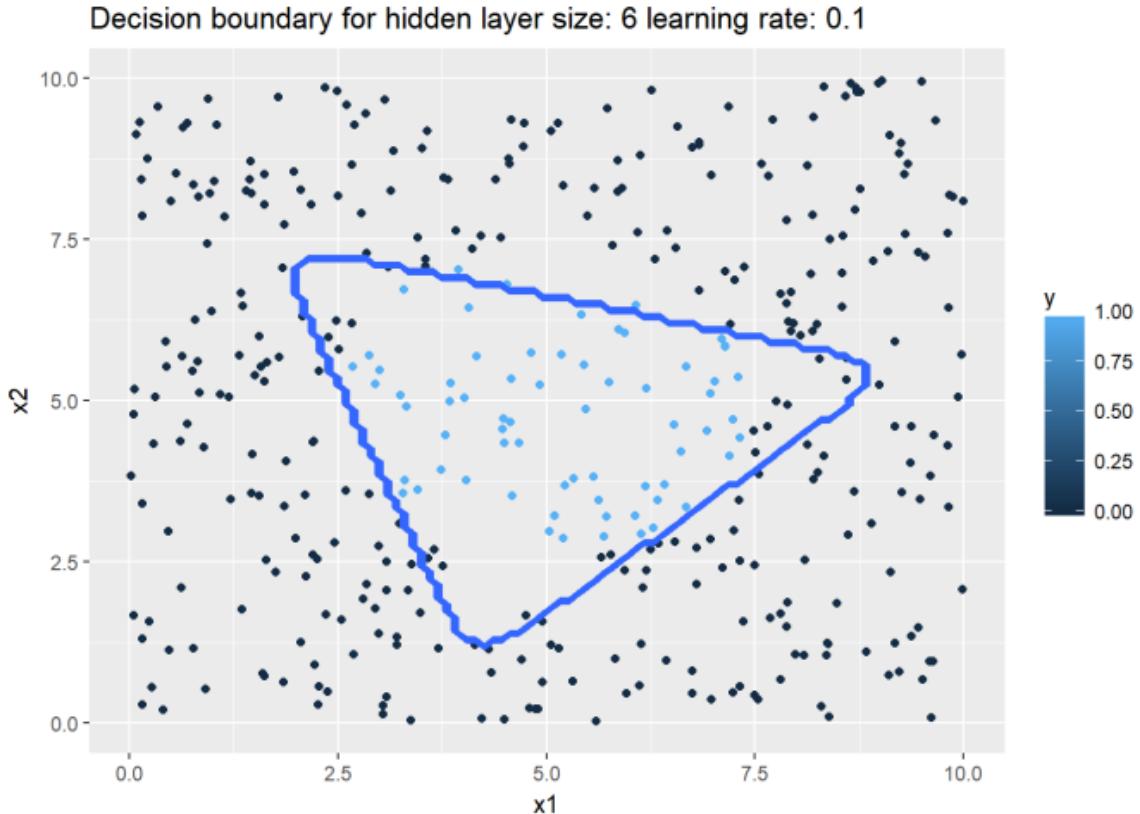
Decision Boundary for hidden layer size:12 and learning rate:2.2



8.2a 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 0.5, is also a triangular shape in R

```
1 source("DLfunctions2_1.R")
2 z <- as.matrix(read.csv("data1.csv", header=FALSE)) # N
3 x <- z[,1:2]
4 y <- z[,3]
5 x1 <- t(x)
6 y1 <- t(y)
7
8 # Execute the 3 layer Neural network
9 nn <- computeNN(x1, y1, 9, learningRate=0.5, numIterations=10000) # Triangular
10 ## [1] 0.8398838
11 ## [1] 0.3303621
12 ## [1] 0.3127731
13 ## [1] 0.3012791
14 ## [1] 0.3305543
15 ## [1] 0.3303964
16 ## [1] 0.2334615
17 ## [1] 0.1920771
18 ## [1] 0.2341225
19 ## [1] 0.2188118
20 ## [1] 0.2082687
21
22 #Plot the decision boundary
23 plotDecisionBoundary(z,nn,6,0.1)
```



8.2b Decision boundary with hidden units=8 and learning rate = 0.1 (R)

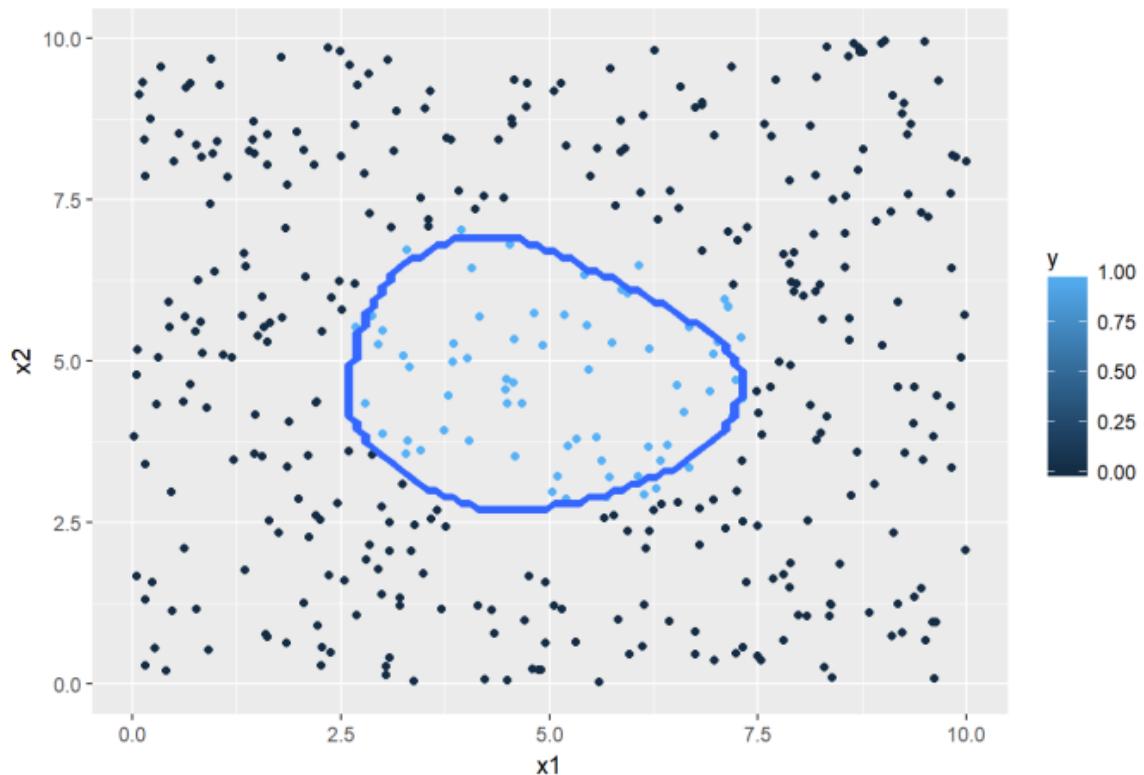
With 8 hidden units the decision boundary is quite circular.

```

1 source("DLfunctions2_1.R")
2 z <- as.matrix(read.csv("data1.csv",header=FALSE)) # N
3 x <- z[,1:2]
4 y <- z[,3]
5 x1 <- t(x)
6 y1 <- t(y)
7
8 # Execute the 3 layer Neural network
9 nn <- computeNN(x1, y1, 8, learningRate=0.1,numIterations=10000) # Hemisphere
10 ## [1] 0.7273279
11 ## [1] 0.3169335
12 ## [1] 0.2378464
13 ## [1] 0.1688635
14 ## [1] 0.1368466
15 ## [1] 0.120664
16 ## [1] 0.111211
17 ## [1] 0.1043362
18 ## [1] 0.09800573
19 ## [1] 0.09126161
20 ## [1] 0.0840379
21
22 #Plot the decision boundary
plotDecisionBoundary(z,nn,8,0.1)

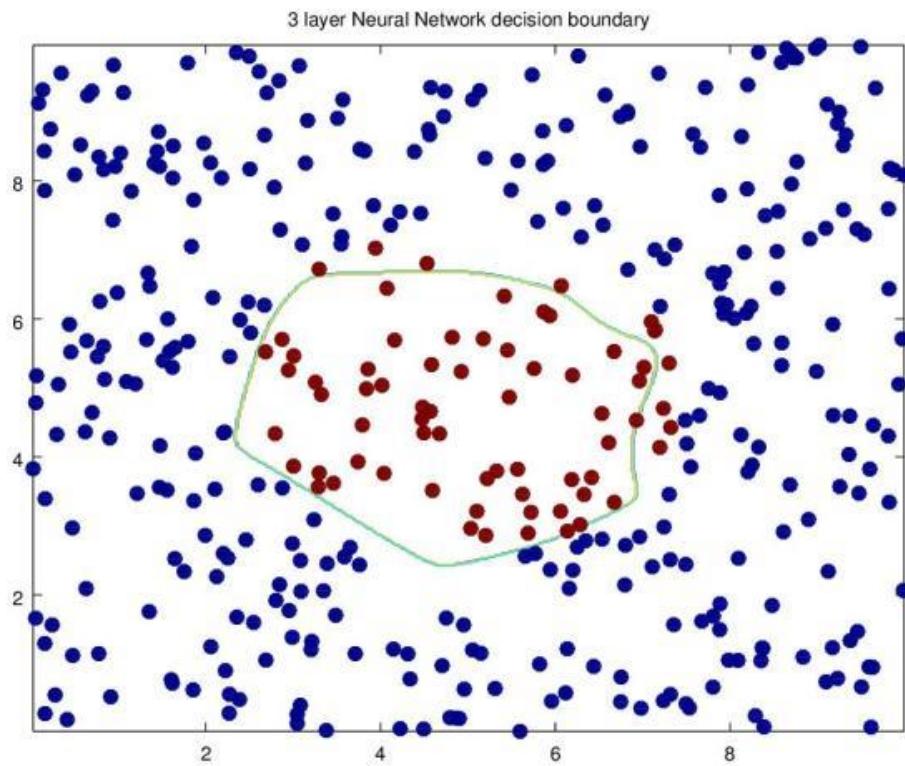
```

Decision boundary for hidden layer size: 8 learning rate: 0.1



8.3a Decision boundary with hidden units=12 and learning rate = 1.5 (Octave)

```
1 source("DL-function2.m")
2 # Read the data
3 data=csvread("data1.csv");
4 X=data(:,1:2);
5 Y=data(:,3);
6
7 # Make sure that the model parameters are correct. Take the transpose of x &
8 Y
9 # Execute the 3 layer Neural network
10 [w1,b1,w2,b2,costs]= computeNN(X', Y',12, learningRate=1.5, numIterations =
11 10000);
12
13 #Plot the decision boundary
14 plotDecisionBoundary(data, w1,b1,w2,b2)
15 print -djpeg figure.jpg
```



9. Conclusion

This chapter 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.

3. Building a L-Layer Deep Learning Network

“Once upon a time, I, Chuang Tzu, dreamt I was a butterfly, fluttering hither and thither, to all intents and purposes a butterfly. I was conscious only of following my fancies as a butterfly, and was unconscious of my individuality as a man. Suddenly, I awoke, and there I lay, myself again. Now I do not know whether I was then a man dreaming I was a butterfly, or whether I am now a butterfly dreaming that I am a man.”

from The Brain: The Story of you – David Eagleman

“Thought is a great big vector of neural activity”

Prof Geoffrey Hinton

1. Introduction

This is the third chapter in my series on Deep Learning from first principles in Python, R and Octave. In this third chapter, I implement a multi-layer, Deep Learning (DL) network of arbitrary depth (any number of hidden layers), and arbitrary height (any number of activation units in each hidden layer). The implementation in the 3rd part is for an L-layer Deep Network, but without any regularization, early stopping, momentum or learning rate adaptation techniques. However even this barebones multi-layer DL, is a handful and has enough hyper-parameters to fine-tune and adjust. The implementation of the vectorized L-layer Deep Learning network in Python, R and Octave is quite challenging, as we need to keep track of the indices, layer number and matrix dimensions. Some challenges because of the differences in the languages

1. Python and Octave allow multiple return values to be unpacked in a single statement. With R, unpacking multiple return values from a list, requires the entire list to be returned, and then unpacked separately. I did see that there is a package gsubfn (<https://stackoverflow.com/questions/1826519/how-to-assign-from-a-function-which-returns-more-than-one-value>), which does this.
2. Python and R can save and return dissimilar elements from functions using dictionaries or lists respectively. However, there is no real equivalent in Octave. The closest I got to this functionality in Octave, was the ‘cell array’. But, the cell array can be accessed only by the index, and not with the key as in a Python dictionary or R list. This makes things just a bit more difficult in Octave.
3. Python and Octave include implicit broadcasting. In R, broadcasting is not implicit, but R has a nifty function, the sweep(), with which we can broadcast either by columns or by rows
4. The closest equivalent of Python’s dictionary, or R’s list, in Octave is the cell array. However I had to manage separate cell arrays for weights and biases and during gradient descent and separate gradients dW and dB
5. In Python, the rank-1 numpy arrays can be annoying at times. This issue is not present in R and Octave.

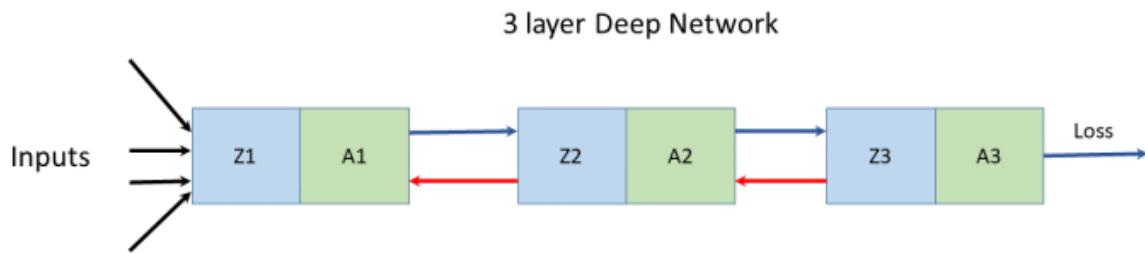
The current vectorized implementation supports the relu, sigmoid and tanh activation functions.

While testing with different hyper-parameters namely i) the number of hidden layers, ii) the number of activation units in each layer, iii) the activation function and iv) the number iterations, I found the L-layer Deep Learning Network to be very sensitive to these hyper-parameters. It is not easy to tune the parameters. Adding more hidden layers, or more units per layer, does not help always, and sometimes results in gradient descent getting stuck in some local minima. It does take a fair amount of trial and error and very close observation on how the Deep Learning network performs for logical changes. We then can zero in on the most the optimal solution. A much more detailed approach for tuning hyper-paramaters is discussed in chapter 8. The Python,R and Octave functions used in this chapter are implemented in Appendix 3 - Building a L- Layer Deep Learning Network

Feel free to download/fork my code from Github DeepLearningFromFirstPrinciples (<https://github.com/tvganesh/DeepLearningFromFirstPrinciples/tree/master/Chap3-L-LayerDeepLearningNetwork>) and play around with the hyper-parameters for your own problems.

2. Derivation of a Multi Layer Deep Learning Network

Let's take a simple 3 layer Neural network with 3 hidden layers and an output layer



In the forward propagation cycle the equations are

$$\begin{aligned} Z_1 &= W_1 A_0 + b_1 \text{ and } A_1 = g(Z_1) \\ Z_2 &= W_2 A_1 + b_2 \text{ and } A_2 = g(Z_2) \\ Z_3 &= W_3 A_2 + b_3 \text{ and } A_3 = g(Z_3) \end{aligned}$$

The loss function is given by

$$L = -(y \log A_3 + (1 - y) \log(1 - A_3))$$

and $dL/dA_3 = -(Y/A_3 + (1 - Y)/(1 - A_3))$

For a binary classification, the output activation function is the sigmoid function given by $A_3 = 1/(1 + e^{-Z_3})$. Hence

$$dA_3/dZ_3 = A_3(1 - A_3) \text{ see equation (2) in chapter 1}$$

$$\partial L/\partial Z_3 = \partial L/\partial A_3 * \partial A_3/\partial Z_3 = A_3 - Y \text{ see equation (f) in chapter 1}$$

and since

$$\partial L/\partial A_2 = \partial L/\partial Z_3 * \partial Z_3/\partial A_2 = (A_3 - Y) * W_3 \text{ because } \partial Z_3/\partial A_2 = W_3 \quad -(1a)$$

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

$$\partial L / \partial A_1 = \partial L / \partial Z_2 * \partial Z_2 / \partial A_1 = \partial L / \partial Z_2 * W_2 \quad -(2a)$$

$$\partial L / \partial Z_1 = \partial L / \partial A_1 * \partial A_1 / \partial Z_1 = \partial L / \partial A_1 * W_2 * g'(Z_1) \quad -(2b)$$

$$\partial L / \partial W_1 = \partial L / \partial Z_1 * A_0 \quad -(2c)$$

$$\partial L / \partial b_1 = \partial L / \partial Z_1 \quad -(2d)$$

Inspecting the above equations (1a – 1d & 2a-2d), we can discern a pattern in these equations and we can write the equations for any layer 'l' as

$$Z_l = W_l A_{l-1} + b_l \quad \text{and} \quad A_l = g(Z_l)$$

The equations for the backward propagation have the general form

$$\partial L / \partial A_l = \partial L / \partial Z_{l+1} * W^{l+1}$$

$$\partial L / \partial Z_l = \partial L / \partial A_l * g'(Z_l)$$

$$\partial L / \partial W_l = \partial L / \partial Z_l * A^{l-1}$$

$$\partial L / \partial b_l = \partial L / \partial Z_l$$

Other important results

The derivatives of the activation functions in the implemented Deep Learning network

$$g(z) = \text{sigmoid}(z) = 1 / (1 + e^{-z}) = a \quad g'(z) = a(1-a) \quad \text{See equation (2) chapter 1}$$

$$g(z) = \tanh(z) = a \quad g'(z) = 1 - a^2 \quad \text{See equation (d) chapter 2}$$

$$g(z) = \text{relu}(z) = z \quad \text{when } z > 0 \text{ and } 0 \text{ when } z \leq 0$$

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 Deep Learning network with the layer dimensions. So, for example layersDimension Vector 'V' of length L indicating 'L' layers where

$$V \text{ (in Python)} = [v_0, v_1, v_2 \dots v_{L-1}]$$

$$V \text{ (in R)} = c(v_1, v_2, v_3 \dots v_L)$$

$$V \text{ (in Octave)} = [v_1 \ v_2 \ v_3 \dots v_L]$$

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

2.1a 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

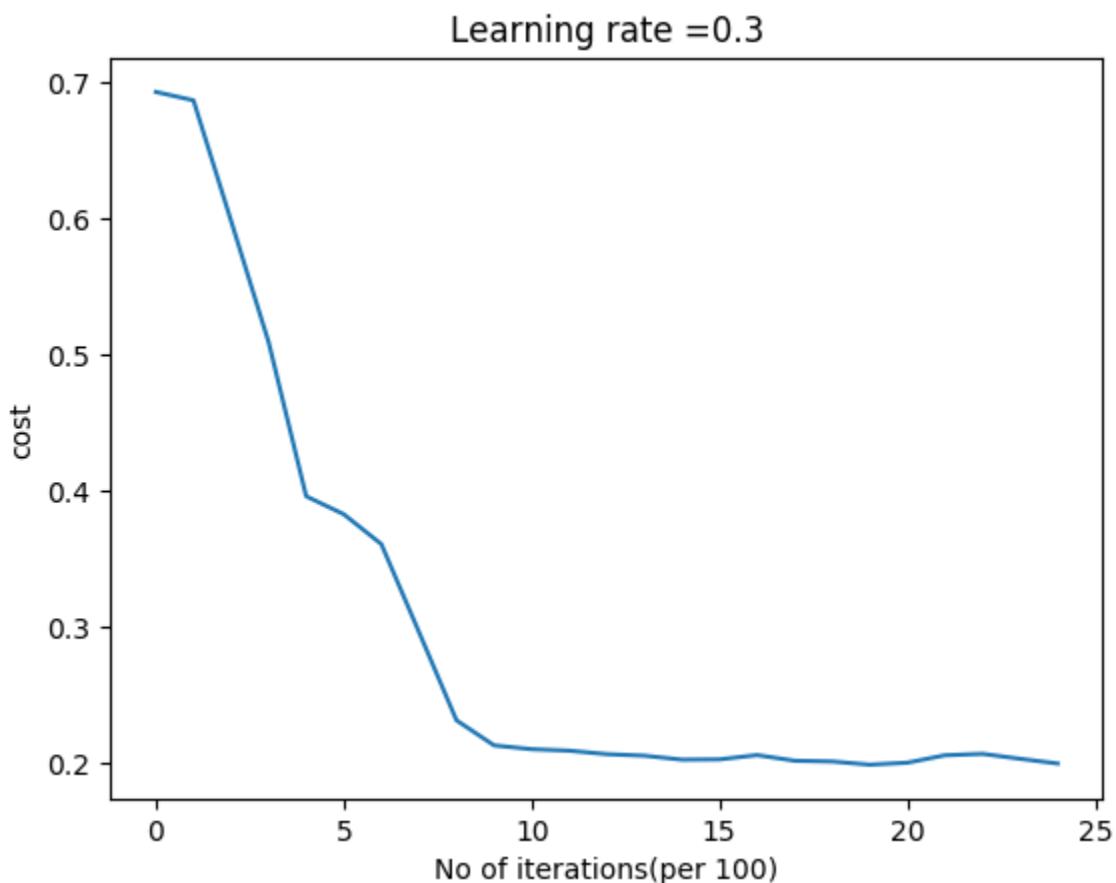
```

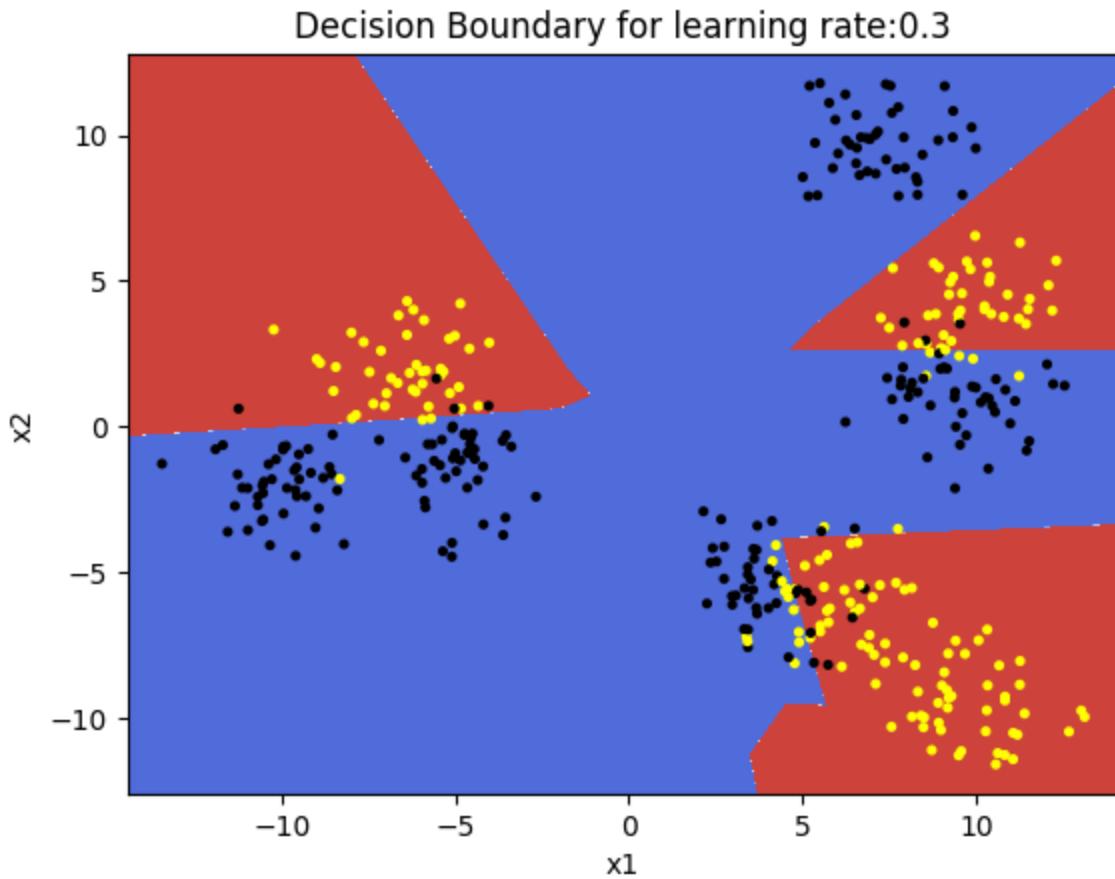
1 import os
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import matplotlib.colors
5 import sklearn.linear_model
6
7 from sklearn.model_selection import train_test_split
8 from sklearn.datasets import make_classification, make_blobs
9 from matplotlib.colors import ListedColormap
10 import sklearn
11 import sklearn.datasets
12
13 #from DLfunctions import plot_decision_boundary
14 execfile("./DLfunctions34.py") #
15
16 # Create clusters of 2 classes
17 X1, Y1 = make_blobs(n_samples = 400, n_features = 2, centers = 9,
18 cluster_std = 1.3, random_state = 4)
19
20 #Create 2 classes
21 Y1=Y1.reshape(400,1)
22 Y1 = Y1 % 2
23 X2=X1.T
24 Y2=Y1.T
25
26 # Set the dimensions of DL Network
27 # Below we have
28 # 2 - 2 input features
29 # 9,9 - 2 hidden layers with 9 activation units per layer and
30 # 1 - 1 sigmoid activation unit in the output layer as this is a binary
31 classification
32 # The activation in the hidden layer is the 'relu' specified in
33 L_Layer_DeepModel
34
35 layersDimensions = [2, 9, 9,1] # 4-layer model
36
37 # Execute the L-layer Deep Learning network
38 # Hidden layer activation unit - relu
```

```

39 # Learning rate = 0.3
40 parameters = L_Layer_DeepModel(x2, Y2,
41 layersDimensions,hiddenActivationFunc='relu', learning_rate =
42 0.3,num_iterations = 2500, fig="fig1.png")
43
44 #Plot the decision boundary
45 plot_decision_boundary(lambda x: predict(parameters, x.T),
46 x2,Y2,str(0.3),"fig2.png")
47
48 # Compute the confusion matrix
49 yhat = predict(parameters,X2)
50 from sklearn.metrics import confusion_matrix
51 a=confusion_matrix(Y2.T,yhat.T)
52
53 #Print the output
54 from sklearn.metrics import accuracy_score, precision_score, recall_score,
55 f1_score
56 print('Accuracy: {:.2f}'.format(accuracy_score(Y2.T, yhat.T)))
57 print('Precision: {:.2f}'.format(precision_score(Y2.T, yhat.T)))
58 print('Recall: {:.2f}'.format(recall_score(Y2.T, yhat.T)))
59 print('F1: {:.2f}'.format(f1_score(Y2.T, yhat.T)))
60 ## Accuracy: 0.90
61 ## Precision: 0.91
62 ## Recall: 0.87
63 ## F1: 0.89

```





2.1b Classification with Multi-layer Deep Learning Network – Relu activation(R)

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

```

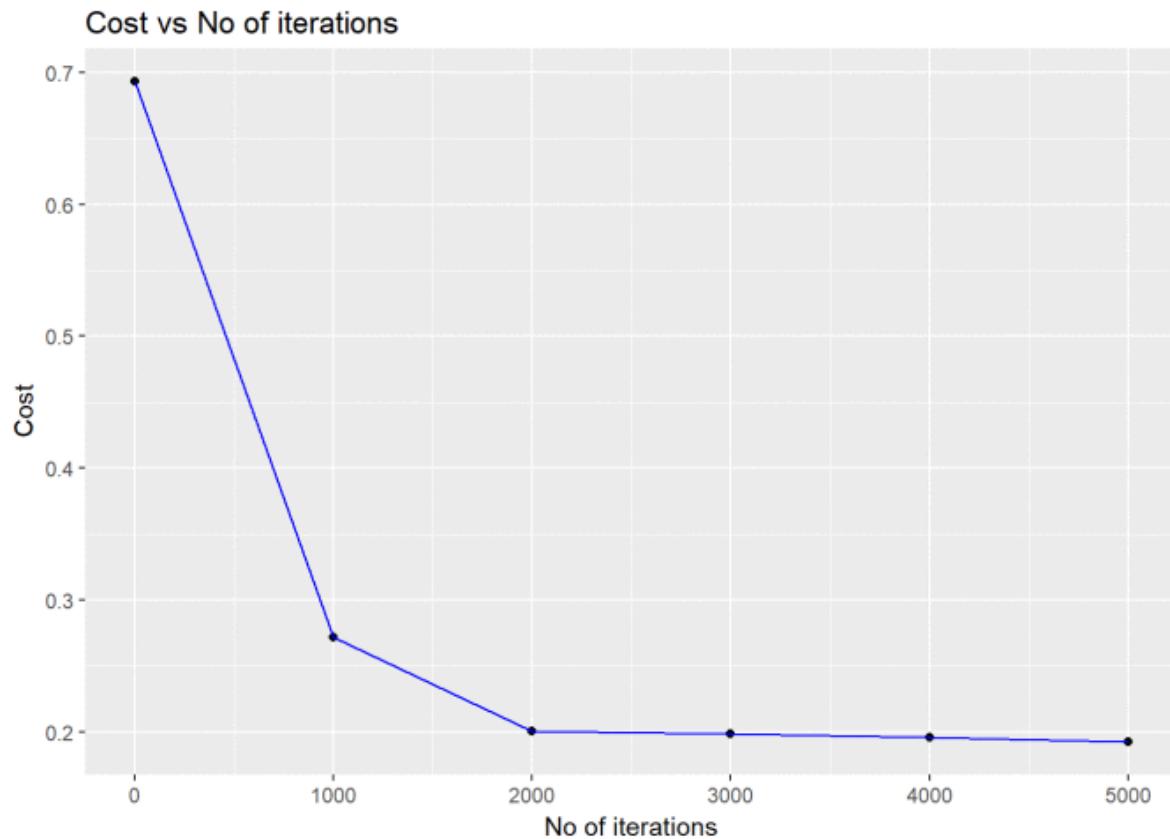
1 library(ggplot2)
2
3 # Read the data
4 z <- as.matrix(read.csv("data.csv", header=FALSE))
5 x <- z[,1:2]
6 y <- z[,3]
7 X1 <- t(x)
8 Y1 <- t(y)
9
10 # Set the dimensions of the Deep Learning network
11 # 2 - No of input features
12 # 9,9 - 2 hidden layers with 9 activation units
13 # 1 - 1 sigmoid activation unit at output layer
14 layersDimensions = c(2, 9, 9,1)
15
16 # Execute the L-layer Deep Learning Neural Network
17 # Hidden layer activation unit - relu
18 # Learning rate - 0.3
19 retvals = L_Layer_DeepModel(X1, Y1, layersDimensions,
20                             hiddenActivationFunc='relu',

```

```

21                                         learningRate = 0.3,
22                                         numIterations = 5000,
23                                         print_cost = True)
24
25
26 library(ggplot2)
27 source("DLfunctions33.R")
28
29 # Get the computed costs
30 costs <- retvals[['costs']]
31 # Create a sequence of iterations
32 numIterations=5000
33 iterations <- seq(0,numIterations,by=1000)
34 df <- data.frame(iterations,costs)
35
36 # Plot the Costs vs number of iterations
37 ggplot(df,aes(x=iterations,y=costs)) + geom_point() +geom_line(color="blue")
38 + xlab('No of iterations') + ylab('Cost') + ggtitle("Cost vs No of
39 iterations")

```

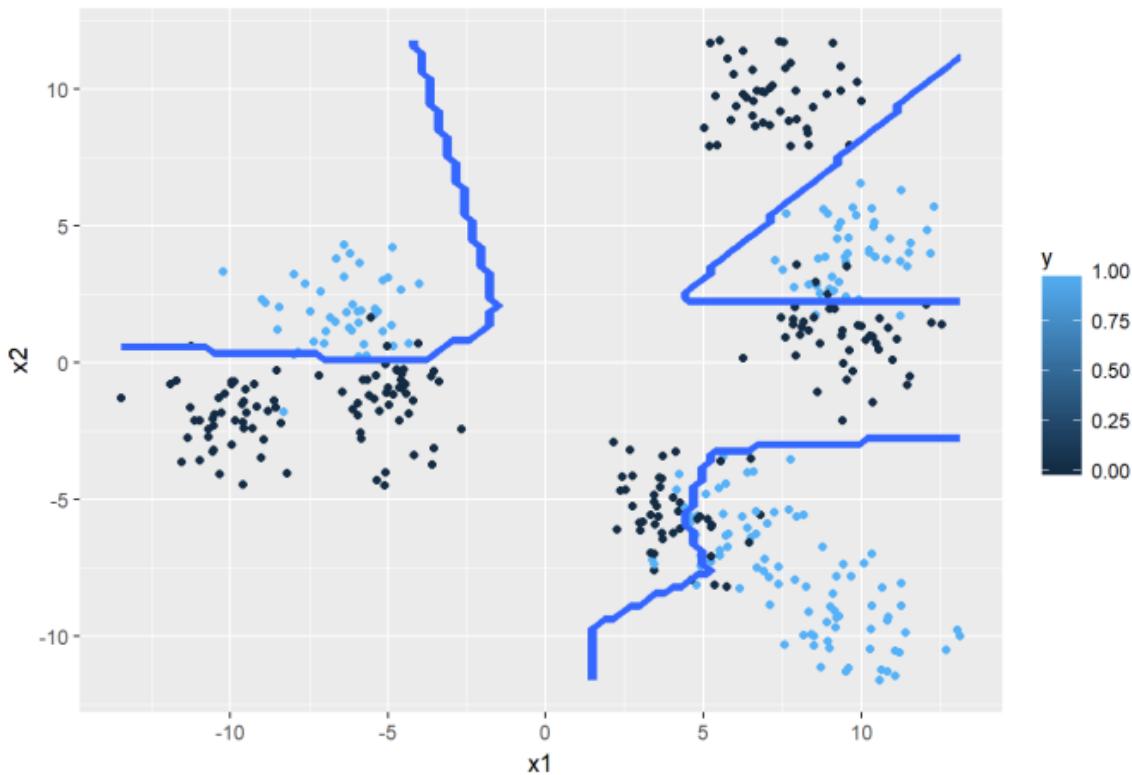


```

1 # Plot the decision boundary
2 plotDecisionBoundary(z,retvals,hiddenActivationFunc="relu",0.3)

```

Decision boundary for learning rate: 0.3



```

1 library(caret)
2 # Predict the output for the data values
3 yhat <- predict(retvals$parameters,x1,hiddenActivationFunc="relu")
4 yhat[yhat==FALSE]=0
5 yhat[yhat==TRUE]=1
6 # Compute the confusion matrix
7 confusionMatrix(yhat,Y1)
8 ## Confusion Matrix and Statistics
9 ##
10 ##          Reference
11 ## Prediction  0   1
12 ##           0 201  10
13 ##           1  21 168
14 ##
15 ##                               Accuracy : 0.9225
16 ##                               95% CI : (0.8918, 0.9467)
17 ## No Information Rate : 0.555
18 ## P-Value [Acc > NIR] : < 2e-16
19 ##
20 ##                               Kappa : 0.8441
21 ## Mcnemar's Test P-Value : 0.07249
22 ##
23 ##                               Sensitivity : 0.9054
24 ##                               Specificity : 0.9438
25 ## Pos Pred Value : 0.9526
26 ## Neg Pred Value : 0.8889
27 ## Prevalence : 0.5550
28 ## Detection Rate : 0.5025
29 ## Detection Prevalence : 0.5275
30 ## Balanced Accuracy : 0.9246
31 ##

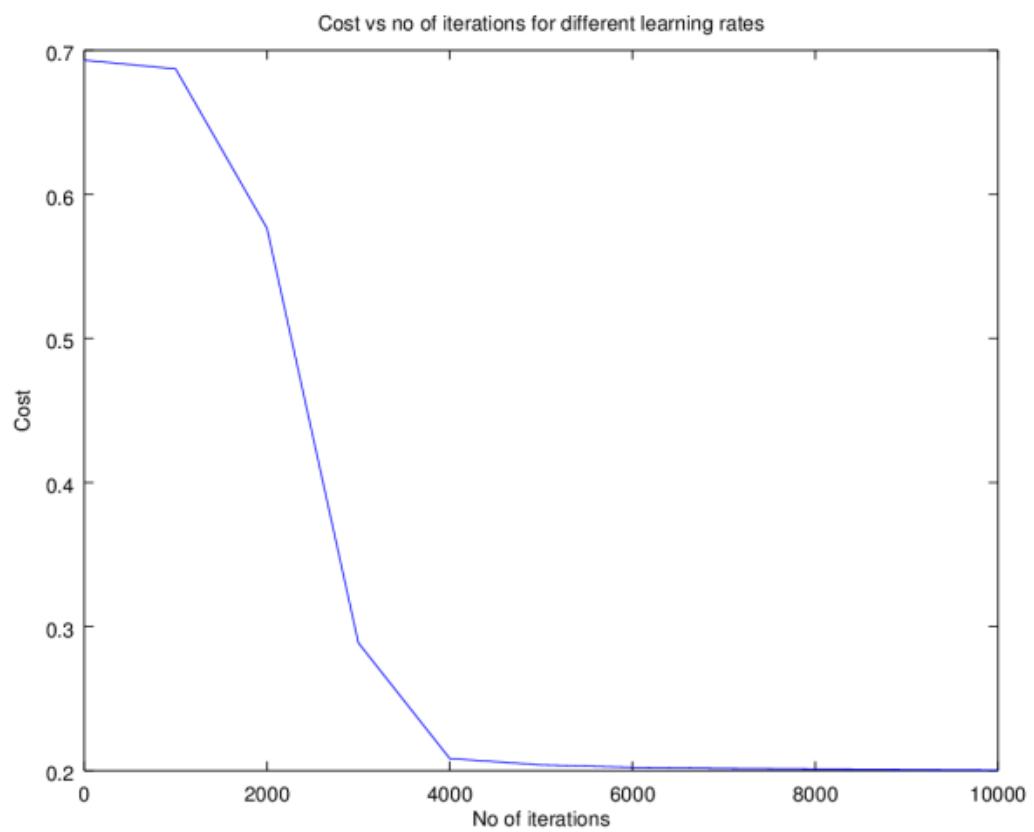
```

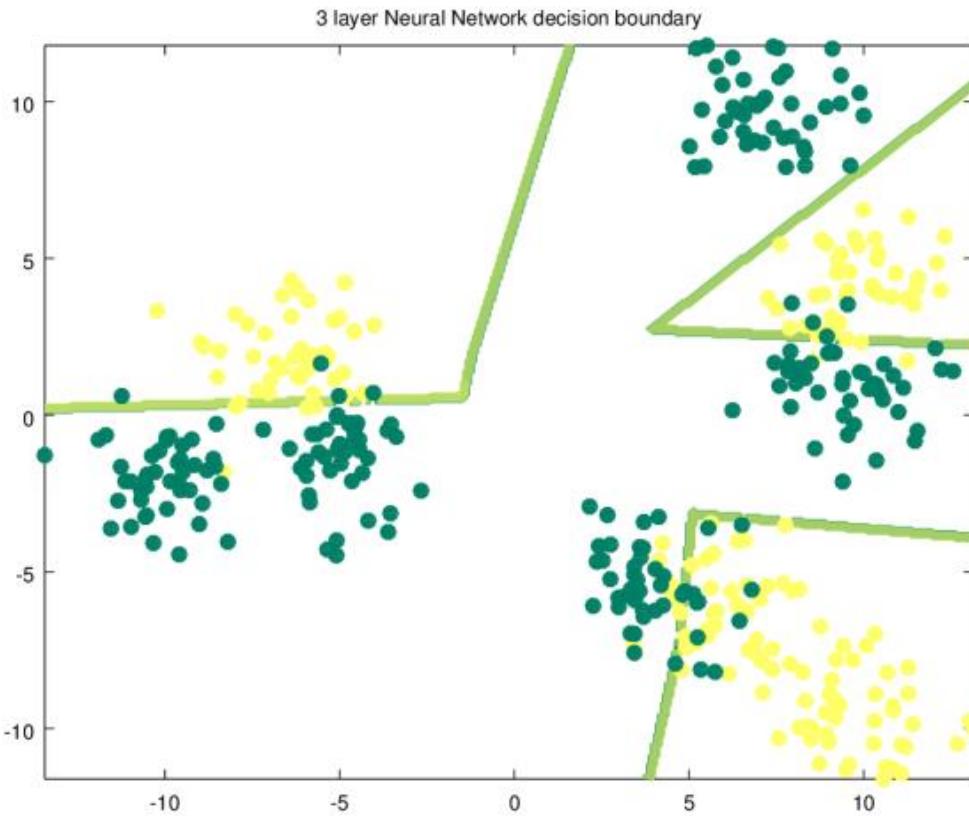
```
32 ##          'Positive' Class : 0  
33 ##
```

2.1c 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.

```
1 # Read the data  
2 data=csvread("data.csv");  
3 X=data(:,1:2);  
4 Y=data(:,3);  
5  
6 # Set layer dimensions  
7 # 2 - 2 input features  
8 # 9 7 - 2 hidden layers with 9 and 7 hidden units  
9 # 1 - 1 sigmoid activation unit at output layer  
10 layersDimensions = [2 9 7 1]  
11  
12 # Execute the L-Layer Deep Network  
13 # Hidden unit - relu  
14 #learning rate - 0.1  
15 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,  
16 hiddenActivationFunc='relu',  
17 learningRate = 0.1,  
18 numIterations = 10000);  
19  
20 #Plot cost vs iterations  
21 plotCostVsIterations(10000,costs);  
22  
23 #Plot the decision boundary  
24 plotDecisionBoundary(data,weights, biases,hiddenActivationFunc="tanh")  
25
```





2.2a Classification with Multi-layer Deep Learning Network – Tanh activation(Python)

The code below uses the tanh activation function to perform the same classification. I found the Tanh activation required a simpler Neural Network of 3 layers.

```

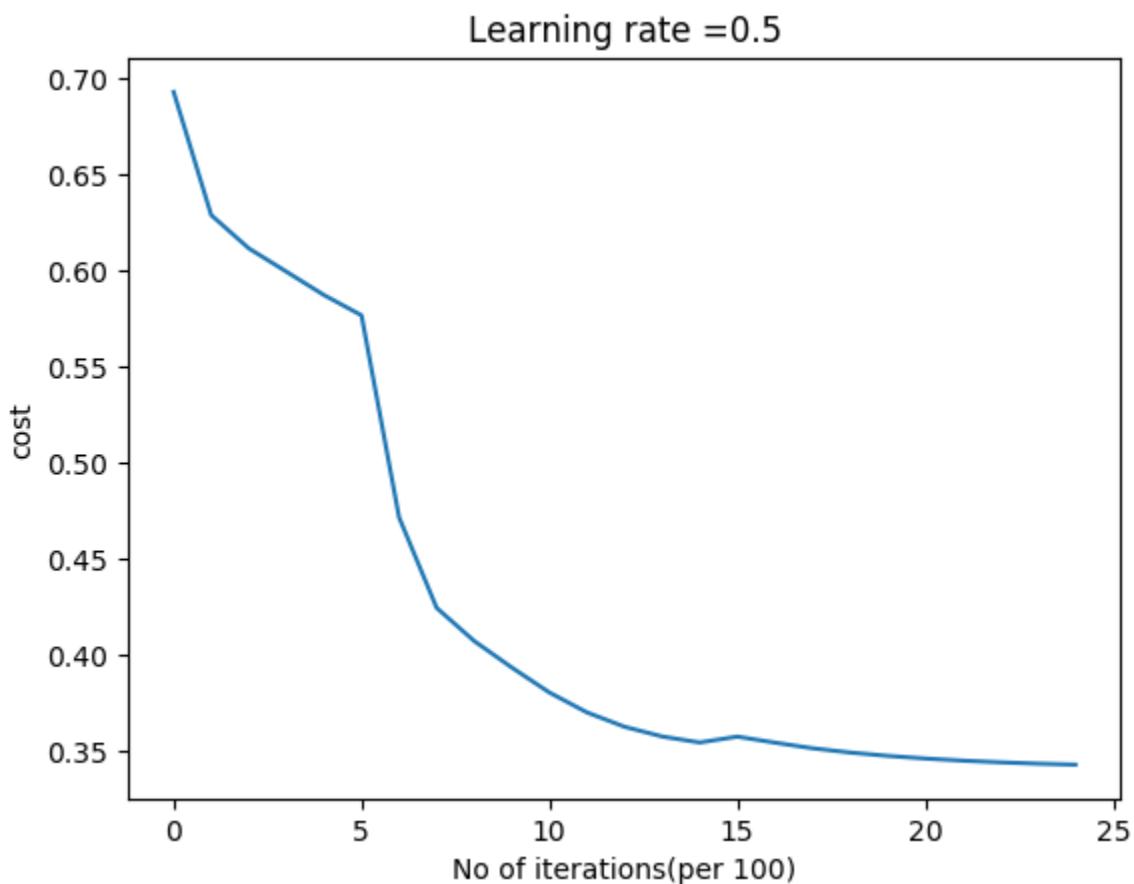
1 # Tanh activation
2 import os
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import matplotlib.colors
6 import sklearn.linear_model
7
8 from sklearn.model_selection import train_test_split
9 from sklearn.datasets import make_classification, make_blobs
10 from matplotlib.colors import ListedColormap
11 import sklearn
12 import sklearn.datasets
13 execfile("./DLfunctions34.py")
14
15 # Create the dataset
16 X1, Y1 = make_blobs(n_samples = 400, n_features = 2, centers = 9,
17                     cluster_std = 1.3, random_state = 4)
18
19 #Create 2 classes

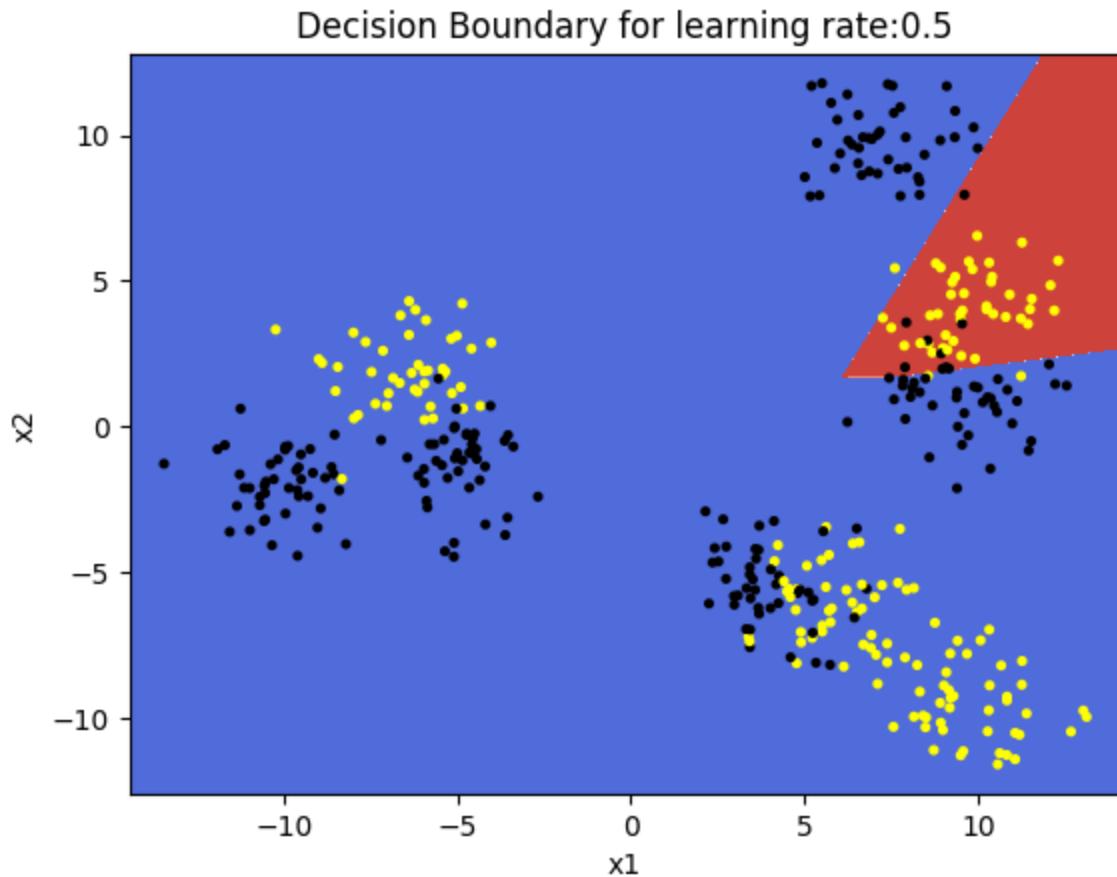
```

```

20 Y1=Y1.reshape(400,1)
21 Y1 = Y1 % 2
22 X2=X1.T
23 Y2=Y1.T
24
25 # Set the dimensions of the Neural Network
26 # 2 - input features
27 # 4 - 1 hidden layer with 4 units
28 # 1 - 1 sigmoid activation unit at output layer
29 layersDimensions = [2, 4, 1] # 3-layer model
30
31 # Execute the L-layer Deep Learning network
32 # hidden layer activation function - tanh
33 # learning rate - 0.5
34 parameters = L_Layer_DeepModel(X2, Y2, layersDimensions,
35 hiddenActivationFunc='tanh', learning_rate = .5,num_iterations =
36 2500,fig="fig3.png")
37
38 #Plot the decision boundary
39 plot_decision_boundary(lambda x: predict(parameters, x.T),
40 X2,Y2,str(0.5),"fig4.png")

```





2.2b 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

```

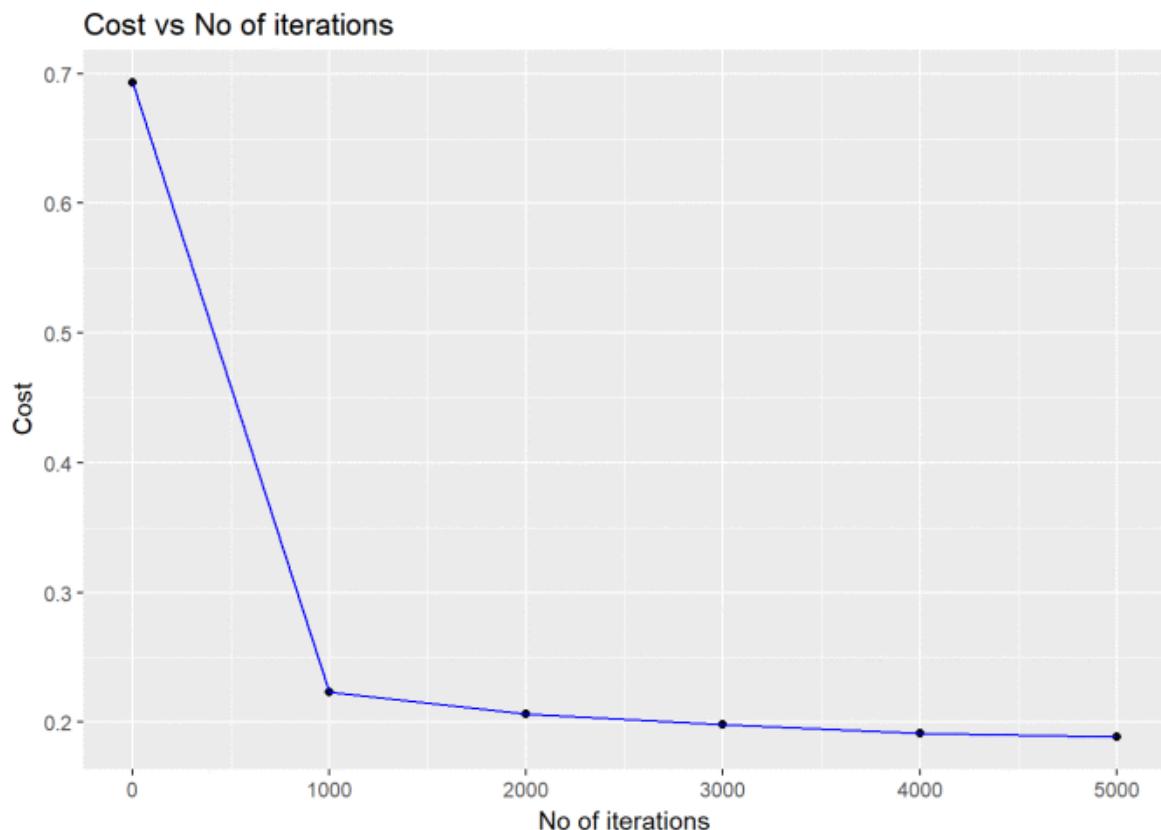
1 #Set the dimensions of the Neural Network
2 library(ggplot2)
3
4 # Read the data
5 z <- as.matrix(read.csv("data.csv",header=FALSE))
6 x <- z[,1:2]
7 y <- z[,3]
8 X1 <- t(x)
9 Y1 <- t(y)
10
11 # Set layer dimensions
12 # 2 - 2 inputr features
13 # 9, 9 - 2 hidden layers with 9 activation units
14 # 1 - 1 sigmoid output activation unit
15 layersDimensions = c(2, 9, 9,1)
16
17 # Execute the L-layer Deep Model
18 # Hidden layer activation function - tanh
19 # learning rate - 0.3
20 retvals = L_Layer_DeepModel(X1, Y1, layersDimensions,

```

```

21             hiddenActivationFunc='tanh',
22             learningRate = 0.3,
23             numIterations = 5000,
24             print_cost = True)
25
26 # Get the costs
27 costs <- retvals[['costs']]
28 # Set iterations
29 iterations <- seq(0,numIterations,by=1000)
30 df <- data.frame(iterations,costs)
31
32 # Plot Cost vs number of iterations
33 ggplot(df,aes(x=iterations,y=costs)) + geom_point() +geom_line(color="blue")
34 + xlab('No of iterations') + ylab('Cost') + ggtitle("Cost vs No of
35 iterations")

```

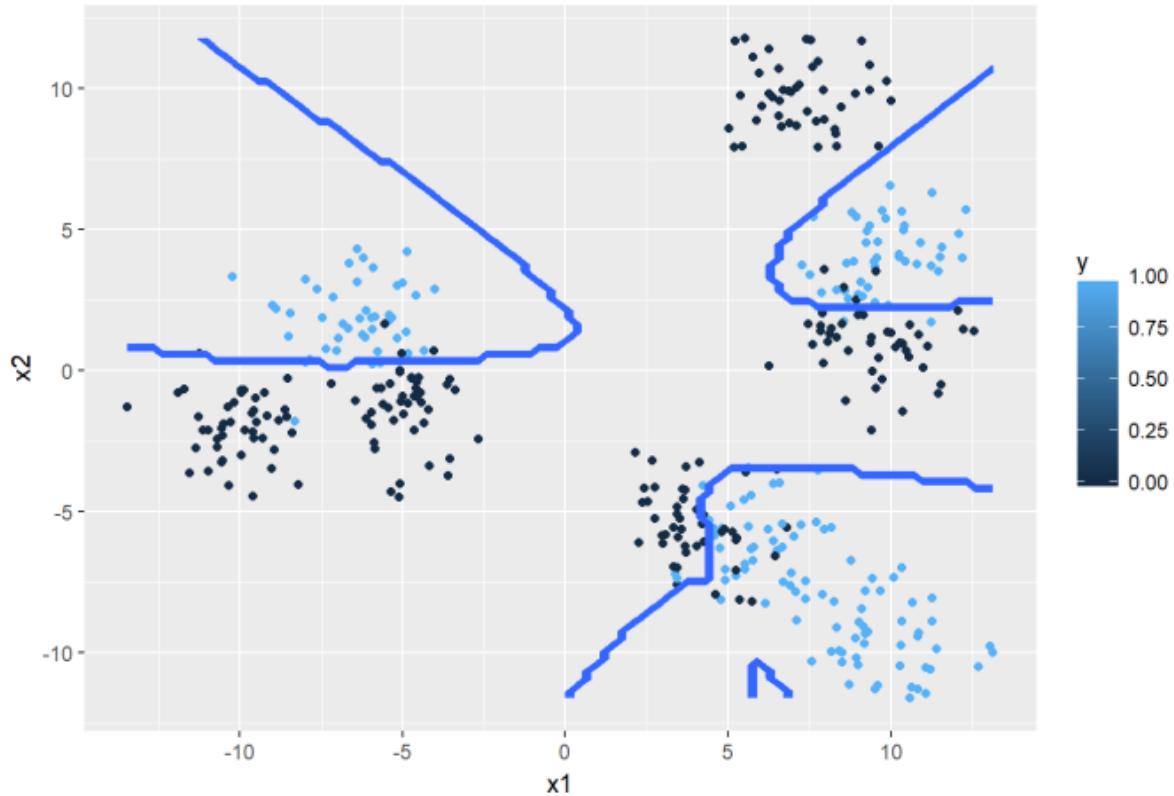


```

1 #Plot the decision boundary
2 plotDecisionBoundary(z,retvals,hiddenActivationFunc="tanh",0.3)

```

Decision boundary for learning rate: 0.3

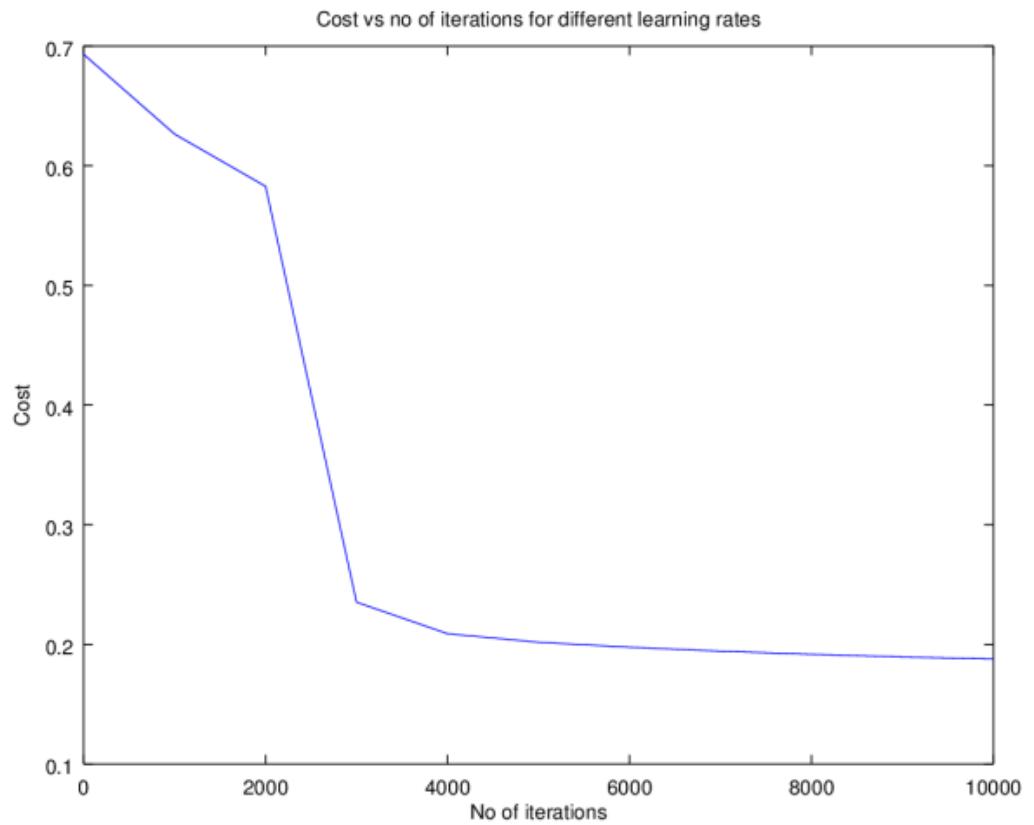


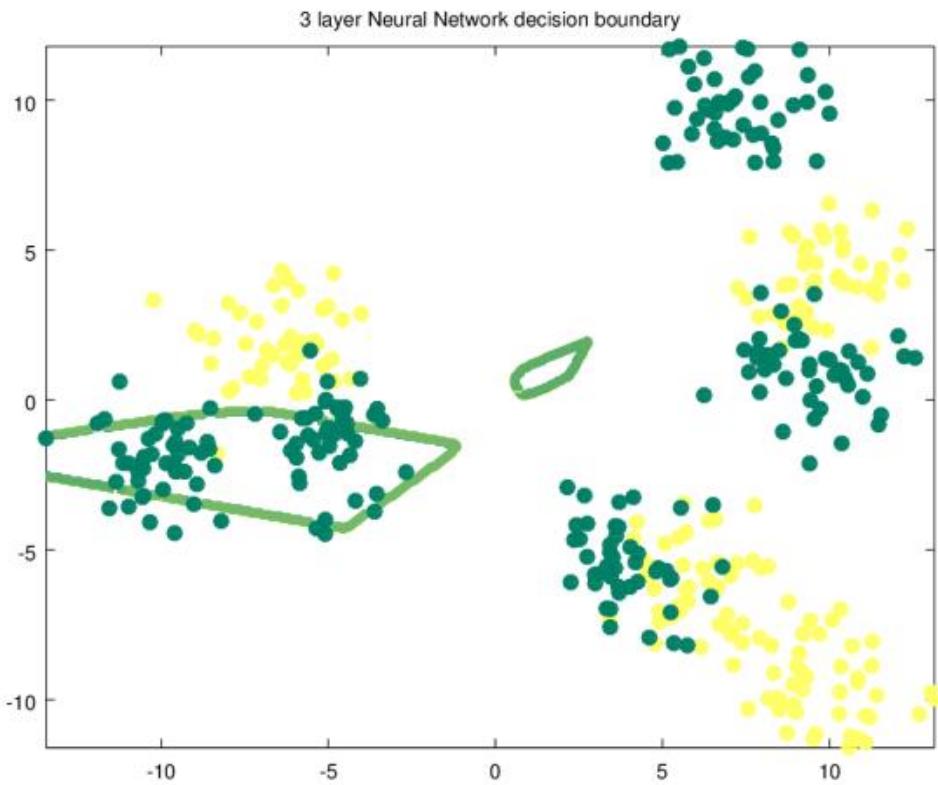
2.2c Classification with Multi-layer Deep Learning Network – Tanh activation(Octave)

The code below uses the Tanh activation in the hidden layers for Octave

```
1 # Read the data
2 data=csvread("data.csv");
3 X=data(:,1:2);
4 Y=data(:,3);
5
6 # Set layer dimensions
7 # 2 - input features
8 # 9 7 - 2 hidden layers with 9 and 7 hidden units
9 # 1 - 1 sigmoid unit at output layer
10 layersDimensions = [2 9 7 1] #tanh=-0.5(ok), #relu=0.1 best!
11
12 # Execute the L-Layer Deep Learning Network
13 # hidden layer activation function - tanh
14 # Learning rate - 0.1
15 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
16 hiddenActivationFunc='tanh',
17 learningRate = 0.1,
18 numIterations = 10000);
19
20 #Plot cost vs iterations
21 plotCostVsIterations(10000, costs);
```

```
22  
23 #Plot the decision boundary  
24 plotDecisionBoundary(data,weights, biases,hiddenActivationFunc="tanh")
```

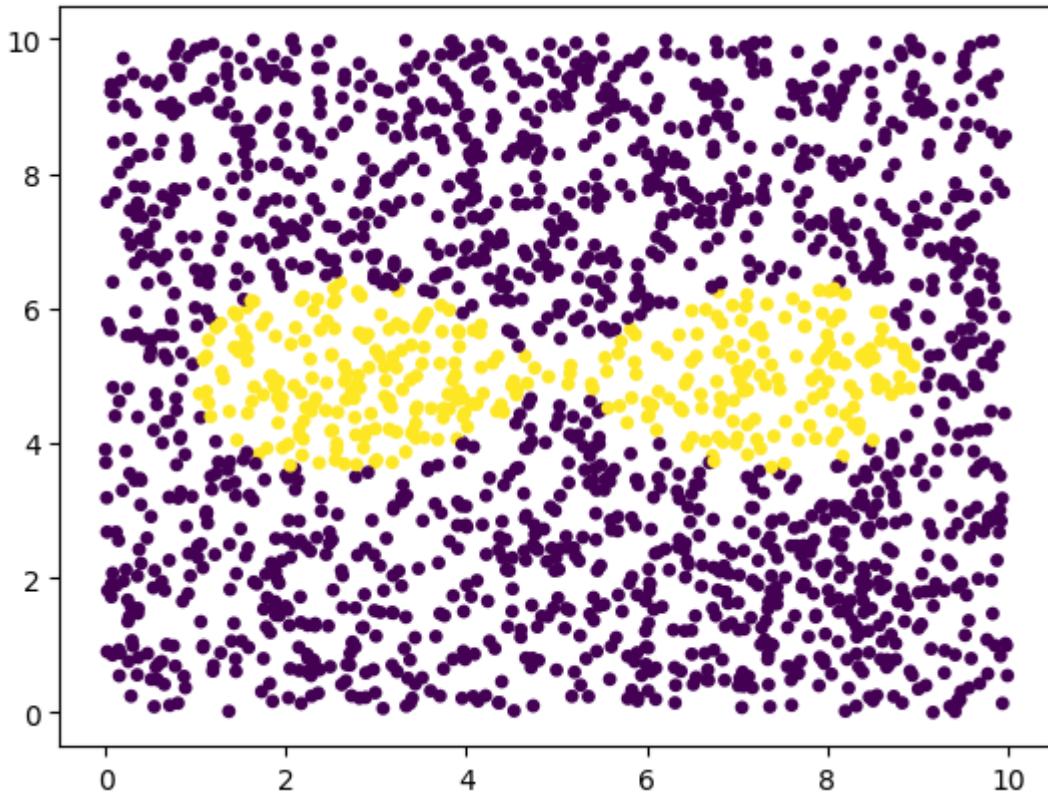




3. Bernoulli's Lemniscate

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)

```

1 import os
2 import numpy as np
3 import matplotlib.pyplot as plt
4 os.chdir("C:\\software\\DeepLearning-Posts\\part3")
5 execfile("./DLfunctions33.py")
6
7 # Create data set
8 x1=np.random.uniform(0,10,2000).reshape(2000,1)
9 x2=np.random.uniform(0,10,2000).reshape(2000,1)
10 X=np.append(x1,x2, axis=1)
11 X.shape
12
13 # Create a subset of values where squared is <0.4. Perform ravel() to flatten
14 this vector
15 # Create the equation
16 #  $(x^2 + y^2)^2 - 2a^2 * (x^2 - y^2) \leq 0$ 
17 a=np.power(np.power(X[:,0]-5,2) + np.power(X[:,1]-5,2),2)
18 b=np.power(X[:,0]-5,2) - np.power(X[:,1]-5,2)
19 c= a - (b*np.power(4,2)) <=0
20 Y=c.reshape(2000,1)
21
22 # Create a scatter plot of the lemniscate
23 plt.scatter(X[:,0], X[:,1], c=Y, marker= 'o', s=15,cmap="viridis")
24 Z=np.append(X,Y, axis=1)
25 plt.savefig("fig50.png",bbox_inches='tight')
26 plt.clf()

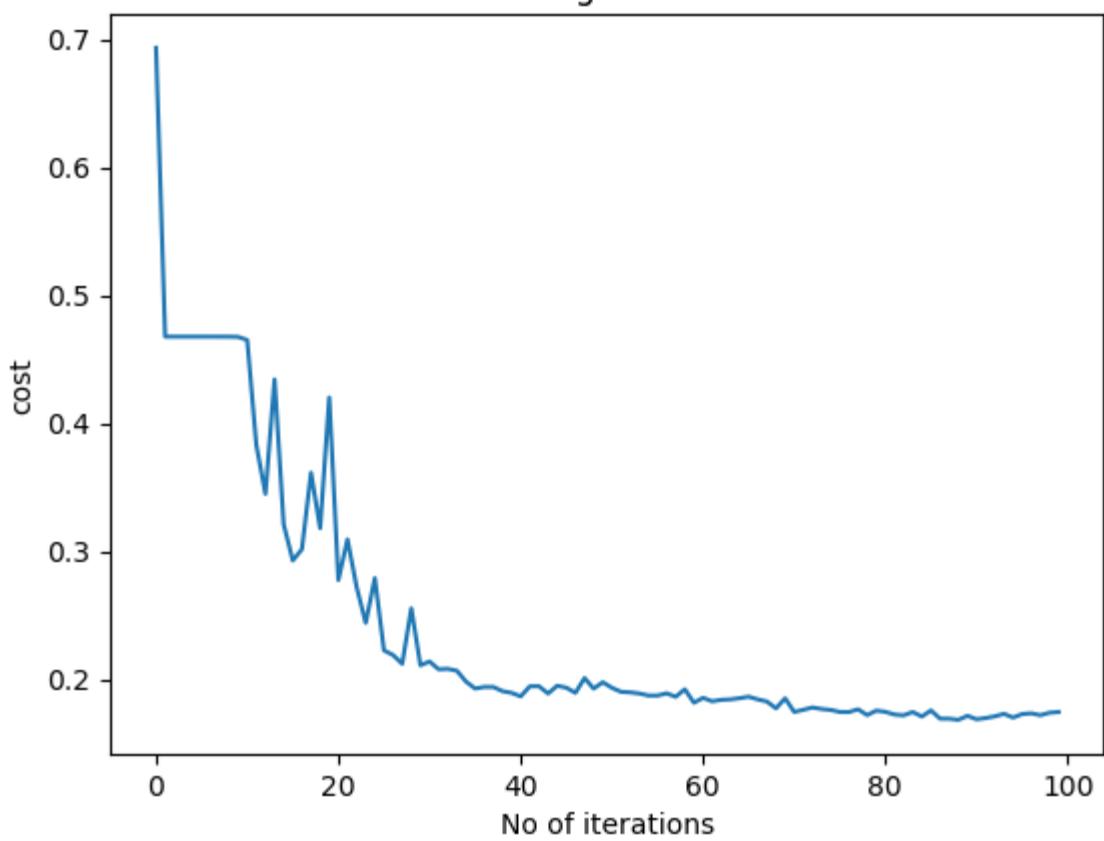
```

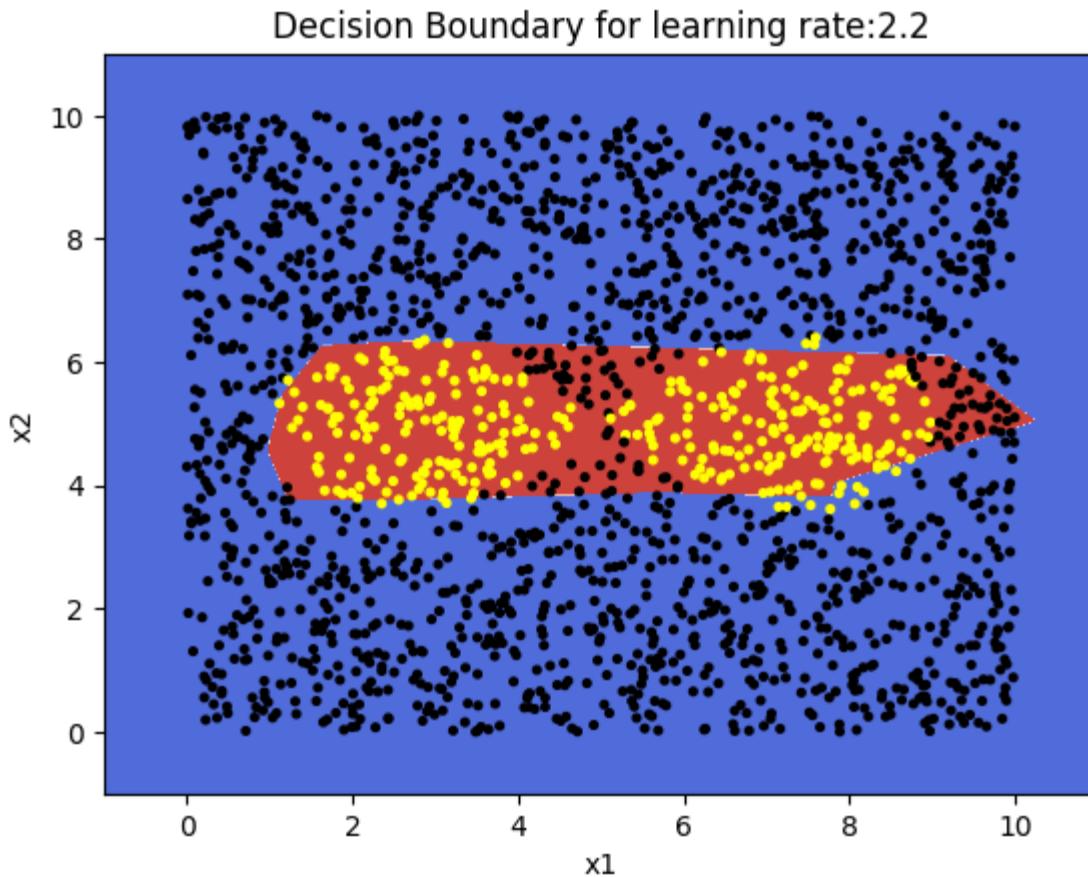
```

27
28 # Set the data for classification
29 X2=X.T
30 Y2=Y.T
31
32 # These settings work the best
33 # Set the Deep Learning layer dimensions for a Relu activation
34 # 2 - input features
35 # 7,4 - 2 hidden layers with 7 and 4 hidden units
36 # 1 - 1 sigmoid activation unit at output layer
37 layersDimensions = [2,7,4,1]
38
39
40 #Execute the L-layer DL network
41 # hidden layer activation function - relu
42 # learning rate - 0.5
43 parameters = L_Layer_DeepModel(X2, Y2, layersDimensions,
44 hiddenActivationFunc='relu', learning_rate = 0.5,num_iterations = 10000,
45 fig="fig5.png")
46
47 #Plot the decision boundary
48 plot_decision_boundary(lambda x: predict(parameters, x.T), X2,
49 Y2,str(2.2),"fig6.png")
50
51 # Compute the Confusion matrix
52 yhat = predict(parameters,X2)
53 from sklearn.metrics import confusion_matrix
54 a=confusion_matrix(Y2.T,yhat.T)
55
56 #Print accuracy,precision, recall and F1 score
57 from sklearn.metrics import accuracy_score, precision_score, recall_score,
58 f1_score
59 print('Accuracy: {:.2f}'.format(accuracy_score(Y2.T, yhat.T)))
60 print('Precision: {:.2f}'.format(precision_score(Y2.T, yhat.T)))
61 print('Recall: {:.2f}'.format(recall_score(Y2.T, yhat.T)))
62 print('F1: {:.2f}'.format(f1_score(Y2.T, yhat.T)))
63 ## Accuracy: 0.93
64 ## Precision: 0.77
65 ## Recall: 0.76
66 ## F1: 0.76

```

Learning rate =0.5





We could get better performance by tuning further. Do play around with the code.

Note: The lemniscate data is saved as a CSV and then read in R and also in Octave.

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

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

```

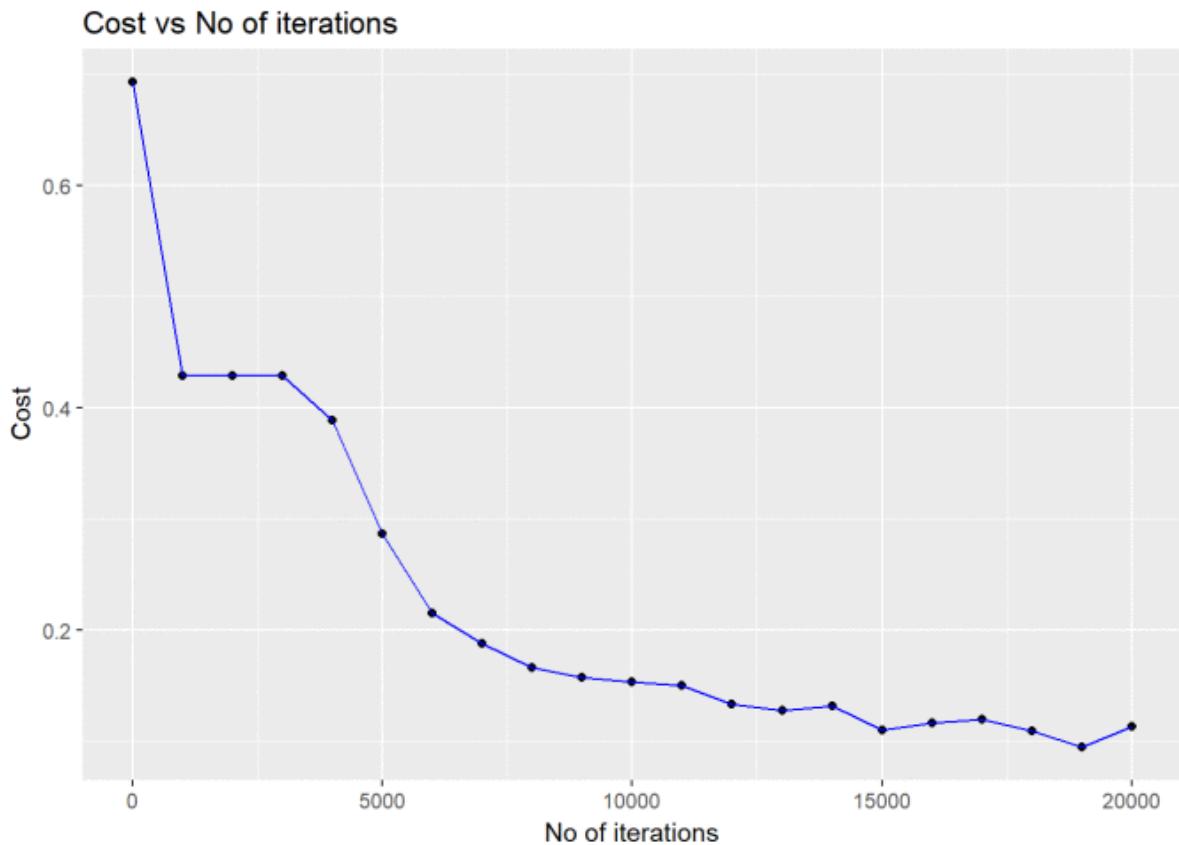
1 # Read lemniscate data
2 Z <- as.matrix(read.csv("lemniscate.csv",header=FALSE))
3 z1=data.frame(Z)
4
5 # Create a scatter plot of the lemniscate
6 ggplot(z1,aes(x=v1,y=v2,col=v3)) +geom_point()
7 #Set the data for the DL network
8 X=Z[,1:2]
9 Y=Z[,3]
10 X1=t(X)
11 Y1=t(Y)
12
13 # Set the layer dimensions for the tanh activation function
14 # 2 - No of input features
15 #5, 4 - 2 hidden layers with 5 and 4 hidden units

```

```

16 # 1 - 1 sigmoid output activation function in output layer
17 layersDimensions = c(2,5,4,1)
18
19 # Execute the L layer Deep Learning network
20 # Activation function in hidden layer - Tanh activation
21 # learning rate=0.3
22 retvals = L_Layer_DeepModel(x1, Y1, layersDimensions,
23                             hiddenActivationFunc='tanh',
24                             learningRate = 0.3,
25                             numIterations = 20000, print_cost = True)
26
27 # Plot cost vs iteration
28 costs <- retvals[['costs']]
29 numIterations = 20000
30 iterations <- seq(0,numIterations,by=1000)
31 df <- data.frame(iterations,costs)
32 ggplot(df,aes(x=iterations,y=costs)) + geom_point() +geom_line(color="blue")
33 + xlab('No of iterations') + ylab('Cost') + ggtitle("Cost vs No of
34 iterations")
35

```

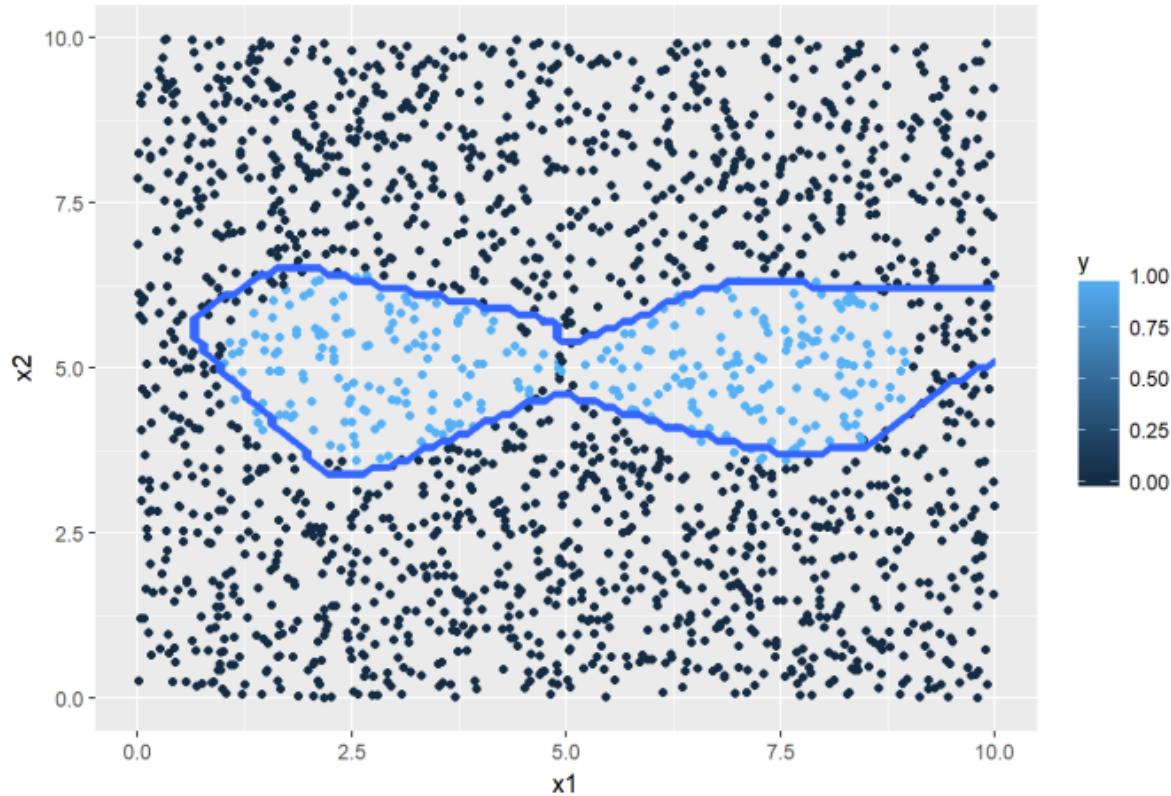


```

1 #Plot the decision boundary
2 plotDecisionBoundary(z,retvals,hiddenActivationFunc="tanh",0.3)

```

Decision boundary for learning rate: 0.3

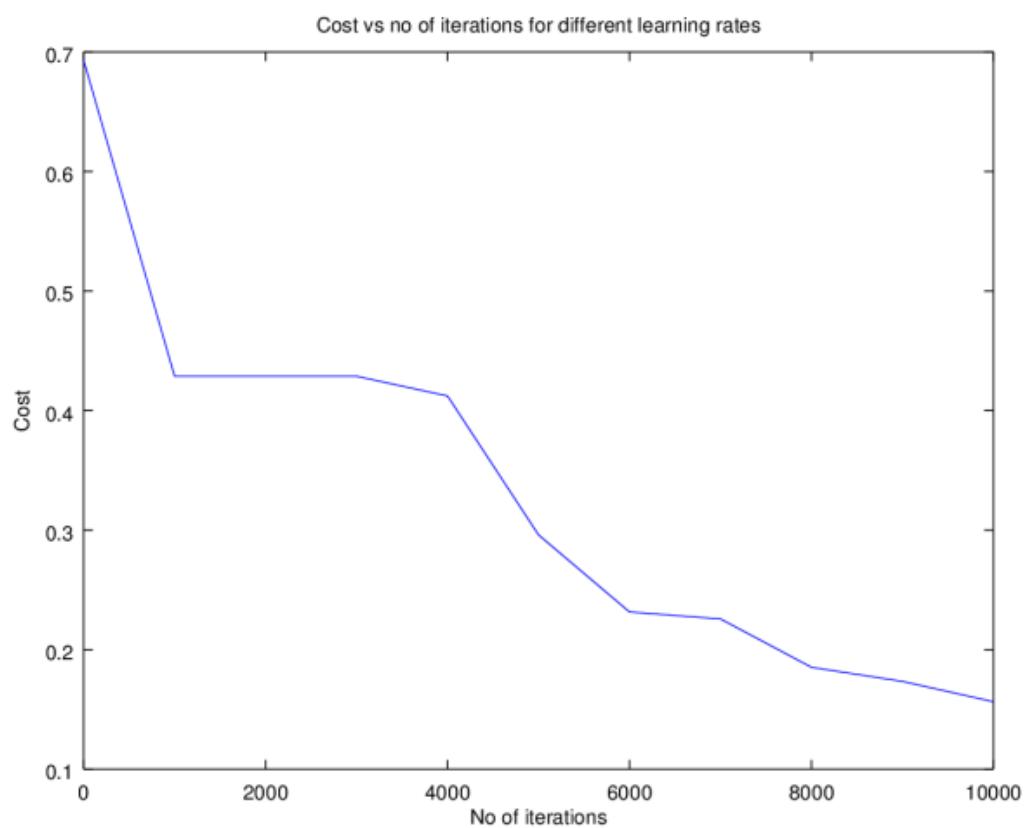


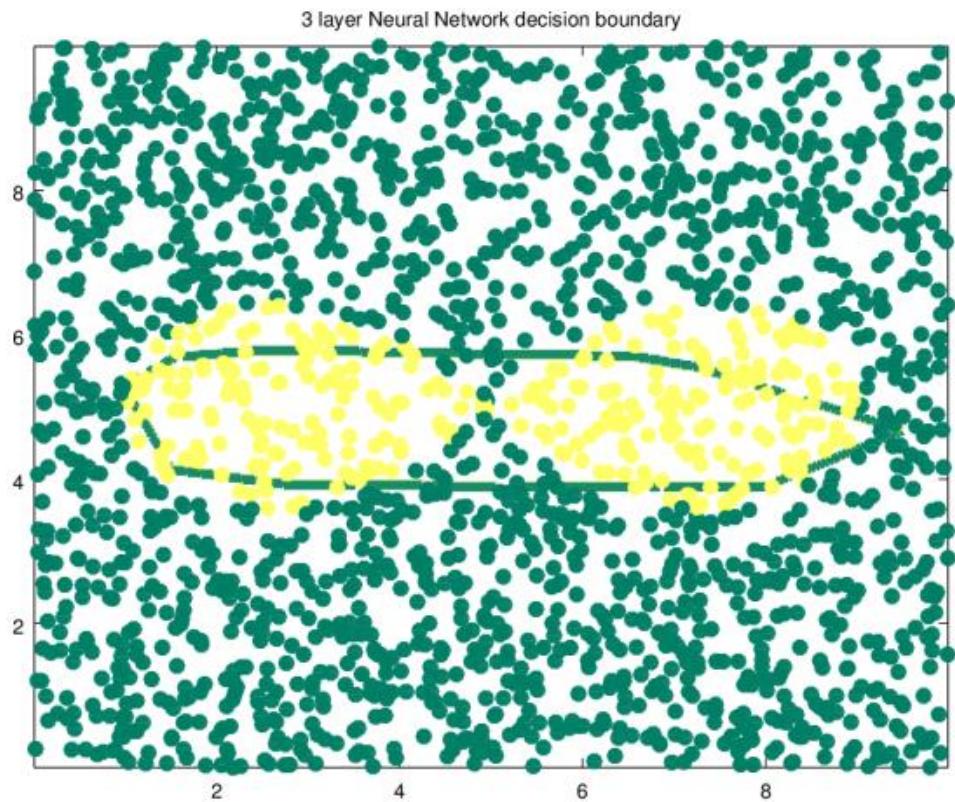
3c. Classifying a lemniscate with Deep Learning Network – Relu activation (Octave code)

Octave is used to generate the non-linear lemniscate boundary.

```
1 # Read the data
2 data=csvread("lemniscate.csv");
3 X=data(:,1:2);
4 Y=data(:,3);
5
6 # Set the dimensions of the layers
7 # 2 - no of input features
8 # 9 7 - 2 hidden layers with 9 and 7 activation units respectively
9 # 1 - 1 activation unit in output layer (sigmoid)
10 layersDimensions = [2 9 7 1]
11
12 # Execute the L-layer the DL network
13 #hidden activation function - relu
14 # learning rate- 0.20
15 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
16 hiddenActivationFunc='relu',
17 learningRate = 0.20,
18 numIterations = 10000);
19
20 #Plot the cost vs iterations
21 plotCostVsIterations(10000,costs);
```

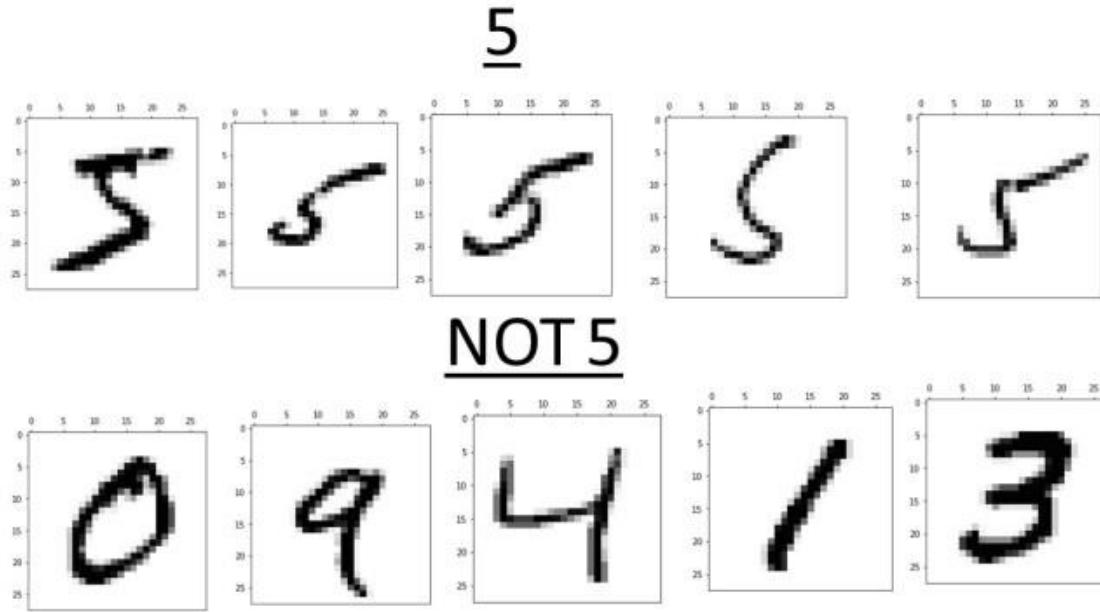
```
22  
23 #Plot the descion boundary  
24 plotDecisionBoundary(data,weights, biases,hiddenActivationFunc="relu")
```





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 perform the proper classification of all digits using the Softmax classifier in chapter 5.

```

1 import os
2 import numpy as np
3 import matplotlib.pyplot as plt
4 execfile("./DLfunctions34.py")
5
6 #Load MNIST
7 execfile("./load_mnist.py")
8 # Set the training and test data and labels
9 training=list(read(dataset='training',path='./mnist'))
10 test=list(read(dataset='testing',path='./mnist'))
11 lbls=[]
12 pxls=[]
13 print(len(training))
14
15 # Select the first 10000 training data and the labels
16 for i in range(10000):
17     l,p=training[i]
18     lbls.append(l)
19     pxls.append(p)
20 labels= np.array(lbls)
21 pixels=np.array(pxls)
22
23 # Set y=1 when labels == 5 and 0 otherwise
24 y=(labels==5).reshape(-1,1)
25 X=pixels.reshape(pixels.shape[0],-1)
26
27 # Create the necessary feature and target variable
28 X1=X.T
29 Y1=y.T
30

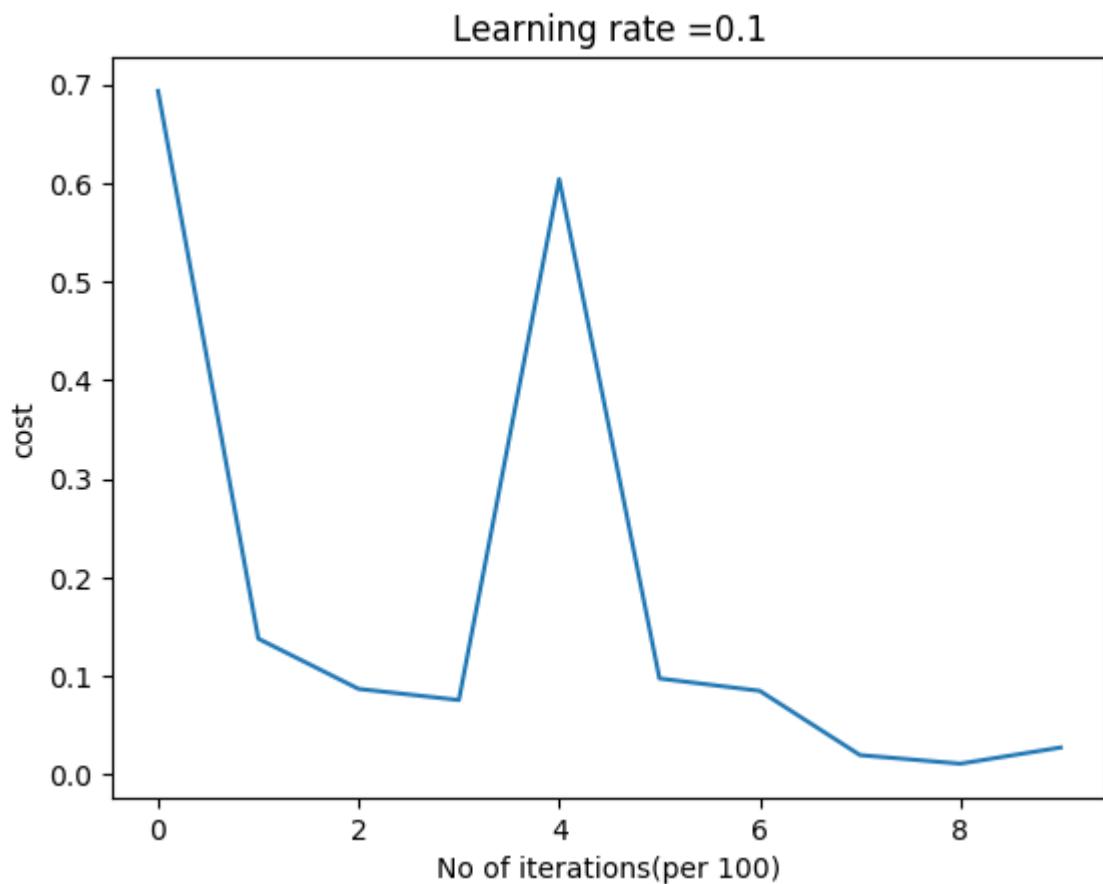
```

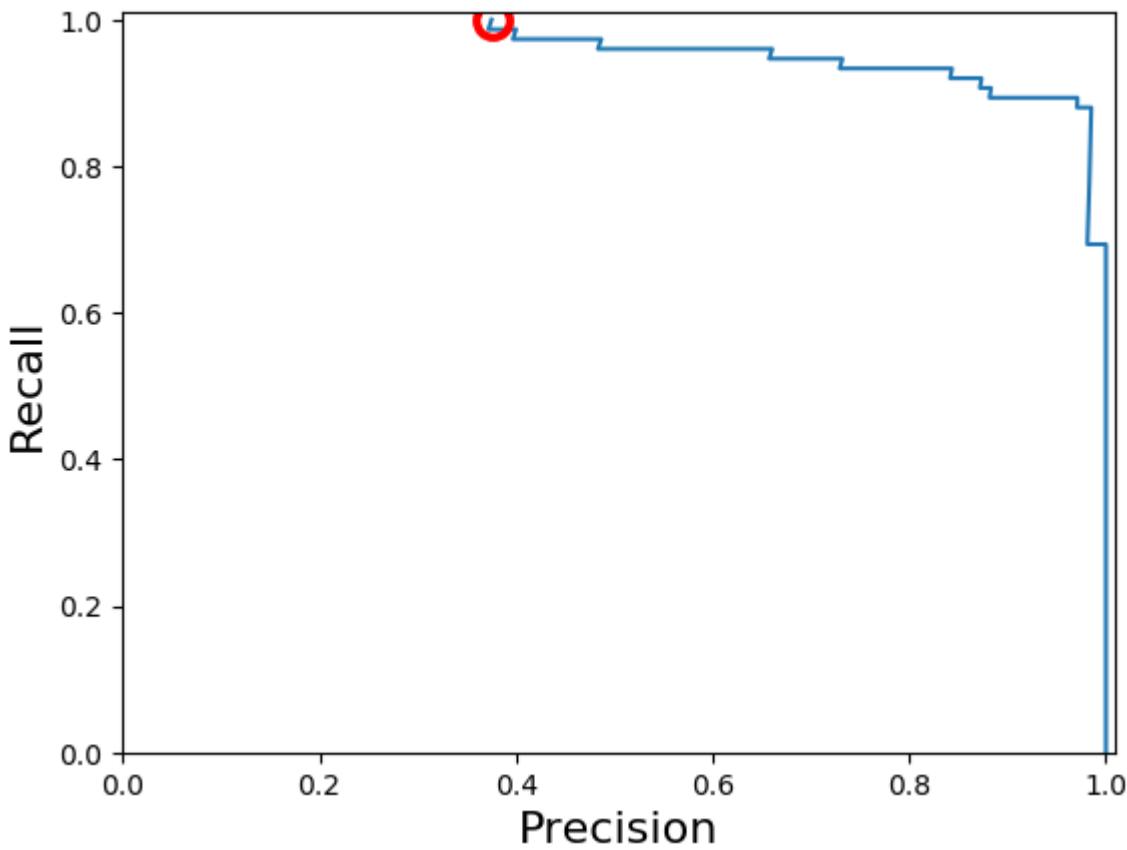
```

31 # Create the layer dimensions. The number of features are 28 x 28 = 784 since
32 the 28 x 28
33 # pixels is flattened to single vector of length 784.
34 # 784 - No of input features = 28 x28
35 # 15, 9 - 2 hidden layers with 15 and 9 hidden activation units
36 # 1 - 1 activation unit in the output layer (sigmoid)
37 layersDimensions=[784, 15,9,7,1] #
38
39 #Execute the L-Layer Deep Learning Network
40 # hidden activation function - relu
41 #learning reate - 0.1
42 parameters = L_Layer_DeepModel(x1, y1, layersDimensions,
43 hiddenActivationFunc='relu', learning_rate = 0.1,num_iterations = 1000,
44 fig="fig7.png")
45
46 # Read the Test data and labels
47 lbls1=[]
48 pxls1=[]
49 for i in range(800):
50     l,p=test[i]
51     lbls1.append(l)
52     pxls1.append(p)
53
54 testLabels=np.array(lbls1)
55 testData=np.array(pxls1)
56
57 ytest=(testLabels==5).reshape(-1,1)
58 Xtest=testData.reshape(testData.shape[0],-1)
59 Xtest1=Xtest.T
60 Ytest1=ytest.T
61
62 # Predict based on test data
63 yhat = predict(parameters,Xtest1)
64 from sklearn.metrics import confusion_matrix
65
66 #Compute the confusion matrix
67 a=confusion_matrix(Ytest1.T,yhat.T)
68
69 #Print accuracy, precision, recall and F1 score
70 from sklearn.metrics import accuracy_score, precision_score, recall_score,
71 f1_score
72 print('Accuracy: {:.2f}'.format(accuracy_score(Ytest1.T, yhat.T)))
73 print('Precision: {:.2f}'.format(precision_score(Ytest1.T, yhat.T)))
74 print('Recall: {:.2f}'.format(recall_score(Ytest1.T, yhat.T)))
75 print('F1: {:.2f}'.format(f1_score(Ytest1.T, yhat.T)))
76
77 # Plot the Precision-Recall curve
78 probs=predict_proba(parameters,Xtest1)
79 from sklearn.metrics import precision_recall_curve
80
81 precision, recall, thresholds = precision_recall_curve(Ytest1.T, probs.T)
82 closest_zero = np.argmin(np.abs(thresholds))
83 closest_zero_p = precision[closest_zero]
84 closest_zero_r = recall[closest_zero]
85
86 #Plot precision-recall curve
87 plt.xlim([0.0, 1.01])
88 plt.ylim([0.0, 1.01])
89 plt.plot(precision, recall, label='Precision-Recall Curve')
90 plt.plot(closest_zero_p, closest_zero_r, 'o', markersize = 12, fillstyle =
91 'none', c='r', mew=3)
92 plt.xlabel('Precision', fontsize=16)
93 plt.ylabel('Recall', fontsize=16)
94 plt.savefig("fig8.png",bbox_inches='tight')

```

```
95  
96 ## Accuracy: 0.99  
97 ## Precision: 0.96  
98 ## Recall: 0.89  
99 ## 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.

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>)

```

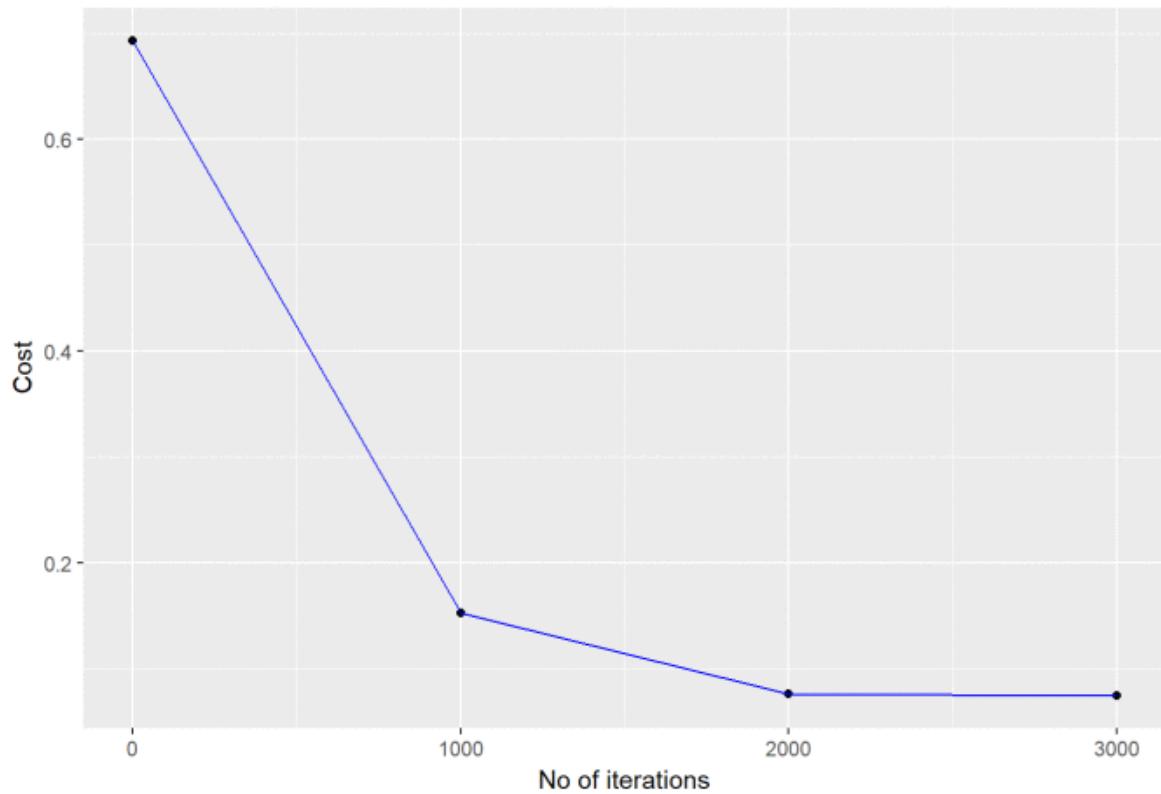
1 source("mnist.R")
2 #Load the MNIST data
3 load_mnist()
4 #show_digit(train$x[2,]
5
6 #Set the layer dimensions
7 # 784 - no of input features = 28 x 28
8 # 7, 7, 3 - 3 hidden layers with 7,7,3 activation units
9 # 1 - 1 sigmoid activation unit at output layer
10 layersDimensions=c(784, 7,7,3,1) # Works at 1500
11
12
13 x <- t(train$x)
14 # Choose only 5000 training data
15 x2 <- x[,1:5000]
16
```

```

17 # Classify the data
18 y <- train$y
19
20 # Set labels for all digits that are 'not 5' to 0
21 y[y!=5] <- 0
22
23 # Set labels of digit 5 as 1
24 y[y==5] <- 1
25
26 # Set the data
27 y1 <- as.matrix(y)
28 y2 <- t(y1)
29
30 # Choose the 1st 5000 data
31 y3 <- y2[,1:5000]
32
33 # Execute the L-Layer Deep Learning Model
34 # hidden activation function - relu
35 # learning rate - 0.3
36 retvals = L_Layer_DeepModel(x2, y3, layersDimensions,
37                             hiddenActivationFunc='tanh',
38                             learningRate = 0.3,
39                             numIterations = 3000, print_cost = True)
40
41 # Setup costs and iterations
42 costs <- retvals[['costs']]
43 numIterations = 3000
44 iterations <- seq(0,numIterations,by=1000)
45 df <- data.frame(iterations,costs)
46
47 # Plot cost vs iterations
48 ggplot(df,aes(x=iterations,y=costs)) + geom_point() +geom_line(color="blue")
49 + xlab('No of iterations') + ylab('Cost') + ggtitle("Cost vs No of
50 iterations")

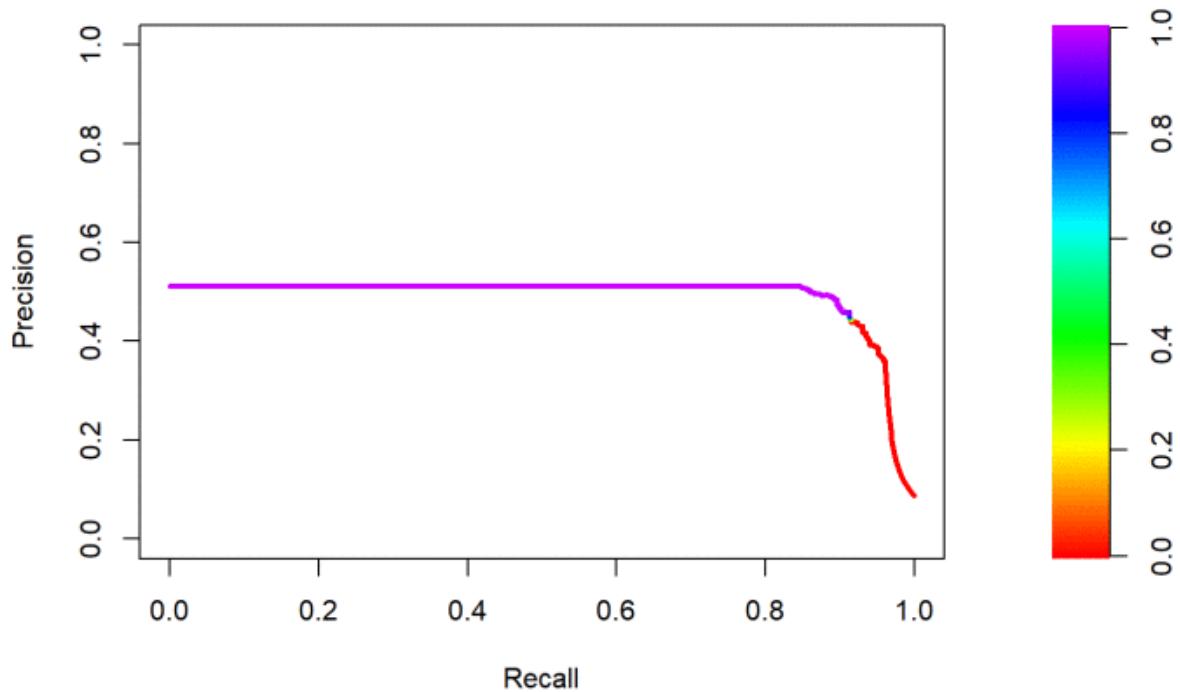
```

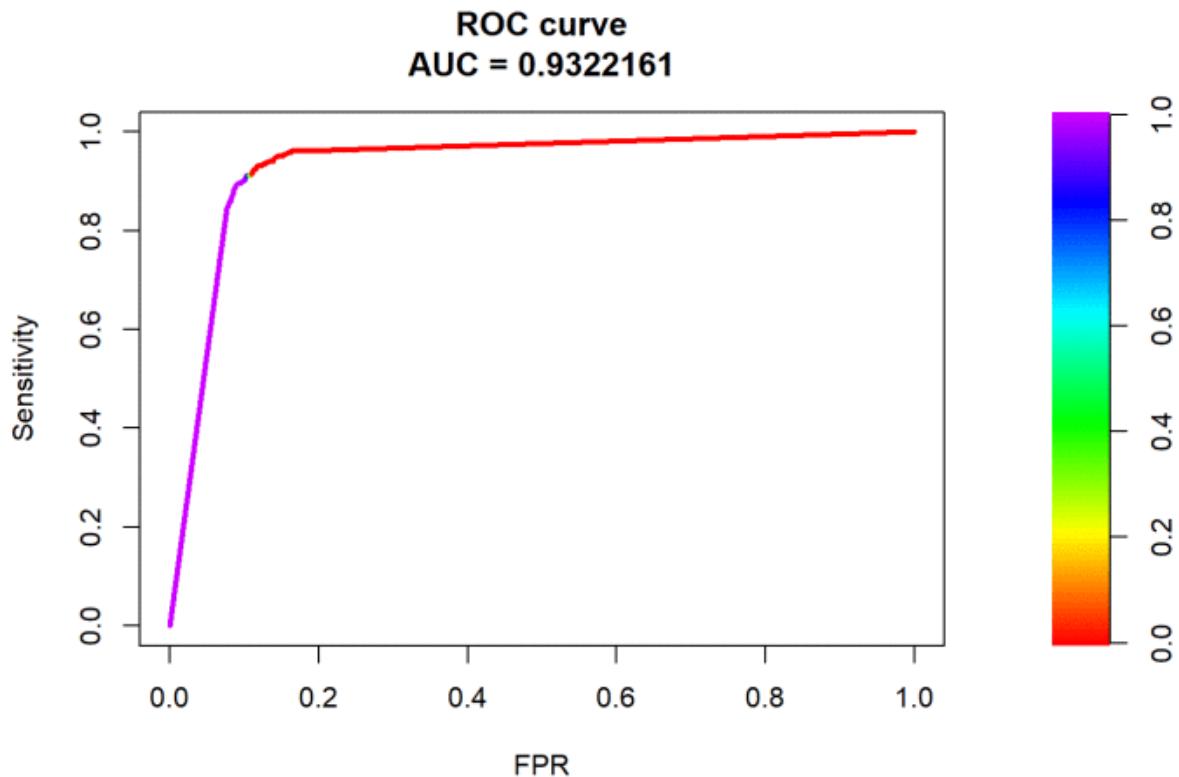
Cost vs No of iterations



```
1 # Compute probability scores
2 scores <- computeScores(retvals$parameters, x2,hiddenActivationFunc='relu')
3 a=y3==1
4 b=y3==0
5
6 # Compute probabilities of class 0 and class 1
7 class1=scores[a]
8 class0=scores[b]
9
10 # Plot ROC curve
11 pr <-pr.curve(scores.class0=class1,
12                 scores.class1=class0,
13                 curve=T)
14
15 plot(pr)
```

PR curve
AUC = 0.4907586





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.

```

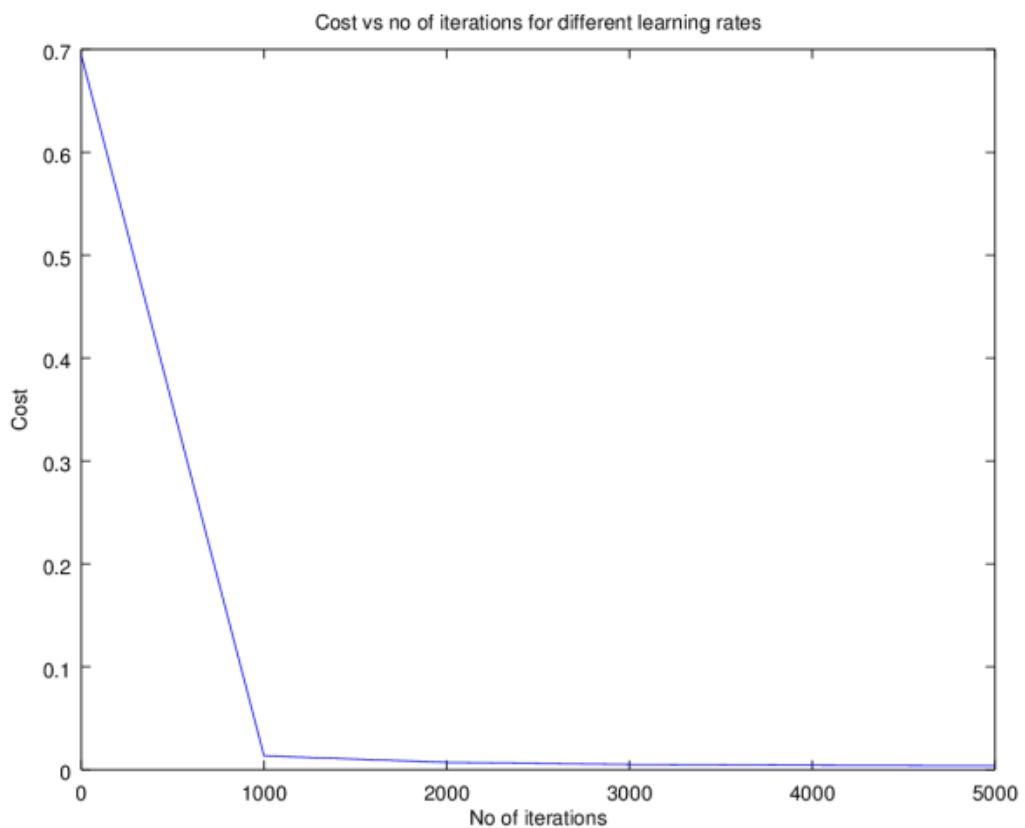
1 # Load the MNIST data
2 load('./mnist/mnist.txt.gz');
3
4 # Classify digits as 5 and not 5
5 # Subset the 'not 5' digits
6 a=(trainY != 5);
7 # Subset '5'
8 b=(trainY == 5);
9 #make a copy of trainY
10 #Set 'not 5' as 0 and '5' as 1
11 y=trainY;
12 y(a)=0;
13 y(b)=1;
14 X=trainX(1:5000,:);
15 Y=y(1:5000);
16
17 # Set the dimensions of layer
18 # 784 - number of input features = 28 x 28
19 # 7, 7, 3 - 3 hidden layers with 7,7, 3 hidden activation units respectively

```

```

20 # 1 - 1 sigmoid activation unit at output layer
21 layersDimensions=[784, 7, 7, 3, 1];
22
23 # Execute the L-Layer the DL network
24 # hidden activation function - relu
25 #learning rate - 0.1
26 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
27 hiddenActivationFunc='relu',
28 learningRate = 0.1,
29 numIterations = 5000);

```



5. Conclusion

It was quite a challenge coding a Deep Learning Network in Python, R and Octave. The Deep Learning network implementation, in this chapter is the ‘no-frills’ Deep Learning network. It does not include initialization techniques, regularization or gradient optimizations methods. These will be discussed in chapter 6 and chapter 7. This L-layer Deep Learning network will be enhanced in the later chapters. 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.
- f. I also did come across situations of ‘exploding/vanishing gradient’; cost went to Inf/-Inf.

Feel free to fork/clone the code from Github DeepLearningFromFirstPrinciples (<https://github.com/tvganesh/DeepLearningFromFirstPrinciples/tree/master/Chap3-LayerDeepLearningNetwork>) and take the DL network apart and play around with it.

4. Deep Learning network with the Softmax

In this fourth chapter I explore the details of creating a multi-class classifier using the Softmax activation unit in a neural network. This fourth chapter takes a swing at multi-class classification and uses the Softmax as the activation unit in the output layer. Inclusion of the Softmax activation unit in the activation layer requires us to compute the derivative of Softmax, or rather the “Jacobian” of the Softmax function, besides also computing the log loss for this Softmax activation during backward propagation. Since the derivation of the Jacobian of a Softmax and the computation of the Cross Entropy/log loss is very involved, I have implemented a basic neural network with just 1 hidden layer with the Softmax activation at the output layer. I also perform multi-class classification based on the ‘spiral’ data set from CS231n Convolutional Neural Networks (<http://cs231n.github.io/neural-networks-case-study/>) Stanford course, to test the performance and correctness of the implementations in Python, R and Octave. The vectorized implementations of the functions in this chapter are at Appendix 4 - Deep Learning network with the Softmax

You can clone download the code for the Python, R and Octave implementations from Github at DeepLearningFromFirstPrinciples (<https://github.com/tvganesh/DeepLearningFromFirstPrinciples/tree/master/Chap4-MulticlassDeepLearningNetwork>)

The Softmax function takes an N dimensional vector as input and generates a N-dimensional vector as output.

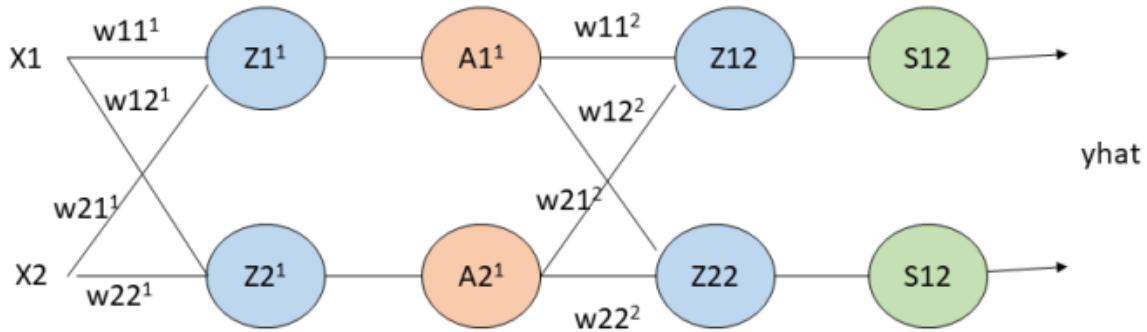
The Softmax function is given by (4a) below

$$S_j = \frac{e_j}{\sum_i^N e_k} \quad - (4a)$$

There is a probabilistic interpretation of the Softmax, since the sum of the Softmax values of a set of vectors will always add up to 1, given that each Softmax value is divided by the total of all values.

As mentioned earlier, the Softmax takes a vector input and returns a vector of outputs. For e.g. the Softmax of a vector $a = [1, 3, 6]$, is another vector $S = [0.0063, 0.0471, 0.9464]$. Notice that vector output is proportional to the input vector. Also, taking the derivative of a vector by another vector, is known as the Jacobian. By the way, The Matrix Calculus You Need For Deep Learning (<https://arxiv.org/pdf/1802.01528.pdf>) by Terence Parr and Jeremy Howard, is very good paper that distills all the main mathematical concepts for Deep Learning in one place.

Let us take a simple 2 layered neural network with just 2 activation units in the hidden layer is shown below



$$Z_1^1 = W_{11}^1 x_1 + W_{21}^1 x_2 + b_1^1$$

$$Z_2^1 = W_{12}^1 x_1 + W_{22}^1 x_2 + b_2^1$$

and

$$A_1^1 = g'(Z_1^1)$$

$$A_2^1 = g'(Z_2^1)$$

where $g'()$ is the activation unit in the hidden layer which can be a relu, sigmoid or a tanh function

Note: The superscript denotes the layer. The equations are applicable for layer 1 of the neural network. For layer 2 with the Softmax activation, the equations are

$$Z_1^2 = W_{11}^2 x_1 + W_{21}^2 x_2 + b_1^2$$

$$Z_2^2 = W_{12}^2 x_1 + W_{22}^2 x_2 + b_2^2$$

and

$$A_1^2 = S(A_1^1)$$

$$A_2^2 = S(A_2^1)$$

where S is the Softmax activation function and using equation (4a) above

$$S = \begin{pmatrix} S(Z_1^2) \\ S(Z_2^2) \end{pmatrix}$$

$$S = \begin{pmatrix} \frac{e^{Z_1}}{e^{Z_1} + e^{Z_2}} \\ \frac{e^{Z_2}}{e^{Z_1} + e^{Z_2}} \end{pmatrix}$$

The Jacobian of the softmax ‘S’ is given by

$$\begin{pmatrix} \frac{\partial S_1}{\partial Z_1} & \frac{\partial S_1}{\partial Z_2} \\ \frac{\partial S_2}{\partial Z_1} & \frac{\partial S_2}{\partial Z_2} \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial Z_1} \frac{e^{Z_1}}{e^{Z_1} + e^{Z_2}} & \frac{\partial}{\partial Z_2} \frac{e^{Z_1}}{e^{Z_1} + e^{Z_2}} \\ \frac{\partial}{\partial Z_1} \frac{e^{Z_2}}{e^{Z_1} + e^{Z_2}} & \frac{\partial}{\partial Z_2} \frac{e^{Z_2}}{e^{Z_1} + e^{Z_2}} \end{pmatrix}$$

Now the ‘division-rule’ of derivatives is as follows. If u and v are functions of x , then

$$\frac{d}{dx} \frac{u}{v} = \frac{v du - u dv}{v^2}$$

Using this we can compute each element of the above Jacobian matrix. It can be seen that when $i=j$ we have

$$\frac{\partial}{\partial Z_1} \frac{e^{Z_1}}{e^{Z_1} + e^{Z_2}} = \frac{\sum e^{Z_1} - e^{Z_1}^2}{\sum^2}$$

and when $i \neq j$

$$\frac{\partial}{\partial z_i} \frac{e^{z_2}}{e^{z_1} + e^{z_2}} = \frac{0 - e^{z_1} e^{z_2}}{\sum^2}$$

This is of the general form

$$\frac{\partial S_i}{\partial z_i} = S_i(1 - S_j) \text{ when } i=j$$

and

$$\frac{\partial S_i}{\partial z_i} = -S_i S_j \text{ when } i \neq j$$

Note: Since the Softmax essentially gives the probability the following notation is also used

$$\frac{\partial p_i}{\partial z_i} = p_i(1 - p_j) \text{ when } i=j$$

and

$$\frac{\partial p_i}{\partial z_i} = -p_i p_j \text{ when } i \neq j$$

If you throw the “Kronecker delta” into the equation, then the above equations can be expressed even more concisely as

$$\frac{\partial p_i}{\partial z_i} = p_i(\delta_{ij} - p_j)$$

where $\delta_{ij} = 1$ when $i=j$ and 0 when $i \neq j$

This reduces the Jacobian of the simple 2 output softmax vectors equation (A) as

$$\begin{pmatrix} p_1(1 - p_1) & -p_1 p_2 \\ -p_2 p_1 & p_2(1 - p_2) \end{pmatrix}$$

The loss of Softmax is given by

$$L = - \sum y_i \log(p_i)$$

For the 2 valued Softmax output this is

$$\frac{dL}{dp1} = -\frac{y_1}{p_1}$$

$$\frac{dL}{dp2} = -\frac{y_2}{p_2}$$

Using the chain rule we can write

$$\frac{\partial L}{\partial w_{pq}} = \sum_i \frac{\partial L}{\partial p_i} \frac{\partial p_i}{\partial w_{pq}} \quad (1)$$

In expanded form this is

Also

$$\frac{\partial L}{\partial Z_i} = \sum_i \frac{\partial L}{\partial p} \frac{\partial p}{\partial Z_i}$$

Therefore

$$\frac{\partial L}{\partial z_1} = -\frac{y_1}{p_1} p_1(1 - p_1) - \frac{y_2}{p_2} * (-p_2 p_1)$$

Since

$$\frac{\partial p_j}{\partial z_i} = p_i(1 - p_j) \text{ when } i=j$$

and

$$\frac{\partial p_j}{\partial z_i} = -p_i p_j \text{ when } i \neq j$$

which simplifies to

$$\frac{\partial L}{\partial Z_1} = -y_1 + y_1 p_1 + y_2 p_1 =$$

$$p_1 \sum (y_1 + y_2) - y_1$$

$$\frac{\partial L}{\partial Z_1} = p_1 - y_1$$

Since

$$\sum_i y_i = 1$$

Similarly

$$\frac{\partial L}{\partial z_2} = -\frac{y_1}{p_1} * (p_1 p_2) - \frac{y_2}{p_2} * p_2 (1 - p_2)$$

$$y_1 p_2 + y_2 p_2 - y_2$$

$$\frac{\partial L}{\partial Z_2} = p_2 \sum (y_1 + y_2) - y_2$$

$$= p_2 - y_2$$

In general this is of the form

$$\frac{\partial L}{\partial z_i} = p_i - y_i \quad - (A)$$

For e.g. if the probabilities computed were $p=[0.1, 0.7, 0.2]$ then this implies that the class with probability 0.7 is the likely class. This would imply that the 'One hot encoding' for y_i would be $y_i=[0,1,0]$ therefore the gradient $p_i - y_i = [0.1, -0.3, 0.2]$

Note: Further, we could extend this derivation for a Softmax activation output that outputs 3 classes

$$S = \begin{pmatrix} \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}} \\ \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}} \\ \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}} \end{pmatrix}$$

We could derive

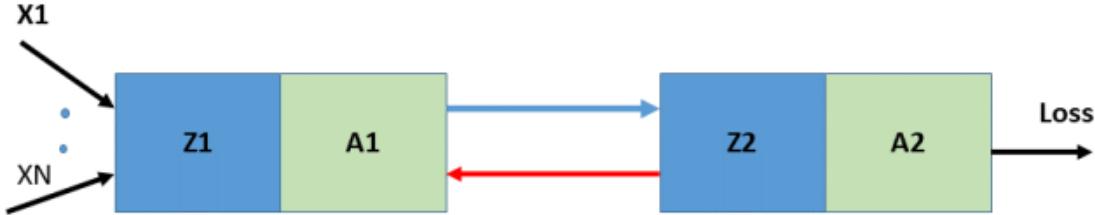
$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial p_1} \frac{\partial p_1}{\partial z_1} + \frac{\partial L}{\partial p_2} \frac{\partial p_2}{\partial z_1} + \frac{\partial L}{\partial p_3} \frac{\partial p_3}{\partial z_1} \text{ which similarly reduces to}$$

$$\frac{\partial L}{\partial z_1} = -\frac{y_1}{p_1} p_1 (1 - p_1) - \frac{y_2}{p_2} * (-p_2 p_1) - \frac{y_3}{p_3} * (-p_3 p_1)$$

$$-y_1 + y_1 p_1 + y_2 p_1 + y_3 p_1 = p_1 \sum (y_1 + y_2 + y_3) - y_1 = p_1 - y_1$$

interestingly, despite the lengthy derivations the final result is simple and intuitive!

As seen in chapter 3, the key equations for forward and backward propagation are



Forward propagation equations layer 1

$$Z_1 = W_1 X + b_1 \quad \text{and} \quad A_1 = g(Z_1)$$

Forward propagation equations layer 1

$$Z_2 = W_2 A_1 + b_2 \quad \text{and} \quad A_2 = S(Z_2)$$

Using the result (A) in the back propagation equations below we have

Backward propagation equations layer 2

$$\partial L / \partial W_2 = \partial L / \partial Z_2 * A_1 = (p_2 - y_2) * A_1$$

$$\partial L / \partial b_2 = \partial L / \partial Z_2 = p_2 - y_2$$

$$\partial L / \partial A_1 = \partial L / \partial Z_2 * W_2 = (p_2 - y_2) * W_2$$

Backward propagation equations layer 1

$$\partial L / \partial W_1 = \partial L / \partial Z_1 * A_0 = (p_1 - y_1) * A_0$$

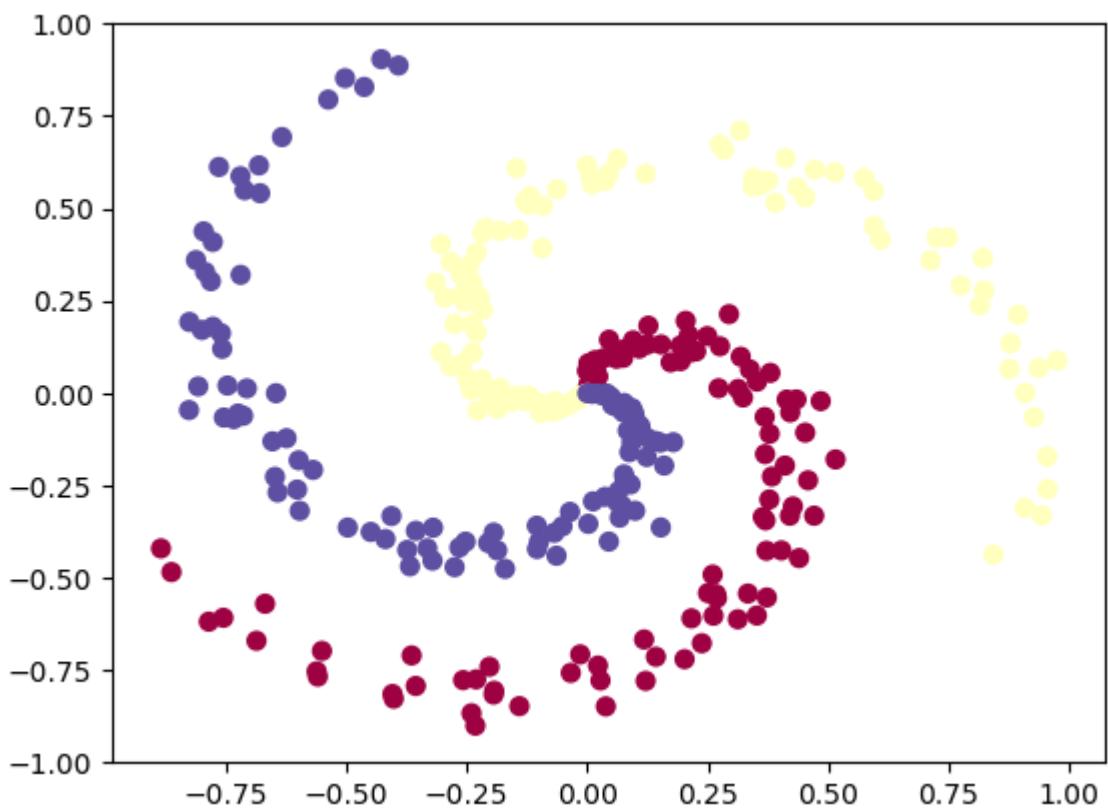
$$\partial L / \partial b_1 = \partial L / \partial Z_1 = (p_1 - y_1)$$

2. Spiral data set

As I mentioned earlier, I will be using the ‘spiral’ data from CS231n Convolutional Neural Networks (<http://cs231n.github.io/neural-networks-case-study/>) to ensure that my vectorized implementations in Python, R and Octave are correct. Here is the ‘spiral’ data set.

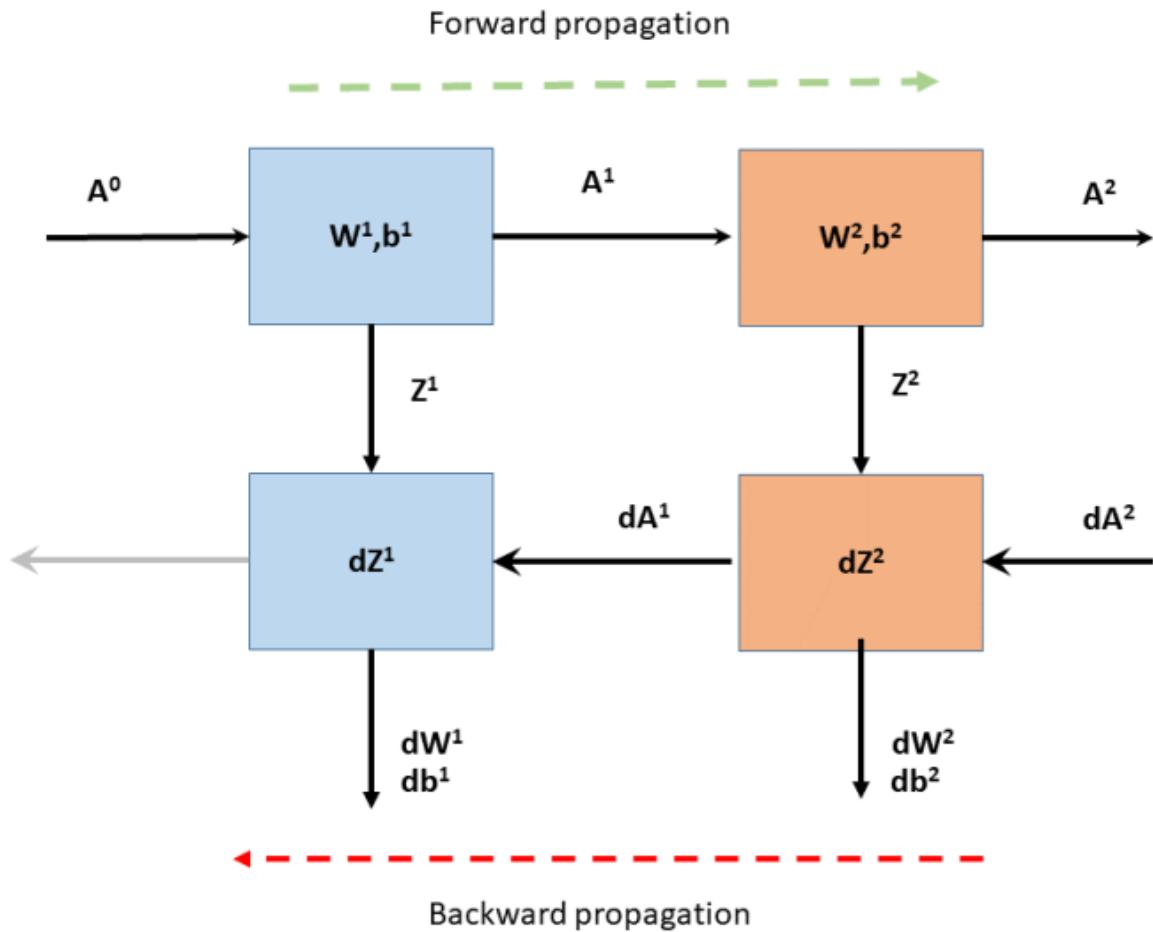
```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 exec(open("./DLfunctions41.py").read())
4
5 # Create an input data set - Taken from CS231n Convolutional Neural networks
6 # http://cs231n.github.io/neural-networks-case-study/
7 N = 100 # number of points per class
8 D = 2 # dimensionality
9 K = 3 # number of classes
10
11 x = np.zeros((N*K,D)) # data matrix (each row = single example)
12 y = np.zeros(N*K, dtype='uint8') # class labels
13 for j in range(K):
14     ix = range(N*j,N*(j+1))
15     r = np.linspace(0.0,1,N) # radius
16     t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
17     x[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
18     y[ix] = j
19
20 # Plot the data
21 plt.scatter(x[:, 0], x[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
22 plt.savefig("fig1.png", bbox_inches='tight')
```



The implementations of the vectorized Python, R and Octave code are shown diagrammatically

below



2.1 Multi-class classification with Softmax – Python code

A simple 2-layer Neural network with a single hidden layer, with 100 Relu activation units in the hidden layer and the Softmax activation unit in the output layer is used for multi-class classification. This Deep Learning Network, plots the non-linear boundary of the 3 classes as shown below

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import os
4 os.chdir("C:/junk/dl-4/dl-4")
5 exec(open("./DLfunctions41.py").read())
6
7 # Read the input data
8 N = 100 # number of points per class
9 D = 2 # dimensionality
10 K = 3 # number of classes
11 X = np.zeros((N*K,D)) # data matrix (each row = single example)
12 y = np.zeros(N*K, dtype='uint8') # class labels
13
14 #Loop
15 for j in range(K):

```

```

16 ix = range(N*j,N*(j+1))
17 r = np.linspace(0.0,1,N) # radius
18 t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
19 X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
20 y[ix] = j
21
22 # Set the number of features, hidden units in hidden layer and number of
23 # classes
24 numHidden=100 # No of hidden units in hidden layer
25 numFeats= 2 # dimensionality
26 numOutput = 3 # number of classes
27
28 # Initialize the model
29 parameters=initializeModel(numFeats,numHidden,numOutput)
30 w1= parameters['w1']
31 b1= parameters['b1']
32 w2= parameters['w2']
33 b2= parameters['b2']
34
35 # Set the learning rate
36 learningRate=0.6
37
38 # Initialize losses
39 losses=[]
40
41 # Perform Gradient descent
42 for i in range(10000):
43     # Forward propagation through hidden layer with Relu units
44     A1,cache1= layerActivationForward(X.T,w1,b1,'relu')
45
46     # Forward propagation through output layer with Softmax
47     A2,cache2 = layerActivationForward(A1,w2,b2,'softmax')
48
49     # No of training examples
50     numTraining = X.shape[0]
51     # Compute log probs. Take the log prob of correct class based on output y
52     correct_logprobs = -np.log(A2[range(numTraining),y])
53     # Compute loss
54     loss = np.sum(correct_logprobs)/numTraining
55
56     # Print the loss
57     if i % 1000 == 0:
58         print("iteration %d: loss %f" % (i, loss))
59         losses.append(loss)
60 dA=0
61
62     # Backward propagation through output layer with Softmax
63     dA1,dw2,db2 = layerActivationBackward(dA, cache2, y,
64 activationFunc='softmax')
65     # Backward propagation through hidden layer with Relu unit
66     dA0,dw1,db1 = layerActivationBackward(dA1.T, cache1, y,
67 activationFunc='relu')
68
69     #Update paramaters with the learning rate
70     w1 += -learningRate * dw1
71     b1 += -learningRate * db1
72     w2 += -learningRate * dw2.T
73     b2 += -learningRate * db2.T
74
75 #Plot losses vs iterations
76 i=np.arange(0,10000,1000)
77 plt.plot(i,losses)
78
79 plt.xlabel('Iterations')

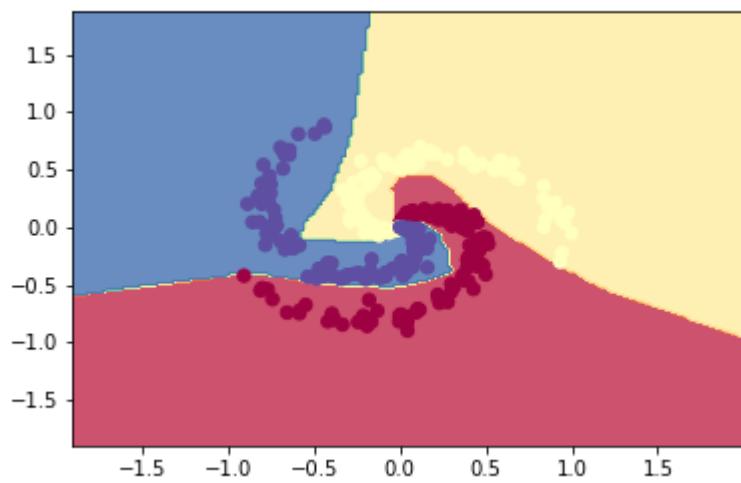
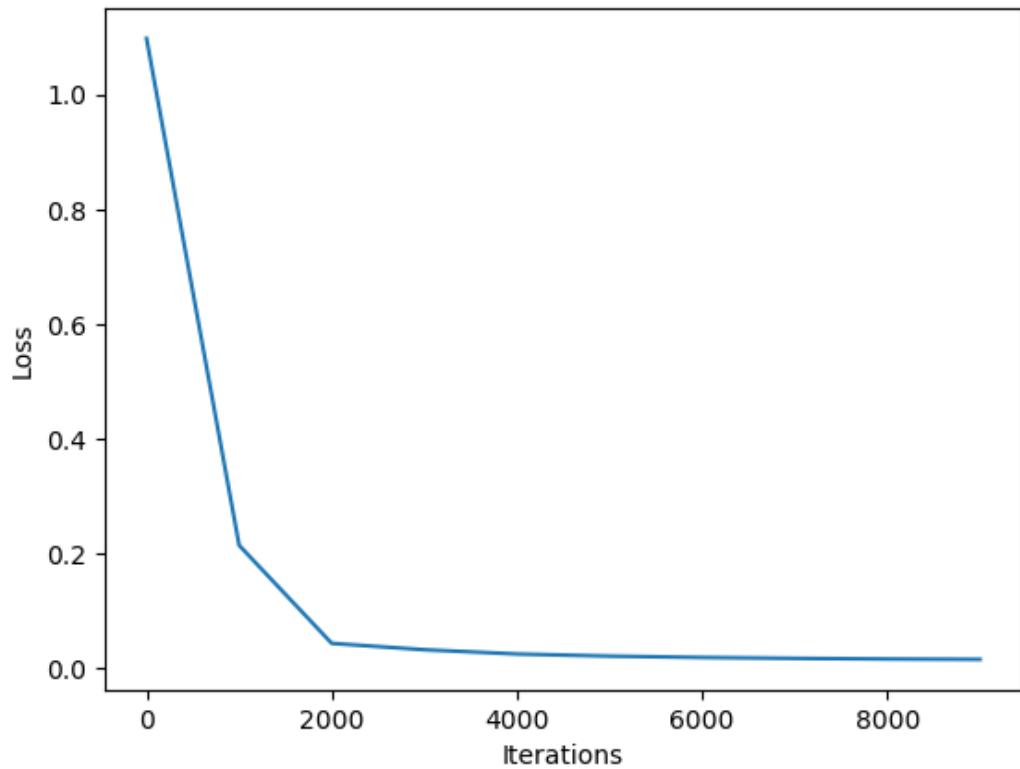
```

```

80 plt.ylabel('Loss')
81 plt.title('Losses vs Iterations')
82 plt.savefig("fig2.png", bbox="tight")
83
84 #Compute the multi-class Confusion Matrix
85 from sklearn.metrics import confusion_matrix
86 from sklearn.metrics import accuracy_score, precision_score, recall_score,
87 f1_score
88
89 # We need to determine the predicted values from the learnt data
90 # Forward propagation through hidden layer with Relu units
91 A1,cache1= layerActivationForward(X.T,w1,b1,'relu')
92
93 # Forward propagation through output layer with Softmax
94 A2,cache2 = layerActivationForward(A1,w2,b2,'softmax')
95 #Compute predicted values from weights and biases
96 yhat=np.argmax(A2, axis=1)
97
98 # Compute the confusion matrix
99 a=confusion_matrix(y.T,yhat.T)
100 print("Multi-class Confusion Matrix")
101 print(a)
102 ## iteration 0: loss 1.098507
103 ## iteration 1000: loss 0.214611
104 ## iteration 2000: loss 0.043622
105 ## iteration 3000: loss 0.032525
106 ## iteration 4000: loss 0.025108
107 ## iteration 5000: loss 0.021365
108 ## iteration 6000: loss 0.019046
109 ## iteration 7000: loss 0.017475
110 ## iteration 8000: loss 0.016359
111 ## iteration 9000: loss 0.015703
112
113 ## Multi-class Confusion Matrix
114 ## [[ 99   1   0]
115 ## [   0 100   0]
116 ## [   0   1  99]]

```

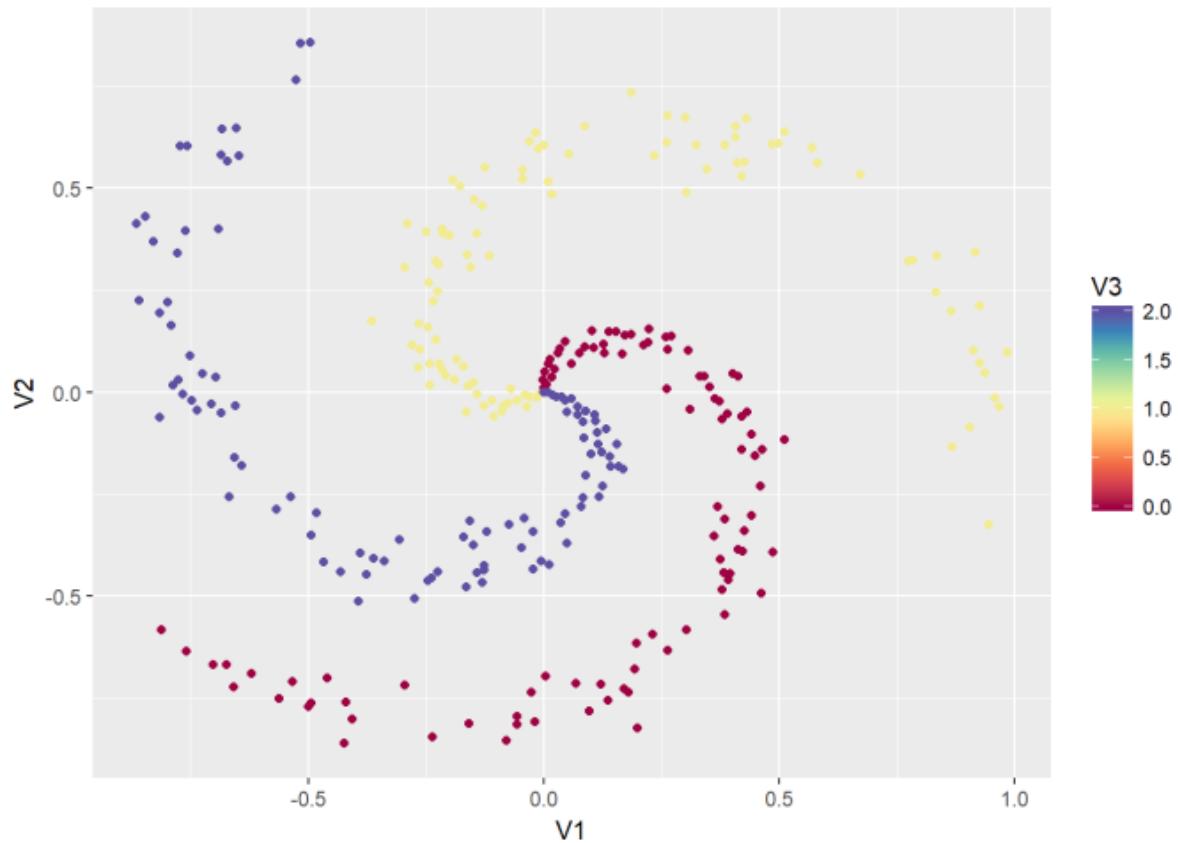
Losses vs Iterations



2.2 Multi-class classification with Softmax – R code

The spiral data set created with Python was saved (spiral.csv), and is used as the input with R code. To compute the softmax derivative I create matrices for the One Hot Encoded y_i and then stack them before subtracting $p_i - y_i$.

```
1 library(ggplot2)
2 library(dplyr)
3 library(RColorBrewer)
4 source("DLfunctions41.R")
5
6 # Read the spiral dataset
7 z <- as.matrix(read.csv("spiral.csv", header=FALSE))
8 z1=data.frame(z)
9
10 #Plot the dataset
11 ggplot(z1,aes(x=v1,y=v2,col=v3)) +geom_point() +
12   scale_colour_gradientn(colours = brewer.pal(10, "Spectral"))
```



```
1 # Setup the data
2 X <- z[,1:2]
3 y <- z[,3]
4 X1 <- t(X)
5 Y1 <- t(y)
6
7 # Initialize number of features, number of hidden units in hidden layer and
8 # number of classes
```

```

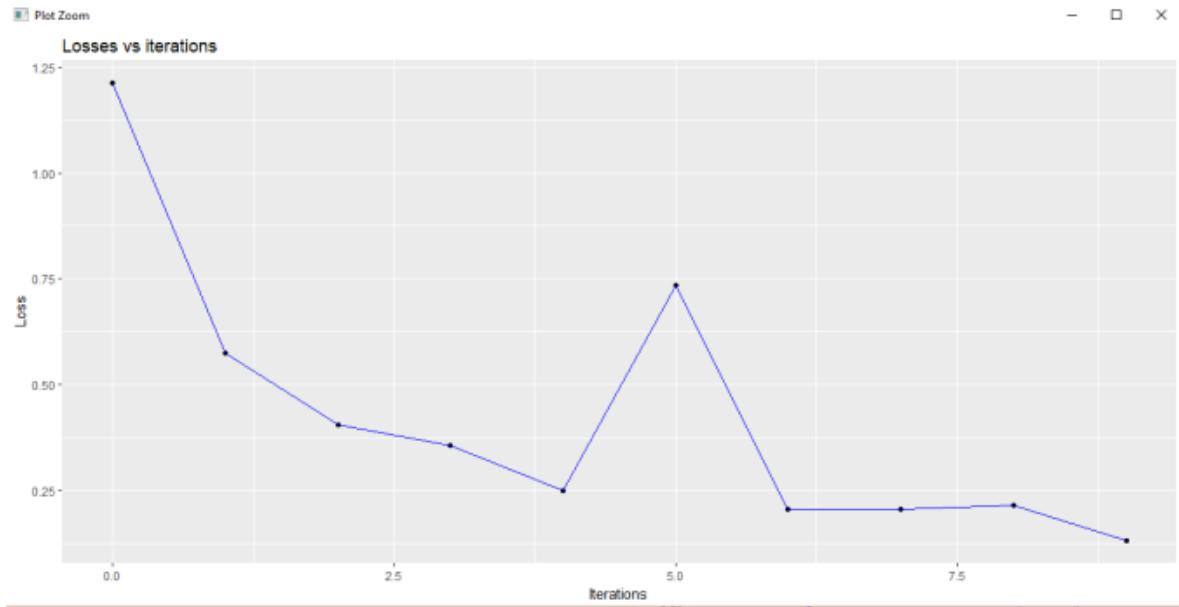
9 numFeats<-2 # No features
10 numHidden<-100 # No of hidden units
11 numOutput<-3 # No of classes
12
13 # Initialize model
14 parameters <- initializeModel(numFeats, numHidden, numOutput)
15
16 # Get the parameters
17 w1 <- parameters[['w1']]
18 b1 <- parameters[['b1']]
19 w2 <- parameters[['w2']]
20 b2 <- parameters[['b2']]
21
22 # Set the learning rate
23 learningRate <- 0.5
24
25 # Initialize losses
26 losses <- NULL
27
28 # Perform gradient descent
29 for(i in 0:9000){
30     # Forward propagation through hidden layer with Relu units
31     retvals <- layerActivationForward(X1,w1,b1,'relu')
32     A1 <- retvals[['A']]
33     cache1 <- retvals[['cache']]
34     forward_cache1 <- cache1[['forward_cache1']]
35     activation_cache <- cache1[['activation_cache']]
36
37     # Forward propagation through output layer with Softmax units
38     retvals = layerActivationForward(A1,w2,b2,'softmax')
39     A2 <- retvals[['A']]
40     cache2 <- retvals[['cache']]
41     forward_cache2 <- cache2[['forward_cache1']]
42     activation_cache2 <- cache2[['activation_cache']]
43
44     # No of training examples
45     numTraining <- dim(X)[1]
46     dA <-0
47
48     # Select the elements where the y values are 0, 1 or 2 and make a
49     vector
50     a=c(A2[y==0,1],A2[y==1,2],A2[y==2,3])
51
52     # Compute probabilities
53     correct_probs = -log(a)
54     # Compute loss
55     loss= sum(correct_probs)/numTraining
56
57     if(i %% 1000 == 0){
58         sprintf("iteration %d: loss %f",i, loss)
59         print(loss)
60     }
61
62     # Backward propagation through output layer with Softmax units
63     retvals = layerActivationBackward(dA, cache2, y,
64     activationFunc='softmax')
65     dA1 = retvals[['dA_prev']]
66     dw2= retvals[['dw']]
67     db2= retvals[['db']]
68
69     # Backward propagation through hidden layer with Relu units
70     retvals = layerActivationBackward(t(dA1), cache1, y,
71     activationFunc='relu')
72     dA0 = retvals[['dA_prev']]

```

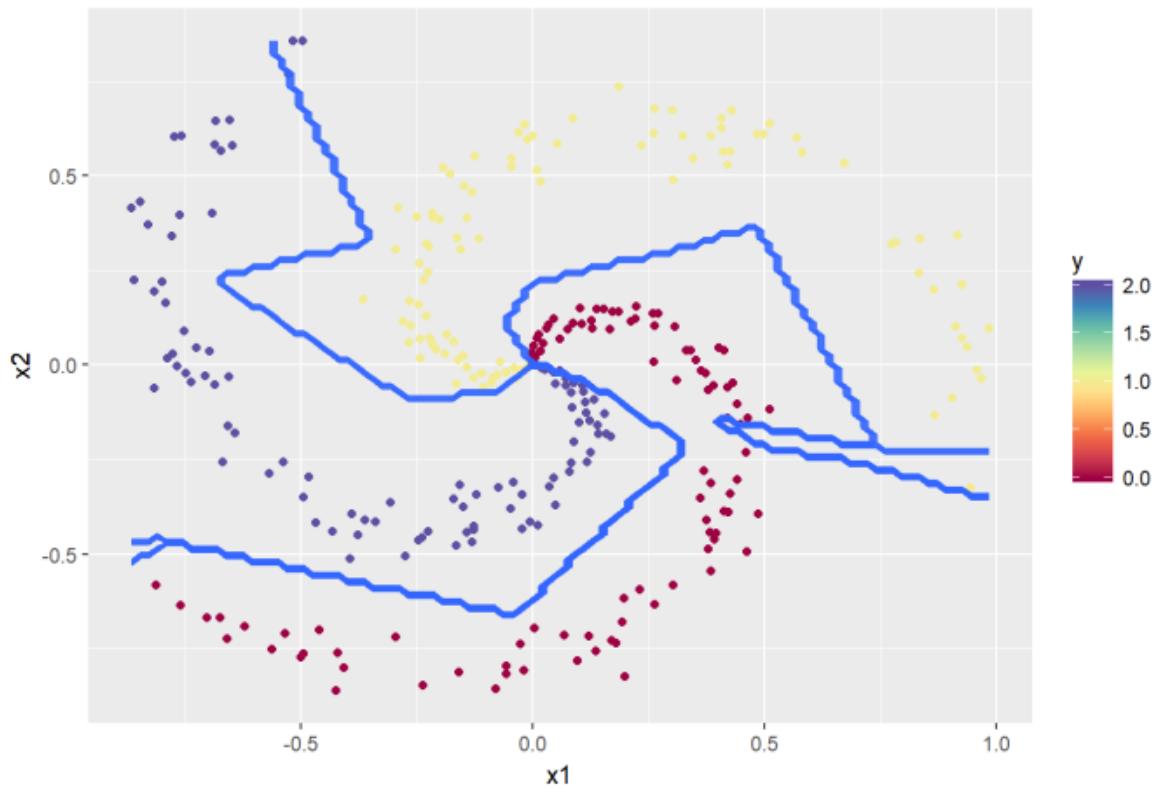
```

73      dw1= retvals[['dw']]
74      db1= retvals[['db']]
75
76      # Update parameters
77      w1 <- w1 - learningRate * dw1
78      b1 <- b1 - learningRate * db1
79      w2 <- w2 - learningRate * t(dw2)
80      b2 <- b2 - learningRate * t(db2)
81
82      ## [1] 1.212487
83      ## [1] 0.5740867
84      ## [1] 0.4048824
85      ## [1] 0.3561941
86      ## [1] 0.2509576
87      ## [1] 0.7351063
88      ## [1] 0.2066114
89      ## [1] 0.2065875
90      ## [1] 0.2151943
91      ## [1] 0.1318807
92
93
94      #Create iterations
95      iterations <- seq(0,10)
96      df=data.frame(iterations,losses)
97
98      # Plot cost vs iterations
99      ggplot(df,aes(x=iterations,y=losses)) + geom_point() +
100         geom_line(color="blue") +
101         ggttitle("Losses vs iterations") + xlab("Iterations") + ylab("Loss")
102
103      #Plot the decision boundary
104      plotDecisionBoundary(z,w1,b1,w2,b2)

```



Decision boundary



Multi-class Confusion Matrix

```

1 library(caret)
2 library(e1071)
3
4 # Forward propagation through hidden layer with Relu units
5 retvals <- layerActivationForward(x1,w1,b1,'relu')
6 A1 <- retvals[['A']]
7
8 # Forward propagation through output layer with softmax units

```

```

9  retvals = layerActivationForward(A1,w2,b2,'softmax')
10 A2 <- retvals[['A']]
11 yhat <- apply(A2, 1,which.max) -1
12 Confusion Matrix and Statistics
13     Reference
14 Prediction 0 1 2
15      0 97 0 1
16      1 2 96 4
17      2 1 4 95
18
19 Overall Statistics
20     Accuracy : 0.96
21     95% CI : (0.9312, 0.9792)
22     No Information Rate : 0.3333
23     P-Value [Acc > NIR] : <2e-16
24
25     Kappa : 0.94
26     Mcnemar's Test P-Value : 0.5724
27 Statistics by Class:
28
29                     class: 0 class: 1 class: 2
30 Sensitivity          0.9700   0.9600   0.9500
31 Specificity          0.9950   0.9700   0.9750
32 Pos Pred Value       0.9898   0.9412   0.9500
33 Neg Pred Value       0.9851   0.9798   0.9750
34 Prevalence           0.3333   0.3333   0.3333
35 Detection Rate       0.3233   0.3200   0.3167
36 Detection Prevalence 0.3267   0.3400   0.3333
37 Balanced Accuracy    0.9825   0.9650   0.9625

```

2.3 Multi-class classification with Softmax – Octave code

A 2-layer neural network with the Softmax activation unit in the output layer is constructed in Octave. The same spiral data (spiral.csv) set is used for Octave also

```

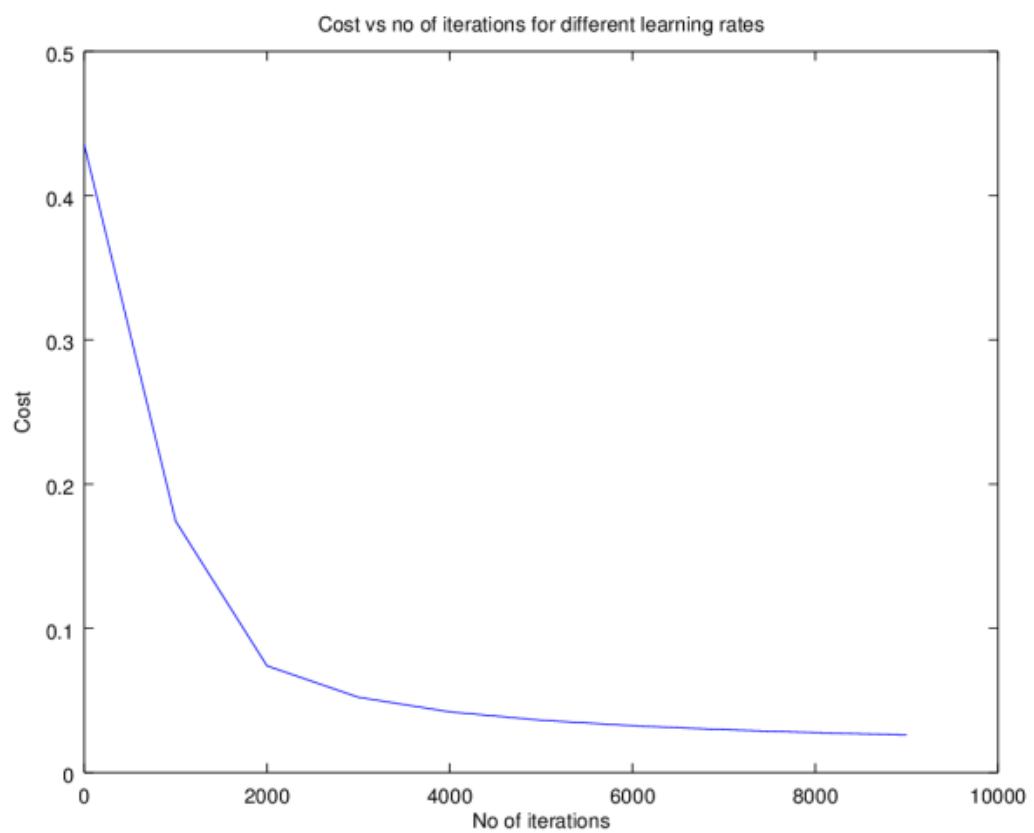
1 source("DL41functions.m")
2 # Read the spiral data
3 data=csvread("spiral.csv");
4 # Setup the data
5 X=data(:,1:2);
6 Y=data(:,3);
7
8 # Set the number of features, number of hidden units in hidden layer and
9 number of classes
10 numFeats=2; #No features
11 numHidden=100; # No of hidden units
12 numOutput=3; # No of classes
13
14 # Initialize model
15 [w1 b1 w2 b2] = initializeModel(numFeats,numHidden,numOutput);
16 # Initialize losses
17 losses=[]
18
19 #Initialize learningRate
20 learningRate=0.5;
21
22 for k =1:10000
23     # Forward propagation through hidden layer with Relu units
24     [A1,cache1 activation_cache1]=
25 layerActivationForward(X',w1,b1,activationFunc ='relu');

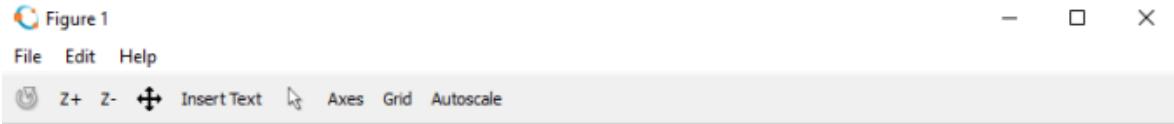
```

```

26      # Forward propagation through output layer with Softmax units
27      [A2,cache2 activation_cache2] =
28      layerActivationForward(A1,w2,b2,activationFunc='softmax');
29
30      # No of training examples
31      numTraining = size(X)(1);
32      # Select rows where Y=0,1, and 2 and concatenate to a long vector
33      a=[A2(Y==0,1) ;A2(Y==1,2) ;A2(Y==2,3)];
34
35      #Select the correct column for log prob
36      correct_probs = -log(a);
37
38      #Compute log loss
39      loss= sum(correct_probs)/numTraining;
40      if(mod(k,1000) == 0)
41          disp(loss);
42          losses=[losses loss];
43      endif
44      dA=0;
45
46      # Backward propagation through output layer with Softmax units
47      [dA1 dw2 db2] = layerActivationBackward(dA, cache2,
48      activation_cache2,Y,activationFunc='softmax');
49
50      # Backward propagation through hidden layer with Relu units
51      [dA0,dw1,db1] = layerActivationBackward(dA1', cache1, activation_cache1,
52      Y, activationFunc='relu');
53      #Update parameters
54      w1 += -learningRate * dw1;
55      b1 += -learningRate * db1;
56      w2 += -learningRate * dw2';
57      b2 += -learningRate * db2';
58  endfor
59
60  # Plot Losses vs Iterations
61  iterations=0:1000:9000
62  plotCostVsIterations(iterations,losses)
63
64  # Plot the decision boundary
65  plotDecisionBoundary( X,Y,w1,b1,w2,b2)
66

```





The code for the Python, R and Octave implementations can be downloaded from Github at DeepLearningFromFirstPrinciples (<https://github.com/tvganesh/DeepLearningFromFirstPrinciples/tree/master/Chap4-MulticlassDeepLearningNetwork>)

Conclusion

In this chapter, I implemented a 2 layer Neural Network with the Softmax classifier. In chapter 3, I implemented a multi-layer Deep Learning Network. The Softmax activation unit is included into the generalized multi-layer Deep Network along with the other activation units of sigmoid, tanh and relu in chapter 5.

5.MNIST classification with Softmax

- a. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
- b. A robot must obey orders given it by human beings except where such orders would conflict with the First Law.
- c. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

Isaac Asimov's Three Laws of Robotics

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C Clarke.

1. Introduction

In this 5th chapter on Deep Learning from first Principles in Python, R and Octave, I solve the MNIST data set of handwritten digits (shown below), from the basics. To do this, I construct a L-Layer, vectorized Deep Learning implementation in Python, R and Octave from scratch and classify the MNIST data set. The MNIST training data set contains 60000 handwritten digits from 0-9, and a test set of 10000 digits. MNIST, is a popular dataset for running Deep Learning tests, and has been rightfully termed as the ‘drosophila’ of Deep Learning, by none other than the venerable Prof Geoffrey Hinton.



This chapter largely builds upon chapter 3, in which I implemented a multi-layer Deep Learning network, with an arbitrary number of hidden layers and activation units per hidden layer and with

the sigmoid output layer. To this, I add the derivation of the Jacobian of a Softmax, the Cross entropy loss and the computations of gradient equations for a multi-class Softmax classifier which I had done in chapter 4

By combining the implementations of chapter 3 & 4 I build a generic, L-layer Deep Learning network, with arbitrary number of hidden layers and hidden units, which can do both binary (sigmoid) and multi-class (softmax) classification. The implementations of the functions invoked in this chapter are in Appendix 5 - MNIST classification with Softmax

The generic, vectorized L-Layer Deep Learning Network implementations in Python, R and Octave can be cloned/downloaded from GitHub at DeepLearningFromFirstPrinciples (<https://github.com/tvganesh/DeepLearningFromFirstPrinciples/tree/master/Chap5-MNISTMultiClassDLNetwork>). This implementation allows for arbitrary number of hidden layers and hidden layer units. The activation function at the hidden layers can be one of sigmoid, relu and tanh. The output activation can do binary classification with the ‘sigmoid’ function or multi-class classification with ‘softmax’. Feel free to download and play around with the code!

Since regular gradient descent on 60,000 training samples on a laptop will result in memory problem on such a large dataset. So, this chapter also implements Stochastic Gradient Descent (SGD) for Python, R and Octave. This chapter also includes the implementation of a numerically stable version of Softmax, to prevent the softmax and its derivative resulting in NaNs.

2. Numerically stable Softmax

The Softmax function $S_j = \frac{e^{Z_j}}{\sum_i^k e^{Z_i}}$ can be numerically unstable because of the division of large exponentials. To handle this problem we have to implement stable Softmax function as below

$$S_j = \frac{e^{Z_j}}{\sum_i^k e^{Z_i}}$$

$$\text{Therefore } S_j = \frac{e^{Z_j+D}}{\sum_i^k e^{Z_i+D}}$$

Here ‘D’ can be anything. A common choice is

$$D = -\max(Z_1, Z_2, \dots, Z_k)$$

Here is the stable Softmax implementation in Python

```

1 # A numerically stable softmax implementation
2 def stableSoftmax(z):
3     #Compute the softmax of vector x in a numerically stable way.
4     shiftz = z.T - np.max(z.T, axis=1).reshape(-1,1)
5     exp_scores = np.exp(shiftz)
6
7     # Normalize them for each example
8     A = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
9     cache=z
10    return A,cache

```

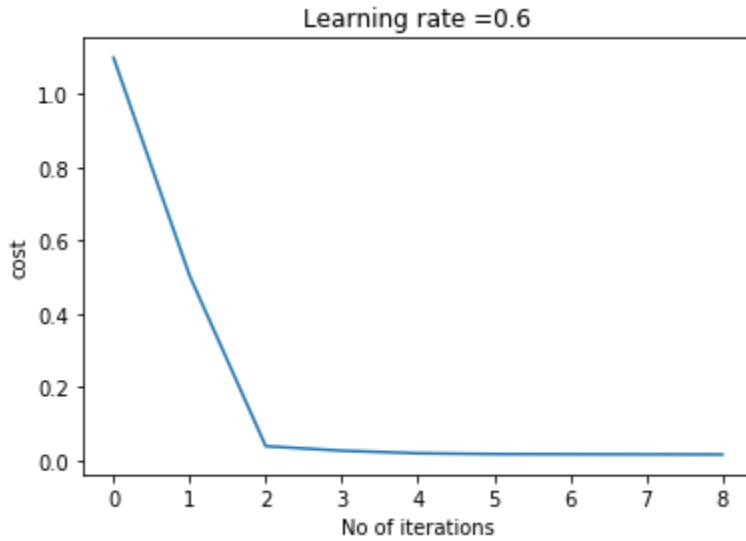
While trying to create an L-Layer generic Deep Learning network in the 3 languages, I found it useful to ensure that the model executed correctly on smaller datasets. You can run into numerous problems while setting up the matrices, which becomes extremely difficult to debug. So in this post, I run the model on 2 smaller data for sets used in my earlier posts (chapter 3 & chapter 4) , in each of the languages, before running the generic model on MNIST.

Here is a fair warning: - if you think you can dive directly into Deep Learning, with just some basic knowledge of Machine Learning, you are bound to run into serious issues. Moreover, your knowledge will be incomplete. It is essential that you have a good grasp of Machine and Statistical Learning, the different algorithms, the measures and metrics for selecting the models etc. It would help to be conversant with all the ML models, ML concepts, validation techniques, classification measures etc. Check out the internet/books for background.

3.1a Random dataset with sigmoid activation – Python

This random data with 9 clusters, used in chapter 3 was used to test the complete L-layer Deep Learning network with Sigmoid activation.

```
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import pandas as pd
5 from sklearn.datasets import make_classification, make_blobs
6 exec(open("DLfunctions51.py").read()) # Cannot import in Rmd
7
8 # Create a random data set with 9 centers
9 X1, Y1 = make_blobs(n_samples = 400, n_features = 2, centers = 9, cluster_std
10 = 1.3, random_state =4)
11
12 #Create 2 classes
13 Y1=Y1.reshape(400,1)
14 Y1 = Y1 % 2
15 X2=X1.T
16 Y2=Y1.T
17
18 # Set the dimensions of L -layer DL network
19 # 2 - number of input features
20 # 9,9 - 2 hidden layers with 9 activation units
21 # 1 - 1 activation unit in the output layer
22 layersDimensions = [2, 9, 9,1] # 4-layer model
23
24 # Execute the L-Layer DL network with hidden activation=relu and sigmoid
25 # output function
26 parameters = L_Layer_DeepModel(X2, Y2, layersDimensions,
27 hiddenActivationFunc='relu', outputActivationFunc="sigmoid", learningRate =
28 0.3,num_iterations = 2500, print_cost = True)
```



3.1b Spiral dataset with Softmax activation – Python

The Spiral data of chapter 4 was used to test the correct functioning of the complete L-layer Deep Learning network for multi-class classification with Softmax activation.

```

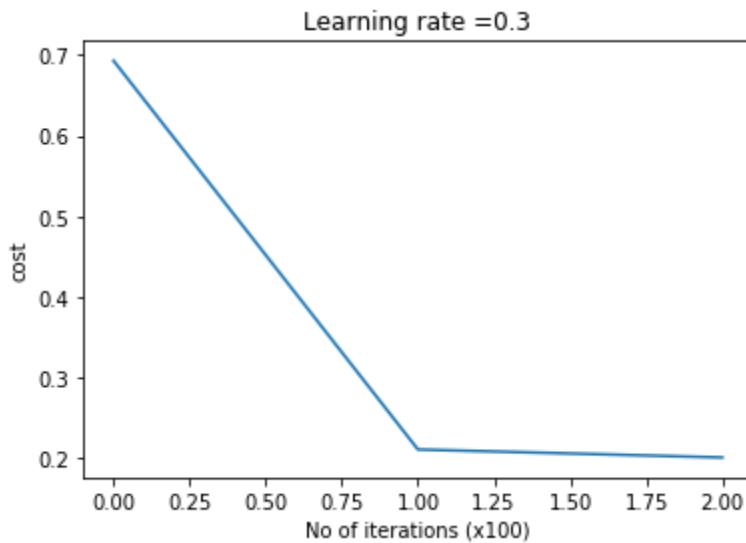
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import pandas as pd
5 from sklearn.datasets import make_classification, make_blobs
6 exec(open("DLfunctions51.py").read())
7
8 # Create an input data set - Taken from CS231n Convolutional Neural networks
9 # http://cs231n.github.io/neural-networks-case-study/
10 N = 100 # number of points per class
11 D = 2 # dimensionality
12 K = 3 # number of classes
13 X = np.zeros((N*K,D)) # data matrix (each row = single example)
14 y = np.zeros(N*K, dtype='uint8') # class labels
15
16 # Loop over K
17 for j in range(K):
18     ix = range(N*j,N*(j+1))
19     r = np.linspace(0.0,1,N) # radius
20     t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
21     X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
22     y[ix] = j
23 X1=X.T
24 Y1=y.reshape(-1,1).T
25
26 Set the parameters of the DL network
27 numHidden=100 # No of hidden units in hidden layer
28 numFeats= 2 # dimensionality
29 numOutput = 3 # number of classes
30 # Set the dimensions of the layers
31 layersDimensions=[numFeats,numHidden,numOutput]
32
33 # Execute the L-layer DL network with hidden activation=relu and softmax
34 output function

```

```

35 parameters = L_Layer_DeepModel(X1, Y1, layersDimensions,
36 hiddenActivationFunc='relu', outputActivationFunc="softmax", learningRate =
37 0.6,num_iterations = 9000, print_cost = True)
38 ## Cost after iteration 0: 1.098759
39 ## Cost after iteration 1000: 0.112666
40 ## Cost after iteration 2000: 0.044351
41 ## Cost after iteration 3000: 0.027491
42 ## Cost after iteration 4000: 0.021898
43 ## Cost after iteration 5000: 0.019181
44 ## Cost after iteration 6000: 0.017832
45 ## Cost after iteration 7000: 0.017452
46 ## Cost after iteration 8000: 0.017161

```



3.1c MNIST dataset with Softmax activation – Python

In the code below, I execute Stochastic Gradient Descent on the MNIST training data of 60000. I used a mini-batch size of 1000. Python takes about 40 minutes to crunch the data. In addition, I also compute the Confusion Matrix and other metrics like Accuracy, Precision and Recall for the MNIST data set. I get an accuracy of 0.93 on the MNIST test set. This accuracy can be improved by choosing more hidden layers or more hidden units and possibly tweaking the learning rate and the number of epochs.

```

1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import pandas as pd
5 import math
6 from sklearn.datasets import make_classification, make_blobs
7 from sklearn.metrics import confusion_matrix
8 from sklearn.metrics import accuracy_score, precision_score, recall_score,
9 f1_score
10 exec(open("DLfunctions51.py").read())
11 exec(open("load_mnist.py").read())
12
13 # Read the MNIST training and test sets
14 training=list(read(dataset='training',path=".\\mnist"))
15 test=list(read(dataset='testing',path=".\\mnist"))

```

```

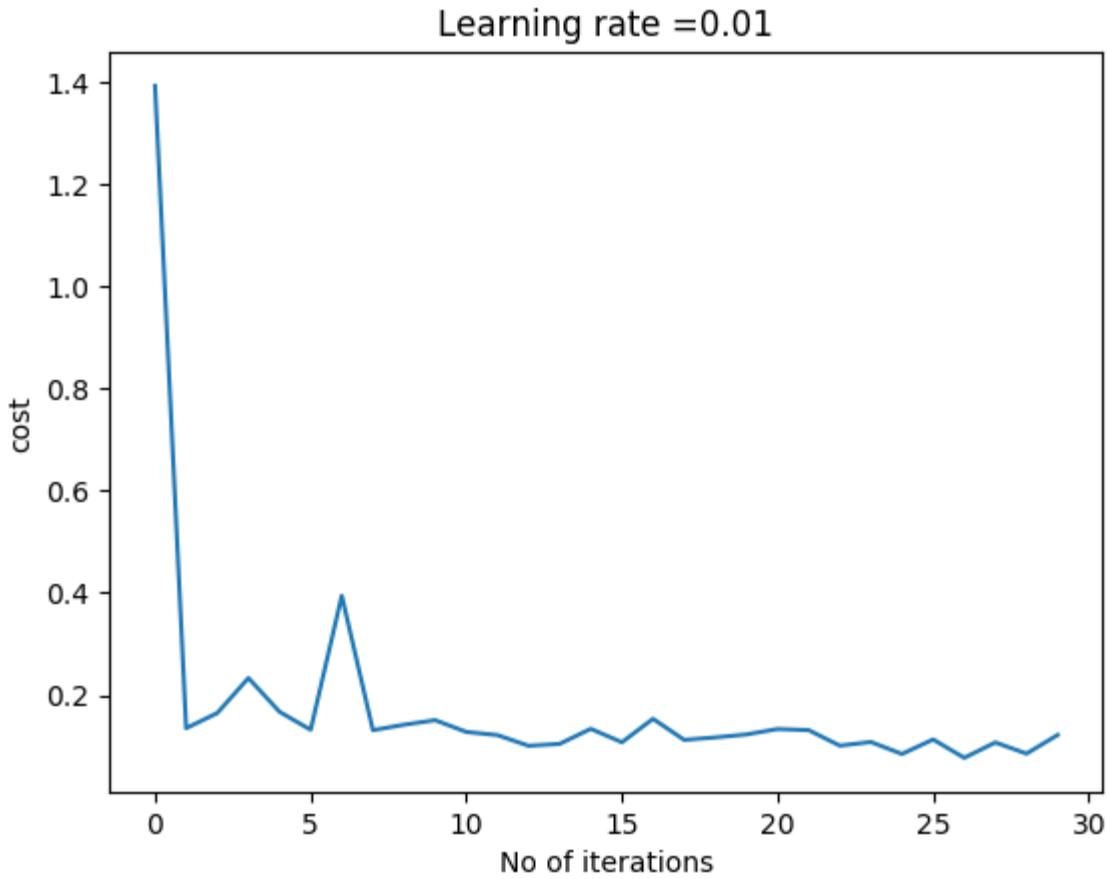
16 # Create labels and pixel arrays
17 lbls=[]
18 pxls=[]
19 print(len(training))
20 #for i in range(len(training)):
21 for i in range(60000):
22     l,p=training[i]
23     lbls.append(l)
24     pxls.append(p)
25 labels= np.array(lbls)
26 pixels=np.array(pxls)
27 y=labels.reshape(-1,1)
28 x=pixels.reshape(pixels.shape[0],-1)
29 X=x.T
30 Y=y.T
31
32 # Set the dimensions of the layers. The MNIST data is 28x28 pixels= 784
33 # Hence input layer is 784. For the 10 digits the softmax classifier
34 # has to handle 10 outputs
35 # 784 - No of input features- 28 x27
36 # 15,9 - 2 hidden layers with 15 & 9 hidden units
37 #10 - 10 outputs at the output layer for 10 digits
38 layersDimensions=[784, 15, 9, 10] # Works very well, lr=0.01,mini_batch =1000,
39 total=20000
40 np.random.seed(1)
41 costs = []
42
43 # Execute the L-Layer Deep Learning network with Stochastic Gradient Descent
44 # with Learning Rate=0.01, mini batch size=1000
45 # number of epochs=3000
46 # hidden units= relu
47 # output activation unit =softmax
48 parameters = L_Layer_DeepModel_SGD(x1, Y1, layersDimensions,
49 hiddenActivationFunc='relu', outputActivationFunc="softmax",learningRate =
50 0.01 ,mini_batch_size =1000, num_epochs = 3000, print_cost = True)
51
52 # Compute the Confusion Matrix on Training set
53 # Compute the training accuracy, precision and recall
54 proba=predict_proba(parameters, X1,outputActivationFunc="softmax")
55 #A2, cache = forwardPropagationDeep(X1, parameters)
56 #proba=np.argmax(A2, axis=0).reshape(-1,1)
57 a=confusion_matrix(Y1.T,proba)
58 print(a)
59 from sklearn.metrics import accuracy_score, precision_score, recall_score,
60 f1_score
61 print('Accuracy: {:.2f}'.format(accuracy_score(Y1.T, proba)))
62 print('Precision: {:.2f}'.format(precision_score(Y1.T,
63 proba,average="micro")))
64 print('Recall: {:.2f}'.format(recall_score(Y1.T, proba,average="micro")))
65
66 # Read the test data
67 lbls=[]
68 pxls=[]
69 print(len(test))
70
71 # Loop
72 for i in range(10000):
73     l,p=test[i]
74     lbls.append(l)
75     pxls.append(p)
76 testLabels= np.array(lbls)
77 testPixels=np.array(pxls)
78 ytest=testLabels.reshape(-1,1)
79 xtest=testPixels.reshape(testPixels.shape[0],-1)

```

```

80 X1test=Xtest.T
81 Y1test=ytest.T
82
83 # Compute the Confusion Matrix on Test set
84 # Compute the test accuracy, precision and recall
85 probaTest=predict_proba(parameters, X1test,outputActivationFunc="softmax")
86 #A2, cache = forwardPropagationDeep(X1, parameters)
87 #proba=np.argmax(A2, axis=0).reshape(-1,1)
88 a=confusion_matrix(Y1test.T,probaTest)
89
90 # Print accuracy, recall, precision and the 10-digit confusion matrix
91 from sklearn.metrics import accuracy_score, precision_score, recall_score,
92 f1_score
93 print('Accuracy: {:.2f}'.format(accuracy_score(Y1test.T, probaTest)))
94 print('Precision: {:.2f}'.format(precision_score(Y1test.T,
95 probaTest,average="micro")))
96 print('Recall: {:.2f}'.format(recall_score(Y1test.T,
97 probaTest,average="micro")))
98
99 ##1. Confusion Matrix of Training set
100    0   1   2   3   4   5   6   7   8   9
101 [[5854  0  19   2  10   7   0   1  24   6]
102 [ 1 6659  30  10   5   3   0  14  20   0]
103 [ 20  24 5805  18   6  11   2  32  37   3]
104 [ 5   4 175 5783  1  27   1  58  60  17]
105 [ 1  21   9   0 5780  0   5   2  12  12]
106 [ 29  9  21 224   6 4824  18  17 245  28]
107 [ 5   4  22   1  32  12 5799  0  43   0]
108 [ 3  13 148 154  18   3   0 5883  4  39]
109 [ 11  34  30  21  13  16   4   7 5703  12]
110 [ 10  4   1  32 135  14   1  92 134 5526]]
111
112 ##2. Accuracy, Precision, Recall of Training set
113 ## Accuracy: 0.96
114 ## Precision: 0.96
115 ## Recall: 0.96
116
117 ##3. Confusion Matrix of Test set
118    0   1   2   3   4   5   6   7   8   9
119 [[ 954  1   8   0   3   3   2   4   4   1]
120 [ 0 1107  6   5   0   0   1   2  14   0]
121 [ 11  7 957 10   5   0   5  20  16   1]
122 [ 2   3 37 925  3  13   0   8  18   1]
123 [ 2   6  1   1 944  0   7   3   4  14]
124 [ 12  5   4  45   2 740  24   8  42  10]
125 [ 8   4   4   2  16   9 903  0  12   0]
126 [ 4  10  27 18   5   1   0 940  1  22]
127 [ 11  13  6 13   9  10   7   2 900  3]
128 [ 8   5   1   7  50   7   0  20  29 882]]
129 ##4. Accuracy, Precision, Recall of Training set
130 ## Accuracy: 0.93
131 ## Precision: 0.93
132 ## Recall: 0.93

```



3.2a Random dataset with sigmoid activation – R code

The random data set used in the Python code above which was saved as a csv(data.csv). The code is used to test the L -Layer DL network with sigmoid Activation in R.

```

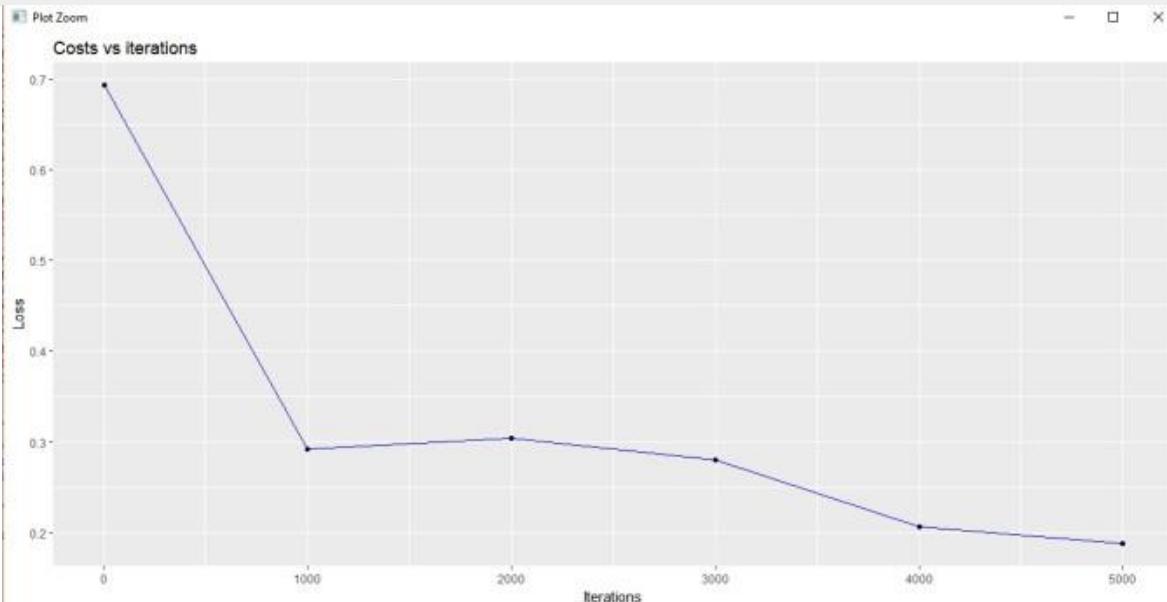
1 source("DLfunctions5.R")
2
3 # Read the random data set
4 z <- as.matrix(read.csv("data.csv",header=FALSE))
5 x <- z[,1:2]
6 y <- z[,3]
7 x <- t(x)
8 Y <- t(y)
9
10 # Set the dimensions of the layers
11 # 2 - number of input features]
12 #9, 9 - 2 hidden layers with 9 hidden units each
13 # 1 - 1 activation unit in the output layer
14 layersDimensions = c(2, 9, 9,1)
15
16 # Execute a L-Layer DL network on the data set with relu hidden unit
17 activation
18 # sigmoid activation unit in the output layer
19 retvals = L_Layer_DeepModel(x, Y, layersDimensions,
20                             hiddenActivationFunc='relu',

```

```

21                               outputActivationFunc="sigmoid",
22                               learningRate = 0.3,
23                               numIterations = 5000,
24                               print_cost = True)
25
26 #Plot the cost vs iterations
27 iterations <- seq(0,5000,1000)
28 costs=retvals$costs
29 df=data.frame(iterations,costs)
30 ggplot(df,aes(x=iterations,y=costs)) + geom_point() + geom_line(color="blue")
31 + ggttitle("Costs vs iterations") + xlab("Iterations") + ylab("Loss")

```



3.2b Spiral dataset with Softmax activation – R

The spiral data set used in the Python code above(spiral.csv) , is reused to test multi-class classification with Softmax.

```

1 source("DLfunctions5.R")
2 Z <- as.matrix(read.csv("spiral.csv",header=FALSE))
3
4 # Setup the data
5 X <- Z[,1:2]
6 y <- Z[,3]
7 X <- t(X)
8 Y <- t(y)
9
10 # Initialize number of features, number of hidden units in hidden layer and
11 # number of classes
12 numFeats<-2 # No features
13 numHidden<-100 # No of hidden units
14 numOutput<-3 # No of classes
15
16 # Set the layer dimensions
17 layersDimensions = c(numFeats,numHidden,numOutput)
18

```

```

19 # Execute the L-Layer Deep Learning network with relu activation unit for
20 hidden layer
21 # and softmax activation in the output
22 retvals = L_Layer_DeepModel(x, Y, layersDimensions,
23                             hiddenActivationFunc='relu',
24                             outputActivationFunc="softmax",
25                             learningRate = 0.5,
26                             numIterations = 9000,
27                             print_cost = True)
28
29
30 #Plot cost vs iterations
31 iterations <- seq(0,9000,1000)
32 costs=retvals$costs
33 df=data.frame(iterations,costs)
34 ggplot(df,aes(x=iterations,y=costs)) + geom_point() + geom_line(color="blue")
35 + ggttitle("Costs vs iterations") + xlab("Iterations") + ylab("Costs")

```

3.2c MNIST dataset with Softmax activation – R

The code below executes a L –Layer Deep Learning network with Softmax output activation, to classify the 10 handwritten digits from MNIST with Stochastic Gradient Descent. The 32768 random samples from the data set was used to train the data.

Having said that, the Confusion Matrix in R dumps a lot of interesting statistics! There is a bunch of statistical measures for each class. For e.g. the Balanced Accuracy for the digits ‘6’ and ‘9’ is around 50%. Looks like, the classifier is confused by the fact that 6 is inverted 9 and vice-versa. The accuracy on the Test data set is just around 75%. I could have played around with the number of layers, number of hidden units, learning rates, epochs etc to get a much higher accuracy. But since each test took about 8+ hours, I may work on this, some other day!

```

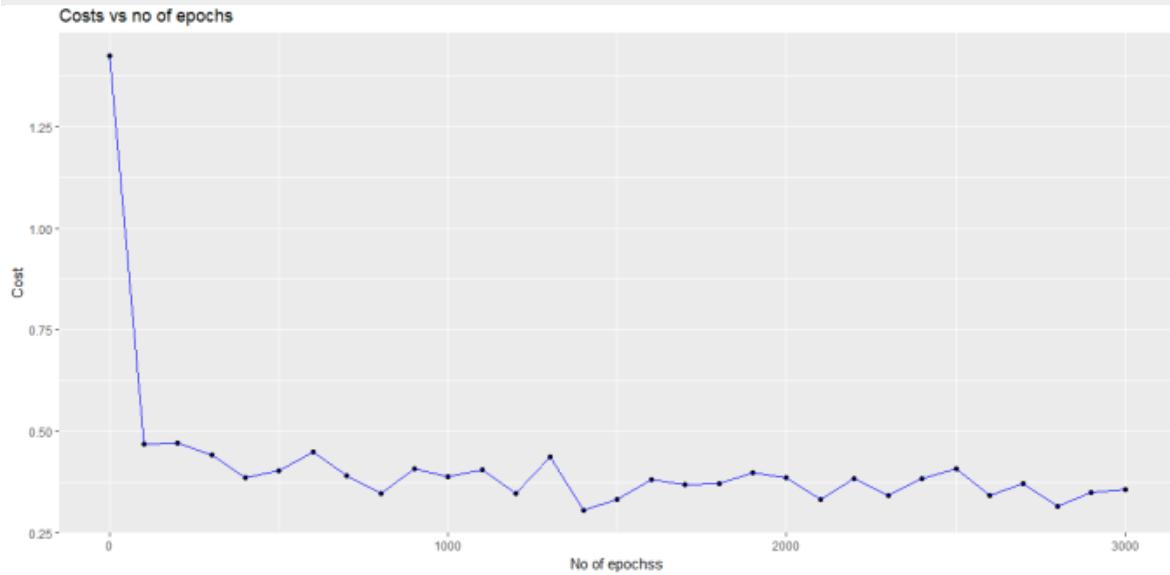
1 source("DLfunctions5.R")
2 source("mnist.R")
3
4 #Load the mnist data
5 load_mnist()
6 show_digit(train$x[2,])
7 #Set the layer dimensions
8 layersDimensions=c(784, 15, 9, 10) # works at 1500
9 x <- t(train$x)
10 x <- x[,1:60000]
11 y <-train$y
12 y1 <- y[1:60000]
13 y2 <- as.matrix(y1)
14 Y=t(y2)
15
16 # Subset 32768 random samples from MNIST
17 permutation = c(sample(2^15))
18 # Randomly shuffle the training data
19 x1 = X[, permutation]
20 y1 = Y[1, permutation]
21 y2 <- as.matrix(y1)
22 Y1=t(y2)
23
24 # Execute Stochastic Gradient Descent on the entire training set
25 # with 'relu' activation in the hidden layer 'softmax' activation and

```

```

26 # learning rate =0.05
27 retvalsSGD= L_Layer_DeepModel_SGD(X1, Y1, layersDimensions,
28                                         hiddenActivationFunc='relu',
29                                         outputActivationFunc="softmax",
30                                         learningRate = 0.05,
31                                         mini_batch_size = 512,
32                                         num_epochs = 1,
33                                         print_cost = True)

```



```

1 # Compute the Confusion Matrix
2 library(caret)
3 library(e1071)
4 predictions=predictProba(retvalsSGD[['parameters']],
5                           X,hiddenActivationFunc='relu',
6                           outputActivationFunc="softmax")
7 confusionMatrix(predictions,Y)
8
9 # Confusion Matrix on the Training set
10 > confusionMatrix(predictions,Y)
11 Confusion Matrix and Statistics
12
13     Reference
14 Prediction   0    1    2    3    4    5    6    7    8    9
15      0 5738    1   21    5   16   17    7   15    9   43
16      1    5 6632   21   24   25    3    2   33   13 392
17      2   12   32 5747  106   25   28    3   27   44 4779
18      3    0   27   12 5715    1   21    1   20    1   13
19      4   10    5   21   18 5677    9   17   30   15 166
20      5  142   21   96 136   93 5306 5884   43   60 413
21      6    0    0    0    0    0    0    0    0    0    0
22      7    6    9   13   13    3    4    0 6085    0    55
23      8    8   12    7   43    1   32    2    7 5703    69
24      9    2    3   20   71    1    1    2    5    6   19
25
26 Overall statistics
27     Accuracy : 0.777
28     95% CI : (0.7737, 0.7804)
29     No Information Rate : 0.1124
30     P-Value [Acc > NIR] : < 2.2e-16
31
32     Kappa : 0.7524

```

```

33  Mcnemar's Test P-Value : NA
34
35  Statistics by Class:
36
37          class: 0 class: 1 class: 2 class: 3 class: 4 class: 5
38  Class: 6
39  Sensitivity           0.96877   0.9837   0.96459   0.93215   0.97176   0.97879
40  0.00000
41  Specificity          0.99752   0.9903   0.90644   0.99822   0.99463   0.87380
42  1.00000
43  Pos Pred Value       0.97718   0.9276   0.53198   0.98348   0.95124   0.43513
44  Nan
45  Neg Pred Value       0.99658   0.9979   0.99571   0.99232   0.99695   0.99759
46  0.90137
47  Prevalence            0.09872   0.1124   0.09930   0.10218   0.09737   0.09035
48  0.09863
49  Detection Rate        0.09563   0.1105   0.09578   0.09525   0.09462   0.08843
50  0.00000
51  Detection Prevalence 0.09787   0.1192   0.18005   0.09685   0.09947   0.20323
52  0.00000
53  Balanced Accuracy     0.98314   0.9870   0.93551   0.96518   0.98319   0.92629
54  0.50000
55          class: 7 class: 8 class: 9
56  Sensitivity           0.9713    0.97471  0.0031938
57  Specificity          0.9981    0.99666  0.9979464
58  Pos Pred Value       0.9834    0.96924  0.1461538
59  Neg Pred Value       0.9967    0.99727  0.9009521
60  Prevalence            0.1044    0.09752  0.0991500
61  Detection Rate        0.1014    0.09505  0.0003167
62  Detection Prevalence 0.1031    0.09807  0.0021667
63  Balanced Accuracy     0.9847    0.98568  0.5005701
64
65  # Confusion Matrix on the Training set xtest <- t(test$x) xtest <-
66  xtest[,1:10000] ytest <- test$y ytest1 <- ytest[1:10000] ytest2 <-
67  as.matrix(ytest1) Ytest=t(ytest2)
68
69  Confusion Matrix and Statistics
70  Reference
71  Prediction
72      0   1   2   3   4   5   6   7   8   9
73      0 950  2   2   3   0   6   9   4   7   6
74      1   3 1110  4   2   9   0   3   12   5  74
75      2   2   6 965  21   9  14   5   16  12 789
76      3   1   2   9 908  2   16   0   21   2   6
77      4   0   1   9   5 938   1   8   6   8   39
78      5  19   5  25  35  20 835  929   8   54   67
79      6   0   0   0   0   0   0   0   0   0   0
80      7   4   4   7  10   2   4   0   952   5   6
81      8   1   5   8  14   2  16   2   3  876  21
82      9   0   0   3  12   0   0   2   6   5   1
83
84  Overall Statistics
85
86          Accuracy : 0.7535
87          95% CI  : (0.7449, 0.7619)
88          No Information Rate : 0.1135
89          P-Value [Acc > NIR] : < 2.2e-16
90
91          Kappa : 0.7262
92          Mcnemar's Test P-Value : NA
93
94  Statistics by Class:
95          class: 0 class: 1 class: 2 class: 3 class: 4 class: 5
96  Class: 6

```

```

97 Sensitivity          0.9694  0.9780  0.9351  0.8990  0.9552  0.9361
98 0.0000
99 Specificity         0.9957  0.9874  0.9025  0.9934  0.9915  0.8724
100 1.0000
101 Pos Pred value    0.9606  0.9083  0.5247  0.9390  0.9241  0.4181
102 Nan
103 Neg Pred value    0.9967  0.9972  0.9918  0.9887  0.9951  0.9929
104 0.9042
105 Prevalence         0.0980  0.1135  0.1032  0.1010  0.0982  0.0892
106 0.0958
107 Detection Rate    0.0950  0.1110  0.0965  0.0908  0.0938  0.0835
108 0.0000
109 Detection Prevalence 0.0989  0.1222  0.1839  0.0967  0.1015  0.1997
110 0.0000
111 Balanced Accuracy 0.9825  0.9827  0.9188  0.9462  0.9733  0.9043
112 0.5000
113                               class: 7  class: 8  class: 9
114 Sensitivity         0.9261  0.8994  0.0009911
115 Specificity        0.9953  0.9920  0.9968858
116 Pos Pred Value    0.9577  0.9241  0.0344828
117 Neg Pred Value    0.9916  0.9892  0.8989068
118 Prevalence         0.1028  0.0974  0.1009000
119 Detection Rate    0.0952  0.0876  0.0001000
120 Detection Prevalence 0.0994  0.0948  0.0029000
121 Balanced Accuracy 0.9607  0.9457  0.4989384
122

```

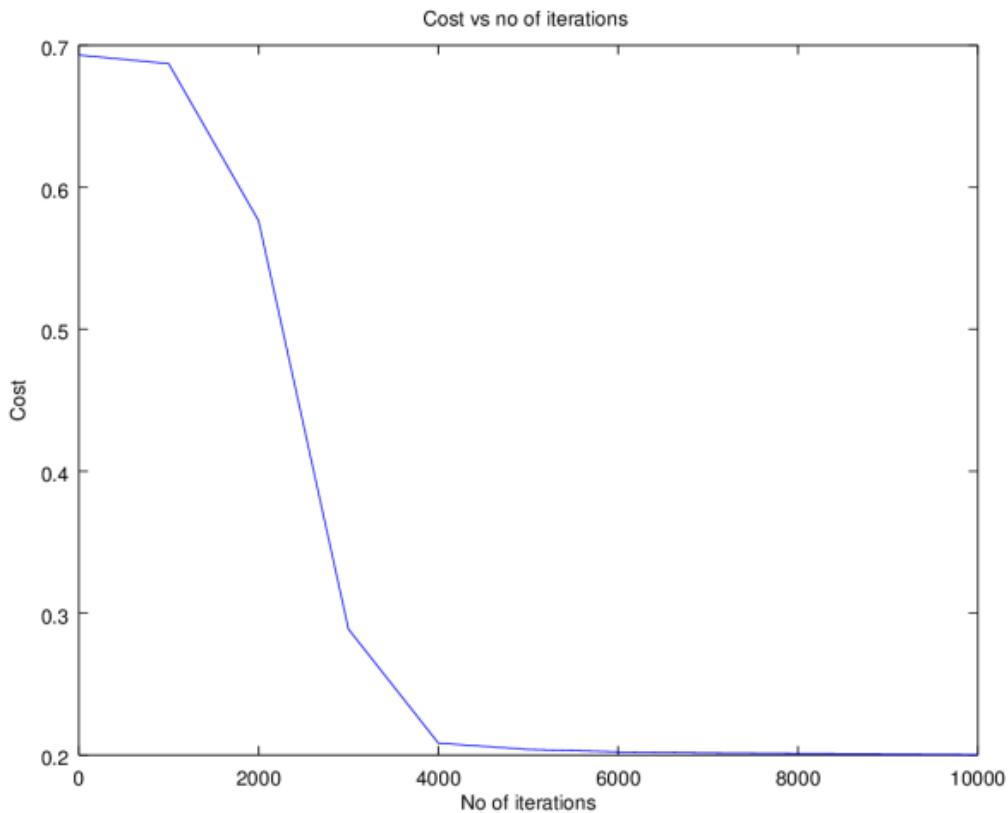
3.3a Random dataset with sigmoid activation – Octave

The Octave code below uses the random data set used by Python(data.csv). The code below implements a L-Layer Deep Learning with Sigmoid Activation.

```

1 source("DL5functions.m")
2
3 # Read the data
4 data=csvread("data.csv");
5
6 #Set up the data
7 X=data(:,1:2);
8 Y=data(:,3);
9
10 #Set the layer dimensions
11 # 2 - 2 input features
12 # 9 7 - 2 hidden layers with 9 and 7 activation units
13 # 1 - 1 sigmoid activation at output layer
14 layersDimensions = [2 9 7 1]; #tanh=-0.5(ok),
15
16 # Execute L-layer Deep Learning Network
17 # with relu as hidden unit and sigmoid as output activation and
18 # learning rate 0.1
19 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
20                                         hiddenActivationFunc='relu',
21                                         outputActivationFunc="sigmoid",
22                                         learningRate = 0.1,
23                                         numIterations = 10000);
24
25 # Plot cost vs iterations
26 plotCostVsIterations(10000,costs);

```



3.3b Spiral dataset with Softmax activation – Octave

The code below uses the spiral data set used by Python above(spiral.csv). The code below implements a L-Layer Deep Learning with Softmax Activation.

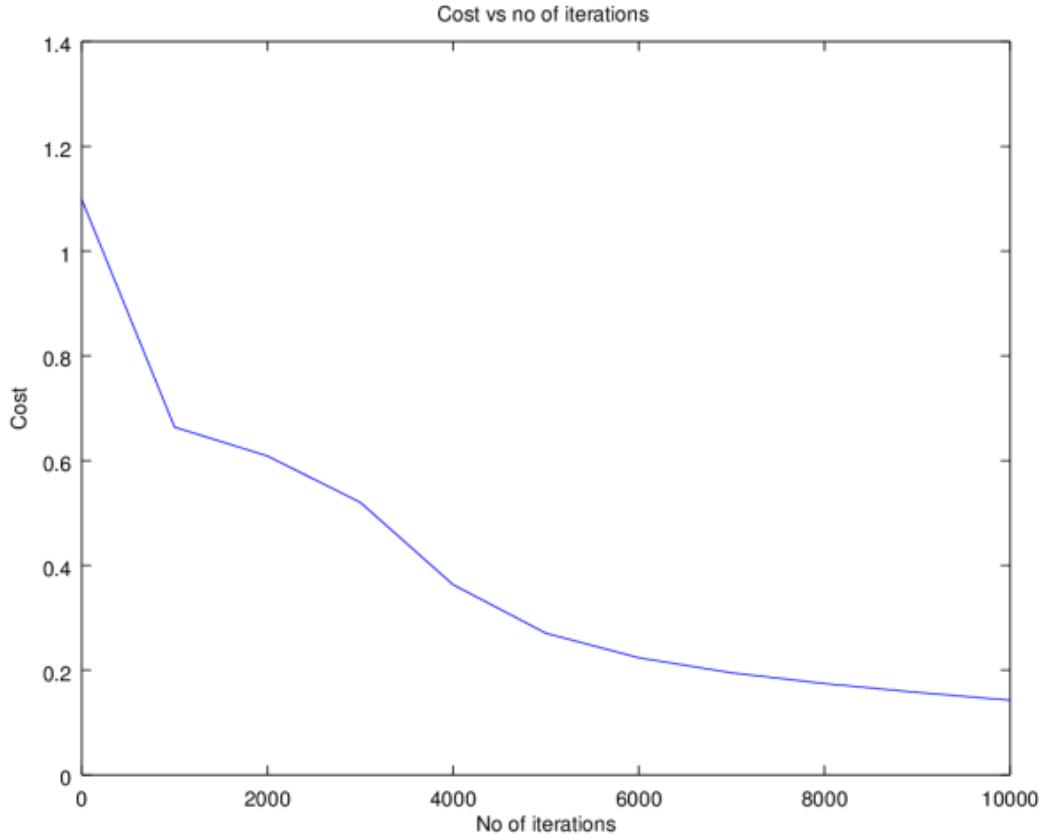
```

1 # Read the data
2 data=csvread("spiral.csv");
3
4 # Setup the data
5 X=data(:,1:2);
6 Y=data(:,3);
7
8 # Set the number of features, number of hidden units in hidden layer and
9 # number of classes
10 numFeats=2; #No features
11 numHidden=100; # No of hidden units
12 numOutput=3; # No of classes
13 # Set the layer dimensions
14 layersDimensions = [numFeats numHidden numOutput];
15
16 #Execute the L-Layer Deep Learning network with 'relu' activation
17 # in the hidden layer and softmax activation unit at the output layer
18 # with learning rate=0.1
19 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
20                                         hiddenActivationFunc='relu',
21                                         outputActivationFunc="softmax",
22                                         )

```

22
23

```
learningRate = 0.1,  
numIterations = 10000);
```



3.3c MNIST dataset with Softmax activation – Octave

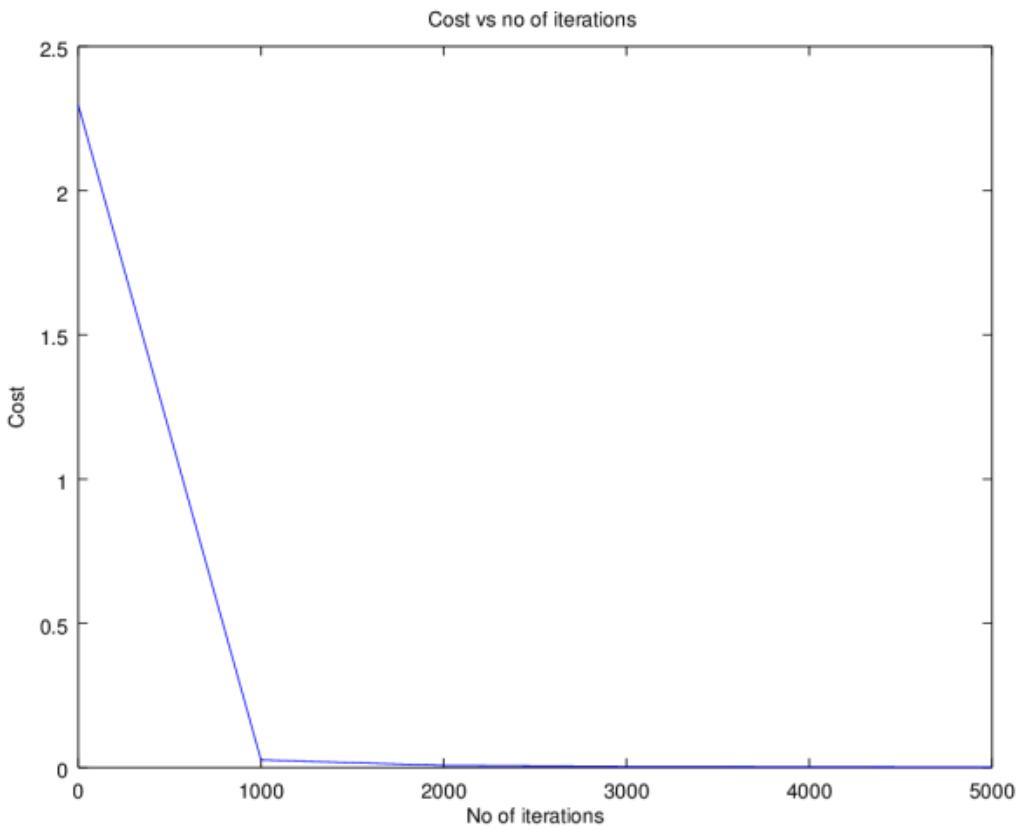
The code below implements a L-Layer Deep Learning Network in Octave with Softmax output activation unit, for classifying the 10 handwritten digits in the MNIST dataset. Unfortunately, Octave can only index to around 10000 training at a time, and I was getting an error ‘*error: out of memory or dimension too large for Octave’s index type error: called from...*’, when I tried to create a batch size of 20000. So, I had to come with a work around to create a batch size of 10000 (randomly) and then use a mini-batch of 1000 samples and execute Stochastic Gradient Descent. The performance was good. Octave takes about 15 minutes, on a batch size of 10000 and a mini batch of 1000.

```
1 # Pseudo code that could be used since Octave only allows 10K batches  
2 # at a time  
3 # Randomly create weights  
4 [weights biases] = initialize_weights()  
5 for i=1:k  
6     # Create a random permutation and create a random batch  
7     permutation = randperm(10000);  
8     X=trainX(permutation,:);  
9     Y=trainY(permutation,:);  
10    # Compute weights from SGD and update weights in the next batch update
```

```

12 [weights biases
13 costs]=L_Layer_DeepModel_SGD(X,Y,mini_bactch=1000,weights, biases,...);
14 ...
15 Endfor
16
17 # Load the MNIST data
18 load('./mnist/mnist.txt.gz');
19
20 #Create a random permutatation from 60K
21 permutation = randperm(10000);
22 disp(length(permutation));
23
24 # Use this 10K as the batch
25 X=trainX(permutation,:);
26 Y=trainY(permutation,:);
27
28 # Set layer dimensions
29 # 784 - Number of input features - 28 x28
30 # 15 9 - 2 hidden layers with 15 and 9 activation units
31 # 10 - 10 activation units at the output layer
32 layersDimensions=[784, 15, 9, 10];
33
34 # Run Stochastic Gradient descent with batch size=10K and
35 mini_batch_size=1000
36 # with 'relu' activation in hidden layer and 'softmax' activation at the
37 output layer
38 [weights biases costs]=L_Layer_DeepModel_SGD(X', Y', layersDimensions,
39 hiddenActivationFunc='relu',
40 outputActivationFunc="softmax",
41 learningRate = 0.01,
42 mini_batch_size = 2000, num_epochs = 5000);

```



4.Final thoughts

Here are some of my final thoughts after working on Python, R and Octave in this series and in other projects

1. Python, with its highly optimized numpy library, is ideally suited for creating Deep Learning Models, which have a lot of matrix manipulations. Python is a real workhorse when it comes to Deep Learning computations.
2. R is somewhat clunky in comparison to its cousin Python in handling matrices or in returning multiple values. But R's statistical libraries, dplyr, and ggplot are really superior to the Python peers. Also, I find R handles dataframes, much better than Python.
3. Octave is a no-nonsense, minimalist language which is very efficient in handling matrices. It is ideally suited for implementing Machine Learning and Deep Learning from scratch. But, Octave has its problems and cannot handle large matrix sizes, and also lacks the statistical libraries of R and Python. They possibly exist in its sibling, Matlab

5.Conclusion

Building a Deep Learning Network from scratch is quite challenging, time-consuming but nevertheless an exciting task. While the statements in the different languages for manipulating matrices, summing up columns, finding columns which have ones don't take more than a single statement, extreme care has to be taken to ensure that the statements work well for any dimension. The lessons learnt from creating L -Layer Deep Learning network are many and well worth it. Give it a try!

6. Initialization, regularization in Deep Learning

“Today you are You, that is truer than true. There is no one alive who is Youer than You.”

Dr. Seuss

“Explanations exist; they have existed for all time; there is always a well-known solution to every human problem — neat, plausible, and wrong.”

H L Mencken

1. Introduction

In this 6th chapter of ‘Deep Learning from first principles’, I look at a couple of different initialization techniques used in Deep Learning, L2 regularization and the ‘dropout’ method. Specifically, I implement “He” initialization” & “Xavier” Initialization.

In addition I also implement L2 regularization and finally dropout. Hence, my generic L-Layer Deep Learning network includes these additional enhancements for enabling/disabling initialization methods, regularization or dropout in the algorithm. It already included sigmoid & softmax output activation for binary and multi-class classification, besides allowing relu, tanh and sigmoid activation for hidden units. The Python,R and Octave functions used in this chapter are implemented in Appendix 6 - Initialization, regularization in Deep Learning

This code for Python, R and Octave can be cloned/downloaded from Github at DeepLearningFromFirstPrinciples (<https://github.com/tvganesh/DeepLearningFromFirstPrinciples/tree/master/Chap6-DLInitializationRegularization>)

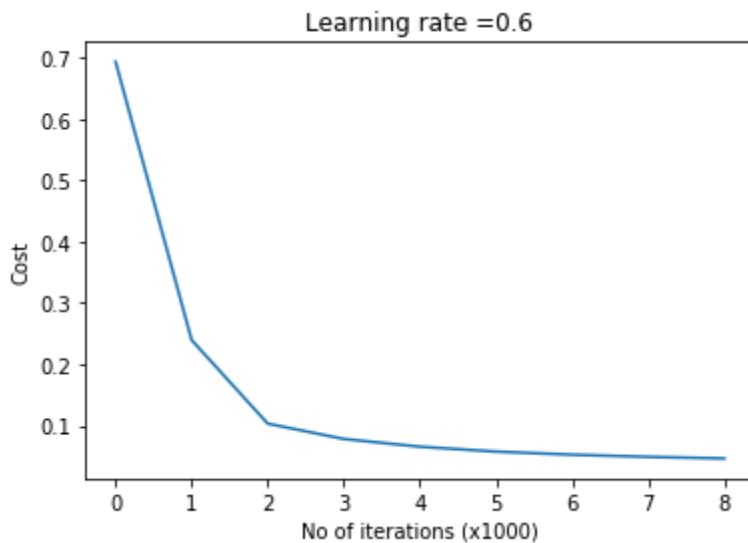
2. Initialization techniques

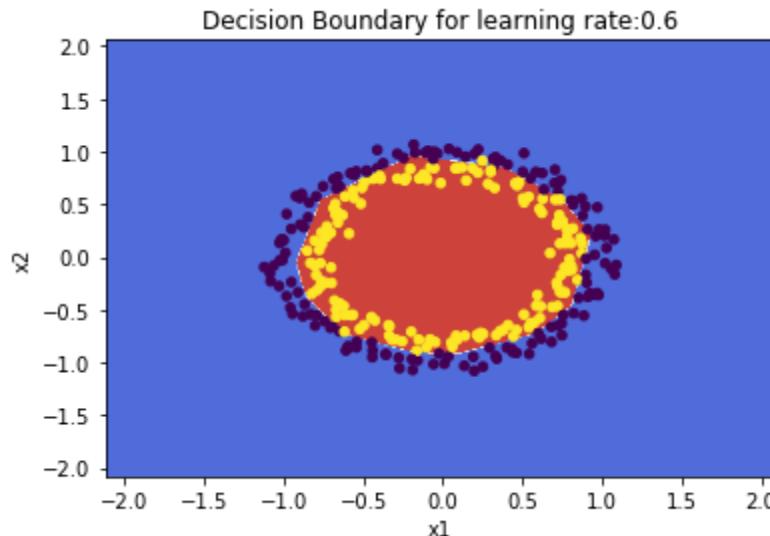
The usual initialization technique is to generate Gaussian or uniform random numbers and multiply it by a small value like 0.01. Two techniques which are used to speed up convergence is the ‘He’ (https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf)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

1.1a Default initialization – Python

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

```
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6 import sklearn
7 import sklearn.datasets
8 exec(open("DLfunctions61.py").read())
9
10 #Load the data
11 train_X, train_Y, test_X, test_Y = load_dataset()
12
13 # Set the layers dimensions
14 # 2 - number of input features
15 # 7 - 1 hidden layer 7 hidden units
16 # 1 - 1 sigmoid activation unit at the output layer
17 layersDimensions = [2,7,1]
18
19 # Train a L-layer deep learning network with random initialization
20 # hidden Activation function - relu
21 # output activation function - sigmoid
22 # learning rate - 0.6
23 parameters = L_Layer_DeepModel(train_X, train_Y, layersDimensions,
24 hiddenActivationFunc='relu', outputActivationFunc="sigmoid",learningRate =
25 0.6, num_iterations = 9000, initType="default", print_cost =
26 True,figure="fig1.png")
27
28 # Clear the plot
29 plt.clf()
30 plt.close()
31
32 # Plot the decision boundary
33 plot_decision_boundary(lambda x: predict(parameters, x.T), train_X,
34 train_Y,str(0.6),figure1="fig2.png")
```





1.1b He initialization – Python

‘He’ initialization attributed to ‘He et al’ (https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper_r.pdf), multiplies the random weights by

$$\sqrt{\frac{2}{\text{dimension of previous layer}}}$$

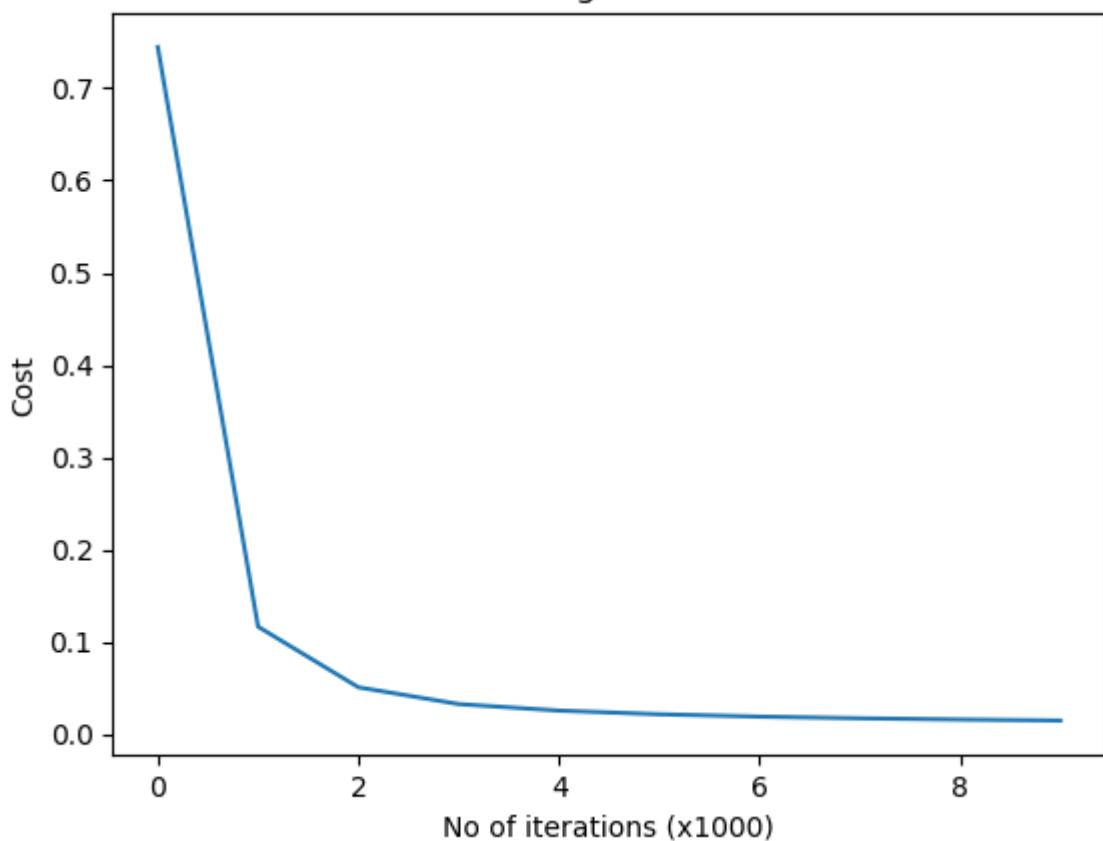
```

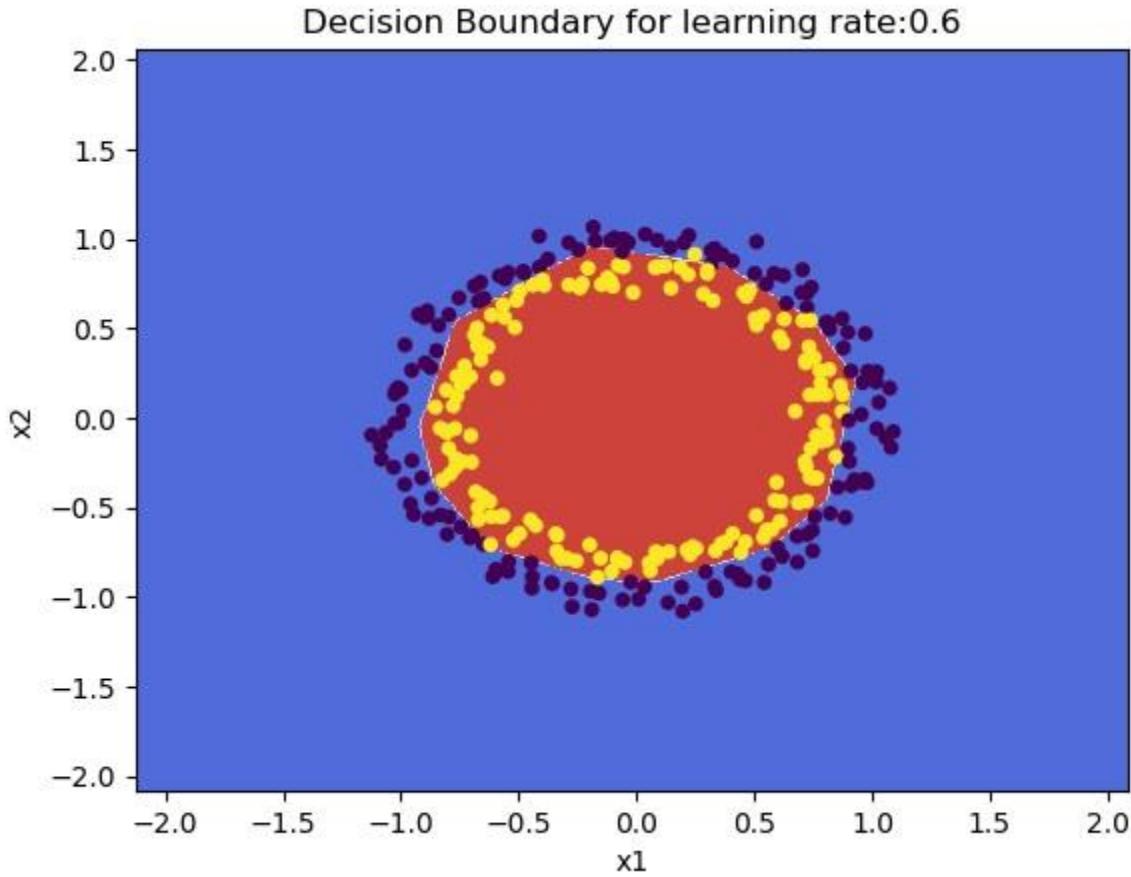
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6 import sklearn
7 import sklearn.datasets
8 exec(open("DLfunctions61.py").read())
9
10 #Load the data
11 train_X, train_Y, test_X, test_Y = load_dataset()
12 # Set the layers dimensions
13 # 2 - number of input features
14 # 7 - 1 hidden layer 7 hidden units
15 # 1 - 1 sigmoid activation unit at the output layer
16 layersDimensions = [2,7,1]
17
18 # Train a L-layer deep learning network with He initialization
19 # hidden Activation function - relu
20 # output activation function - sigmoid
21 # learning rate - 0.6
22 parameters = L_Layer_DeepModel(train_X, train_Y, layersDimensions,
23 hiddenActivationFunc='relu', outputActivationFunc="sigmoid", learningRate
24 =0.6, num_iterations = 10000, initType="He", print_cost = True,
25 figure="fig3.png")
26
27 # clear plot
28 plt.clf()
29 plt.close()
30
31 # Plot the decision boundary

```

```
32 plot_decision_boundary(lambda x: predict(parameters, x.T), train_X,  
33 train_Y,str(0.6),figure1="fig4.png")
```

Learning rate =0.6





1.1c Xavier initialization – Python

‘Xavier’ (<http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>) initialization multiply the random weights by

$$\sqrt{\frac{1}{\text{dimension of previous layer}}}$$

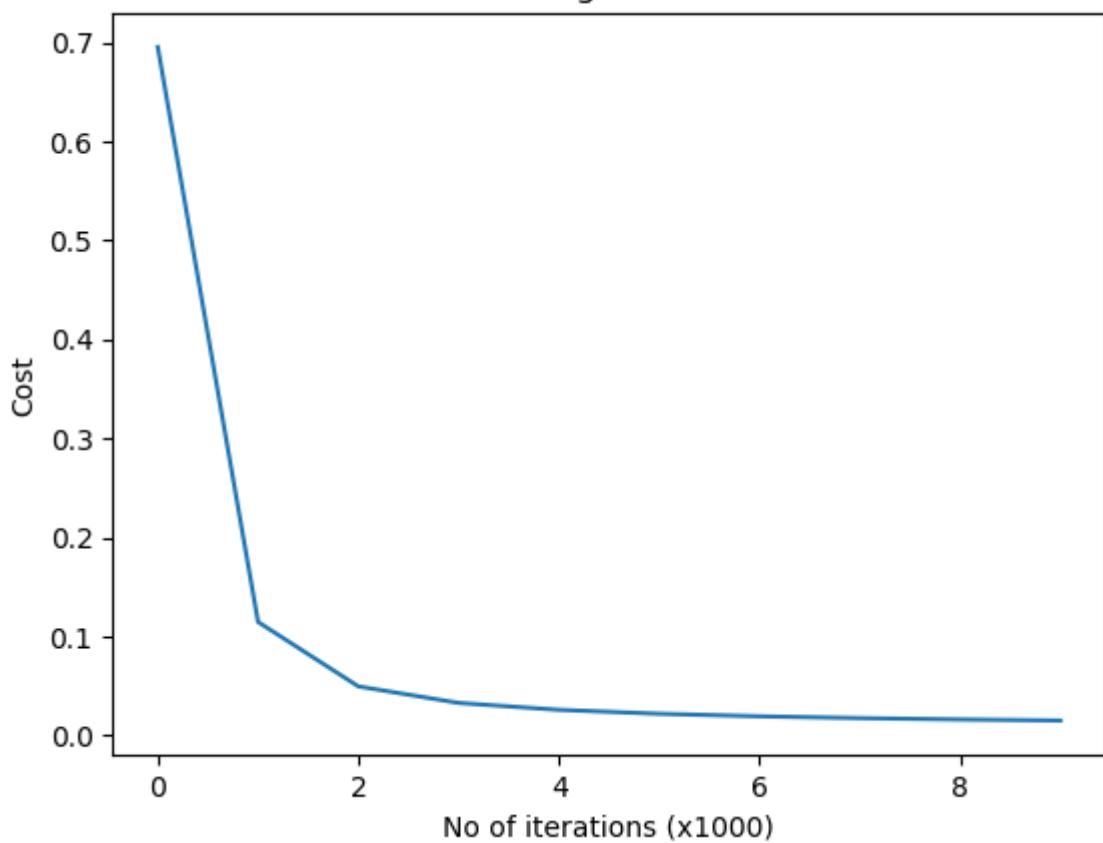
```

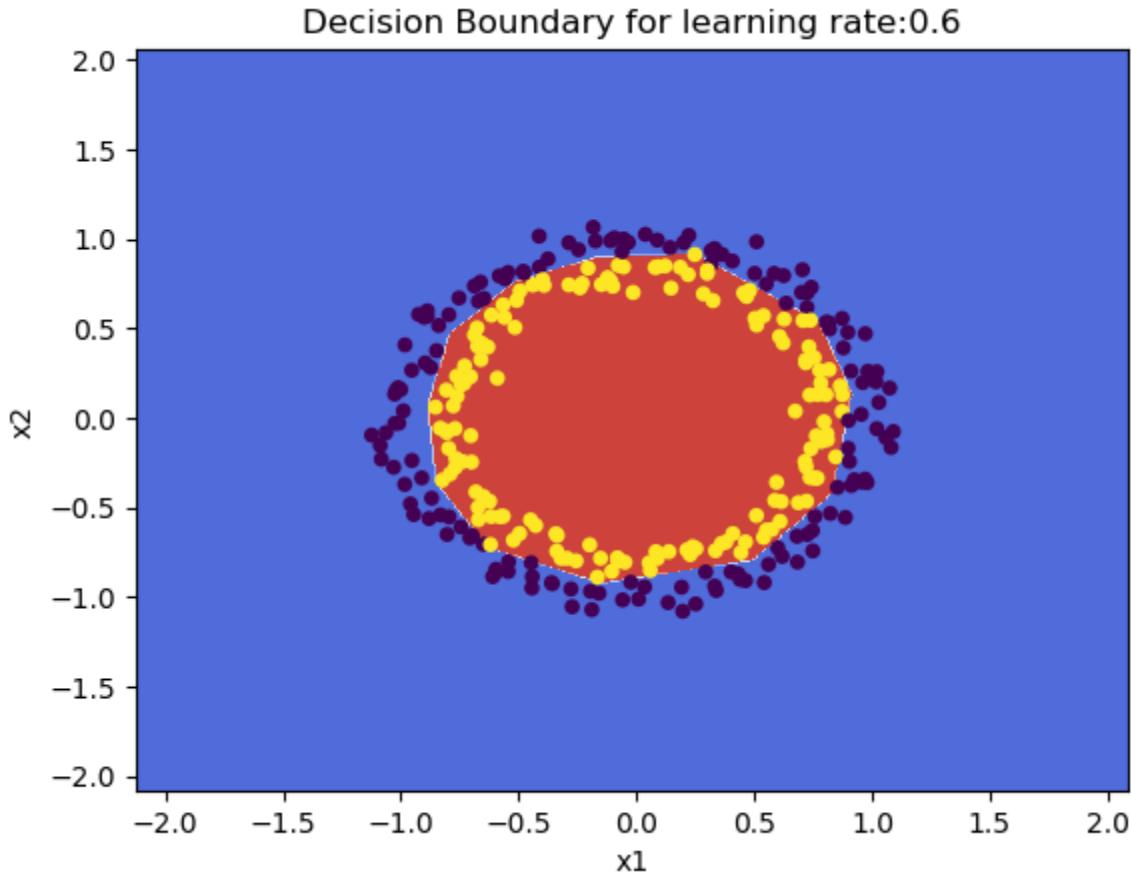
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6 import sklearn
7 import sklearn.datasets
8 exec(open("DLfunctions61.py").read())
9
10 #Load the data
11 train_X, train_Y, test_X, test_Y = load_dataset()
12
13 # Set the layers dimensions
14 # 2 - number of input features
15 # 7 - 1 hidden layer 7 hidden units
16 # 1 - 1 sigmoid activation unit at the output layer
17 layersDimensions = [2,7,1]
18
19 # Train a L-layer Deep Learning network with Xavier initialization
20 # hidden Activation function - relu

```

```
21 # output activation function - sigmoid
22 # learning rate - 0.6
23 parameters = L_Layer_DeepModel(train_X, train_Y, layersDimensions,
24     hiddenActivationFunc='relu', outputActivationFunc="sigmoid",
25     learningRate = 0.6,num_iterations = 10000,
26     initType="Xavier",print_cost = True,
27     figure="fig5.png")
28
29 # Plot the decision boundary
30 plot_decision_boundary(lambda x: predict(parameters, x.T), train_X,
31 train_Y,str(0.6),figure1="fig6.png")
```

Learning rate =0.6





1.2a Default initialization – R

```

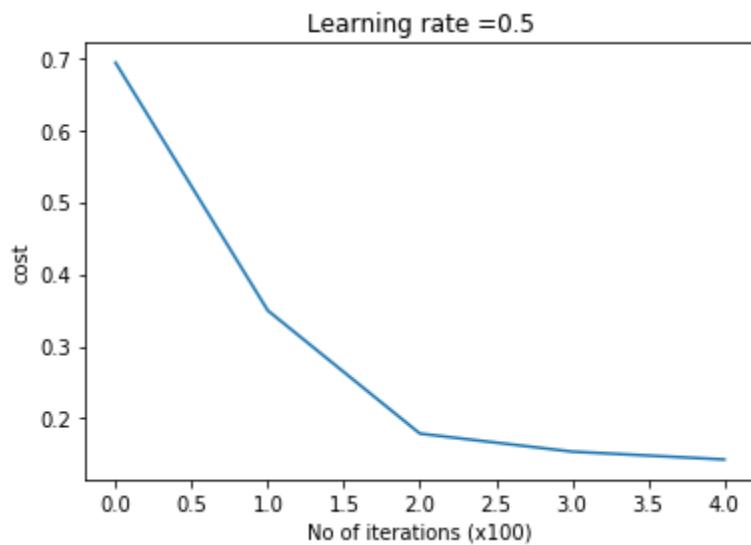
1 source("DLfunctions61.R")
2 #Load the data
3 z <- as.matrix(read.csv("circles.csv",header=FALSE))
4 x <- z[,1:2]
5 y <- z[,3]
6 x <- t(x)
7 Y <- t(y)
8
9 #set the layer dimensions
10 # 2 - number of input features
11 # 11 - 1 hidden layer 11 hidden units
12 # 1 - 1 sigmoid activation unit at the output layer
13 layersDimensions = c(2,11,1)
14
15 # Train a L-layer deep learning network
16 # hidden Activation function - relu
17 # output activation function - sigmoid
18 # learning rate - 0.5
19 retvals = L_Layer_DeepModel(x, Y, layersDimensions,
20                             hiddenActivationFunc='relu',
21                             outputActivationFunc="sigmoid",
22                             learningRate = 0.5,
23                             numIterations = 8000,
24                             initType="default",
25                             print_cost = True)

```

```

26
27 #Plot the cost vs iterations
28 iterations <- seq(0,8000,1000)
29
30 # Plot cost vs iterations
31 costs=retvals$costs
32 df=data.frame(iterations,costs)
33 ggplot(df,aes(x=iterations,y=costs)) + geom_point() + geom_line(color="blue")
34 + ggttitle("Costs vs iterations") + xlab("No of iterations") + ylab("Cost")

```

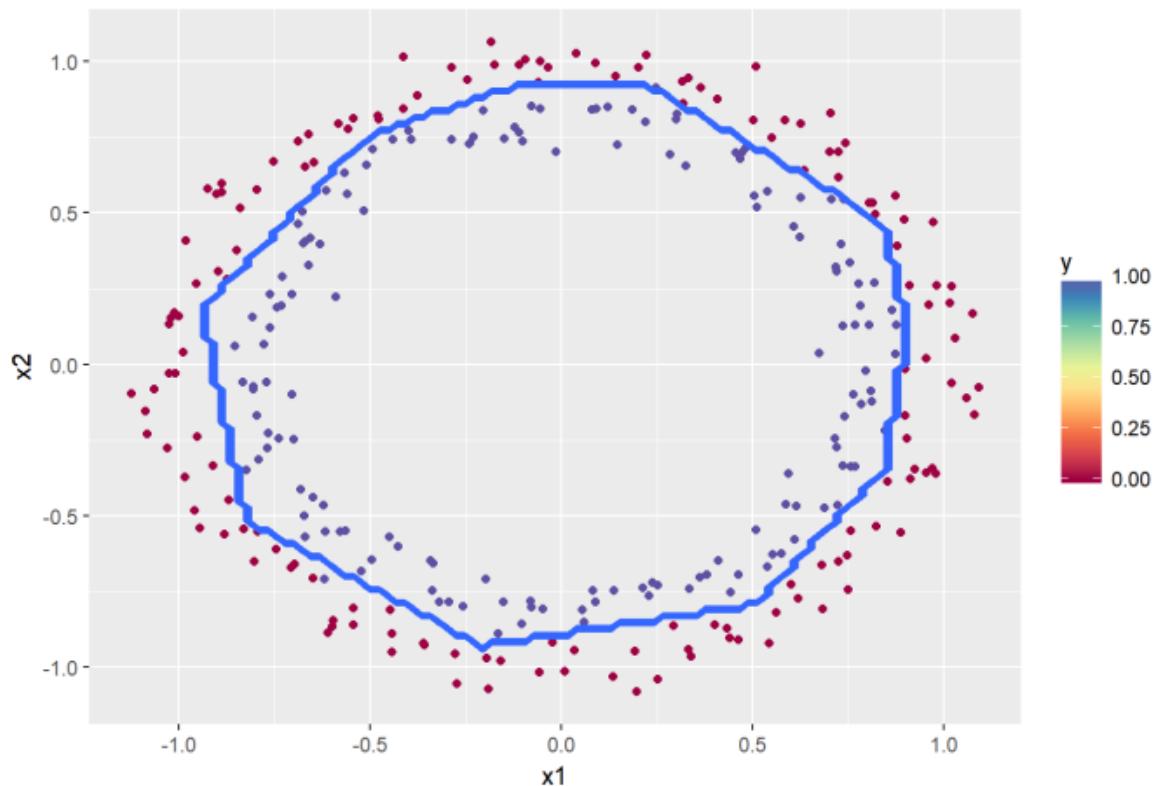


```

1 # Plot the decision boundary
2 plotDecisionBoundary(z,retvals,hiddenActivationFunc="relu",lr=0.5)

```

Decision boundary for learning rate: 0.5



1.2b He initialization – R

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

```

1 # He Initialization model for L layers
2 # Input: List of units in each layer
3 # Returns: Initial weights and biases matrices for all layers
4 # He initialization multiplies the random numbers with
5 # sqrt(2/layerDimensions[previouslayer])
6
7 HeInitializeDeepModel <- function(layerDimensions){
8   set.seed(2)
9
10  # Initialize empty list
11  layerParams <- list()
12
13  # Note the weight matrix at layer '1' is a matrix of size (1,1-1)
14  # The Bias is a vectors of size (1,1)
15
16  # Loop through the layer dimension from 1.. L
17  # Indices in R start from 1
18  for(l in 2:length(layersDimensions)){
19
20    # Initialize a matrix of small random numbers of size 1 x 1-1
21    # Create random numbers of size 1 x 1-1
22    w=rnorm(layersDimensions[l]*layersDimensions[l-1])
23

```

```

24      # Create a weight matrix of size l x l-1 with this initial weights
25 and
26      # Add to list w1,w2... WL
27      # He initialization - Divide by sqrt(2/layerDimensions[previous
28 layer])
29      layerParams[[paste('w',l-1,sep="")]] =
30 matrix(w,nrow=layerDimensions[1],
31 ncol=layerDimensions[l-1])*sqrt(2/layerDimensions[l-1])
32      layerParams[[paste('b',l-1,sep="")]] =
33 matrix(rep(0,layerDimensions[l]),
34 nrow=layerDimensions[l],ncol=1)
35      }
36      return(layerParams)
37  }
38 }
39

```

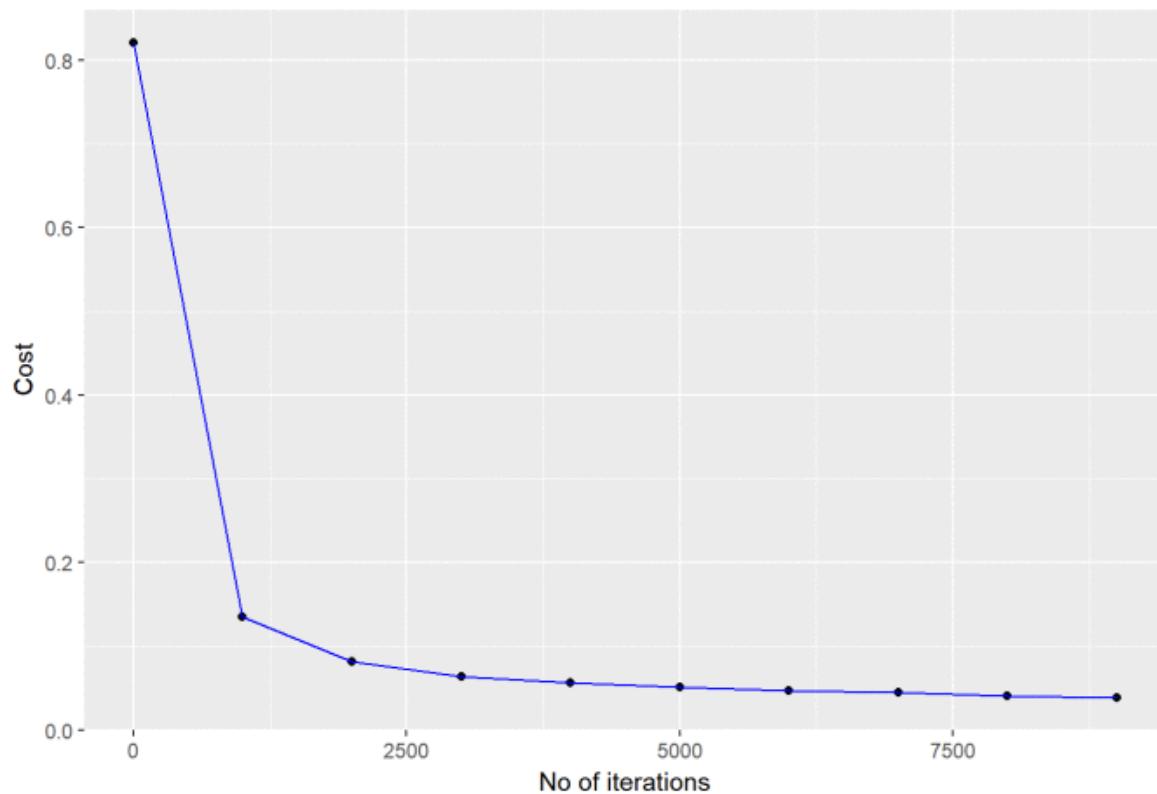
The code in R below uses He initialization to learn the data

```

1 source("DLfunctions61.R")
2 # Load the data
3 z <- as.matrix(read.csv("circles.csv",header=FALSE))
4 x <- z[,1:2]
5 y <- z[,3]
6 X <- t(x)
7 Y <- t(y)
8
9 # Set the layer dimensions
10 # 2 - number of input features
11 # 11 - 1 hidden layer 11 hidden units
12 # 1 - 1 sigmoid activation unit at the output layer
13 layersDimensions = c(2,11,1)
14
15 # Train a L-layer Deep learning network with He initialization
16 # hidden Activation function - relu
17 # output activation function - sigmoid
18 # learning rate - 0.5
19 retvals = L_Layer_DeepModel(x, y, layersDimensions,
20                             hiddenActivationFunc='relu',
21                             outputActivationFunc="sigmoid",
22                             learningRate = 0.5,
23                             numIterations = 9000,
24                             initType="He",
25                             print_cost = True)
26
27 #Plot the cost vs iterations
28 iterations <- seq(0,9000,1000)
29 costs=retvals$costs
30 df=data.frame(iterations,costs)
31 ggplot(df,aes(x=iterations,y=costs)) + geom_point() + geom_line(color="blue")
32 + ggttitle("Costs vs iterations") + xlab("No of iterations") + ylab("Cost")
33

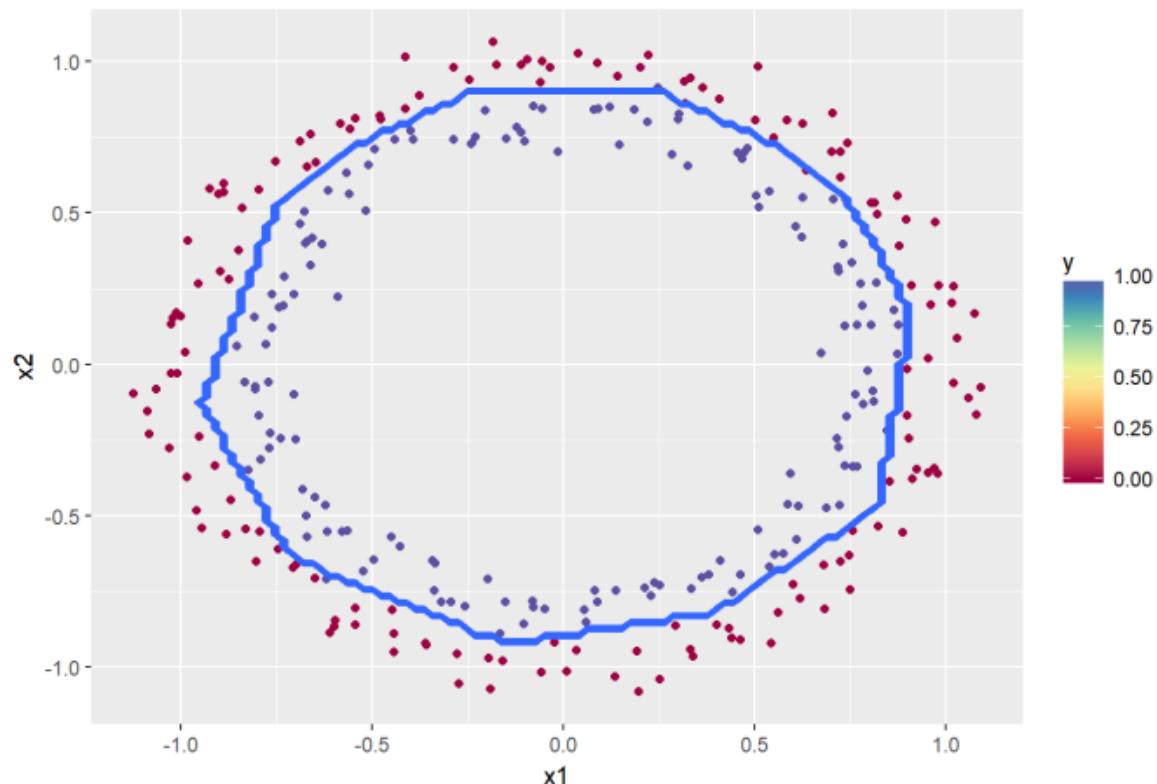
```

Costs vs iterations



```
1 # Plot the decision boundary
2 plotDecisionBoundary(z,retvals,hiddenActivationFunc="relu",0.5,lr=0.5)
```

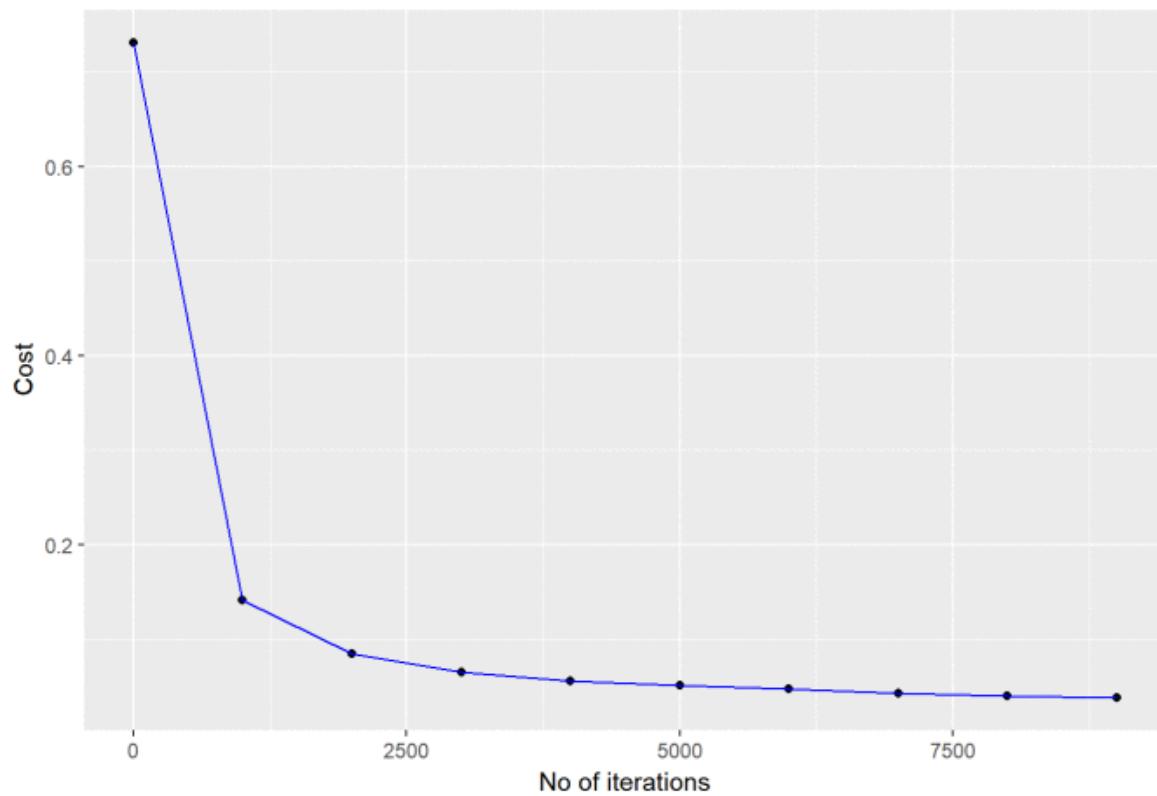
Decision boundary for learning rate: 0.5



1.2c Xavier initialization – R

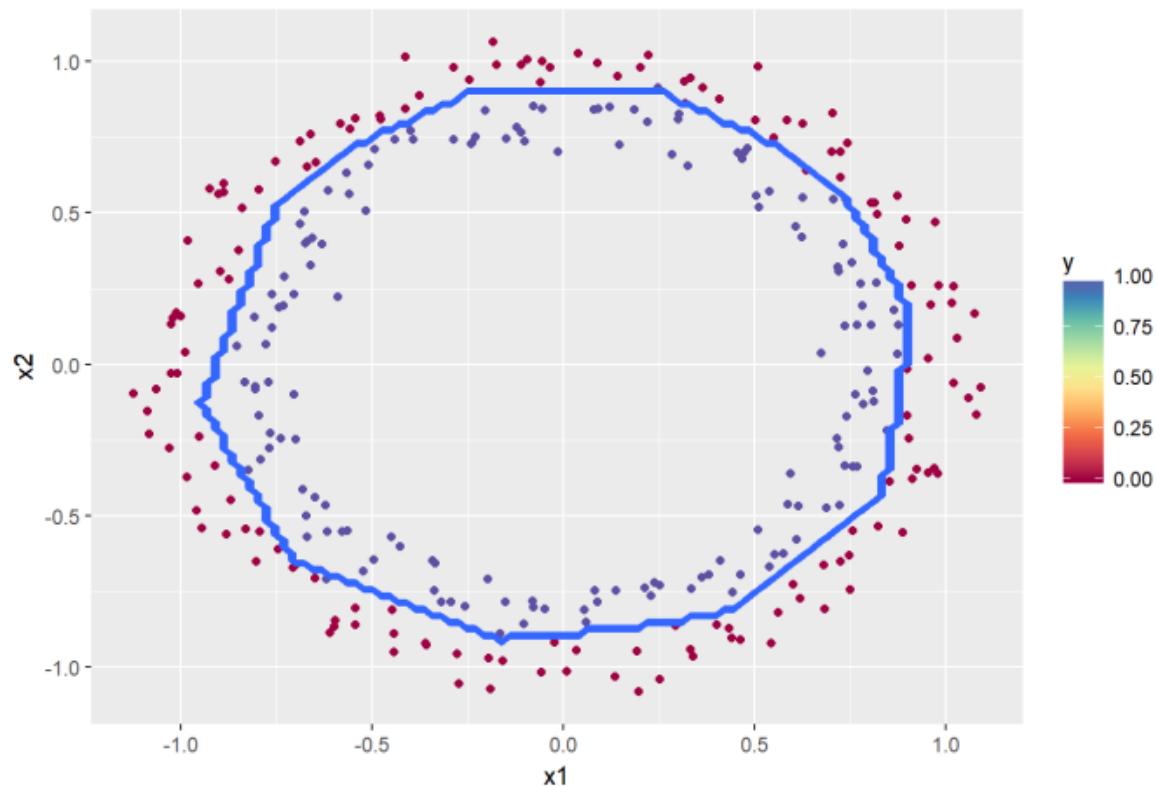
```
1 ## Xav initialization
2 # Set the layer dimensions
3 # 2 - number of input features
4 # 11 - 1 hidden layer 11 hidden units
5 # 1 - 1 sigmoid activation unit at the output layer
6 layersDimensions = c(2,11,1)
7
8 # Train a deep learning network with Xavier initialization
9 # hidden Activation function - relu
10 # output activation function - sigmoid
11 # learning rate - 0.5
12 retvals = L_Layer_DeepModel(x, Y, layersDimensions,
13                             hiddenActivationFunc='relu',
14                             outputActivationFunc="sigmoid",
15                             learningRate = 0.5,
16                             numIterations = 9000,
17                             initType="Xav",
18                             print_cost = True)
19
20 #Plot the cost vs iterations
21 iterations <- seq(0,9000,1000)
22 costs=retvals$costs
23 df=data.frame(iterations,costs)
24 ggplot(df,aes(x=iterations,y=costs)) + geom_point() + geom_line(color="blue")
25 + ggttitle("Costs vs iterations") + xlab("No of iterations") + ylab("Cost")
```

Costs vs iterations



```
1 # Plot the decision boundary
2 plotDecisionBoundary(z,retvals,hiddenActivationFunc="relu",0.5)
```

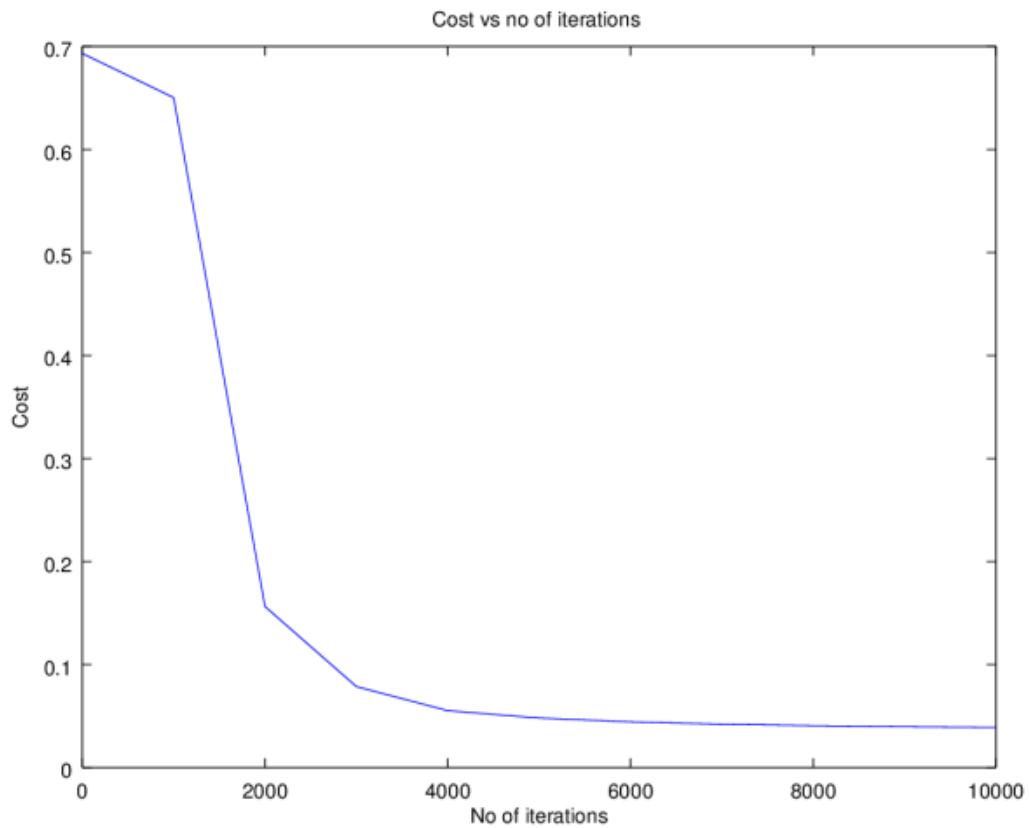
Decision boundary for learning rate: 0.5

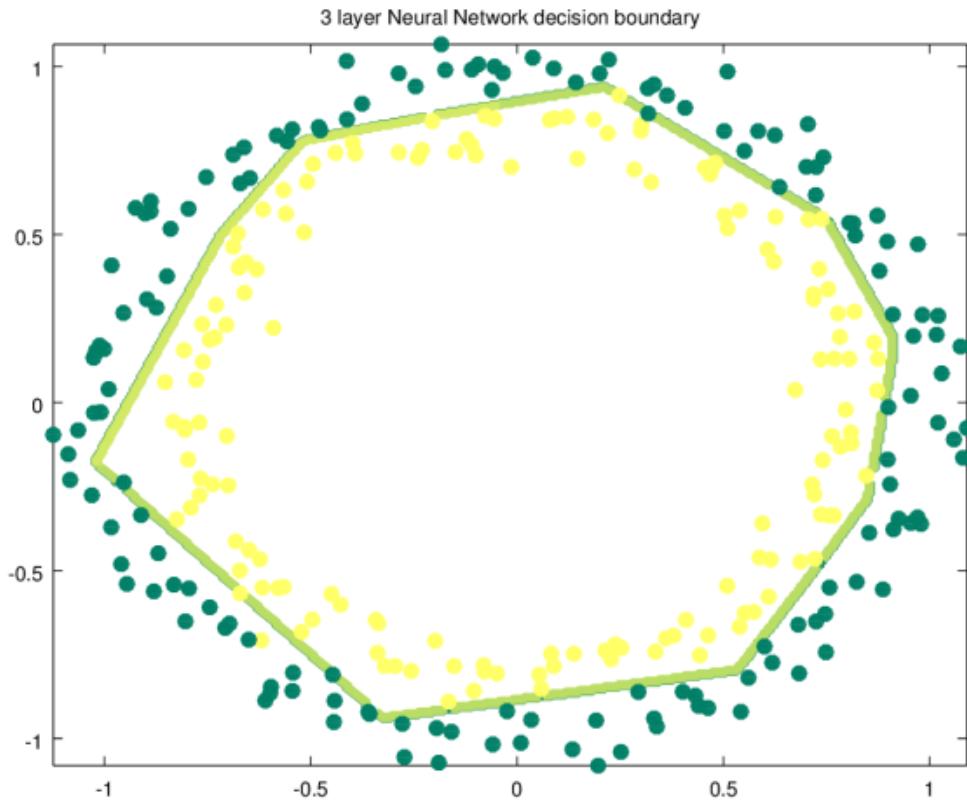


1.3a Default initialization – Octave

```
1 source("DL61functions.m")
2 # Read the data
3 data=csvread("circles.csv");
4
5 X=data(:,1:2);
6 Y=data(:,3);
7 # Set the layer dimensions
8 # 2 - number of input features
9 # 11 - 1 hidden layer 11 hidden units
10 # 1 - 1 sigmoid activation unit at the output layer
11 layersDimensions = [2 11 1];
12
13 # Train a L-layer deep learning network with deafault initialization
14 # hidden Activation function - relu
15 # output activation function - sigmoid
16 # learning rate - 0.5
17 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
18                                         hiddenActivationFunc='relu',
19                                         outputActivationFunc="sigmoid",
20                                         learningRate = 0.5,
21                                         lambd=0,
22                                         keep_prob=1,
23                                         numIterations = 10000,
24                                         initType="default");
25
26 # Plot cost vs iterations
27 plotCostVsIterations(10000,costs)
28
```

```
29 #Plot decision boundary
30 plotDecisionBoundary(data,weights, biases,keep_prob=1,
31 hiddenActivationFunc="relu")
```





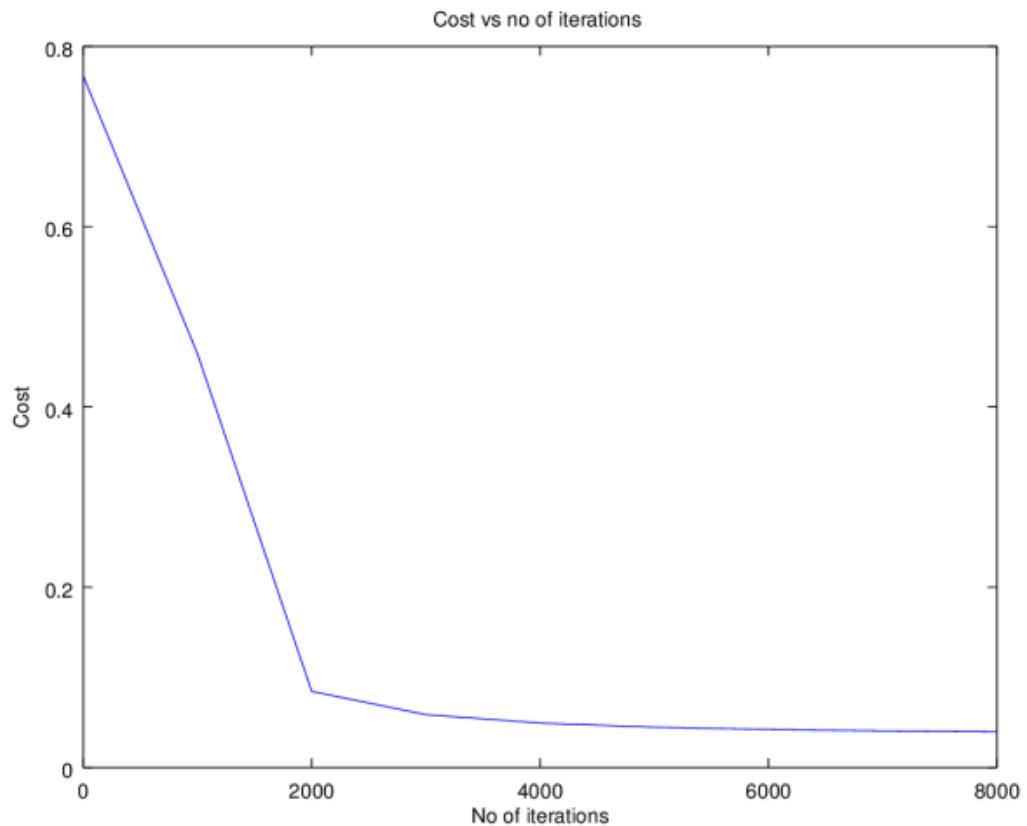
1.3b He initialization – Octave

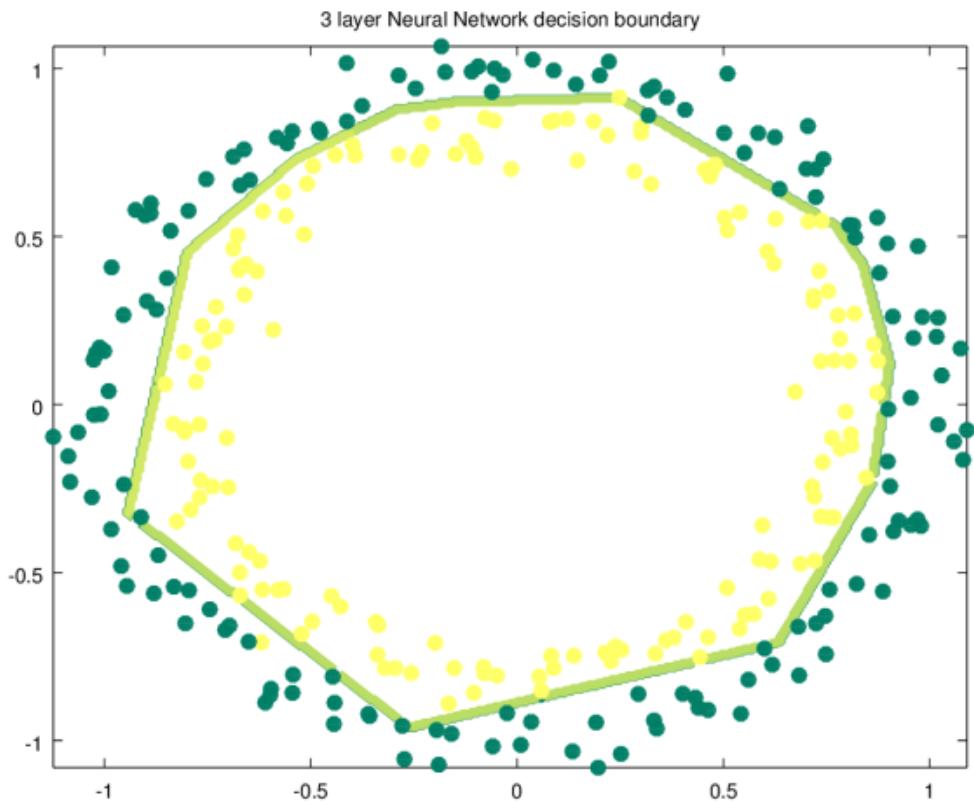
```

1 source("DL61functions.m")
2 #Load data
3 data=csvread("circles.csv");
4 X=data(:,1:2);
5 Y=data(:,3);
6
7 # Set the layer dimensions
8 # 2 - number of input features
9 # 11 - 1 hidden layer 11 hidden units
10 # 1 - 1 sigmoid activation unit at the output layer
11 layersDimensions = [2 11 1];
12
13 # Train a L-layer deep learning network with He initialization
14 # hidden Activation function - relu
15 # output activation function - sigmoid
16 # learning rate - 0.5
17 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
18                                         hiddenActivationFunc='relu',
19                                         outputActivationFunc="sigmoid",
20                                         learningRate = 0.5,
21                                         lambd=0,
22                                         keep_prob=1,
23                                         numIterations = 8000,
24                                         initType="He");
25
26 # Plot cost vs iterations
27 plotCostVsIterations(8000,costs)

```

```
28  
29 #Plot decision boundary  
30 plotDecisionBoundary(data,weights,  
31 biases,keep_prob=1,hiddenActivationFunc="relu")
```





1.3c Xavier initialization – Octave

The code snippet for Xavier initialization in Octave is shown below

```

1 source("DL61functions.m")
2 # Xavier Initialization for L layers
3 # Input: List of units in each layer
4 # Returns: Initial weights and biases matrices for all layers
5 function [w b] = XavInitializeDeepModel(layerDimensions)
6     rand ("seed", 3);
7     # note the weight matrix at layer '1' is a matrix of size (1,1-1)
8     # The Bias is a vectors of size (1,1)
9
10    # Loop through the layer dimension from 1.. L
11    # Create cell arrays for weights and biases
12
13    for l =2:size(layerDimensions)(2)
14        w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*%
15        sqrt(1/layerDimensions(l-1)); # Multiply by .01
16        b{l-1} = zeros(layerDimensions(l),1);
17
18    endfor
19 end

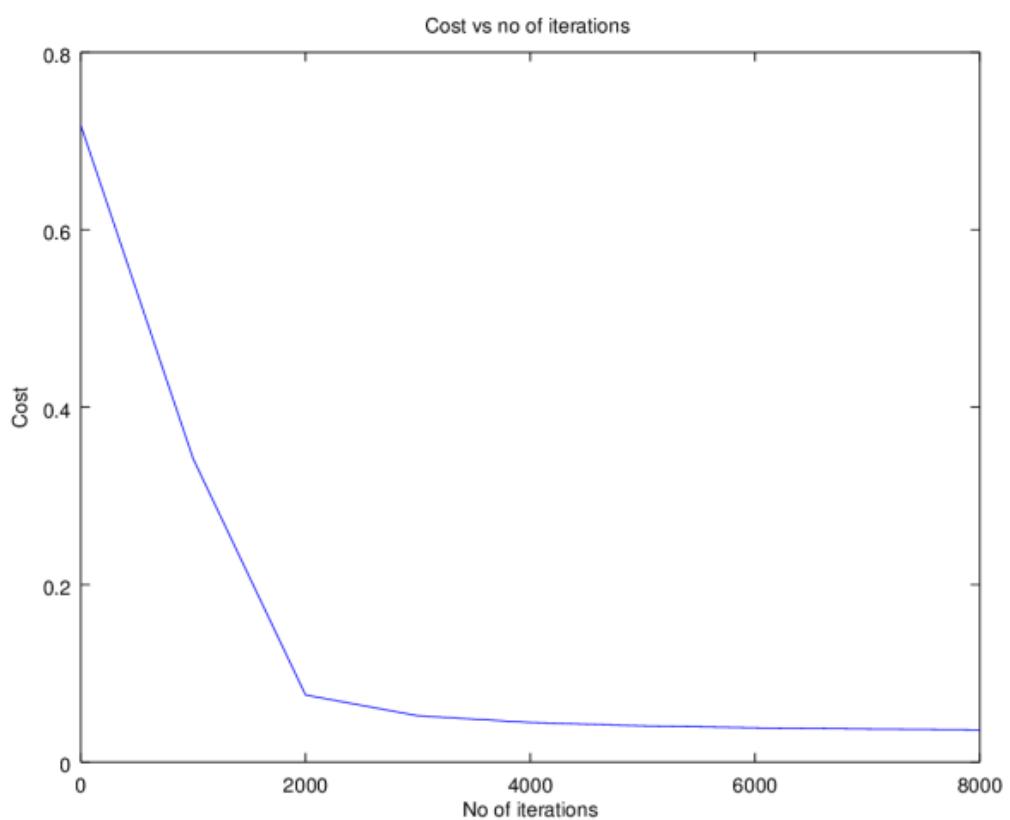
```

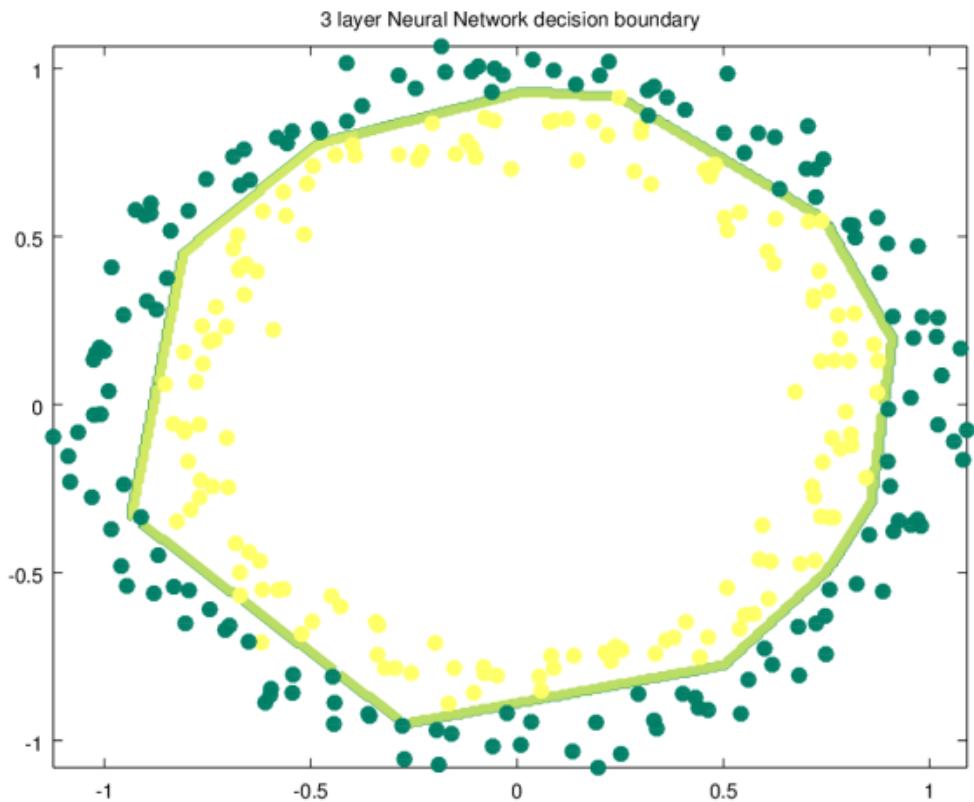
The Octave code below uses Xavier initialization

```

1 source("DL61functions.m")
2 #Load data
3 data=csvread("circles.csv");
4 X=data(:,1:2);
5 Y=data(:,3);
6
7 #Set layer dimensions
8 # 2 - number of input features
9 # 11 - 1 hidden layer 11 hidden units
10 # 1 - 1 sigmoid activation unit at the output layer
11 layersDimensions = [2 11 1
12
13 # Train a L-layer deep learning network with Xavier initialization
14 # hidden Activation function - relu
15 # output activation function - sigmoid
16 # learning rate - 0.5
17 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
18 hiddenActivationFunc='relu',
19 outputActivationFunc="sigmoid",
20 learningRate = 0.5,
21 lambd=0,
22 keep_prob=1,
23 numIterations = 8000,
24 initType="Xav");
25
26 #Plot cost vs iterations
27 plotCostVsIterations(8000,costs)
28
29 # Plot decision boundary
30 plotDecisionBoundary(data,weights,
31 biases,keep_prob=1,hiddenActivationFunc="relu")

```





2.1a Regularization : Circles data – Python

The cross-entropy cost for Logistic classification is given as $J = \frac{1}{m} \sum_{i=1}^m y^i \log((a^L)^{(i)}) + (1 - y^i) \log((1 - a^L)^{(i)})$

```

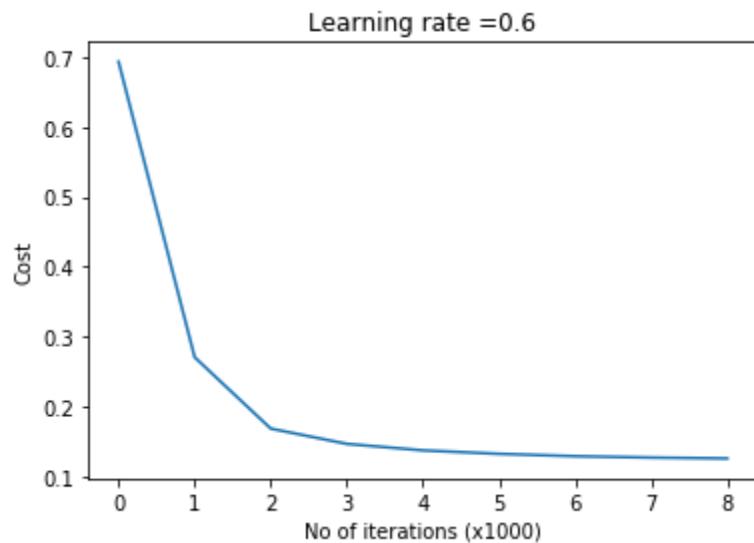
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6 import sklearn
7 import sklearn.datasets
8 exec(open("DLfunctions61.py").read())
9
10 #Load the data
11 train_X, train_Y, test_X, test_Y = load_dataset()
12
13 # Set the layers dimensions
14 # 2 - number of input features
15 # 7 - 1 hidden layer 7 hidden units
16 # 1 - 1 sigmoid activation unit at the output layer
17 layersDimensions = [2,7,1]

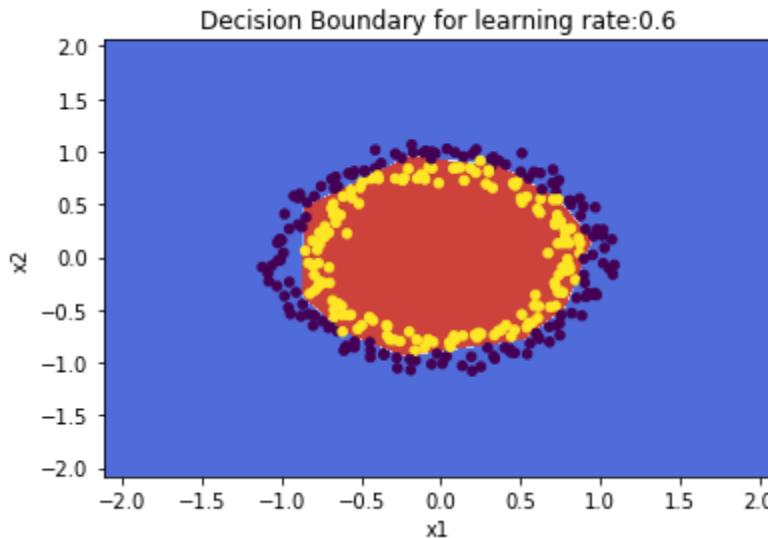
```

```

18
19 # Train a L-layer deep learning network with default initialization
20 # hidden Activation function - relu
21 # output activation function - sigmoid
22 # learning rate - 0.6
23 # lambd=0.1
24 parameters = L_Layer_DeepModel(train_X, train_Y, layersDimensions,
25 hiddenActivationFunc='relu', outputActivationFunc="sigmoid",learningRate =
26 0.6, lambd=0.1, num_iterations = 9000, initType="default", print_cost =
27 True,figure="fig7.png")
28
29 # Clear the plot
30 plt.clf()
31 plt.close()
32
33 # Plot the decision boundary
34 plot_decision_boundary(lambda x: predict(parameters, x.T), train_X,
35 train_Y,str(0.6),figure1="fig8.png")
36
37 plt.clf()
38 plt.close()

```





2.1b Regularization: Spiral data – Python

```

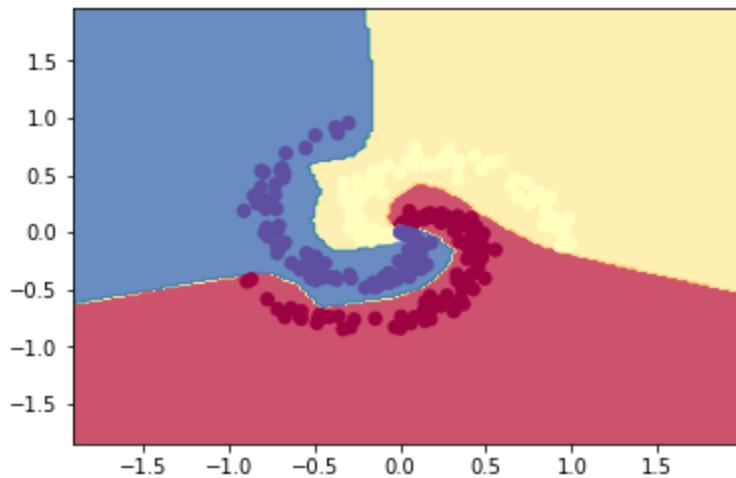
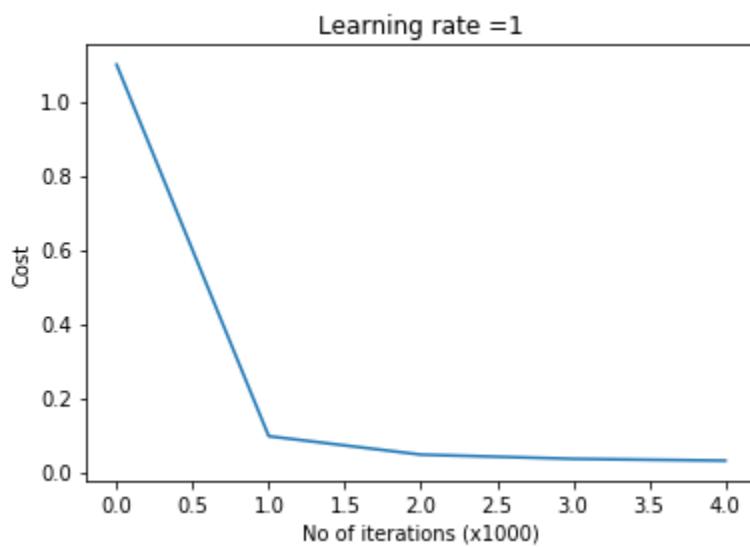
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6 import sklearn
7 import sklearn.datasets
8 exec(open("DLfunctions61.py").read())
9
10 N = 100 # number of points per class
11 D = 2 # dimensionality
12 K = 3 # number of classes
13 X = np.zeros((N*K,D)) # data matrix (each row = single example)
14 y = np.zeros(N*K, dtype='uint8') # class labels
15 for j in range(K):
16     ix = range(N*j,N*(j+1))
17     r = np.linspace(0.0,1,N) # radius
18     t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
19     X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
20     y[ix] = j
21
22 # Plot the data
23 plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
24 plt.clf()
25 plt.close()
26
27 #Set layer dimensions
28 # 2 - number of input features
29 # 100 - 1 hidden layer 100 hidden units
30 # 3 -3 classes with softmax unit at the output layer
31 layersDimensions = [2,100,3]
32 y1=y.reshape(-1,1).T
33
34 # Train a L-layer deep learning network
35 # hidden Activation function - relu
36 # output activation function - softmax
37 # learning rate - 1
38

```

```

39 # lambd=1e-3
40 parameters = L_Layer_DeepModel(x.T, y1, layersDimensions,
41 hiddenActivationFunc='relu', outputActivationFunc="softmax", learningRate =
42 1,lambd=1e-3, num_iterations = 5000, print_cost = True,figure="fig9.png")
43
44 plt.clf()
45 plt.close()
46
47 # Plot decision boundary
48 w1=parameters['w1']
49 b1=parameters['b1']
50 w2=parameters['w2']
51 b2=parameters['b2']
52 plot_decision_boundary1(x, y1,w1,b1,w2,b2,figure2="fig10.png")

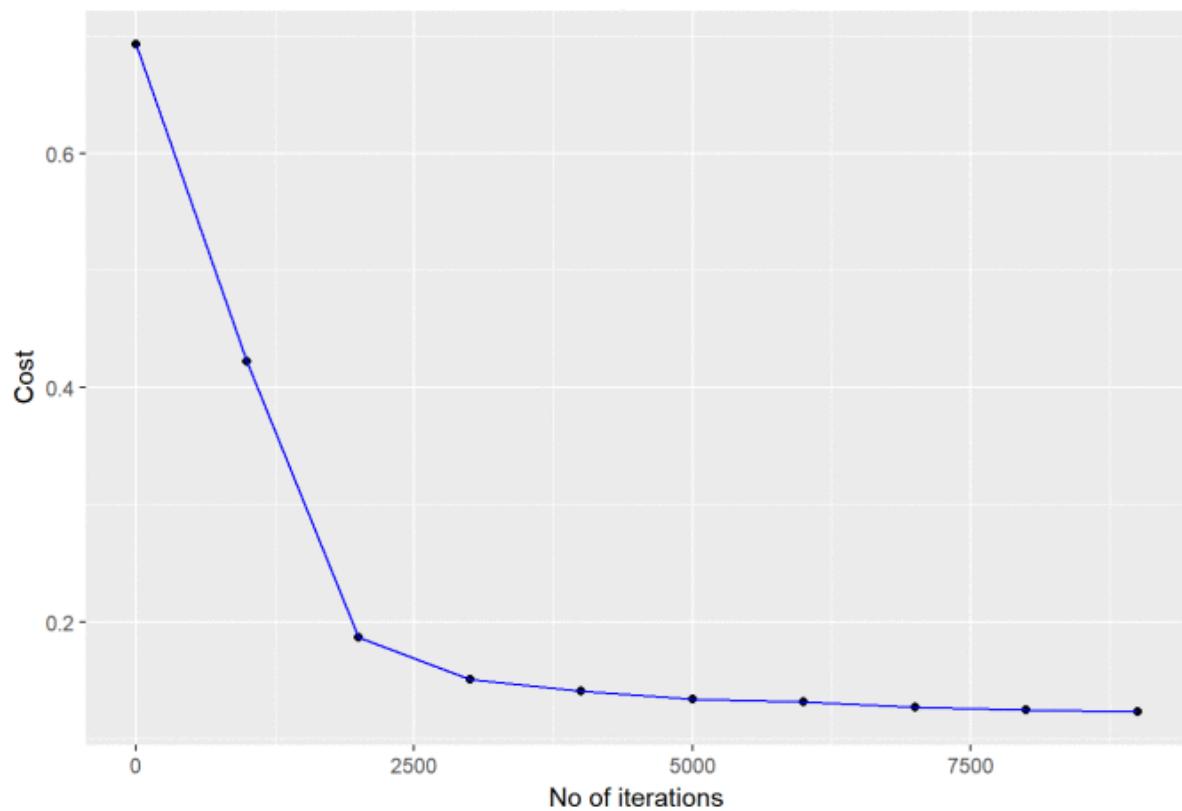
```



2.2a Regularization: Circles data – R

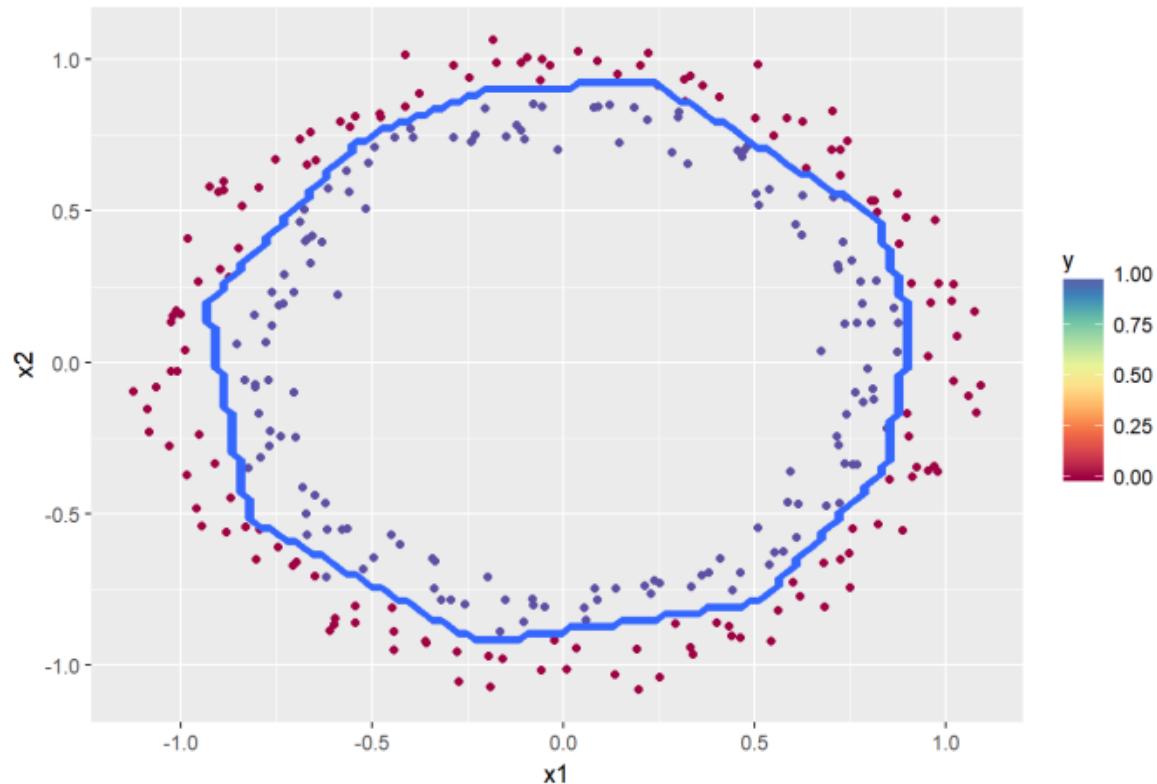
```
1 source("DLfunctions61.R")
2 #Load data
3 df=read.csv("circles.csv",header=FALSE)
4 z <- as.matrix(read.csv("circles.csv",header=FALSE))
5 x <- z[,1:2]
6 y <- z[,3]
7 X <- t(x)
8 Y <- t(y)
9
10 #Set layer dimensions
11 # 2 - number of input features
12 # 11 - 1 hidden layer 11 hidden units
13 # 1 - 1 sigmoid activation unit at the output layer
14 layersDimensions = c(2,11,1)
15
16 # Train a L-layer deep learning network
17 # hidden Activation function - relu
18 # output activation function - sigmoid
19 # learning rate - 0.6
20 # lambd=0.1
21 retvals = L_Layer_DeepModel(x, Y, layersDimensions,
22                             hiddenActivationFunc='relu',
23                             outputActivationFunc="sigmoid",
24                             learningRate = 0.5,
25                             lambd=0.1,
26                             numIterations = 9000,
27                             initType="default",
28                             print_cost = True)
29
30 #Plot the cost vs iterations
31 iterations <- seq(0,9000,1000)
32 costs=retvals$costs
33 df=data.frame(iterations,costs)
34 ggplot(df,aes(x=iterations,y=costs)) + geom_point() + geom_line(color="blue")
35 + ggttitle("Costs vs iterations") + xlab("No of iterations") + ylab("Cost")
```

Costs vs iterations



```
1 # Plot the decision boundary
2 plotDecisionBoundary(z,retvals,hiddenActivationFunc="relu",0.5)
```

Decision boundary for learning rate: 0.5



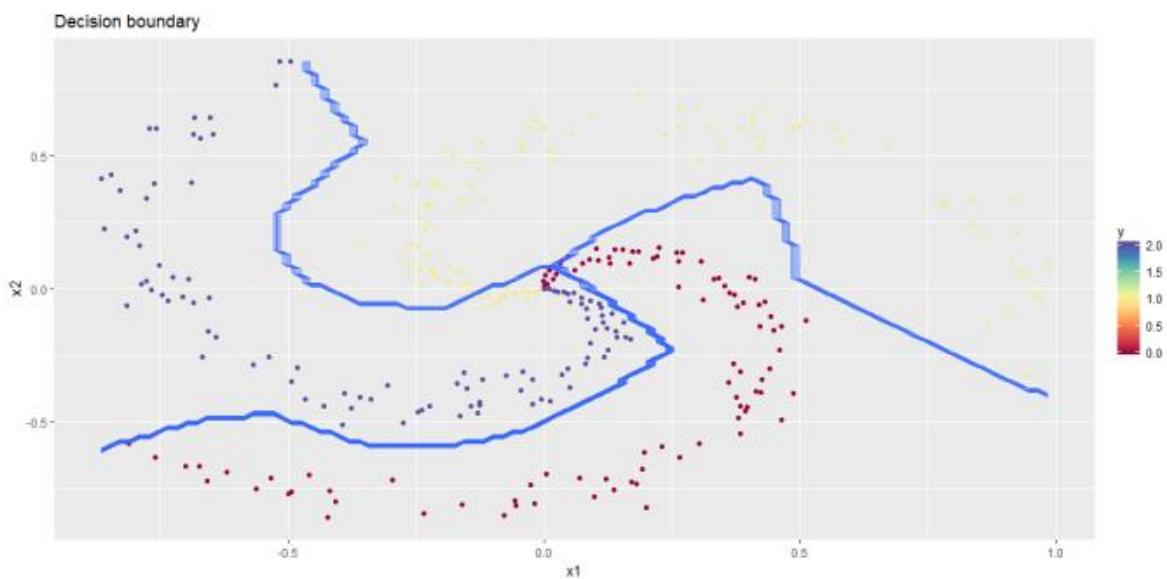
2.2b Regularization:Spiral data – R

```
1 # Read the spiral dataset
2 #Load the data
3 source("DLfunctions61.R")
4 Z <- as.matrix(read.csv("spiral.csv",header=FALSE))
5
6 # Setup the data
7 X <- Z[,1:2]
8 y <- Z[,3]
9 X <- t(X)
10 Y <- t(y)
11
12 # Set layer dimensions
13 layersDimensions = c(2, 100, 3)
14 # 2 - number of input features
15 # 100 - 1 hidden layer 100 hidden units
16 # 3 - 3 softmax classes at the output layer
17
18 # Train a L-layer deep learning network
19 # hidden Activation function - relu
20 # output activation function - sigmoid
21 # learning rate - 0.5
22 # lambd=0.01
23 retvals = L_Layer_DeepModel(X, Y, layersDimensions,
24     hiddenActivationFunc='relu',
25     outputActivationFunc="softmax",
26     learningRate = 0.5,
27     lambd=0.01,
28     numIterations = 9000,
```

```

29     print_cost = True)
30 parameters<-retvals$parameters
31
32 #Plot decision boundary
33 plotDecisionBoundary1(z,parameters)

```

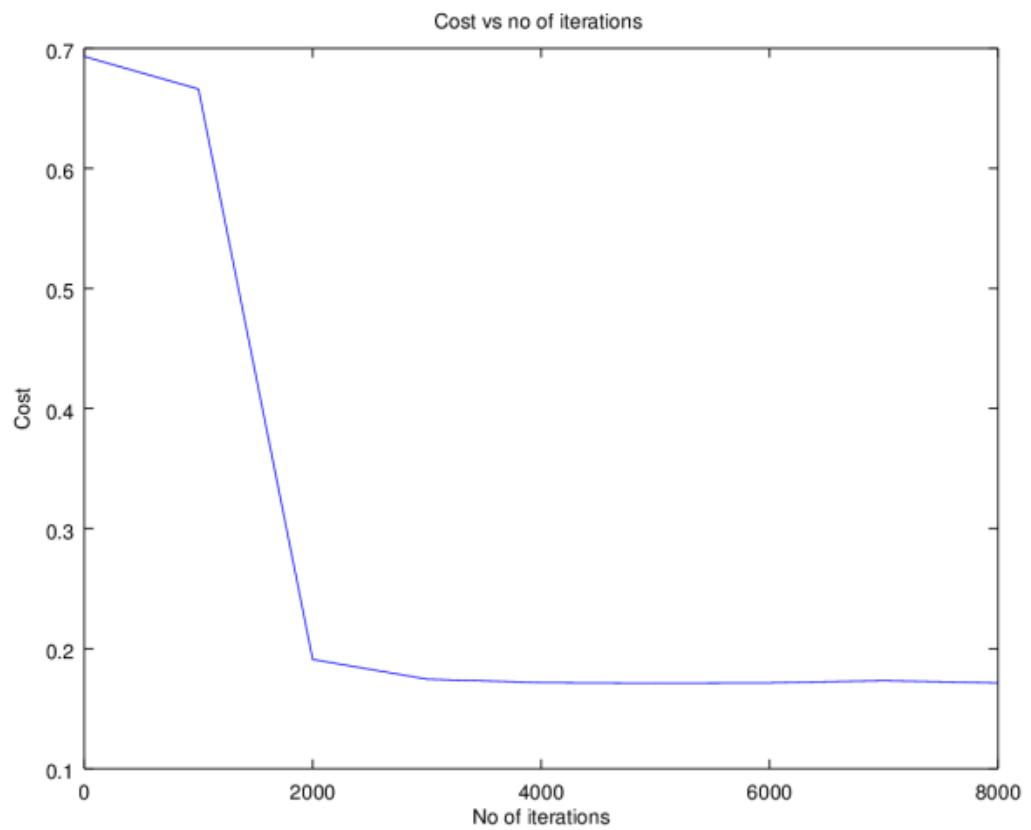


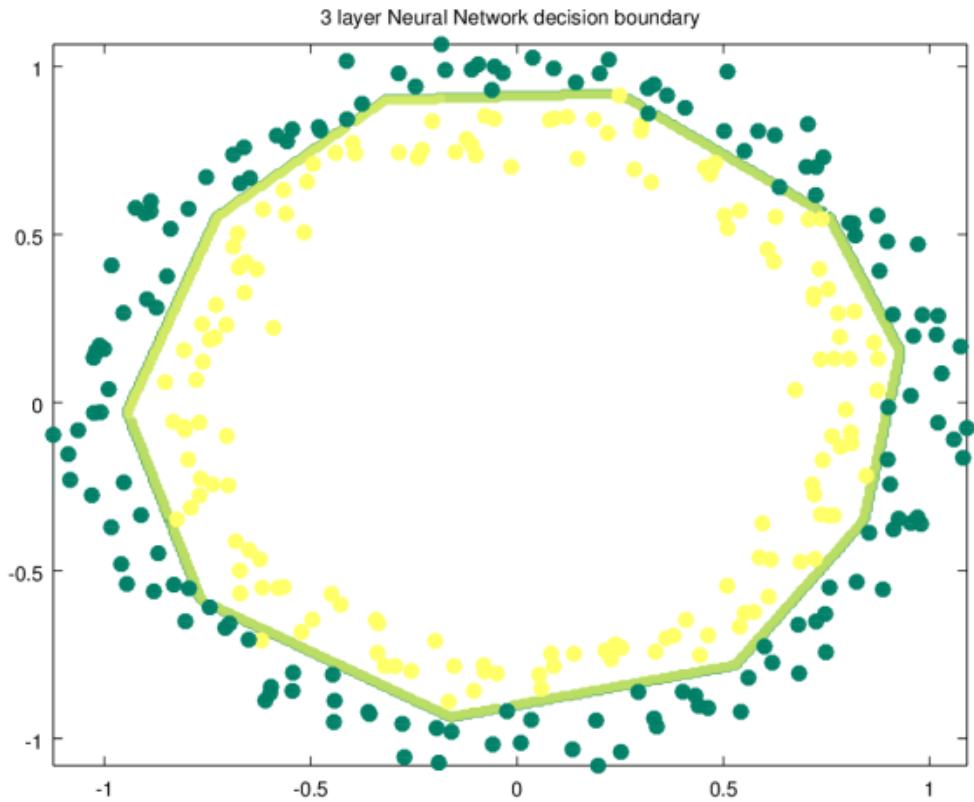
2.3a Regularization: Circles data – Octave

```

1 #
2 source("DL61functions.m")
3 #Load data
4 data=csvread("circles.csv");
5 X=data(:,1:2);
6 Y=data(:,3);
7
8 # Set layer dimensions
9 # 2 - number of input features
10 # 11 - 1 hidden layer 11 hidden units
11 # 1 - 1 sigmoid activation unit at the output layer
12 layersDimensions = [2 11 1]; #tanh=-0.5(ok), #relu=0.1 best!
13
14 # Train a L-layer deep learning network
15 # hidden Activation function - relu
16 # output activation function - sigmoid
17 # learning rate - 0.5
18 # lambd=0.2
19 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
20                                         hiddenActivationFunc='relu',
21                                         outputActivationFunc="sigmoid",
22                                         learningRate = 0.5,
23                                         lambd=0.2,
24                                         keep_prob=1,
25                                         numIterations = 8000,
26                                         initType="default");
27
28 #Plot cost vs iterations
29 plotCostVsIterations(8000,costs)
30
31 #Plot decision boundary
32 plotDecisionBoundary(data,weights,
33 biases,keep_prob=1,hiddenActivationFunc="relu")

```





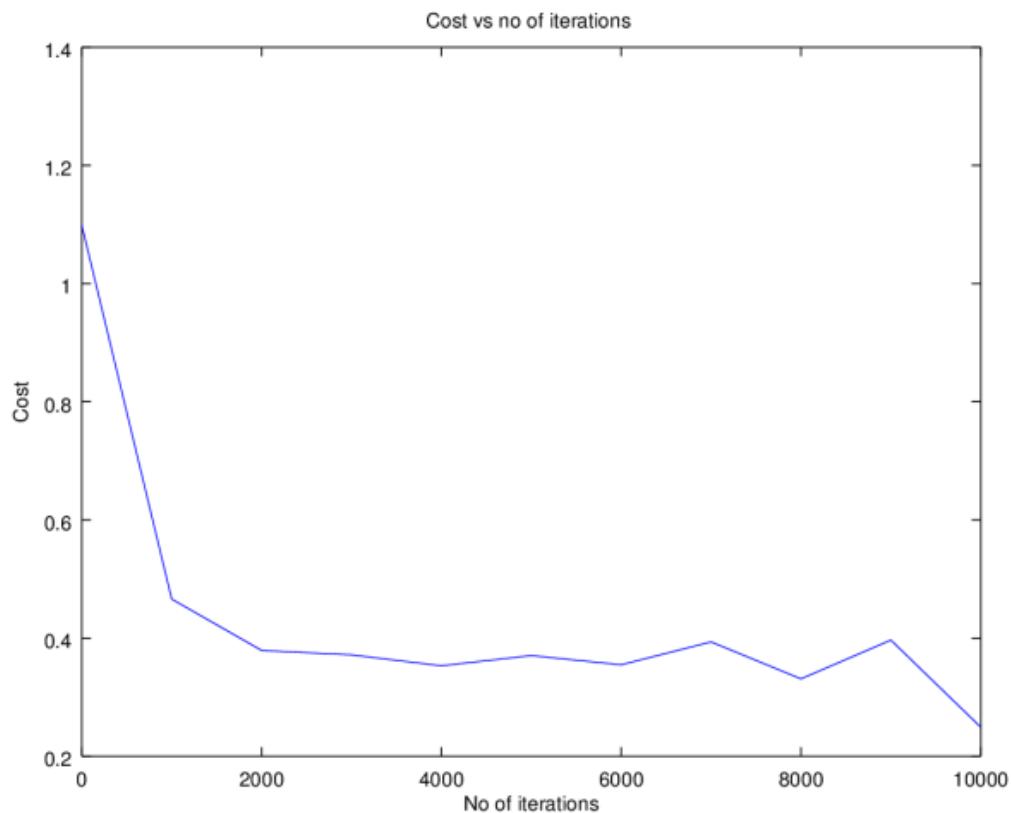
2.3b Regularization:Spiral data 2 – Octave

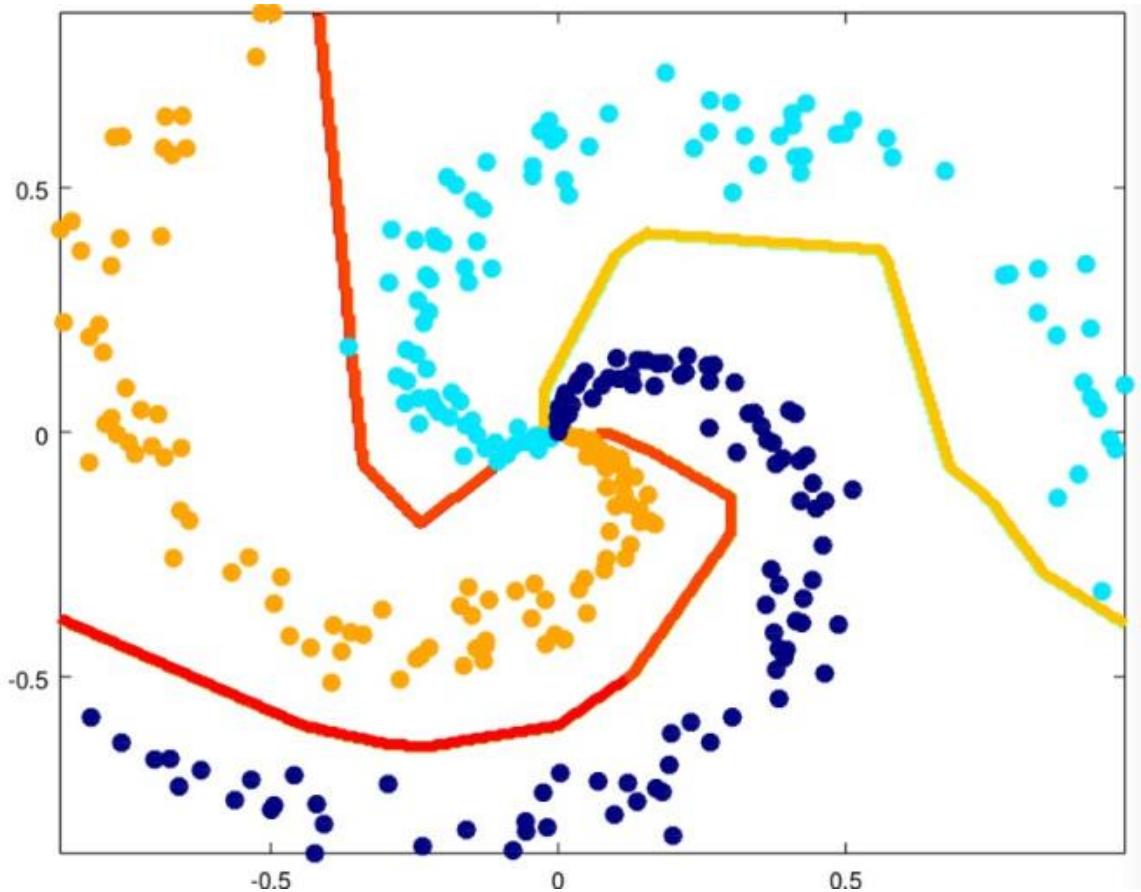
```

1 # 
2 source("DL61functions.m");
3 data=csvread("spiral.csv");
4 
5 # Setup the data
6 X=data(:,1:2);
7 Y=data(:,3);
8 
9 # Set layer dimensions
10 # 2 - number of input features
11 # 100 - 1 hidden layer 100 hidden units
12 # 3 -3 classes with softmax activation unit at the output layer
13 layersDimensions = [2 100 3]
14 
15 # Train a L-layer deep learning network
16 # hidden Activation function - relu
17 # output activation function - sigmoid
18 # learning rate - 0.6
19 # lambd=0.2
20 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
21                                         hiddenActivationFunc='relu',
22                                         outputActivationFunc="softmax",
23                                         LearningRate = 0.6,
24                                         lambd=0.2,
25                                         keep_prob=1,
26                                         numIterations = 10000);

```

```
27  
28 #Plot cost vs iterations  
29 plotCostVsIterations(10000,costs)  
30  
31 #Plot decision boundary  
32 plotDecisionBoundary1(data,weights,  
33 biases,keep_prob=1,hiddenActivationFunc="relu")
```





3.1a Dropout: Circles data – Python

The ‘dropout’ regularization technique was used with great effectiveness, to prevent overfitting by Alex Krizhevsky, Ilya Sutskever and Prof Geoffrey E. Hinton in the ‘Imagenet classification with Deep Convolutional Neural Networks’²

[\(<https://www.nvidia.cn/content/tesla/pdf/machine-learning/imagenet-classification-with-deep-convolutional-nn.pdf>\)](https://www.nvidia.cn/content/tesla/pdf/machine-learning/imagenet-classification-with-deep-convolutional-nn.pdf)

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

```

1 for l =1:L-1
2     ...
3     D=rand(size(A)(1),size(A)(2));
4     D = (D < keep_prob);
5     # Zero out some hidden units
6     A= A .* D;
7     # Divide by keep_prob to keep the expected value of A the same
8     A = A ./ keep_prob;
9     # Store D in a dropoutMat cell array
10    dropoutMat{l}=D;
11    ...
12 endfor

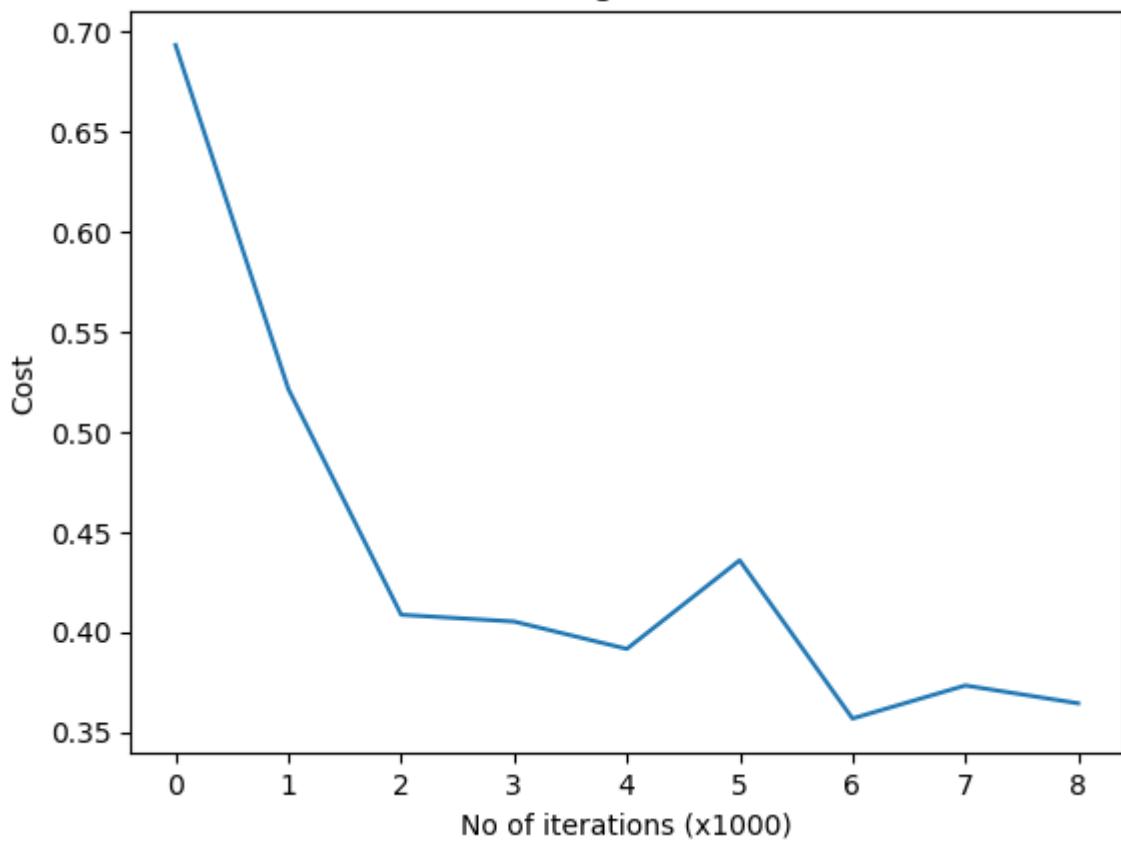
```

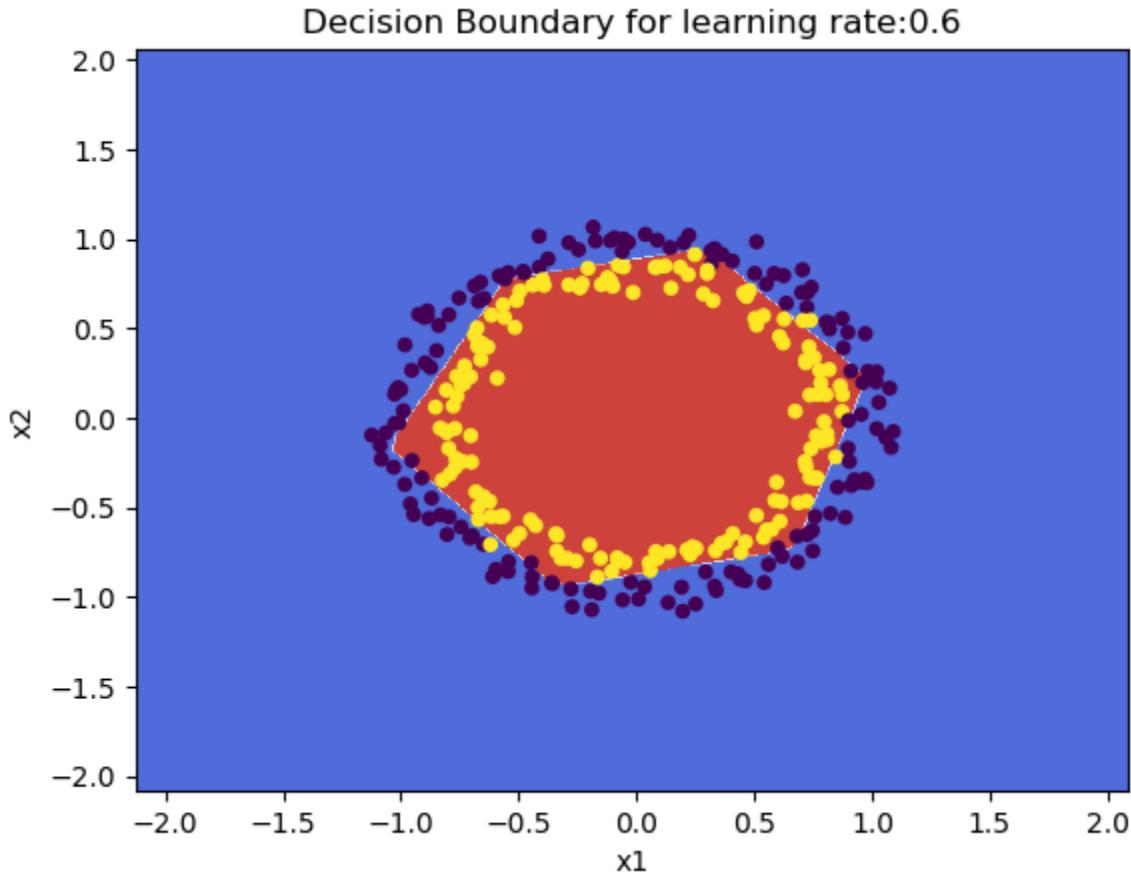
In the backward propagation cycle we have

```
1 for l = (L-1):-1:1
2     ...
3     D = dropoutMat{l};
4     # Zero out the dA1 based on same dropout matrix
5     dA1= dA1 .* D;
6     # Divide by keep_prob to maintain the expected value
7     dA1 = dA1 ./ keep_prob;
8
9 endfor ...
```

```
1 #
2 import numpy as np
3 import matplotlib
4 import matplotlib.pyplot as plt
5 import sklearn.linear_model
6 import pandas as pd
7 import sklearn
8 import sklearn.datasets
9 exec(open("DLfunctions61.py").read())
10
11 #Load the data
12 train_X, train_Y, test_X, test_Y = load_dataset()
13
14 # Set the layers dimensions
15 # 2 - number of input features
16 # 7 - 1 hidden layer 7 hidden units
17 # 1 - 1 sigmoid activation unit at the output layer
18 layersDimensions = [2,7,1]
19
20 # Train a L-layer deep learning network
21 # hidden Activation function - relu
22 # output activation function - sigmoid
23 # learning rate - 0.6
24 #keep_prob=0.7
25 parameters = L_Layer_DeepModel(train_X, train_Y, layersDimensions,
26 hiddenActivationFunc='relu', outputActivationFunc="sigmoid",learningRate =
27 0.6, keep_prob=0.7, num_iterations = 9000, initType="default", print_cost
= True,figure="fig11.png")
28
29 # Clear the plot
30 plt.clf()
31 plt.close()
32
33 # Plot the decision boundary
34 plot_decision_boundary(lambda x: predict(parameters, x.T,keep_prob=0.7),
35 train_X, train_Y,str(0.6),figure1="fig12.png")
```

Learning rate =0.6





3.1b Dropout: Spiral data – Python

```

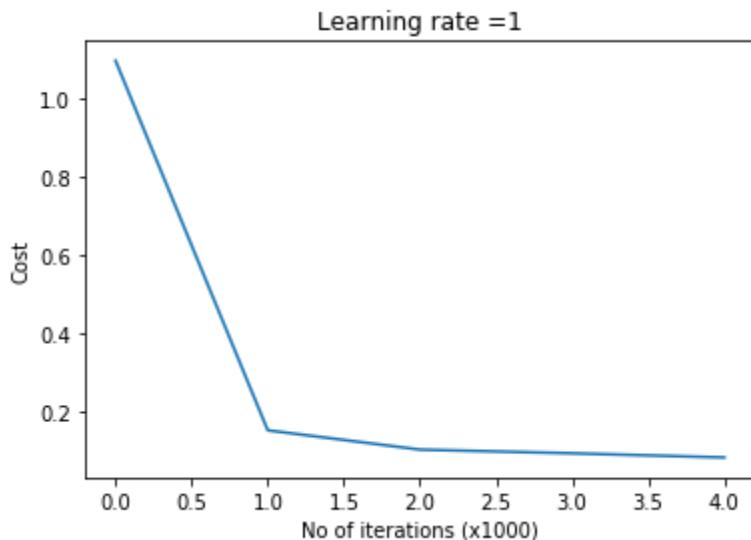
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6 import sklearn
7 import sklearn.datasets
8 exec(open("DLfunctions61.py").read())
9 # Create an input data set - Taken from CS231n Convolutional Neural networks,
10 # http://cs231n.github.io/neural-networks-case-study/
11
12
13 N = 100 # number of points per class
14 D = 2 # dimensionality
15 K = 3 # number of classes
16 X = np.zeros((N*K,D)) # data matrix (each row = single example)
17 y = np.zeros(N*K, dtype='uint8') # class labels
18 for j in range(K):
19     ix = range(N*j,N*(j+1))
20     r = np.linspace(0.0,1,N) # radius
21     t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
22     X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
23     y[ix] = j
24
25

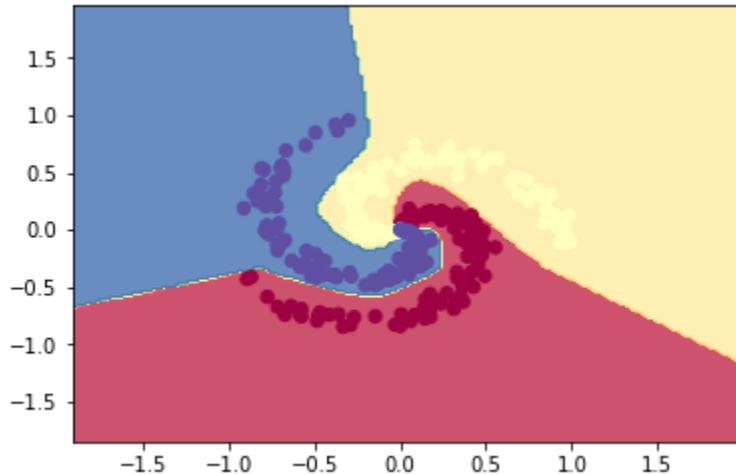
```

```

26 # Plot the data
27 plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
28 plt.clf()
29 plt.close()
30
31 #Set layer dimensions
32 # 2 - number of input features
33 # 100 - 1 hidden layer 100 hidden units
34 # 3 -3 classes with softmax activation unit at the output layer
35 layersDimensions = [2,100,3]
36 y1=y.reshape(-1,1).T
37
38 # Train a L-layer deep learning network
39 # hidden Activation function - relu
40 # output activation function - softmax
41 # learning rate - 1
42 #keep_prob=0.9
43 parameters = L_Layer_DeepModel(X.T, y1, layersDimensions,
44 hiddenActivationFunc='relu', outputActivationFunc="softmax", learningRate =
45 1,keep_prob=0.9, num_iterations = 5000, print_cost = True,figure="fig13.png")
46
47 plt.clf()
48 plt.close()
49 w1=parameters['w1']
50 b1=parameters['b1']
51 w2=parameters['w2']
52 b2=parameters['b2']
53
54 #Plot decision boundary
55 plot_decision_boundary1(x, y1,w1,b1,w2,b2,figure2="fig14.png")

```





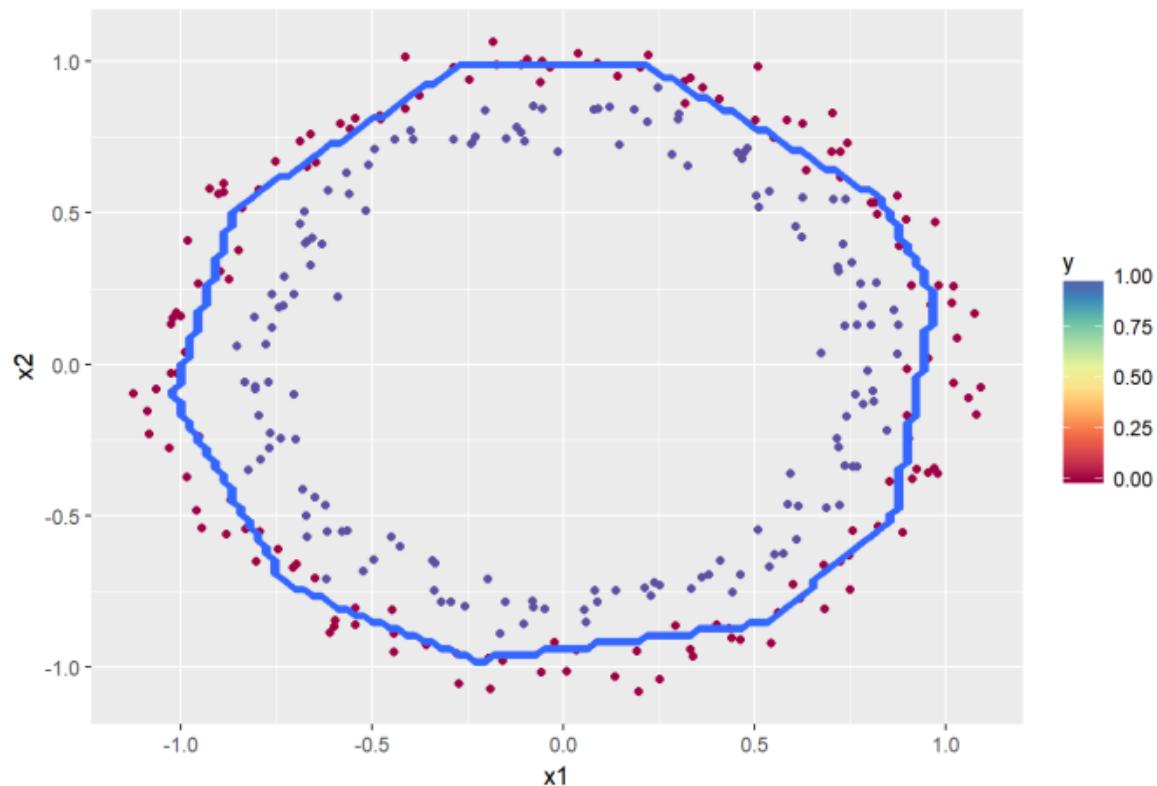
3.2a Dropout: Circles data – R

```

1 source("DLfunctions61.R")
2 #Load data
3 df=read.csv("circles.csv",header=FALSE)
4 z <- as.matrix(read.csv("circles.csv",header=FALSE))
5
6 x <- z[,1:2]
7 y <- z[,3]
8 X <- t(x)
9 Y <- t(y)
10
11 # Set layer dimensions
12 # 2 - number of input features
13 # 11 - 1 hidden layer 11 hidden units
14 # 1 - 1 sigmoid activation unit at the output layer
15 layersDimensions = c(2,11,1)
16
17 # Train a L-layer deep learning network
18 # hidden Activation function - relu
19 # output activation function - sigmoid
20 # learning rate - 0.5
21 # keep_prob=0.8
22 retvals = L_Layer_DeepModel(X, Y, layersDimensions,
23                             hiddenActivationFunc='relu',
24                             outputActivationFunc="sigmoid",
25                             learningRate = 0.5,
26                             keep_prob=0.8,
27                             numIterations = 9000,
28                             initType="default",
29                             print_cost = True)
30
31 # Plot the decision boundary
32 plotDecisionBoundary(z,retvals,keep_prob=0.6,
33 hiddenActivationFunc="relu",0.5)

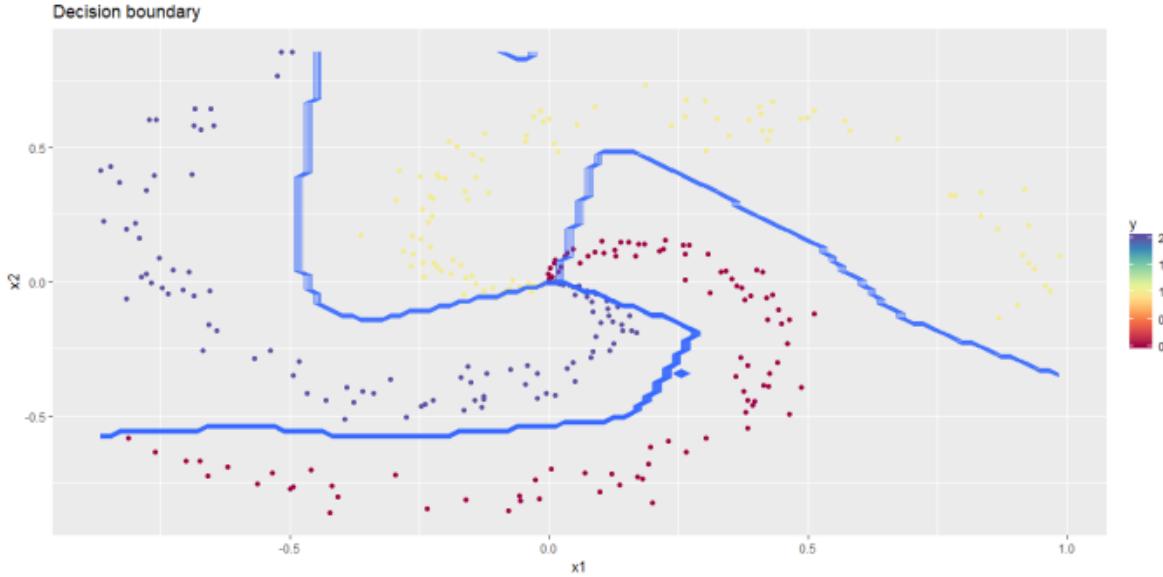
```

Decision boundary for learning rate: 0.5



3.2b Dropout: Spiral data – R

```
1 # Read the spiral dataset
2 source("DLfunctions61.R")
3 # Load data
4 Z <- as.matrix(read.csv("spiral.csv",header=FALSE))
5
6 # Setup the data
7 X <- Z[,1:2]
8 y <- Z[,3]
9 X <- t(X)
10 Y <- t(y)
11
12 # Train a L-layer deep learning network
13 # hidden Activation function - relu
14 # output activation function - softmax
15 # learning rate - 0.1
16 #keep_prob=0.9
17 retvals = L_Layer_DeepModel(X, Y, layersDimensions,
18                             hiddenActivationFunc='relu',
19                             outputActivationFunc="softmax",
20                             learningRate = 0.1,
21                             keep_prob=0.90,
22                             numIterations = 9000,
23                             print_cost = True)
24
25 parameters<-retvals$parameters
26 #Plot decision boundary
27 plotDecisionBoundary1(Z,parameters)
```

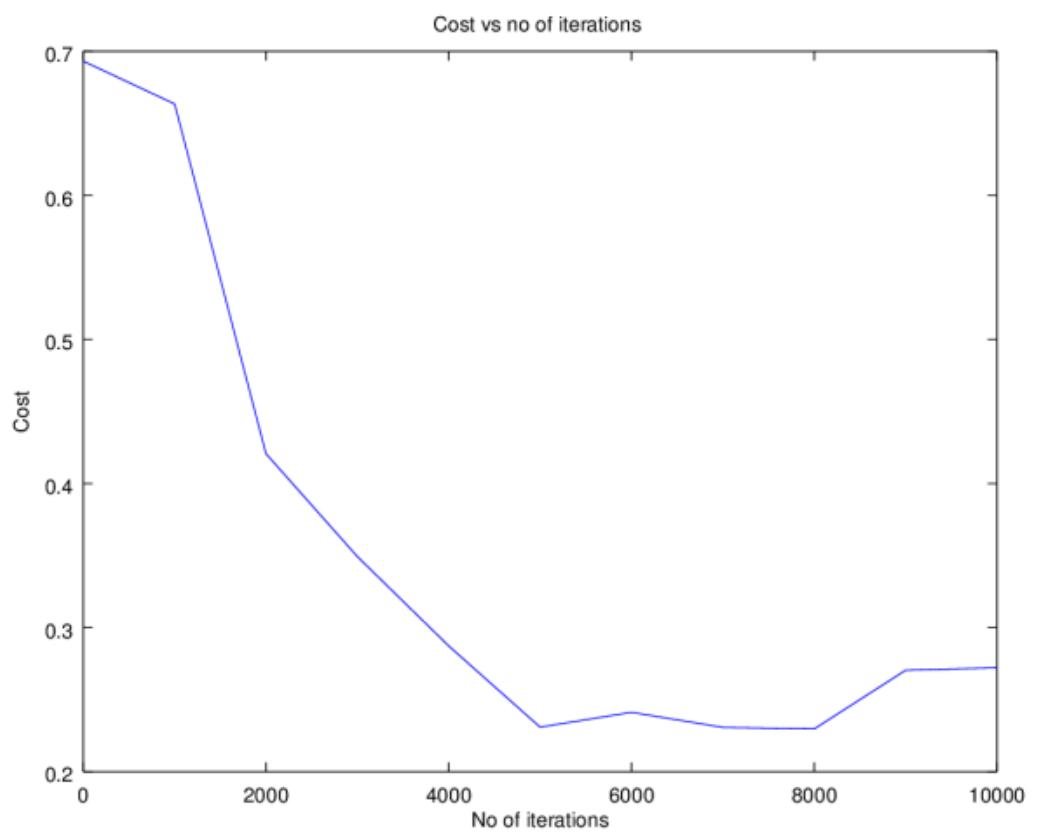


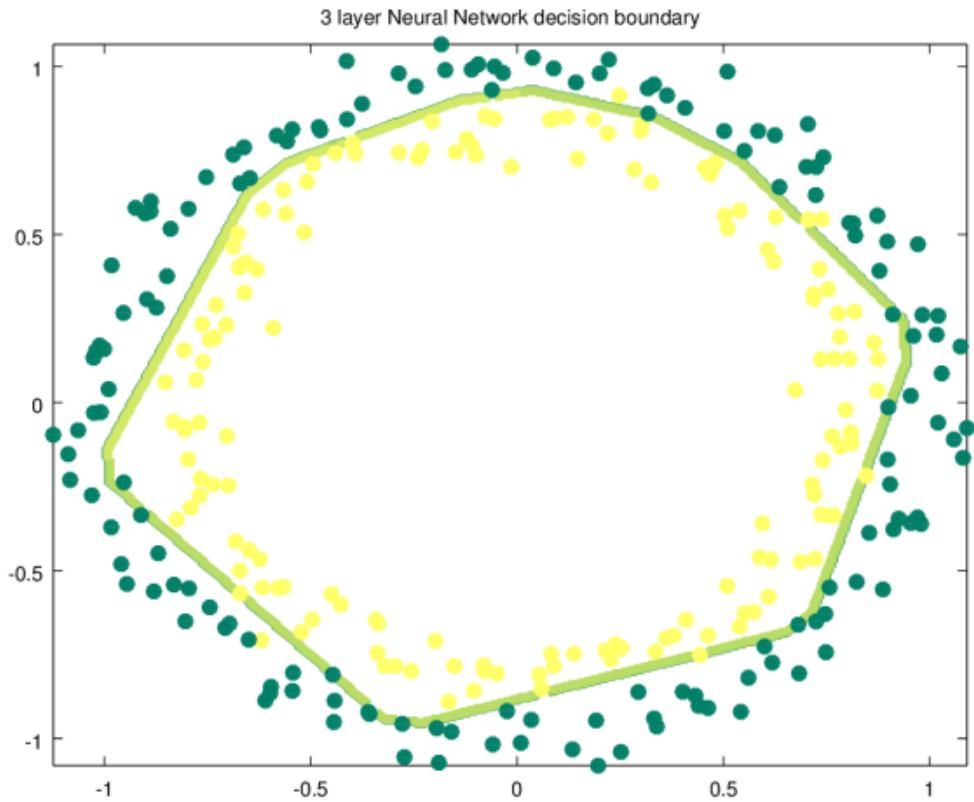
3.3a Dropout: Circles data – Octave

```

1 data=csvread("circles.csv");
2
3 X=data(:,1:2);
4 Y=data(:,3);
5
6 # Set layer dimensions
7 # 2 - number of input features
8 # 11 - 1 hidden layer 11 hidden units
9 # 1 - 1 sigmoid activation unit at the output layer
10 layersDimensions = [2 11 1];
11
12 # Train a L-layer deep learning network
13 # hidden Activation function - relu
14 # output activation function - sigmoid
15 # learning rate - 0.5
16 # keep_prob=0.8
17 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
18                                         hiddenActivationFunc='relu',
19                                         outputActivationFunc="sigmoid",
20                                         learningRate = 0.5,
21                                         lambd=0,
22                                         keep_prob=0.8,
23                                         numIterations = 10000,
24                                         initType="default");
25
26 #Plot cost vs iterations
27 plotCostVsIterations(10000,costs)
28
29 #Plot decision boundary
30 plotDecisionBoundary1(data,weights, biases,keep_prob=1,
31 hiddenActivationFunc="relu")
32

```





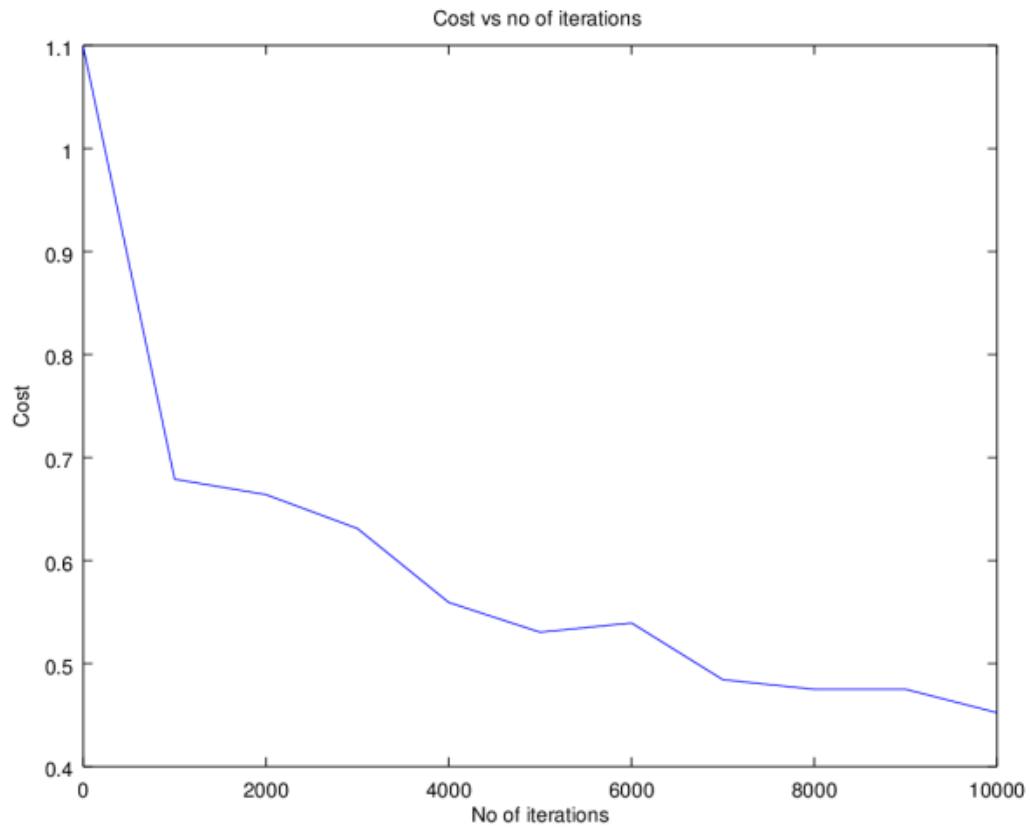
3.3b Dropout Spiral data – Octave

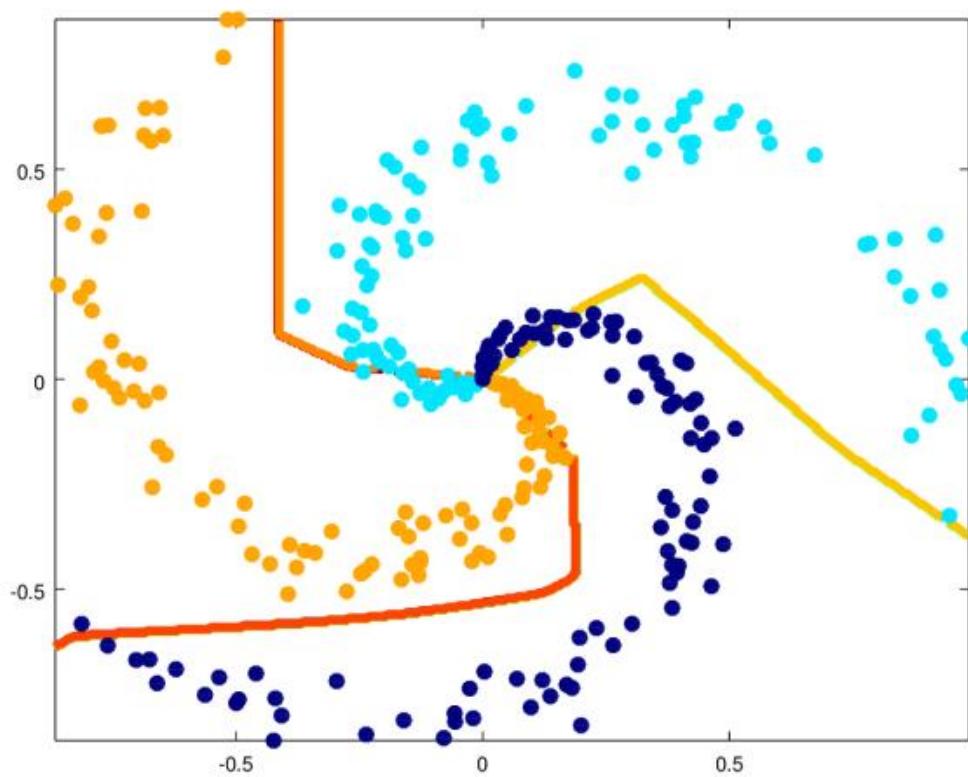
```

1 source("DL61functions.m")
2 data=csvread("spiral.csv");
3
4 # Setup the data
5 X=data(:,1:2);
6 Y=data(:,3);
7
8 # Set layer dimensions
9 layersDimensions = [numFeats numHidden numOutput];
10
11 # Train a L-layer deep learning network
12 # hidden Activation function - relu
13 # output activation function - softmax
14 # learning rate - 0.1
15 #keep_prob=0.8
16 [weights biases costs]=L_Layer_DeepModel(X', Y', layersDimensions,
17                                         hiddenActivationFunc='relu',
18                                         outputActivationFunc="softmax",
19                                         learningRate = 0.1,
20                                         lambd=0,
21                                         keep_prob=0.8,
22                                         numIterations = 10000);
23
24 #Plot cost vs iterations
25 plotCostVsIterations(10000,costs)
26

```

```
27 #Plot decision boundary
28 plotDecisionBoundary1(data,weights, biases,keep_prob=1,
29 hiddenActivationFunc="relu")
```





4. Conclusion

This post further enhances my earlier L-Layer generic implementation of a Deep Learning network to include options for initialization techniques, L2 regularization or dropout regularization

7.Gradient Descent Optimization techniques

Artificial Intelligence is the new electricity. – Prof Andrew Ng

Most of human and animal learning is unsupervised learning. If intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake. We know how to make the icing and the cherry, but we don't know how to make the cake. We need to solve the unsupervised learning problem before we can even think of getting to true AI. – Yann LeCun, March 14, 2016 (Facebook)

1. Introduction

In this chapter 7 of 'Deep Learning from first principles, I implement optimization methods used in Stochastic Gradient Descent (SGD) to speed up the convergence. Specifically, I discuss and implement the following gradient descent optimization techniques

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

This chapter, further enhances my generic L-Layer Deep Learning Network implementations in vectorized Python, R and Octave to also include the Stochastic Gradient Descent optimization techniques. These vectorized implementation are in Appendix 7 - Gradient Descent Optimization techniques

You can clone/download the code from Github at DeepLearningFromFirstPrinciples (<https://github.com/tvganesh/DeepLearningFromFirstPrinciples/tree/master/Chap7-DLGradientDescentOptimization>). Incidentally, a good discussion of the various optimizations methods used in Stochastic Gradient Optimization techniques can be seen at Sebastian Ruder's blog (<http://ruder.io/optimizing-gradient-descent/>)

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

This chapter is largely based of on Prof Andrew Ng's Deep Learning Specialization (<https://www.coursera.org/specializations/deep-learning>). All the optimization techniques discussed here use Stochastic Gradient Descent and are based on the technique of exponentially weighted average method. So, for example if we had some time series data $\theta_1, \theta_2, \theta_3, \dots, \theta_t$ then we can represent the exponentially average value at time 't' as a sequence of the previous

value v_{t-1} and θ_t as shown below

$$v_t = \beta v_{t-1} + (1 - \beta) \theta_t$$

Here v_t represent the average of the data set over $\frac{1}{1-\beta}$. By choosing different values of β , we can average over a larger or smaller number of the data points.

We can write the equations as follows

$$v_t = \beta v_{t-1} + (1 - \beta) \theta_t$$

$$v_{t-1} = \beta v_{t-2} + (1 - \beta) \theta_{t-1}$$

$$v_{t-2} = \beta v_{t-3} + (1 - \beta) \theta_{t-2}$$

and

$$v_{t-k} = \beta v_{t-(k+1)} + (1 - \beta) \theta_{t-k}$$

By substitution we have

$$v_t = (1 - \beta) \theta_t + \beta v_{t-1}$$

$$v_t = (1 - \beta) \theta_t + \beta((1 - \beta) \theta_{t-1}) + \beta v_{t-2}$$

$$v_t = (1 - \beta) \theta_t + \beta((1 - \beta) \theta_{t-1}) + \beta((1 - \beta) \theta_{t-2} + \beta v_{t-3})$$

Hence it can be seen that the v_t is the weighted sum over the previous values θ_k , which is an exponentially decaying function.

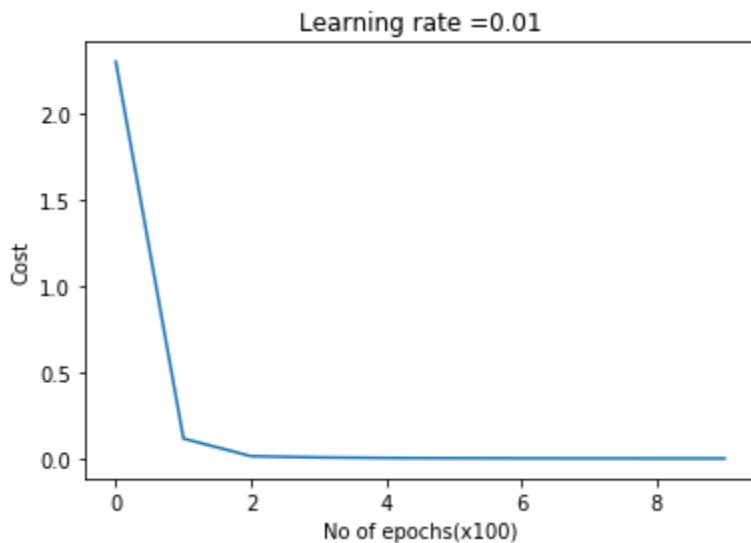
1.1a. Stochastic Gradient Descent (Vanilla) – Python

```
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6 import sklearn
7 import sklearn.datasets
8 exec(open("DLfunctions7.py").read())
9 exec(open("load_mnist.py").read())
10
11 # Read the MNIST training data and labels
12 training=list(read(dataset='training',path=".\\mnist"))
13 test=list(read(dataset='testing',path=".\\mnist"))
14 lbls=[]
15 pxls=[]
16
17 #Loop
18 for i in range(60000):
19     l,p=training[i]
20     lbls.append(l)
21     pxls.append(p)
22 labels= np.array(lbls)
23 pixels=np.array(pxls)
24 y=labels.reshape(-1,1)
25 X=pixels.reshape(pixels.shape[0],-1)
26 X1=X.T
27 Y1=y.T
```

```

29 # Create a list of 1024 random numbers.
30 permutation = list(np.random.permutation(2**10))
31
32 # Subset 1024 training samples from the data
33 X2 = X1[:, permutation]
34 Y2 = Y1[:, permutation].reshape((1,2**10))
35
36 # Set the layer dimensions
37 # 784 - number of input features (28 x28)
38 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
39 # 10 - 10 output classes with softmax activation unit at the output layer
40 layersDimensions=[784, 15,9,10]
41
42 # Execute the L-Layer Deep Network to perform SGD with regular gradient
43 descent
44 # hidden Activation function - relu
45 # output activation function - softmax
46 # learning rate - 0.01
47 # optimizer="gd" # gradient descent
48 # mini_batch_size = 512
49 parameters = L_Layer_DeepModel_SGD(X2, Y2, layersDimensions,
50 hiddenActivationFunc='relu',
51
52 outputActivationFunc="softmax",learningRate = 0.01 ,
53 optimizer="gd", mini_batch_size =512,
54 num_epochs = 1000, print_cost = True,figure="fig1.png")

```



1.1b. Stochastic Gradient Descent (Vanilla) – R

```

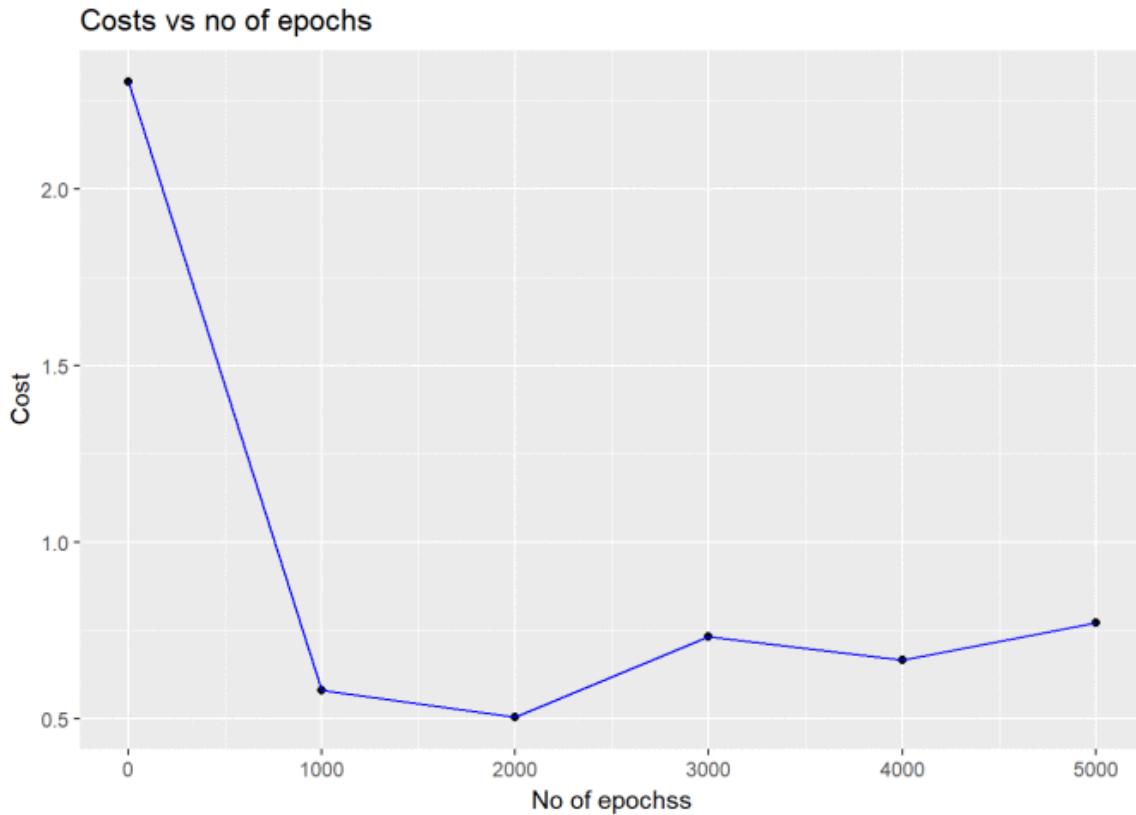
1 source("mnist.R")
2 source("DLfunctions7.R")
3
4 #Load and read MNIST data
5 load_mnist()
6 x <- t(train$x)
7 x <- x[,1:60000]
8 y <- train$y
9 y1 <- y[1:60000]
10 y2 <- as.matrix(y1)
11 Y=t(y2)

```

```

12
13 # Subset 1024 random samples from MNIST
14 permutation = c(sample(2^10))
15
16 # Randomly shuffle the training data
17 X1 = X[, permutation]
18 Y1 = Y[1, permutation]
19 y2 <- as.matrix(y1)
20 Y1=t(y2)
21
22 # Set layer dimensions
23 # 784 - number of input features (28 x28)
24 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
25 # 10 - 10 output classes with softmax activation unit at the output layer
26 layersDimensions=c(784, 15, 9, 10)
27
28 # Execute the L-Layer Deep Network to perform SGD with regular gradient
29 # descent
30 # hidden Activation function - tanh
31 # output activation function - softmax
32 # learning rate - 0.05
33 # optimizer="gd" # gradient descent
34 # mini_batch_size = 512
35 retvalsSGD= L_Layer_DeepModel_SGD(X1, Y1, layersDimensions,
36                                     hiddenActivationFunc='tanh',
37                                     outputActivationFunc="softmax",
38                                     learningRate = 0.05,
39                                     optimizer="gd",
40                                     mini_batch_size = 512,
41                                     num_epochs = 5000,
42                                     print_cost = True)
43
44 #Plot the cost vs number of epochs
45 iterations <- seq(0,5000,1000)
46 costs=retvalsSGD$costs
47 df=data.frame(iterations,costs)
48 ggplot(df,aes(x=iterations,y=costs)) + geom_point() + geom_line(color="blue")
49 + ggttitle("Costs vs no of epochs") + xlab("No of epochss") + ylab("Cost")

```



1.1c. Stochastic Gradient Descent (Vanilla) – Octave

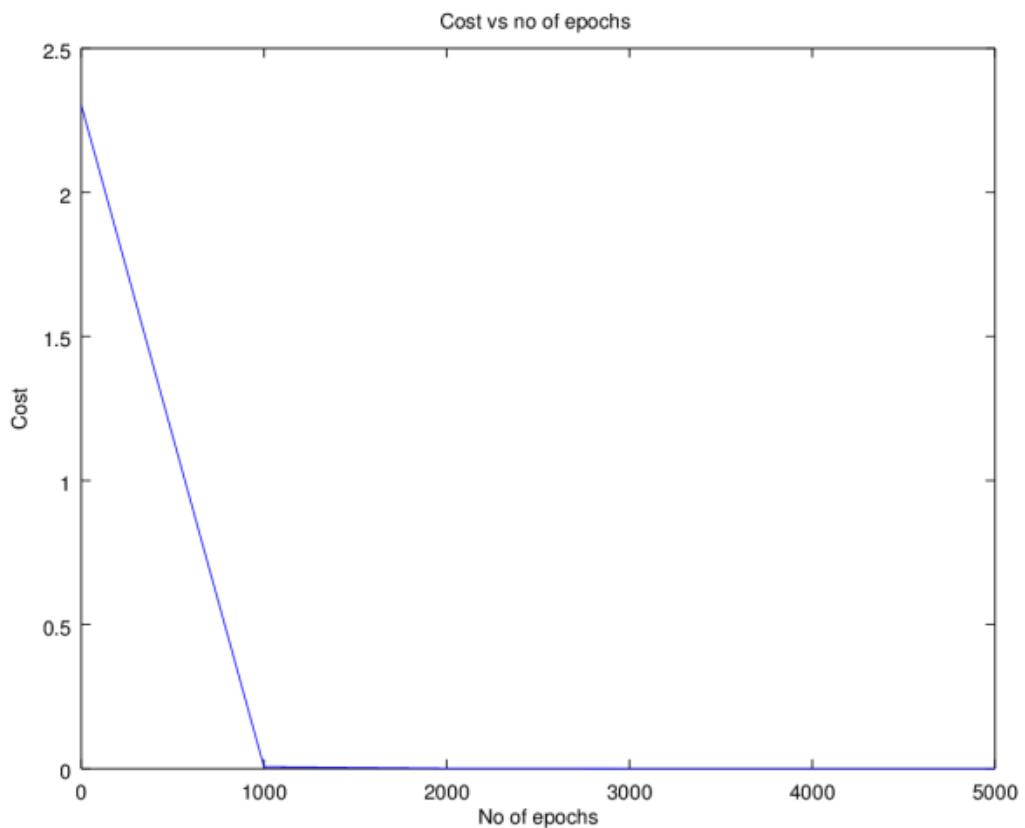
```

1 source("DL7functions.m")
2 #Load and read MNIST
3 load('./mnist/mnist.txt.gz');
4 #Create a random permutatation from 1024
5 permutation = randperm(1024);
6 disp(length(permutation));
7
8 # Use this 1024 as the mini-batch
9 X=trainX(permutation,:);
10 Y=trainY(permutation,:);
11
12 # Set layer dimensions
13 # 784 - number of input features (28 x28)
14 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
15 # 10 - 10 output classes with softmax activation unit at the output layer
16 layersDimensions=[784, 15, 9, 10];
17
18 # Execute the L-Layer Deep Network to perform SGD with regular gradient
19 # descent
20 # hidden Activation function - relu
21 # output activation function - softmax
22 # learning rate - 0.005
23 # optimizer="gd" # gradient descent
24 # mini_batch_size = 512
25 [weights biases costs]=L_Layer_DeepModel_SGD(X', Y', layersDimensions,
26     hiddenActivationFunc='relu',
27     outputActivationFunc="softmax",
28     learningRate = 0.005,
```

```

29      lrDecay=true,
30      decayRate=1,
31      lambda=0,
32      keep_prob=1,
33      optimizer="gd",
34      beta=0.9,
35      beta1=0.9,
36      beta2=0.999,
37      epsilon=10^-8,
38      mini_batch_size = 512,
39      num_epochs = 5000);
40
41 #Plot cost vs number epochs
42 plotCostVsEpochs(5000,costs);

```



2.1. Stochastic Gradient Descent with Learning rate decay

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

with the number of epochs and becomes smaller and smaller, so that gradient descent can take smaller steps towards the minima.

There are several techniques employed in learning rate decay

- a) Exponential decay: $\alpha = \text{decayRate}^{\text{epochNum}} * \alpha_0$
- b) $1/t$ decay: $\alpha = \frac{\alpha_0}{1+\text{decayRate}*\text{epochNum}}$
- c) $\alpha = \frac{\text{decayRate}}{\sqrt{(\text{epochNum})}} * \alpha_0$

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

```
1 if lrDecay == True:
2     #decay the learning rate exponentially
3     learningRate = np.power(decayRate,(num_epochs/1000)) * learningRate
```

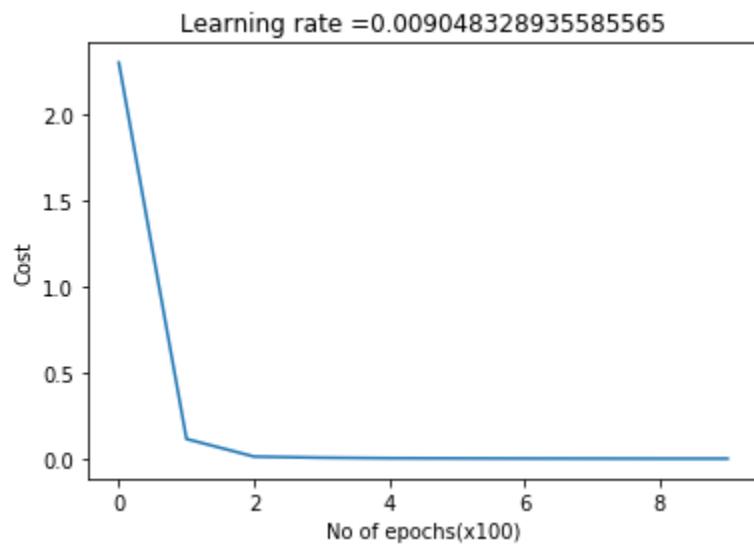
2.1a. Stochastic Gradient Descent with Learning rate decay – Python

```
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6 import sklearn
7 import sklearn.datasets
8 exec(open("DLfunctions7.py").read())
9 exec(open("load_mnist.py").read())
10
11 # Read the MNIST data
12 training=list(read(dataset='training',path=".\\mnist"))
13 test=list(read(dataset='testing',path=".\\mnist"))
14 lbls=[]
15 pxls=[]
16 for i in range(60000):
17     l,p=training[i]
18     lbls.append(l)
19     pxls.append(p)
20 labels= np.array(lbls)
21 pixels=np.array(pxls)
22 y=labels.reshape(-1,1)
23 X=pixels.reshape(pixels.shape[0],-1)
24 X1=X.T
25 Y1=y.T
26
27 # Create a list of random numbers of 1024
28 permutation = list(np.random.permutation(2**10))
```

```

30 # Subset 1024 from the data
31 X2 = X1[, permutation]
32 Y2 = Y1[, permutation].reshape((1,2**10))
33
34 # Set layer dimensions
35 # 784 - number of input features (28 x28)
36 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
37 # 10 - 10 output classes with softmax activation unit at the output layer
38 layersDimensions=[784, 15,9,10]
39
40 # Execute the L-layer Deep Learning network using SGD with learning rate
41 decay
42 # hidden Activation function - relu
43 # output activation function - softmax
44 # learning rate - 0.01
45 # lrDecay=True
46 # decayRate=0.9999
47 # optimizer="gd" # gradient descent
48 # mini_batch_size = 512
49 parameters = L_Layer_DeepModel_SGD(X2, Y2, layersDimensions,
50 hiddenActivationFunc='relu',
51 outputActivationFunc="softmax",
52 learningRate = 0.01 , lrDecay=True, decayRate=0.9999,
53 optimizer="gd", mini_batch_size =512, num_epochs = 1000, print_cost =
54 True,figure="fig2.png")

```



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

```

1 source("mnist.R")
2 source("DLfunctions7.R")
3
4 # Read and load MNIST
5 load_mnist()
6 x <- t(train$x)
7 x <- x[,1:60000]
8 y <-train$y
9 y1 <- y[1:60000]
10 y2 <- as.matrix(y1)

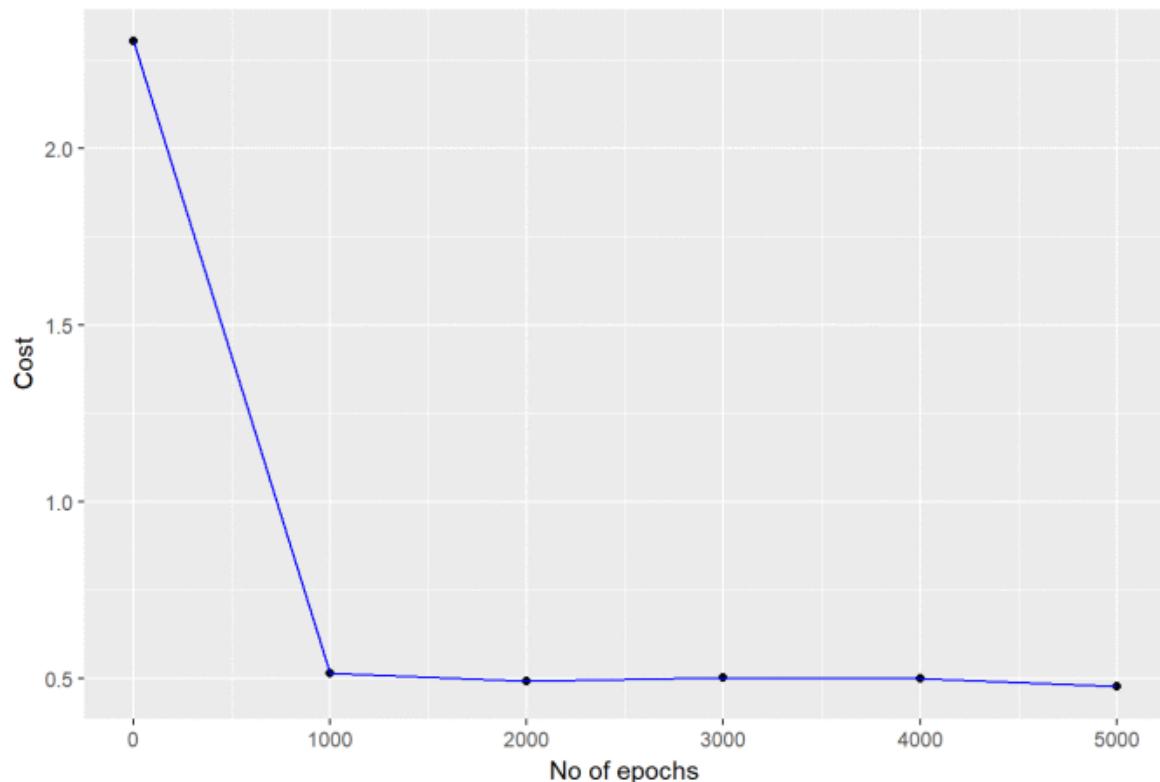
```

```

11 Y=t(y2)
12
13 # Subset 1024 random samples from MNIST
14 permutation = c(sample(2^10))
15
16 # Randomly shuffle the training data
17 X1 = X[, permutation]
18 y1 = Y[1, permutation]
19 y2 <- as.matrix(y1)
20 Y1=t(y2)
21
22 # Set layer dimensions
23 # 784 - number of input features (28 x28)
24 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
25 # 10 - 10 output classes with softmax activation unit at the output layer
26 layersDimensions=c(784, 15, 9, 10)
27
28 # Execute the L-layer Deep Learning network using SGD with learning rate
29 decay
30 # hidden Activation function - tanh
31 # output activation function - softmax
32 # learning rate - 0.05
33 # lrDecay=TRUE
34 # decayRate=0.9999
35 # optimizer="gd" # gradient descent
36 # mini_batch_size = 512
37 retvalsSGD= L_Layer_DeepModel_SGD(X1, Y1, layersDimensions,
38                                         hiddenActivationFunc='tanh',
39                                         outputActivationFunc="softmax",
40                                         learningRate = 0.05,
41                                         lrDecay=TRUE,
42                                         decayRate=0.9999,
43                                         optimizer="gd",
44                                         mini_batch_size = 512,
45                                         num_epochs = 5000,
46                                         print_cost = True)
47
48 #Plot the cost vs number of epochs
49 iterations <- seq(0,5000,1000)
50 costs=retvalsSGD$costs
51 df=data.frame(iterations,costs)
52 ggplot(df,aes(x=iterations,y=costs)) + geom_point() + geom_line(color="blue")
53 + ggttitle("Costs vs number of epochs") + xlab("No of epochs") + ylab("Cost")

```

Costs vs number of epochs



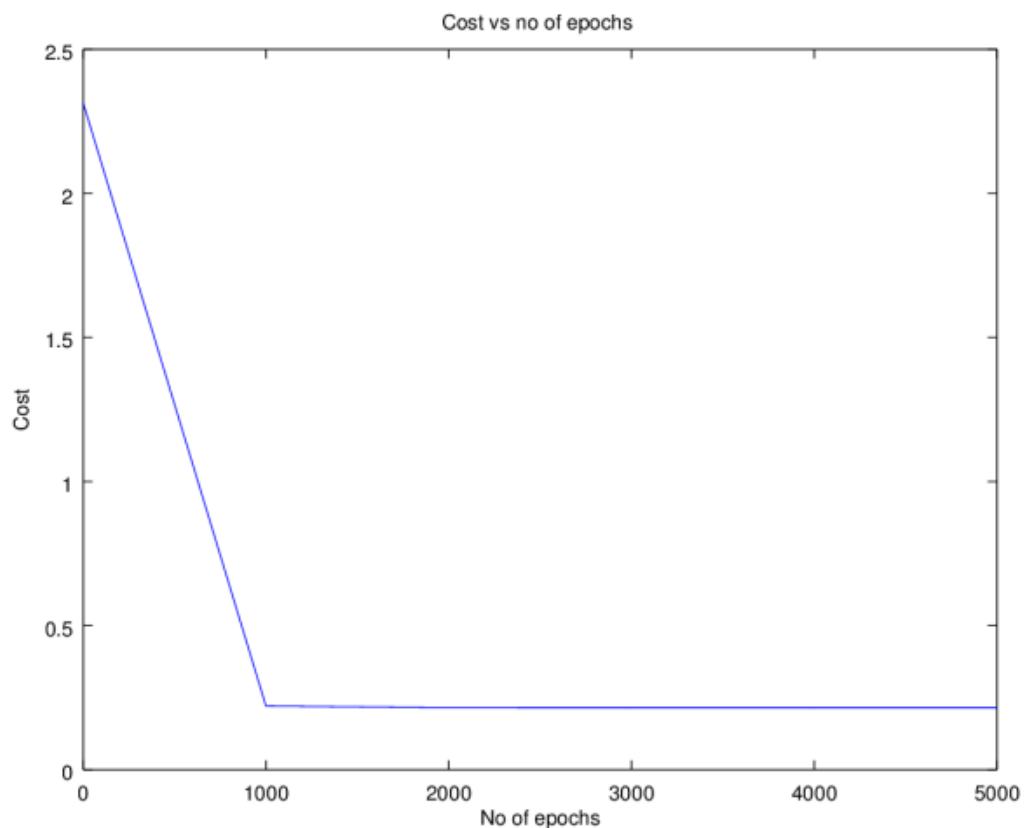
2.1c. Stochastic Gradient Descent with Learning rate decay – Octave

```
1 source("DL7functions.m")
2 #Load and read MNIST
3 load('./mnist/mnist.txt.gz');
4
5 #Create a random permutatation from 1024
6 permutation = randperm(1024);
7 disp(length(permutation));
8
9 # Use this 1024 as the batch
10 X=trainX(permutation,:);
11 Y=trainY(permutation,:);
12
13 # Set layer dimensions
14 # 784 - number of input features (28 x28)
15 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
16 # 10 - 10 output classes with softmax activation unit at the output layer
17 layersDimensions=[784, 15, 9, 10];
18
19 # Execute the L-layer Deep Learning network using SGD with learning rate
20 decay
21 # hidden Activation function - relu
22 # output activation function - softmax
23 # learning rate - 0.01
24 # lrDecay=true
25 # decayRate=0.999
```

```

26 # optimizer="gd" # gradient descent
27 # mini_batch_size = 512
28 [weights biases costs]=L_Layer_DeepModel_SGD(X', Y', layersDimensions,
29     hiddenActivationFunc='relu',
30     outputActivationFunc="softmax",
31     learningRate = 0.01,
32     lrDecay=true,
33     decayRate=0.999,
34     lambd=0,
35     keep_prob=1,
36     optimizer="gd",
37     beta=0.9,
38     beta1=0.9,
39     beta2=0.999,
40     epsilon=10^-8,
41     mini_batch_size = 512,
42     num_epochs = 5000);
43
44 # Plot cost vs number of epochs
45 plotCostVsEpochs(5000,costs)

```



3.1. Stochastic Gradient Descent with Momentum

Stochastic Gradient Descent with Momentum uses the exponentially weighted average method discussed above and hence moves faster into the ravine than across it. The equations are

$$\begin{aligned}v_{dW}^l &= \beta v_{dW}^{l-1} + (1 - \beta) dW^l \\v_{db}^l &= \beta v_{db}^{l-1} + (1 - \beta) db^l \\W^l &= W^l - \alpha v_{dW}^l \\b^l &= b^l - \alpha v_{db}^l\end{aligned}$$

where

v_{dW} and v_{db} are the momentum terms which are exponentially weighted with the corresponding gradients ‘dW’ and ‘db’ at the corresponding layer ‘l’. The code snippet for Stochastic Gradient Descent with momentum in R is shown below

```

1 # Perform Gradient Descent with momentum
2 # Input : weights and biases
3 #           : beta
4 #           : gradients
5 #           : learning rate
6 #           : outputActivationFunc - Activation function at hidden layer
7 sigmoid/softmax
8 #output : Updated weights after 1 iteration
9 gradientDescentWithMomentum <- function(parameters, gradients,v, beta,
10 learningRate,outputActivationFunc="sigmoid"){
11
12     L = length(parameters)/2 # number of layers in the neural network
13     # Update rule for each parameter. Use a for loop.
14     for(l in 1:(L-1)){
15
16         # Compute velocities
17         # v['dwk'] = beta *v['dwk'] + (1-beta)*dwk
18         v[[paste("dw",l, sep="")]] = beta*v[[paste("dw",l, sep="")]] +
19             (1-beta) * gradients[[paste('dw',l,sep="")]]
20         v[[paste("db",l, sep="")]] = beta*v[[paste("db",l, sep="")]] +
21             (1-beta) * gradients[[paste('db',l,sep="")]]
22         #Update parameters with velocities
23         parameters[[paste("w",l,sep="")]] = parameters[[paste("w",l,sep="")]]
24
25         learningRate* v[[paste("dw",l, sep="")]]
26         parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]]
27
28         learningRate* v[[paste("db",l, sep="")]]
29     }
30
31     # Compute for the Lth layer if output activation is sigmoid
32     if(outputActivationFunc=="sigmoid"){
33         v[[paste("dw",L, sep="")]] = beta*v[[paste("dw",L, sep="")]] +
34             (1-beta) * gradients[[paste('dw',L,sep="")]]
35         v[[paste("db",L, sep="")]] = beta*v[[paste("db",L, sep="")]] +
36             (1-beta) * gradients[[paste('db',L,sep="")]]
37
38         parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
39
40         learningRate* v[[paste("dw",l, sep="")]]
41         parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
42
43         learningRate* v[[paste("db",l, sep="")]]
44
45     }else if (outputActivationFunc=="softmax"){ #If output activation is
46     Softmax
47         v[[paste("dw",L, sep="")]] = beta*v[[paste("dw",L, sep="")]] +
48             (1-beta) * t(gradients[[paste('dw',L,sep="")]])
49         v[[paste("db",L, sep="")]] = beta*v[[paste("db",L, sep="")]] +
```

```

50      (1-beta) * t(gradients[[paste('db',L,sep="")]])]
51  parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
52  -
53      learningRate* t(gradients[[paste("dw",L,sep="")]])]
54  parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
55  -
56      learningRate* t(gradients[[paste("db",L,sep="")]])]
57  }
58  return(parameters)
59 }
60

```

3.1a. Stochastic Gradient Descent with Momentum- Python

```

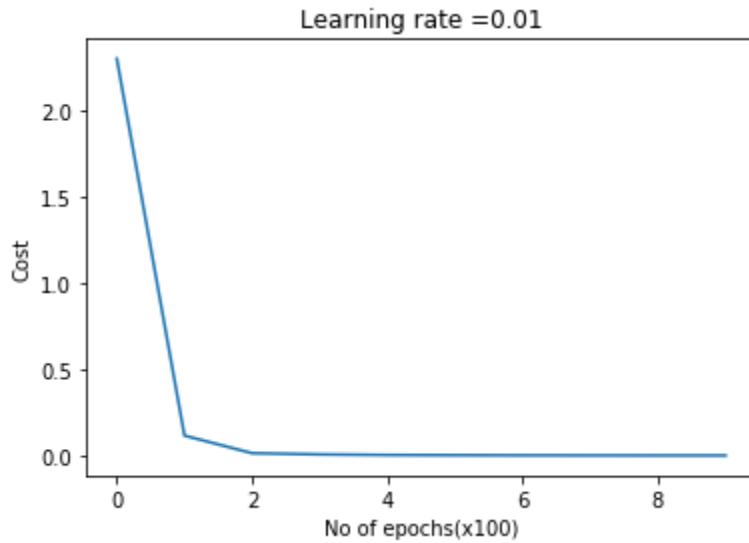
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6 import sklearn
7 import sklearn.datasets
8
9 # Read and load data and labels
10 exec(open("DLfunctions7.py").read())
11 exec(open("load_mnist.py").read())
12 training=list(read(dataset='training',path=".\\mnist"))
13 test=list(read(dataset='testing',path=".\\mnist"))
14 lbls=[]
15 pxls=[]
16 for i in range(60000):
17     l,p=training[i]
18     lbls.append(l)
19     pxls.append(p)
20 labels= np.array(lbls)
21 pixels=np.array(pxls)
22 y=labels.reshape(-1,1)
23 X=pixels.reshape(pixels.shape[0],-1)
24 X1=X.T
25 Y1=y.T
26
27 # Create a list of random numbers of 1024
28 permutation = list(np.random.permutation(2**10))
29 # Subset 1024 random samples from the data
30 X2 = X1[:, permutation]
31 Y2 = Y1[:, permutation].reshape((1,2**10))
32
33 # Set layer dimensions
34 # 784 - number of input features (28 x28)
35 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
36 # 10 - 10 output classes with softmax activation unit at the output layer
37 layersDimensions=[784, 15,9,10]
38
39 # Execute the L-Layer Deep Learning network using SGD with momentum
40 # hidden Activation function - relu
41 # output activation function - softmax
42 # learning rate - 0.01
43 # optimizer="momentum"
44 # beta =0.9

```

```

45 # mini_batch_size = 512
46 parameters = L_Layer_DeepModel_SGD(X2, Y2, layersDimensions,
47 hiddenActivationFunc='relu',
48 outputActivationFunc="softmax",learningRate = 0.01 ,
49 optimizer="momentum", beta=0.9,
50 mini_batch_size =512, num_epochs = 1000, print_cost = True,figure="fig3.png")

```



3.1b. Stochastic Gradient Descent with Momentum- R

```

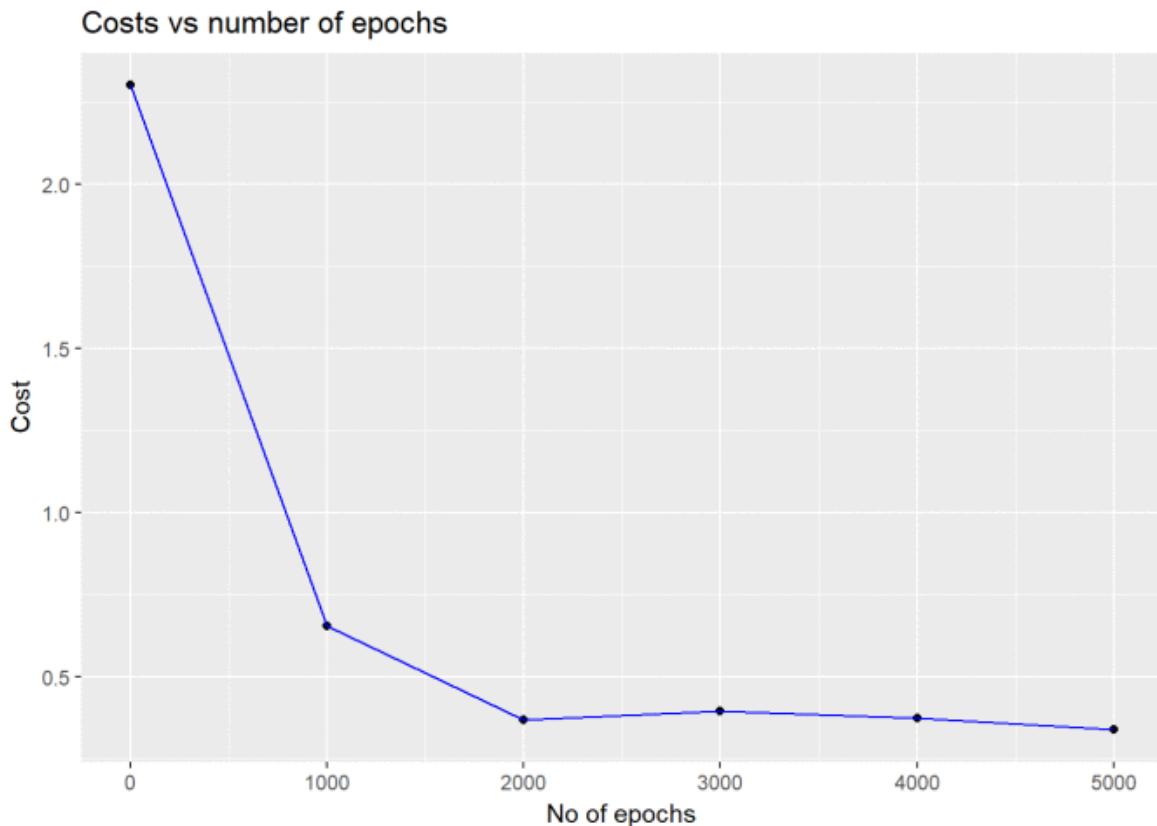
1 source("mnist.R")
2 source("DLfunctions7.R")
3 load_mnist()
4 x <- t(train$x)
5 X <- x[,1:60000]
6 y <-train$y
7 y1 <- y[1:60000]
8 y2 <- as.matrix(y1)
9 Y=t(y2)
10
11 # Subset 1024 random samples from MNIST
12 permutation = c(sample(2^10))
13
14 # Randomly shuffle the training data
15 X1 = X[, permutation]
16 y1 = Y[1, permutation]
17 y2 <- as.matrix(y1)
18 Y1=t(y2)
19
20 # Set layer dimensions
21 # 784 - number of input features (28 x28)
22 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
23 # 10 - 10 output classes with softmax activation unit at the output layer
24 layersDimensions=c(784, 15,9, 10)
25
26 # Execute the L-Layer Deep Learning network using SGD with momentum
27 # hidden Activation function - tanh
28 # output activation function - softmax
29 # learning rate - 0.05
30 # optimizer="momentum"
31 # beta =0.9

```

```

32 # mini_batch_size = 512
33 retvalsSGD= L_Layer_DeepModel_SGD(X1, Y1, layersDimensions,
34                                     hiddenActivationFunc='tanh',
35                                     outputActivationFunc="softmax",
36                                     learningRate = 0.05,
37                                     optimizer="momentum",
38                                     beta=0.9,
39                                     mini_batch_size = 512,
40                                     num_epochs = 5000,
41                                     print_cost = True)
42
43 #Plot the cost vs number of epochs
44 iterations <- seq(0,5000,1000)
45 costs=retvalsSGD$costs
46 df=data.frame(iterations,costs)
47 ggplot(df,aes(x=iterations,y=costs)) + geom_point() + geom_line(color="blue")
48 + ggttitle("Costs vs number of epochs") + xlab("No of epochs") + ylab("Cost")

```



3.1c. Stochastic Gradient Descent with Momentum- Octave

```

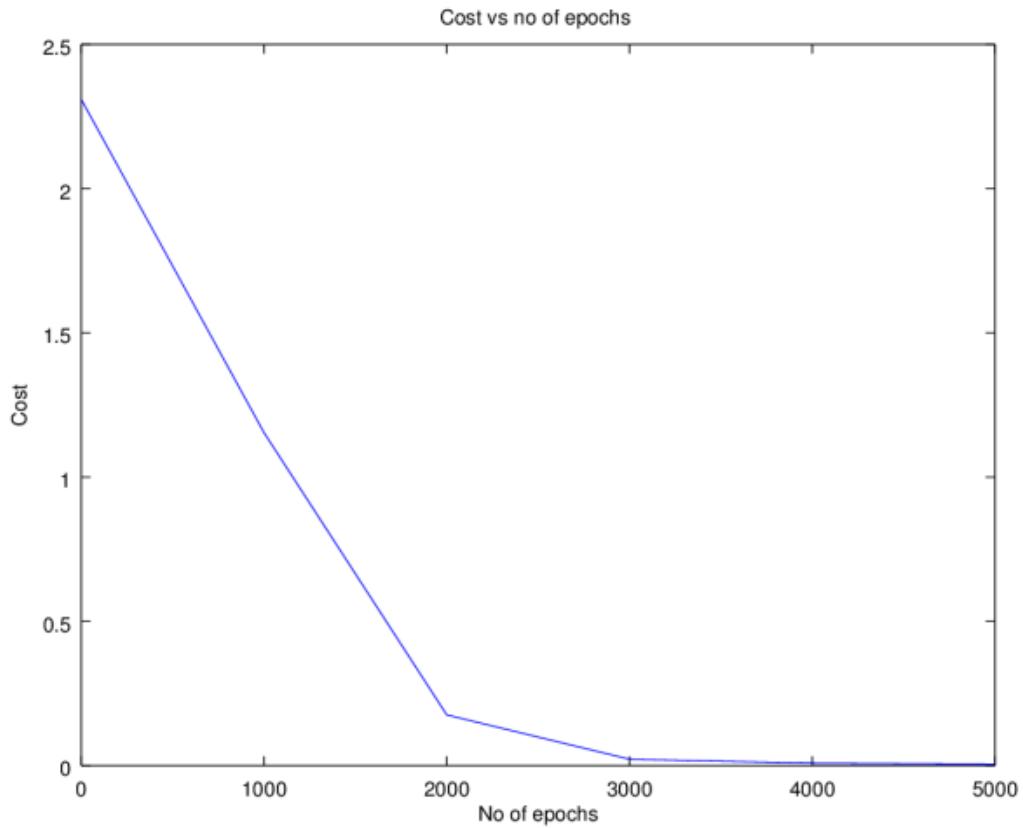
1 source("DL7functions.m")
2
3 #Load and read MNIST
4 load('./mnist/mnist.txt.gz');
5
6 #Create a random permutatation from 60K
7 permutation = randperm(1024);
8 disp(length(permutation));
9
10 # Use this 1024 as the batch

```

```

11 X=trainX(permuation,:);
12 Y=trainY(permuation,:);
13
14 # Set layer dimensions
15 # 784 - number of input features (28 x28)
16 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
17 # 10 - 10 output classes with softmax activation unit at the output layer
18 layersDimensions=[784, 15, 9, 10];
19
20 # Execute the L-Layer Deep Learning network using SGD with momentum
21 # hidden Activation function - relu
22 # output activation function - softmax
23 # learning rate - 0.01
24 # optimizer="momentum"
25 # beta =0.9
26 # mini_batch_size = 512
27 [weights biases costs]=L_Layer_DeepModel_SGD(X', Y', layersDimensions,
28     hiddenActivationFunc='relu',
29     outputActivationFunc="softmax",
30     learningRate = 0.01,
31     lrDecay=false,
32     decayRate=1,
33     lambd=0,
34     keep_prob=1,
35     optimizer="momentum",
36     beta=0.9,
37     beta1=0.9,
38     beta2=0.999,
39     epsilon=10^-8,
40     mini_batch_size = 512,
41     num_epochs = 5000);
42
43 #Plot the cost vs number of epochs
44 plotCostVsEpochs(5000,costs)

```



4.1. Stochastic Gradient Descent with RMSProp

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

The equations are

$$\begin{aligned}s_{dW}^l &= \beta_1 s_{dW}^l + (1 - \beta_1)(dW^l)^2 \\ s_{db}^l &= \beta_1 s_{db}^l + (1 - \beta_1)(db^l)^2 \\ W^l &= W^l - \frac{\alpha s_{dW}^l}{\sqrt{(s_{dW}^l + \epsilon)}} \\ b^l &= b^l - \frac{\alpha s_{db}^l}{\sqrt{(s_{db}^l + \epsilon)}}\end{aligned}$$

where s_{dW} and s_{db} are the RMSProp terms which are exponentially weighted with the corresponding gradients 'dW' and 'db' at the corresponding layer 'l'

The code snippet in Octave is shown below

```

1 # Update parameters with RMSProp
2 # Input : parameters
3 #           : gradients
4 #           : s
```

```

5      : beta
6      : learningRate
7 #output : Updated parameters RMSProp
8
9 function [weights biases] = gradientDescentwithRMSProp(weights,
10    biases,gradsDW,gradsDB, sdW, sdB, beta1, epsilon,
11    learningRate,outputActivationFunc="sigmoid")
12    L = size(weights)(2); # number of layers in the neural network
13
14    # Update rule for each parameter.
15    for l=1:(L-1)
16        sdW{l} = beta1*sdW{l} + (1 -beta1) .* gradsDW{l} .* gradsDW{l};
17        sdB{l} = beta1*sdB{l} + (1 -beta1) .* gradsDB{l} .* gradsDB{l};
18        weights{l} = weights{l} - learningRate* gradsDW{l} ./ sqrt(sdW{l} +
19        epsilon);
20        biases{l} = biases{l} - learningRate* gradsDB{l} ./ sqrt(sdB{l} +
21        epsilon);
22    endfor
23
24    #Update for Lth layer is output is sigmoid
25    if (strcmp(outputActivationFunc,"sigmoid"))
26        sdW{L} = beta1*sdW{L} + (1 -beta1) .* gradsDW{L} .* gradsDW{L};
27        sdB{L} = beta1*sdB{L} + (1 -beta1) .* gradsDB{L} .* gradsDB{L};
28        weights{L} = weights{L} -learningRate* gradsDW{L} ./ sqrt(sdW{L} +
29        epsilon);
30        biases{L} = biases{L} -learningRate* gradsDB{L} ./ sqrt(sdB{L} +
31        epsilon);
32    elseif (strcmp(outputActivationFunc,"softmax")) #Update for Lth layer is
33    output is softmax
34        sdW{L} = beta1*sdW{L} + (1 -beta1) .* gradsDW{L}' .* gradsDW{L}';
35        sdB{L} = beta1*sdB{L} + (1 -beta1) .* gradsDB{L}' .* gradsDB{L}';
36        weights{L} = weights{L} -learningRate* gradsDW{L}' ./ sqrt(sdW{L} +
37        epsilon);
38        biases{L} = biases{L} -learningRate* gradsDB{L}' ./ sqrt(sdB{L} +
39        epsilon);
40    endif
41 end

```

4.1a. Stochastic Gradient Descent with RMSProp – Python

```

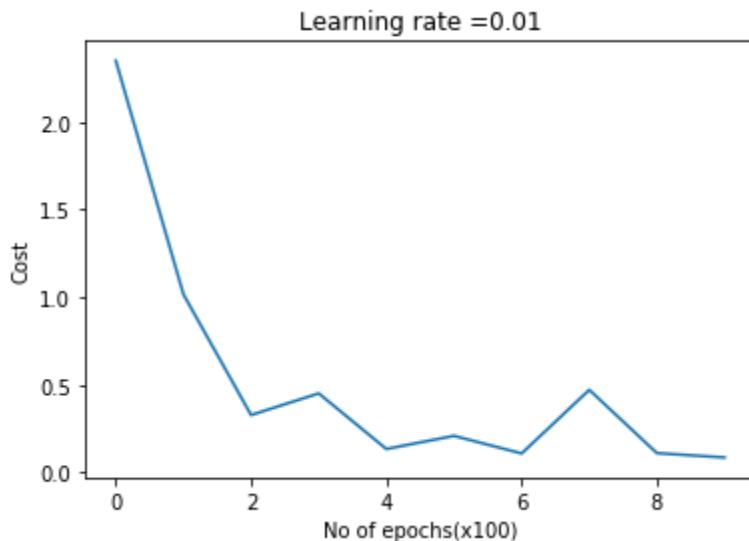
1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6 import sklearn
7 import sklearn.datasets
8 exec(open("DLfunctions7.py").read())
9 exec(open("load_mnist.py").read())
10
11 # Read and load MNIST
12 training=list(read(dataset='training',path=".\\mnist"))
13 test=list(read(dataset='testing',path=".\\mnist"))
14 lbls=[]
15 pxls=[]
16 for i in range(60000):
17     l,p=training[i]
18     lbls.append(l)

```

```

19     pxls.append(p)
20 labels= np.array(lbls)
21 pixels=np.array(pxls)
22 y=labels.reshape(-1,1)
23 X=pixels.reshape(pixels.shape[0],-1)
24 X1=X.T
25 Y1=y.T
26
27 print("X1=",X1.shape)
28 print("Y1=",Y1.shape)
29
30 # Create a list of random numbers of 1024
31 permutation = list(np.random.permutation(2**10))
32 # Subset 1024 from the data
33 X2 = X1[:, permutation]
34 Y2 = Y1[:, permutation].reshape((1,2**10))
35
36 # Set layer dimensions
37 # 784 - number of input features (28 x28)
38 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
39 # 10 - 10 output classes with softmax activation unit at the output layer
40 layersDimensions=[784, 15, 9, 10]
41
42 # Execute the L-layer Deep learning network using SGD with RMSProp
43 # hidden Activation function - relu
44 # output activation function - softmax
45 # learning rate - 0.01
46 # optimizer="rmsprop"
47 # beta1 =0.7
48 # epsilon=1e-8
49 # mini_batch_size = 512
50 parameters = L_Layer_DeepModel_SGD(X2, Y2, layersDimensions,
51 hiddenActivationFunc='relu',
52 outputActivationFunc="softmax",learningRate = 0.01 ,
53 optimizer="rmsprop", beta1=0.7, epsilon=1e-8,
54 mini_batch_size =512, num_epochs = 1000, print_cost = True,figure="fig4.png")

```



4.1b. Stochastic Gradient Descent with RMSProp – R

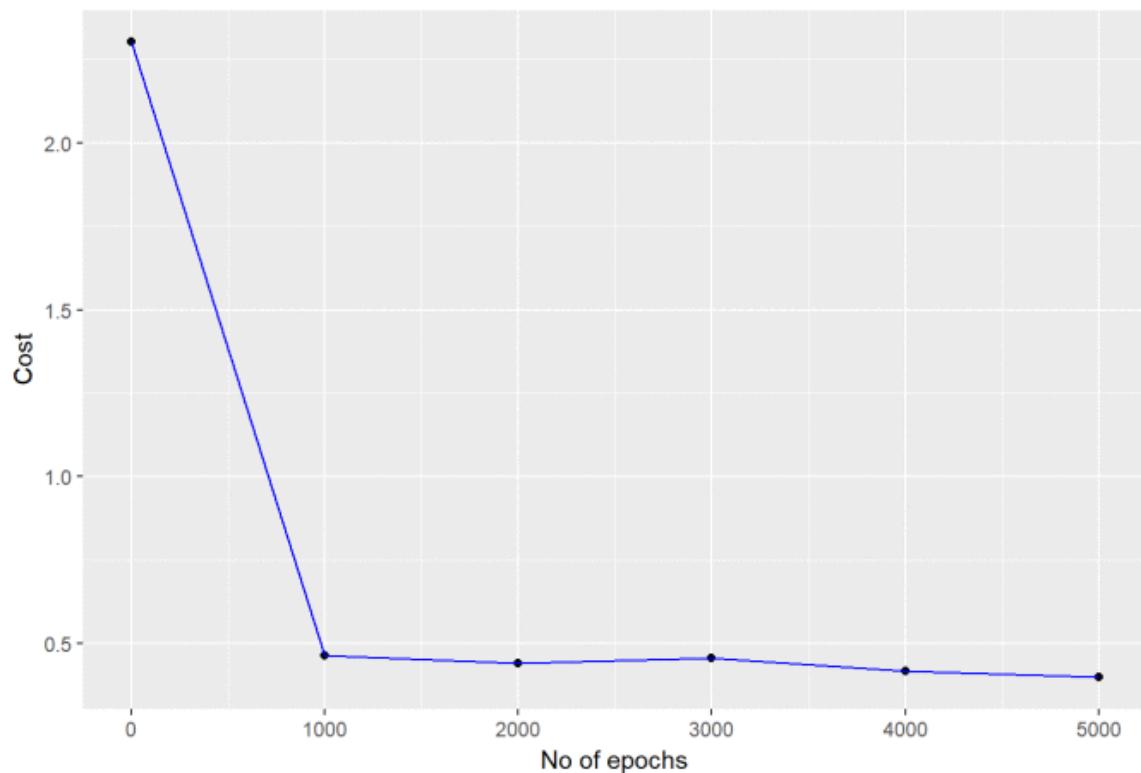
```
1 source("mnist.R")
```

```

2 source("DLfunctions7.R")
3 load_mnist()
4 x <- t(train$x)
5 x <- x[,1:60000]
6 y <- train$y
7 y1 <- y[1:60000]
8 y2 <- as.matrix(y1)
9 Y=t(y2)
10
11 # Subset 1024 random samples from MNIST
12 permutation = c(sample(2^10))
13
14 # Randomly shuffle the training data
15 x1 = x[, permutation]
16 y1 = Y[1, permutation]
17 y2 <- as.matrix(y1)
18 Y1=t(y2)
19
20 # Set layer dimensions
21 # 784 - number of input features (28 x28)
22 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
23 # 10 - 10 output classes with softmax activation unit at the output layer
24 layersDimensions=c(784, 15, 9, 10)
25
26 # Execute the L-layer Deep learning network using SGD with RMSProp
27 # hidden Activation function - tanh
28 # output activation function - softmax
29 # learning rate - 0.001
30 # optimizer="rmsprop"
31 # beta1 =0.9
32 # epsilon=1e-8
33 # mini_batch_size = 512
34 retvalsSGD= L_Layer_DeepModel_SGD(x1, Y1, layersDimensions,
35                                     hiddenActivationFunc='tanh',
36                                     outputActivationFunc="softmax",
37                                     learningRate = 0.001,
38                                     optimizer="rmsprop",
39                                     beta1=0.9,
40                                     epsilon=10^-8,
41                                     mini_batch_size = 512,
42                                     num_epochs = 5000 ,
43                                     print_cost = True)
44
45 #Plot the cost vs number of epochs
46 iterations <- seq(0,5000,1000)
47 costs=retvalsSGD$costs
48 df=data.frame(iterations,costs)
49 ggplot(df,aes(x=iterations,y=costs)) + geom_point() + geom_line(color="blue")
50 + ggttitle("Costs vs number of epochs") + xlab("No of epochs") + ylab("Cost")

```

Costs vs number of epochs



4.1c. Stochastic Gradient Descent with RMSProp – Octave

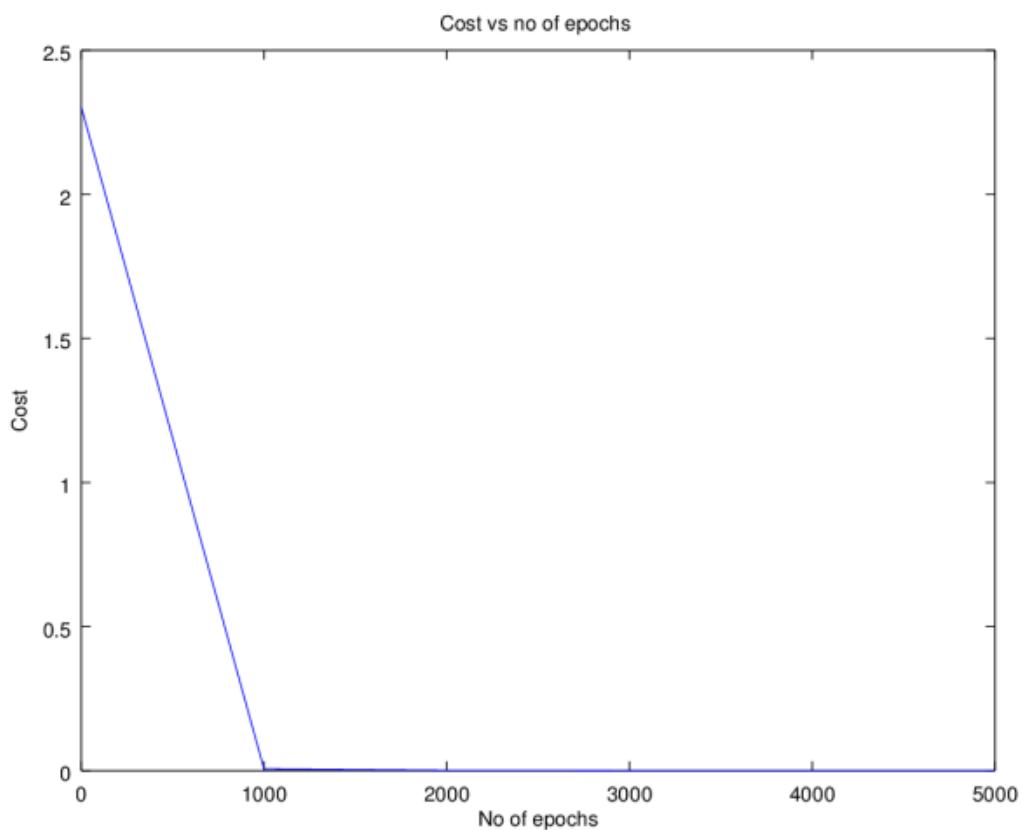
```

1 source("DL7functions.m")
2 load('./mnist/mnist.txt.gz');
3 #Create a random permutatation from 1024
4 permutation = randperm(1024);
5
6 # Use this 1024 as the batch
7 X=trainX(permutation,:);
8 Y=trainY(permutation,:);
9
10 # Set layer dimensions
11 # Set layer dimensions
12 # 784 - number of input features (28 x28)
13 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
14 # 10 - 10 output classes with softmax activation unit at the output layer
15 layersDimensions=[784, 15, 9, 10];
16
17 # Execute the L-layer Deep learning network using SGD with RMSProp
18 # hidden Activation function - relu
19 # output activation function - softmax
20 # learning rate - 0.005
21 # optimizer="rmsprop"
22 # beta1 =0.9
23 # epsilon=1
24 # mini_batch_size = 512
25 [weights biases costs]=L_Layer_DeepModel_SGD(X', Y', layersDimensions,
26     hiddenActivationFunc='relu',
27     outputActivationFunc="softmax",
28     learningRate = 0.005,
```

```

29      lrDecay=false,
30      decayRate=1,
31      lambd=0,
32      keep_prob=1,
33      optimizer="rmsprop",
34      beta=0.9,
35      beta1=0.9,
36      beta2=0.999,
37      epsilon=1,
38      mini_batch_size = 512,
39      num_epochs = 5000);
40
41 #Plot cost vs number of epochs
42 plotCostVsEpochs(5000,costs)

```



5.1. Stochastic Gradient Descent with Adam

Adaptive Moment Estimate is a combination of the momentum (1st moment) and RMSProp (2nd moment). The equations for Adam are below

The moving average for the 1st moment

$$v_{dW}^l = \beta_1 v_{dW}^{l-1} + (1 - \beta_1) dW^l$$

$$v_{db}^l = \beta_1 v_{db}^{l-1} + (1 - \beta_1) db^l$$

The bias corrections for the 1st moment

$$v_{Corrected}^l_{dW} = \frac{v_{dW}^l}{1 - \beta_1^t}$$

$$v_{Corrected}^l_{db} = \frac{v_{db}^l}{1 - \beta_1^t}$$

Similarly, the moving average for the 2nd moment- RMSProp

$$s_{dW}^l = \beta_2 s_{dW}^{l-1} + (1 - \beta_2) (dW^l)^2$$

$$s_{db}^l = \beta_2 s_{db}^{l-1} + (1 - \beta_2) (db^l)^2$$

The bias corrections for the 2nd moment

$$s_{Corrected}^l_{dW} = \frac{s_{dW}^l}{1 - \beta_2^t}$$

$$s_{Corrected}^l_{db} = \frac{s_{db}^l}{1 - \beta_2^t}$$

The Adam Gradient Descent is given by

$$W^l = W^l - \frac{\alpha v_{Corrected}^l_{dW}}{\sqrt{(s_{dW}^l + \epsilon)}}$$

$$b^l = b^l - \frac{\alpha v_{Corrected}^l_{db}}{\sqrt{(s_{db}^l + \epsilon)}}$$

The code snippet of Adam in R is included be

```

1 # Perform Gradient Descent with Adam
2 # Input : Weights and biases
3 #           : beta1
4 #           : epsilon
5 #           : gradients
6 #           : learning_rate
7 #           : outputActivationFunc - Activation function at hidden layer
8 # sigmoid/softmax
9 #output : Updated weights after 1 iteration
10 gradientDescentWithAdam <- function(parameters, gradients,v, s, t,
11                                     beta1=0.9, beta2=0.999, epsilon=10^-8,
12                                     learningRate=0.1,outputActivationFunc="sigmoid"){
13
14     L = length(parameters)/2 # number of layers in the neural network
15     v_corrected <- list()
16     s_corrected <- list()
17     # Update rule for each parameter.
18     for(l in 1:(L-1)){
19         # v['dwk'] = beta *v['dwk'] + (1-beta)*dwk
20         v[[paste("dw",l, sep="")]] = beta1*v[[paste("dw",l, sep="")]] +
21             (1-beta1) * gradients[[paste('dw',l,sep="")]]
22         v[[paste("db",l, sep="")]] = beta1*v[[paste("db",l, sep="")]] +
23             (1-beta1) * gradients[[paste('db',l,sep="")]]
24
25         # Compute bias-corrected first moment estimate.
26         v_corrected[[paste("dw",l, sep="")]] = v[[paste("dw",l, sep="")]]/(1-
27         beta1^t)
28         v_corrected[[paste("db",l, sep="")]] = v[[paste("db",l, sep="")]]/(1-
29         beta1^t)
```

```

30
31      # Element wise multiply of gradients
32      s[[paste("dw",1, sep="")]] = beta2*s[[paste("dw",1, sep="")]] +
33          (1-beta2) * gradients[[paste('dw',1,sep="")]] *
34 gradients[[paste('dw',1,sep="")]]
35      s[[paste("db",1, sep="")]] = beta2*s[[paste("db",1, sep="")]] +
36          (1-beta2) * gradients[[paste('db',1,sep="")]] *
37 gradients[[paste('db',1,sep="")]]
38
39      # Compute bias-corrected second moment estimate.
40      s_corrected[[paste("dw",1, sep="")]] = s[[paste("dw",1, sep="")]]/(1-
41 beta2^t)
42      s_corrected[[paste("db",1, sep="")]] = s[[paste("db",1, sep="")]]/(1-
43 beta2^t)
44
45      # Update parameters.
46      d1=sqrt(s_corrected[[paste("dw",1, sep="")]]+epsilon)
47      d2=sqrt(s_corrected[[paste("db",1, sep="")]]+epsilon)
48
49      parameters[[paste("w",1,sep="")]] = parameters[[paste("w",1,sep="")]] -
50
51          learningRate * v_corrected[[paste("dw",1, sep="")]]/d1
52      parameters[[paste("b",1,sep="")]] = parameters[[paste("b",1,sep="")]] -
53
54          learningRate*v_corrected[[paste("db",1, sep="")]]/d2
55  }
56  # Compute for the Lth layer for sigmoid activation
57  if(outputActivationFunc=="sigmoid"){
58      v[[paste("dw",L, sep="")]] = beta1*v[[paste("dw",L, sep="")]] +
59          (1-beta1) * gradients[[paste('dw',L,sep="")]]
60      v[[paste("db",L, sep="")]] = beta1*v[[paste("db",L, sep="")]] +
61          (1-beta1) * gradients[[paste('db',L,sep="")]]
62
63      # Compute bias-corrected first moment estimate.
64      v_corrected[[paste("dw",L, sep="")]] = v[[paste("dw",L, sep="")]]/(1-
65 beta1^t)
66      v_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-
67 beta1^t)
68
69      # Element wise multiply of gradients
70      s[[paste("dw",L, sep="")]] = beta2*s[[paste("dw",L, sep="")]] +
71          (1-beta2) * gradients[[paste('dw',L,sep="")]] *
72 gradients[[paste('dw',L,sep="")]]
73      s[[paste("db",L, sep="")]] = beta2*s[[paste("db",L, sep="")]] +
74          (1-beta2) * gradients[[paste('db',L,sep="")]] *
75 gradients[[paste('db',L,sep="")]]
76
77      # Compute bias-corrected second moment estimate.
78      s_corrected[[paste("dw",L, sep="")]] = s[[paste("dw",L, sep="")]]/(1-
79 beta2^t)
80      s_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-
81 beta2^t)
82
83      # Update parameters.
84      d1=sqrt(s_corrected[[paste("dw",L, sep="")]]+epsilon)
85      d2=sqrt(s_corrected[[paste("db",L, sep="")]]+epsilon)
86
87      parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]] -
88
89          learningRate * v_corrected[[paste("dw",L, sep="")]]/d1
90      parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] -
91
92          learningRate*v_corrected[[paste("db",L, sep="")]]/d2
93

```

```

94     }else if (outputActivationFunc=="softmax"){ #Compute for the Lth layer
95     for softmax activation
96         v[[paste("dw",L, sep="")]] = beta1*v[[paste("dw",L, sep="")]] + 
97             (1-beta1) * t(gradients[[paste('dw',L,sep="")]]) 
98         v[[paste("db",L, sep="")]] = beta1*v[[paste("db",L, sep="")]] + 
99             (1-beta1) * t(gradients[[paste('db',L,sep="")]]) 
100
101        # Compute bias-corrected first moment estimate.
102        v_corrected[[paste("dw",L, sep="")]] = v[[paste("dw",L, sep="")]]/(1-
103 beta1^t)
104        v_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-
105 beta1^t)
106
107        # Element wise multiply of gradients
108        s[[paste("dw",L, sep="")]] = beta2*s[[paste("dw",L, sep="")]] + 
109            (1-beta2) * t(gradients[[paste('dw',L,sep="")]]) *
110            t(gradients[[paste('dw',L,sep="")]])
111        s[[paste("db",L, sep="")]] = beta2*s[[paste("db",L, sep="")]] + 
112            (1-beta2) * t(gradients[[paste('db',L,sep="")]]) *
113            t(gradients[[paste('db',L,sep="")]])
114
115        # Compute bias-corrected second moment estimate.
116        s_corrected[[paste("dw",L, sep="")]] = s[[paste("dw",L, sep="")]]/(1-
117 beta2^t)
118        s_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-
119 beta2^t)
120
121        # Update parameters.
122        d1=sqrt(s_corrected[[paste("dw",L, sep="")]]+epsilon)
123        d2=sqrt(s_corrected[[paste("db",L, sep="")]]+epsilon)
124
125        parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]] 
126        -
127            learningRate * v_corrected[[paste("dw",L, sep="")]]/d1
128        parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] 
129        -
130            learningRate*v_corrected[[paste("db",L, sep="")]]/d2
131    }
132    return(parameters)
133 }
```

5.1a. Stochastic Gradient Descent with Adam – Python

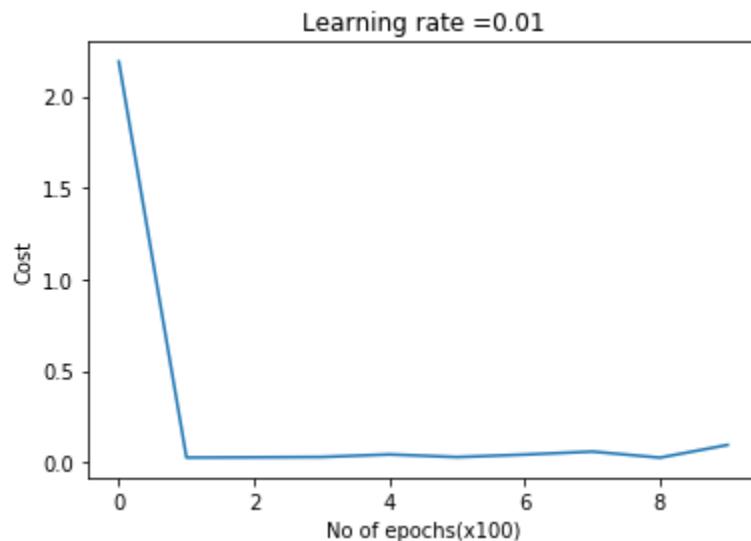
```

1 import numpy as np
2 import matplotlib
3 import matplotlib.pyplot as plt
4 import sklearn.linear_model
5 import pandas as pd
6 import sklearn
7 import sklearn.datasets
8
9 exec(open("DLfunctions7.py").read())
10
11 #Load MNIST
12 exec(open("load_mnist.py").read())
13 training=list(read(dataset='training',path=".\\mnist"))
14 test=list(read(dataset='testing',path=".\\mnist"))
15 lbls=[]
16 pxls=[]
```

```

17 print(len(training))
18 #for i in range(len(training)):
19 for i in range(60000):
20     l,p=training[i]
21     lbls.append(l)
22     pxls.append(p)
23 labels= np.array(lbls)
24 pixels=np.array(pxls)
25 y=labels.reshape(-1,1)
26 X=pixels.reshape(pixels.shape[0],-1)
27 X1=X.T
28 Y1=y.T
29
30
31 # Create a list of random numbers of 1024
32 permutation = list(np.random.permutation(2**10))
33 # Subset 1024 from the data
34 X2 = X1[:, permutation]
35 Y2 = Y1[:, permutation].reshape((1,2**10))
36
37 # Set layer dimensions
38 # 784 - number of input features (28 x28)
39 # 15, 9 - 2 hidden layers with 15, 9 hidden units respectively
40 # 10 - 10 output classes with softmax activation unit at the output layer
41 layersDimensions=[784, 15, 9, 10]
42
43 #Execute a L-layer Deep Learning network using SGD with Adam optimization
44 # hidden Activation function - relu
45 # output activation function - softmax
46 # learning rate - 0.01
47 # optimizer="adam"
48 # beta1 =0.9
49 # beta2=0.9
50 # epsilon=1e-8
51 # mini_batch_size = 512
52 parameters = L_Layer_DeepModel_SGD(X2, Y2, layersDimensions,
53 hiddenActivationFunc='relu',
54 outputActivationFunc="softmax",learningRate = 0.01 ,
55 optimizer="adam", beta1=0.9, beta2=0.9, epsilon = 1e-8,
56 mini_batch_size =512, num_epochs = 1000, print_cost = True,
57 figure="fig5.png")

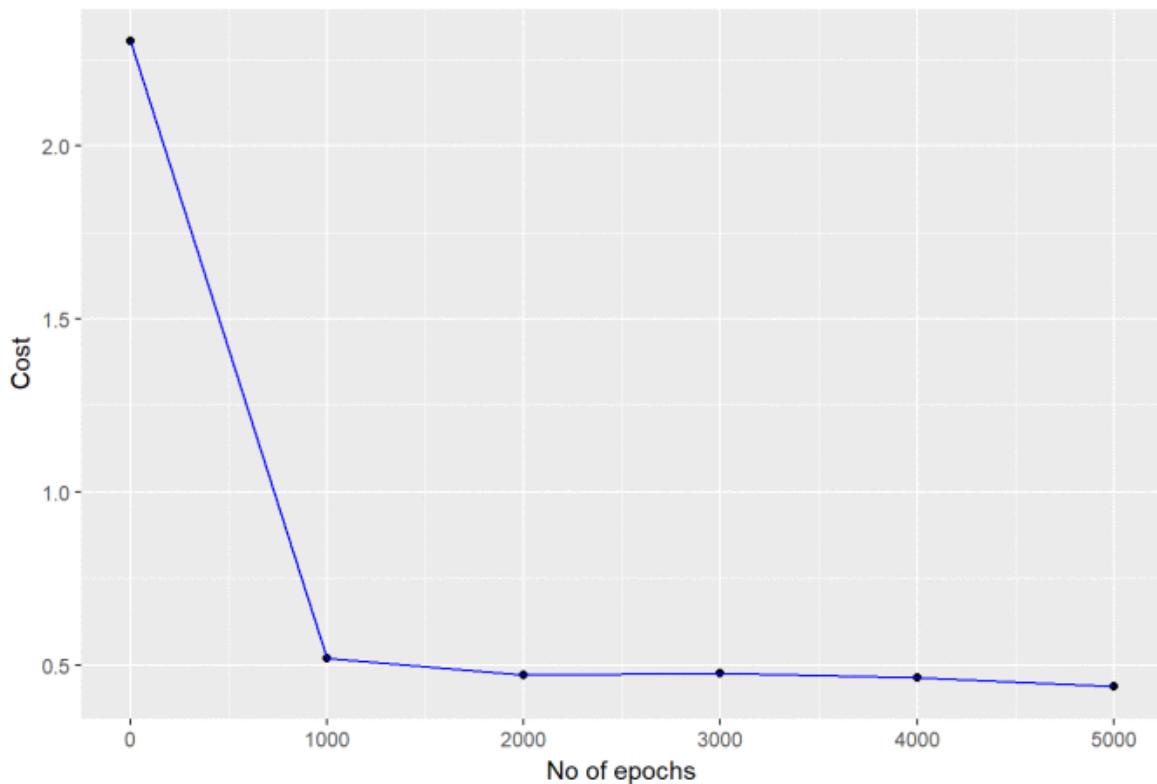
```



5.1b. Stochastic Gradient Descent with Adam – R

```
1 source("mnist.R")
2 source("DLfunctions7.R")
3 load_mnist()
4 x <- t(train$x)
5 x <- x[,1:60000]
6 y <- train$y
7 y1 <- y[1:60000]
8 y2 <- as.matrix(y1)
9 Y=t(y2)
10
11 # Subset 1024 random samples from MNIST
12 permutation = c(sample(2^10))
13 # Randomly shuffle the training data
14 X1 = X[, permutation]
15 y1 = Y[1, permutation]
16 y2 <- as.matrix(y1)
17 Y1=t(y2)
18
19 # Set layer dimensions
20 # 784 - number of input features (28 x28)
21 # 15, 9 - 2 hidden layers with 15,9 hidden units respectively
22 # 10 - 10 output classes with softmax activation unit at the output layer
23 layersDimensions=c(784, 15, 9, 10)
24
25 #Execute a L-layer Deep Learning network using SGD with Adam optimization
26 # hidden Activation function - tanh
27 # output activation function - softmax
28 # learning rate - 0.005
29 # optimizer="adam"
30 # beta1 =0.7
31 # beta2=0.9
32 # epsilon=1e-8
33 # mini_batch_size = 512
34 retvalsSGD= L_Layer_DeepModel_SGD(x1, Y1, layersDimensions,
35                                     hiddenActivationFunc='tanh',
36                                     outputActivationFunc="softmax",
37                                     learningRate = 0.005,
38                                     optimizer="adam",
39                                     beta1=0.7,
40                                     beta2=0.9,
41                                     epsilon=10^-8,
42                                     mini_batch_size = 512,
43                                     num_epochs = 5000 ,
44                                     print_cost = True)
45
46 #Plot the cost vs number of epochs
47 iterations <- seq(0,5000,1000)
48 costs=retvalsSGD$costs
49 df=data.frame(iterations,costs)
50 ggplot(df,aes(x=iterations,y=costs)) + geom_point() + geom_line(color="blue")
51 + ggttitle("Costs vs number of epochs") + xlab("No of epochs") + ylab("Cost")
```

Costs vs number of epochs



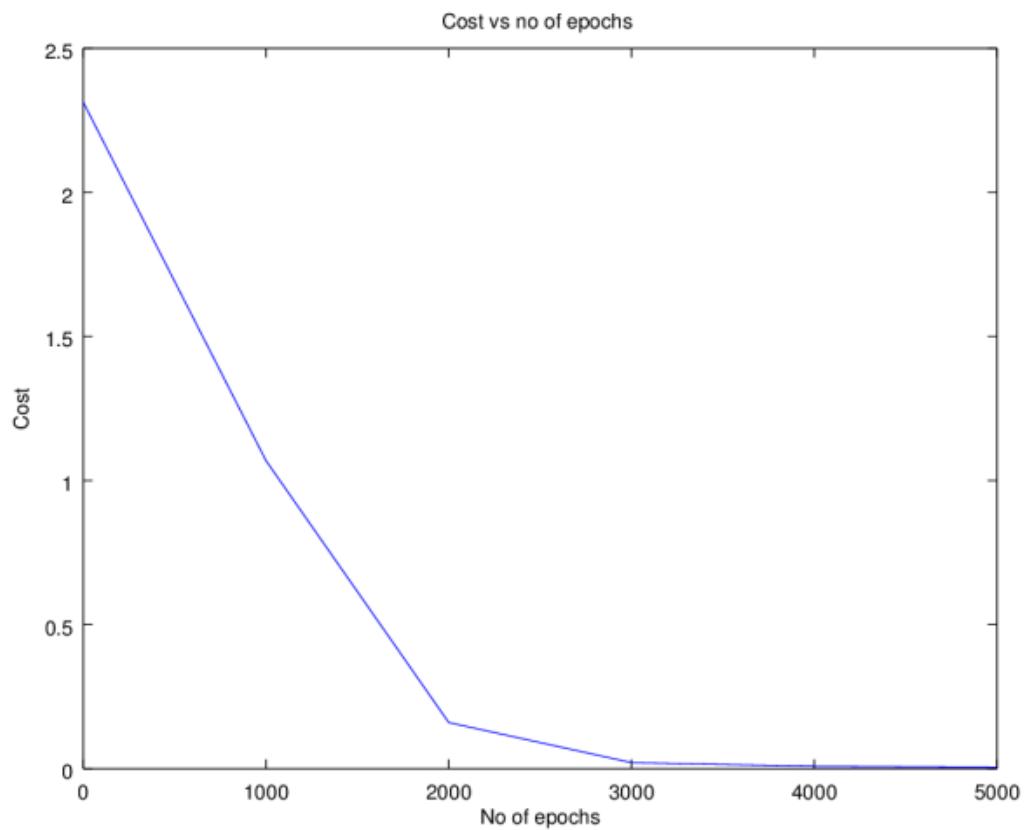
5.1c. Stochastic Gradient Descent with Adam – Octave

```
1 source("DL7functions.m")
2 load('./mnist/mnist.txt.gz');
3 #Create a random permutation from 1024
4 permutation = randperm(1024);
5 disp(length(permutation));
6
7 # Use this 1024 as the batch
8 X=trainX(permutation,:);
9 Y=trainY(permutation,:);
10 # Set layer dimensions
11 # 784 - number of input features (28 x28)
12 # 15, 9 - 2 hidden layers with 15,9 hidden units respectively
13 # 10 - 10 output classes with softmax activation unit at the output layer
14 layersDimensions=[784, 15, 9, 10];
15
16 # Note the high value for epsilon.
17 #Otherwise GD with Adam does not seem to converge
18 #Execute a L-layer Deep Learning network using SGD with Adam optimization
19 # hidden Activation function - relu
20 # output activation function - softmax
21 # learning rate - 0.01
22 # optimizer="adam"
23 # beta1 =0.9
24 # beta2=0.9
25 # epsilon=100
26 # mini_batch_size = 512
27 [weights biases costs]=L_Layer_DeepModel_SGD(X', Y', layersDimensions,
28 hiddenActivationFunc='relu',
```

```

29          outputActivationFunc="softmax",
30          learningRate = 0.1,
31          lrDecay=false,
32          decayRate=1,
33          lambd=0,
34          keep_prob=1,
35          optimizer="adam",
36          beta=0.9,
37          beta1=0.9,
38          beta2=0.9,
39          epsilon=100,
40          mini_batch_size = 512,
41          num_epochs = 5000);
42
43 #Plot cost vs number of epochs
44 plotCostVsEpochs(5000,costs)

```



6.Conclusion

In this chapter I discuss and implement several Stochastic Gradient Descent optimization methods. The implementation of these methods enhance my already existing generic L-Layer Deep Learning Network implementation in vectorized Python, R and Octave.

8.Gradient Check in Deep Learning

You don't understand anything until you learn it more than one way. Marvin Minsky
No computer has ever been designed that is ever aware of what it's doing; but most of the time, we aren't either. Marvin Minsky

A wealth of information creates a poverty of attention. Herbert Simon

In this final chapter, I discuss and implement a key functionality needed while building Deep Learning networks viz. ‘Gradient Checking’. Gradient Checking is an important method to check the correctness of your implementation, specifically the forward propagation and the backward propagation cycles of an implementation. In addition, I also discuss some tips for tuning hyperparameters of a Deep Learning network based on my experience.

This chapter implements a critical function for ensuring the correctness of a L-Layer Deep Learning network implementation using Gradient Checking

Gradient Checking is based on the following approach. One iteration of Gradient Descent computes and updates the parameters θ by doing

$$\theta := \theta - \frac{d}{d\theta} J(\theta)$$

To minimize the cost we will need to minimize $J(\theta)$

Let $g(\theta)$ be a function that computes the derivative $\frac{d}{d\theta} J(\theta)$. Gradient Checking allows us to numerically evaluate the implementation of the function $g(\theta)$ and verify its correctness.

We know the derivative of a function is given by

$$\frac{d}{d\theta} J(\theta) = \lim_{\epsilon \rightarrow 0} \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}$$

Note: The above derivative is based on the 2-sided derivative. The 1-sided derivative is given

$$\frac{d}{d\theta} J(\theta) = \lim_{\epsilon \rightarrow 0} \frac{J(\theta + \epsilon) - J(\theta)}{\epsilon}$$

Gradient Checking is based on the 2-sided derivative because the error is of the order $O(\epsilon^2)$ as opposed $O(\epsilon)$ for the 1-sided derivative.

Hence Gradient Check uses the 2-sided derivative as follows.

$$g(\theta) = \lim_{\epsilon \rightarrow 0} \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}$$

In Gradient Check the following is done

A) Run one normal cycle of your implementation by doing the following

a) Compute the output activation by running 1 cycle of forward propagation

b) Compute the cost using the output activation

c) Compute the gradients using backpropagation (grad)

B) Perform gradient check steps as below

a) Set θ . Flatten all ‘weights’ and ‘bias’ matrices and vectors to a column vector.

b) Initialize $\theta+$ by nudging θ up by adding ϵ ($\theta + \epsilon$)

c) Perform forward propagation with $\theta+$

- d) Compute cost with $\theta+$ i.e. $J(\theta+)$
- e) Initialize $\theta-$ by nudging θ down by subtracting $\epsilon (\theta - \epsilon)$
- f) Perform forward propagation with $\theta-$
- g) Compute cost with $\theta-$ i.e. $J(\theta-)$
- h) Compute $\frac{d}{d\theta} J(\theta)$ or ‘gradapprox’ as $\frac{J(\theta+) - J(\theta-)}{2\epsilon}$ using the 2 sided derivative.
- i) Compute L2norm or the Euclidean distance between ‘grad’ and ‘gradapprox’.

If the difference is of the order of 10^{-5} or 10^{-7} the implementation is correct. In the Deep Learning Specialization (<https://www.coursera.org/specializations/deep-learning>) Prof Andrew Ng mentions that if the difference is of the order of 10^{-7} then the implementation is correct. A difference of 10^{-5} is also ok. Anything more than that is a cause of worry and you should look at your code more closely. To see more details click [Gradient checking and advanced optimization](http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanced_optimization) (http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanced_optimization)

The implementations of the functions invoked in this chapter are in Appendix 8 – Gradient Check

You can clone/download the code from Github at DeepLearningFromFirstPrinciples (<https://github.com/tvganesh/DeepLearningFromFirstPrinciples/tree/master/Chap8-DLGradientCheck>)

After spending a better part of 3 days, I now realize how critical Gradient Check is for ensuring the correctness of your implementation. Initially I was getting very high difference and did not know how to understand the results or debug my implementation. After many hours of staring at the results, I was able to finally arrive at a way, to localize issues in the implementation. In fact, I did catch a small bug in my Python code, which did not exist in the R and Octave implementations. I will demonstrate this below

1.1a Gradient Check – Sigmoid Activation – Python

```

1 import numpy as np
2 import matplotlib
3
4 exec(open("DLfunctions8.py").read())
5 exec(open("testcases.py").read())
6
7 #Load the circles data set
8 train_X, train_Y, test_X, test_Y = load_dataset()
9
10 #Set layer dimensions
11 # 2 - number of input features
12 # 4 - 1 hidden layer with 4 activation units
13 # 1 - 1 sigmoid activation unit at the output layer
14 layersDimensions = [2,4,1]
15
16 # Initialize parameters
17 parameters = initializeDeepModel(layersDimensions)
18
19 #Perform forward propagation
20 AL, caches, dropoutMat = forwardPropagationDeep(train_X, parameters,
21 keep_prob=1, hiddenActivationFunc="relu",outputActivationFunc="sigmoid")
22
23 #Compute cost

```

```

24 cost = computeCost(AL, train_Y, outputActivationFunc="sigmoid")
25 print("cost=",cost)
26
27 #Perform backprop and get gradients
28 gradients = backwardPropagationDeep(AL, train_Y, caches, dropoutMat, lambd=0,
29 keep_prob=1,
30 hiddenActivationFunc="relu",outputActivationFunc="sigmoid")
31
32 # Set values
33 epsilon = 1e-7
34 outputActivationFunc="sigmoid"
35
36 # Set-up variables
37 # Flatten parameters to a vector
38 parameters_values, _ = dictionary_to_vector(parameters)
39
40 #Flatten gradients to a vector
41 grad = gradients_to_vector(parameters,gradients)
42 num_parameters = parameters_values.shape[0]
43
44 #Initialize J_plus, J_minus and gradapprox
45 J_plus = np.zeros((num_parameters, 1))
46 J_minus = np.zeros((num_parameters, 1))
47 gradapprox = np.zeros((num_parameters, 1))
48
49 # Compute gradapprox using 2-sided derivative
50 for i in range(num_parameters):
51
52     # Compute J_plus[i].
53     thetaplus = np.copy(parameters_values)
54     thetaplus[i][0] = thetaplus[i][0] + epsilon
55     AL, caches, dropoutMat = forwardPropagationDeep(train_X,
56 vector_to_dictionary(parameters,thetaplus), keep_prob=1,
57 hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
58     J_plus[i] = computeCost(AL, train_Y,
59 outputActivationFunc=outputActivationFunc)
60
61
62     # Compute J_minus[i].
63     thetaminus = np.copy(parameters_values)
64     thetaminus[i][0] = thetaminus[i][0] - epsilon
65     AL, caches, dropoutMat = forwardPropagationDeep(train_X,
66 vector_to_dictionary(parameters,thetaminus), keep_prob=1,
67 hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
68     J_minus[i] = computeCost(AL, train_Y,
69 outputActivationFunc=outputActivationFunc)
70
71     # Compute gradapprox[i]
72     gradapprox[i] = (J_plus[i] - J_minus[i])/(2*epsilon)
73
74 # Compare gradapprox to gradients from backprop by computing euclidean
75 difference.
76 numerator = np.linalg.norm(grad-gradapprox)
77 denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)
78 difference = numerator/denominator
79
80 #Check the difference
81 if difference > 1e-5:
82     print ("\u033[93m" + "There is a mistake in the backward propagation!
83 difference = " + str(difference) + "\u033[0m")
84 else:
85     print ("\u033[92m" + "Your backward propagation works perfectly fine!
86 difference = " + str(difference) + "\u033[0m")
87 print(difference)

```

```

88 print("\n")
89 # The technique below can be used to identify
90 # which of the parameters are in error
91
92 # Convert grad to dictionary
93 m=vector_to_dictionary2(parameters,grad)
94 print("Gradients from backprop")
95 print(m)
96 print("\n")
97
98 # Convert gradapprox to dictionary
99 n=vector_to_dictionary2(parameters,gradapprox)
100 print("Gradapprox from gradient check")
101 print(n)
102 ## (300, 2)
103 ## (300,)
104 ## cost= 0.6931455556341791
105 ## [Your backward propagation works perfectly fine! difference =
106 1.1604150683743381e-06[0m
107 ## 1.1604150683743381e-06
108 ##
109 ##
110 ##
111 ## Gradients from backprop
112 ## {'dw1': array([[-6.19439955e-06, -2.06438046e-06],
113 ## [-1.50165447e-05, 7.50401672e-05],
114 ## [ 1.33435433e-04, 1.74112143e-04],
115 ## [-3.40909024e-05, -1.38363681e-04]]), 'db1': array([[ 7.31333221e-
116 07],
117 ## [ 7.98425950e-06],
118 ## [ 8.15002817e-08],
119 ## [-5.69821155e-08]]), 'dw2': array([[2.73416304e-04, 2.96061451e-04,
120 7.51837363e-05, 1.01257729e-04]]), 'db2': array([-7.22232235e-06]])}
121 ##
122 ##
123 ## Gradapprox from gradient check
124 ## {'dw1': array([[-6.19448937e-06, -2.06501483e-06],
125 ## [-1.50168766e-05, 7.50399742e-05],
126 ## [ 1.33435485e-04, 1.74112391e-04],
127 ## [-3.40910633e-05, -1.38363765e-04]]), 'db1': array([[ 7.31081862e-
128 07],
129 ## [ 7.98472399e-06],
130 ## [ 8.16013923e-08],
131 ## [-5.71764858e-08]]), 'dw2': array([[2.73416290e-04, 2.96061509e-04,
132 7.51831930e-05, 1.01257891e-04]]), 'db2': array([-7.22255589e-06])}

```

1.1b Gradient Check – Softmax Activation – Python (Error!!)

In the code below I show, how I managed to spot a bug in your implementation

```

1 import numpy as np
2 exec(open("DLfunctions8.py").read())
3
4 # Create spiral dataset
5 N = 100 # number of points per class
6 D = 2 # dimensionality
7 K = 3 # number of classes
8 X = np.zeros((N*K,D)) # data matrix (each row = single example)
9 y = np.zeros(N*K, dtype='uint8') # class labels

```

```

10 for j in range(K):
11     ix = range(N*j,N*(j+1))
12     r = np.linspace(0.0,1,N) # radius
13     t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
14     X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
15     y[ix] = j
16
17
18 # Plot the data
19 #plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
20
21 # Set layer dimensions
22 # 2 - number of input features
23 # 3 - 1 hidden layer with 3 activation units
24 # 3 - 3 classes with softmax activation unit at the output layer
25 layersDimensions = [2,3,3]
26
27 # Set params
28 y1=y.reshape(-1,1).T
29 train_X=X.T
30 train_Y=y1
31
32 # Initialize the model
33 parameters = initializeDeepModel(layersDimensions)
34
35 #Compute forward prop
36 AL, caches, dropoutMat = forwardPropagationDeep(train_X, parameters,
37 keep_prob=1,
38
39 hiddenActivationFunc="relu",outputActivationFunc="softmax")
40
41 #Compute cost
42 cost = computeCost(AL, train_Y, outputActivationFunc="softmax")
43 print("cost=",cost)
44
45 #Compute gradients from backprop
46 gradients = backwardPropagationDeep(AL, train_Y, caches, dropoutMat, lambd=0,
47 keep_prob=1,
48
49 hiddenActivationFunc="relu",outputActivationFunc="softmax")
50
51 # Note the transpose of the gradients for Softmax has to be taken
52 L= len(parameters)//2
53 print(L)
54 gradients['dw'+str(L)]=gradients['dw'+str(L)].T
55 gradients['db'+str(L)]=gradients['db'+str(L)].T
56
57 # Perform gradient check
58 gradient_check_n(parameters, gradients, train_X, train_Y, epsilon = 1e-
59 ,outputActivationFunc="softmax")
60
61 cost= 1.0986187818144022
62 There is a mistake in the backward propagation! difference =
63 0.7100295155692544
64 0.7100295155692544
65
66
67 Gradients from backprop
68 {'dw1': array([[ 0.00050125,  0.00045194],
69                 [ 0.00096392,  0.00039641],
70                 [-0.00014276, -0.00045639]]), 'db1': array([[ 0.00070082],
71                 [-0.00224399],
72                 [ 0.00052305]]), 'dw2': array([[-8.40953794e-05, -9.52657769e-04, -
73                 1.10269379e-04],
```

```

74      [-7.45469382e-04,  9.49795606e-04,  2.29045434e-04],
75      [ 8.29564761e-04,  2.86216305e-06, -1.18776055e-04]], ,
76      'db2': array([[[-0.00253808],
77      [-0.00505508],
78      [ 0.00759315]]])
79
80
81 Gradapprox from gradient check
82 {'dw1': array([[ 0.00050125,  0.00045194],
83     [ 0.00096392,  0.00039641],
84     [-0.00014276, -0.00045639]]], 'db1': array([[ 0.00070082],
85     [-0.00224399],
86     [ 0.00052305]]], 'dw2': array([[-8.40960634e-05, -9.52657953e-04,
87     1.10268461e-04],
88     [-7.45469242e-04,  9.49796908e-04,  2.29045671e-04],
89     [ 8.29565305e-04,  2.86104473e-06, -1.18776100e-04]]),
90     'db2': array([[-8.46211989e-06],
91     [-1.68487446e-05],
92     [ 2.53108645e-05]])}

```

Gradient Check gives a high value of the difference of 0.7100295. Inspecting the Gradients and Gradapprox we can see there is a very big discrepancy in db2. After I went over my code I discovered that my computation in the function layerActivationBackward for Softmax was

```

1 # Erroneous code
2     if activationFunc == 'softmax':
3         dw = 1/numtraining * np.dot(A_prev,dz)
4         db = np.sum(dz, axis=0, keepdims=True)
5         dA_prev = np.dot(dz,w)
6
7 instead of
8
9 # Fixed code
10    if activationFunc == 'softmax':
11        dw = 1/numtraining * np.dot(A_prev,dz)
12        db = 1/numtraining * np.sum(dz, axis=0, keepdims=True)
13        dA_prev = np.dot(dz,w)

```

After fixing this error when I ran the Gradient Check as shown below

1.1c Gradient Check – Softmax Activation – Python (Corrected!!)

```

1 import numpy as np
2 exec(open("DLfunctions8.py").read())
3
4 # Create Spiral dataset
5 N = 100 # number of points per class
6 D = 2 # dimensionality
7 K = 3 # number of classes
8 X = np.zeros((N*K,D)) # data matrix (each row = single example)
9 y = np.zeros(N*K, dtype='uint8') # class labels
10 for j in range(K):
11     ix = range(N*j,N*(j+1))
12     r = np.linspace(0.0,1,N) # radius
13     t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta

```

```

14 x[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
15 y[ix] = j
16
17
18 # Plot the data
19 #plt.scatter(x[:, 0], x[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
20 # Set layer dimensions
21 # 2 - number of input features
22 # 3 - 1 hidden layer with 3 activation units
23 # 3 - 3 classes with softmax activation unit at the output layer
24 layersDimensions = [2,3,3]
25 y1=y.reshape(-1,1).T
26 train_X=X.T
27 train_Y=y1
28
29 # Initialize model
30 parameters = initializeDeepModel(layersDimensions)
31
32 #Perform forward prop
33 AL, caches, dropoutMat = forwardPropagationDeep(train_X, parameters,
34 keep_prob=1,
35
36 hiddenActivationFunc="relu",outputActivationFunc="softmax")
37
38 #Compute cost
39 cost = computeCost(AL, train_Y, outputActivationFunc="softmax")
40 print("cost=",cost)
41
42 #Compute gradients from backprop
43 gradients = backwardPropagationDeep(AL, train_Y, caches, dropoutMat, lambd=0,
44 keep_prob=1,
45
46 hiddenActivationFunc="relu",outputActivationFunc="softmax")
47
48 # Note the transpose of the gradients for softmax has to be taken
49 L= len(parameters)//2
50 print(L)
51 gradients['dw'+str(L)]=gradients['dw'+str(L)].T
52 gradients['db'+str(L)]=gradients['db'+str(L)].T
53
54 #Perform gradient check
55 gradient_check_n(parameters, gradients, train_X, train_Y, epsilon = 1e-
56 7,outputActivationFunc="softmax")
57 ## cost= 1.0986193170234435
58 ## 2
59 ## [92mYour backward propagation works perfectly fine! difference =
60 5.268804859613151e-07[0m
61 ## 5.268804859613151e-07
62 ##
63 ##
64 ## Gradients from backprop
65 ## {'dw1': array([[ 0.00053206,  0.00038987],
66 ##                 [ 0.00093941,  0.00038077],
67 ##                 [-0.00012177, -0.0004692 ]]), 'db1': array([[ 0.00072662],
68 ##                 [-0.00210198],
69 ##                 [ 0.00046741]]), 'dw2': array([[-7.83441270e-05, -9.70179498e-04, -
70 1.08715815e-04],
71 ##                 [-7.70175008e-04,  9.54478237e-04,  2.27690198e-04],
72 ##                 [ 8.48519135e-04,  1.57012608e-05, -1.18974383e-04]]), 'db2':
73 array([[-8.52190476e-06],
74 ##                 [-1.69954294e-05],
75 ##                 [ 2.55173342e-05]])}
76 ##
77 ##

```

```

78 ## Gradapprox from gradient check
79 ## {'dw1': array([[ 0.00053206,  0.00038987],
80 ##                  [ 0.00093941,  0.00038077],
81 ##                  [-0.00012177, -0.0004692 ]]), 'db1': array([[ 0.00072662],
82 ##                  [-0.00210198],
83 ##                  [ 0.00046741]]), 'dw2': array([[-7.83439980e-05, -9.70180603e-04,
84 1.08716369e-04],
85 ##                  [-7.70173925e-04,  9.54478718e-04,  2.27690089e-04],
86 ##                  [ 8.48520143e-04,  1.57018842e-05, -1.18973720e-04]]), 'db2':
87 array([[-8.52096171e-06],
88 ##                  [-1.69964043e-05],
89 ##                  [ 2.55162558e-05]])}

```

1.2a Gradient Check – Sigmoid Activation – R

```

1 source("DLfunctions8.R")
2 z <- as.matrix(read.csv("circles.csv",header=FALSE))
3
4 x <- z[,1:2]
5 y <- z[,3]
6 X <- t(x)
7 Y <- t(y)
8
9 #Set layer dimensions
10 layersDimensions = c(2,5,1)
11
12 #Initialize model
13 parameters = initializeDeepModel(layersDimensions)
14
15 #Perform forward prop
16 retvals = forwardPropagationDeep(X, parameters,keep_prob=1,
17 hiddenActivationFunc="relu",
18 outputActivationFunc="sigmoid")
19 AL <- retvals[['AL']]
20 caches <- retvals[['caches']]
21 dropoutMat <- retvals[['dropoutMat']]
22
23 #Compute cost
24 cost <- computeCost(AL, Y,outputActivationFunc="sigmoid",
25 numClasses=layersDimensions[length(layersDimensions)])
26 print(cost)
27 ## [1] 0.6931447
28
29 # Perform backward propagation and get gradients
30 gradients = backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd=0,
31 keep_prob=1, hiddenActivationFunc="relu",
32 outputActivationFunc="sigmoid",numClasses=layersDimensions[length(layersDimensions)])
33
34 # Set values
35 epsilon = 1e-07
36 outputActivationFunc="sigmoid"
37
38 #Convert parameter list to vector
39 parameters_values = list_to_vector(parameters)
40
41 #Convert gradient list to vector
42 grad = gradients_to_vector(parameters,gradients)
43 num_parameters = dim(parameters_values)[1]
44
45 #Initialize J_plus, J_minus and gradapprox

```

```

48 J_plus = matrix(rep(0,num_parameters),
49                 nrow=num_parameters,ncol=1)
50 J_minus = matrix(rep(0,num_parameters),
51                  nrow=num_parameters,ncol=1)
52 gradapprox = matrix(rep(0,num_parameters),
53                      nrow=num_parameters,ncol=1)
54
55 # Compute J_plus, J_minus and
56 for(i in 1:num_parameters){
57
58     # Compute J_plus[i].
59     thetaplus = parameters_values
60     thetaplus[i][1] = thetaplus[i][1] + epsilon
61     retvals = forwardPropagationDeep(X, vector_to_list(parameters,thetaplus),
62     keep_prob=1,
63     hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
64
65     AL <- retvals[['AL']]
66     J_plus[i] = computeCost(AL, Y, outputActivationFunc=outputActivationFunc)
67
68
69     # Compute J_minus[i].
70     thetaminus = parameters_values
71     thetaminus[i][1] = thetaminus[i][1] - epsilon
72     retvals = forwardPropagationDeep(X,
73     vector_to_list(parameters,thetaminus), keep_prob=1,
74     hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
75     AL <- retvals[['AL']]
76     J_minus[i] = computeCost(AL, Y,
77     outputActivationFunc=outputActivationFunc)
78
79     # Compute gradapprox[i]
80     gradapprox[i] = (J_plus[i] - J_minus[i])/(2*epsilon)
81 }
82 # Compare gradapprox to backprop gradients by computing Euclidean difference.
83 #Compute L2Norm
84 numerator = L2NormVec(grad-gradapprox)
85 denominator = L2NormVec(grad) + L2NormVec(gradapprox)
86 difference = numerator/denominator
87
88 # Output difference
89 if(difference > 1e-5){
90     cat("There is a mistake, the difference is too high",difference)
91 } else{
92     cat("The implementations works perfectly", difference)
93 }
94 ## The implementations works perfectly 1.279911e-06
95
96 # Technique to check gradients
97 print("Gradients from backprop")
98 ## [1] "Gradients from backprop"
99 vector_to_list2(parameters,grad)
100 ## $dw1
101 ## [,1]      [,2]
102 ## [1,] -7.641588e-05 -3.427989e-07
103 ## [2,] -9.049683e-06  6.906304e-05
104 ## [3,]  3.401039e-06 -1.503914e-04
105 ## [4,]  1.535226e-04 -1.686402e-04
106 ## [5,] -6.029292e-05 -2.715648e-04
107 ##
108 ## $db1
109 ## [,1]
110 ## [1,]  6.930318e-06
111 ## [2,] -3.283117e-05

```

```

112 ## [3,] 1.310647e-05
113 ## [4,] -3.454308e-05
114 ## [5,] -2.331729e-08
115 ##
116 ## $dw2
117 ## [,1] [,2] [,3] [,4] [,5]
118 ## [1,] 0.0001612356 0.0001113475 0.0002435824 0.000362149 2.874116e-05
119 ##
120 ## $db2
121 ## [,1]
122 ## [1,] -1.16364e-05
123 print("Grad approx from gradient check")
124 ## [1] "Grad approx from gradient check"
125 vector_to_list2(parameters,gradapprox)
126 ## $dw1
127 ## [,1] [,2]
128 ## [1,] -7.641554e-05 -3.430589e-07
129 ## [2,] -9.049428e-06 6.906253e-05
130 ## [3,] 3.401168e-06 -1.503919e-04
131 ## [4,] 1.535228e-04 -1.686401e-04
132 ## [5,] -6.029288e-05 -2.715650e-04
133 ##
134 ## $db1
135 ## [,1]
136 ## [1,] 6.930012e-06
137 ## [2,] -3.283096e-05
138 ## [3,] 1.310618e-05
139 ## [4,] -3.454237e-05
140 ## [5,] -2.275957e-08
141 ##
142 ## $dw2
143 ## [,1] [,2] [,3] [,4] [,5]
144 ## [1,] 0.0001612355 0.0001113476 0.0002435829 0.0003621486 2.87409e-05
145 ##
146 ## $db2
147 ## [,1]
148 ## [1,] -1.16368e-05

```

1.2b Gradient Check – Softmax Activation – R

```

1 source("DLfunctions8.R")
2 Z <- as.matrix(read.csv("spiral.csv",header=FALSE))
3
4 # Setup the data
5 X <- Z[,1:2]
6 y <- Z[,3]
7 X <- t(X)
8 Y <- t(y)
9
10 # Set layer dimensions
11 # 2 - number of input features
12 # 3 - 1 hidden layer with 3 activation units
13 # 3 - 3 classes with softmax activation unit at the output layer
14 layersDimensions = c(2, 3, 3)
15
16 #Initialize model
17 parameters = initializeDeepModel(layersDimensions)
18
19 #Perform forward propagation
20 retvals = forwardPropagationDeep(X, parameters,keep_prob=1,
21 hiddenActivationFunc="relu",

```

```

22                                     outputActivationFunc="softmax")
23 AL <- retvals[['AL']]
24 caches <- retvals[['caches']]
25 dropoutMat <- retvals[['dropoutMat']]
26
27 #Compute cost
28 cost <- computeCost(AL, Y, outputActivationFunc="softmax",
29                         numClasses=layersDimensions[length(layersDimensions)])
30 print(cost)
31 ## [1] 1.098618
32
33 # Perform Backward propagation.
34 gradients = backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd=0,
35 keep_prob=1, hiddenActivationFunc="relu",
36 outputActivationFunc="softmax", numClasses=layersDimensions[length(layersDimensions)])
37
38 # Need to take transpose of the last layer for Softmax
39 L=length(parameters)/2
40 gradients[[paste('dw',L,sep="")]] = t(gradients[[paste('dw',L,sep="")]])
41 gradients[[paste('db',L,sep="")]] = t(gradients[[paste('db',L,sep="")]])
42
43 #Perform gradient check
44 gradient_check_n(parameters, gradients, X, Y,
45                     epsilon = 1e-7, outputActivationFunc="softmax")
46 ## The implementations works perfectly 3.903011e-07[1] "Gradients from
47 backprop"
48 ## $dw1
49
50 ## [,1]      [,2]
51 ## [1,] 0.0007962367 -0.0001907606
52 ## [2,] 0.0004444254  0.0010354412
53 ## [3,] 0.0003078611  0.0007591255
54 ##
55 ## $db1
56 ## [,1]
57 ## [1,] -0.0017305136
58 ## [2,] 0.0005393734
59 ## [3,] 0.0012484550
60 ##
61 ## $dw2
62 ## [,1]      [,2]      [,3]
63 ## [1,] -3.515627e-04 7.487283e-04 -3.971656e-04
64 ## [2,] -6.381521e-05 -1.257328e-06 6.507254e-05
65 ## [3,] -1.719479e-04 -4.857264e-04 6.576743e-04
66 ##
67 ## $db2
68 ## [,1]
69 ## [1,] -5.536383e-06
70 ## [2,] -1.824656e-05
71 ## [3,] 2.378295e-05
72 ##
73 ## [1] "Grad approx from gradient check"
74 ## $dw1
75 ## [,1]      [,2]
76 ## [1,] 0.0007962364 -0.0001907607
77 ## [2,] 0.0004444256  0.0010354406
78 ## [3,] 0.0003078615  0.0007591250
79 ##
80 ## $db1
81 ## [,1]
82 ## [1,] -0.0017305135
83 ## [2,] 0.0005393741
84 ## [3,] 0.0012484547
85 ##

```

```

86 ## $dw2
87 ## [,1]      [,2]      [,3]
88 ## [1,] -3.515632e-04 7.487277e-04 -3.971656e-04
89 ## [2,] -6.381451e-05 -1.257883e-06 6.507239e-05
90 ## [3,] -1.719469e-04 -4.857270e-04 6.576739e-04
91 ##
92 ## $db2
93 ## [,1]
94 ## [1,] -5.536682e-06
95 ## [2,] -1.824652e-05
96 ## [3,] 2.378209e-05

```

1.3a Gradient Check – Sigmoid Activation – Octave

```

1 source("DL8functions.m")
2
3 # Read circles data
4 data=csvread("circles.csv");
5 X=data(:,1:2);
6 Y=data(:,3);
7
8 #Set layer dimensions
9 # 2 - number of input features
10 # 5 - 1 hidden layer with 5 activation units
11 # 1 - 1 sigmoid activation unit at the output layer
12 layersDimensions = [2 5 1];
13
14 #Initialize model
15 [weights biases] = initializeDeepModel(layersDimensions);
16
17 #Perform forward prop
18 [AL forward_caches activation_caches dropoutMat] = forwardPropagationDeep(X',
19 weights, biases,keep_prob=1,
20 hiddenActivationFunc="relu",
21 outputActivationFunc="sigmoid");
22
23 #Compute cost
24 cost = computeCost(AL,
25 Y',outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size
26 (layersDimensions)(2)));
27 disp(cost);
28
29 #Compute gradients from cost
30 [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y',
31 activation_caches,forward_caches, dropoutMat, lambd=0, keep_prob=1,
32 hiddenActivationFunc="relu",
33 outputActivationFunc="sigmoid",
34 numClasses=layersDimensions(size(layersDimensions)(2)));
35
36 #Set values
37 epsilon = 1e-07;
38 outputActivationFunc="sigmoid";
39
40 # Convert paramters cell array to vector
41 parameters_values = cellArray_to_vector(weights, biases);
42
43 #Convert gradient cell array to vector
44 grad = gradients_to_vector(gradsDW,gradsDB);
45 num_parameters = size(parameters_values)(1);
46
47 #Initialize J_plus, J_minus and gradapprox

```

```

49 J_plus = zeros(num_parameters, 1);
50 J_minus = zeros(num_parameters, 1);
51 gradapprox = zeros(num_parameters, 1);
52
53 # Compute gradapprox
54 for i = 1:num_parameters
55
56     # Compute J_plus[i].
57     thetaplus = parameters_values;
58     thetaplus(i,1) = thetaplus(i,1) + epsilon;
59     [weights1 biases1] = vector_to_cellArray(weights, biases, thetaplus);
60     [AL forward_caches activation_caches dropoutMat] =
61     forwardPropagationDeep(X', weights1, biases1, keep_prob=1,
62     hiddenActivationFunc="relu", outputActivationFunc=outputActivationFunc);
63     J_plus(i) = computeCost(AL, Y',
64     outputActivationFunc=outputActivationFunc);
65
66     # Compute J_minus[i].
67     thetaminus = parameters_values;
68     thetaminus(i,1) = thetaminus(i,1) - epsilon ;
69     [weights1 biases1] = vector_to_cellArray(weights, biases, thetaminus);
70     [AL forward_caches activation_caches dropoutMat] =
71     forwardPropagationDeep(X', weights1, biases1, keep_prob=1,
72
73     hiddenActivationFunc="relu", outputActivationFunc=outputActivationFunc);
74     J_minus(i) = computeCost(AL, Y',
75     outputActivationFunc=outputActivationFunc);
76
77     # Compute gradapprox[i]
78     gradapprox(i) = (J_plus(i) - J_minus(i))/(2*epsilon);
79
80 endfor
81
82 #Compute L2Norm or the Euclidean distance between gradients and gradapprox
83 numerator = L2NormVec(grad-gradapprox);
84 denominator = L2NormVec(grad) + L2NormVec(gradapprox);
85 difference = numerator/denominator;
86 disp(difference);
87
88 #Check difference
89 if difference > 1e-04
90     printf("There is a mistake in the implementation ");
91     disp(difference);
92 else
93     printf("The implementation works perfectly");
94     disp(difference);
95 endif
96
97 # Technique to compare the weights and biases and localize issue
98 # Convert grad to cell Array
99 [weights1 biases1] = vector_to_cellArray(weights, biases, grad);
100 printf("Gradients from back propagation");
101 disp(weights1);
102 disp(biases1);
103
104 # Convert gradapprox to cell array
105 [weights2 biases2] = vector_to_cellArray(weights, biases, gradapprox);
106 printf("Gradients from gradient check");
107
108 # Display
109 disp(weights2);
110 disp(biases2);
111
112 0.69315

```

```

113 1.4893e-005
114 The implementation works perfectly 1.4893e-005
115 Gradients from back propagation
116 {
117 [1,1] =
118 5.0349e-005 2.1323e-005
119 8.8632e-007 1.8231e-006
120 9.3784e-005 1.0057e-004
121 1.0875e-004 -1.9529e-007
122 5.4502e-005 3.2721e-005
123 [1,2] =
124 1.0567e-005 6.0615e-005 4.6004e-005 1.3977e-004 1.0405e-004
125 }
126 {
127 [1,1] =
128 -1.8716e-005
129 1.1309e-009
130 4.7686e-005
131 1.2051e-005
132 -1.4612e-005
133 [1,2] = 9.5808e-006
134 }
135 Gradients from gradient check
136 {
137 [1,1] =
138 5.0348e-005 2.1320e-005
139 8.8485e-007 1.8219e-006
140 9.3784e-005 1.0057e-004
141 1.0875e-004 -1.9762e-007
142 5.4502e-005 3.2723e-005
143 [1,2] =
144 [1,2] =
145 1.0565e-005 6.0614e-005 4.6007e-005 1.3977e-004 1.0405e-004
146 }
147 {
148 [1,1] =
149 -1.8713e-005
150 1.1102e-009
151 4.7687e-005
152 1.2048e-005
153 -1.4609e-005
154 [1,2] = 9.5790e-006
155 }
156

```

1.3b Gradient Check – Softmax Activation – Octave

```

1 source("DL8functions.m")
2 data=csvread("spiral.csv");
3
4 # Setup the data
5 X=data(:,1:2);
6 Y=data(:,3);
7
8 # Set the layer dimensions
9 # 2 - number of input features
10 # 3 - 1 hidden layer with 3 activation units

```

```

11 # 3 - 3 classes with softmax activation unit at the output
12 layersDimensions = [2 3 3];
13 [weights biases] = initializeDeepModel(layersDimensions);
14
15 # Run forward prop
16 [AL forward_caches activation_caches dropoutMat] = forwardPropagationDeep(X',
17 weights, biases, keep_prob=1,
18 hiddenActivationFunc="relu",
19 outputActivationFunc="softmax");
20
21 # Compute cost
22 cost = computeCost(AL,
23 Y', outputActivationFunc=outputActivationFunc, numClasses=layersDimensions(size
24 (layersDimensions)(2)));
25 disp(cost);
26
27 # Perform backward prop
28 [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y',
29 activation_caches, forward_caches, dropoutMat, lambd=0, keep_prob=1,
30 hiddenActivationFunc="relu",
31 outputActivationFunc="softmax",
32 numClasses=layersDimensions(size(layersDimensions)(2)));
33
34 #Take transpose of last layer for Softmax
35 L=size(weights)(2);
36 gradsDW{L}= gradsDW{L}';
37 gradsDB{L}= gradsDB{L}';
38
39 #Perform gradient check
40 difference= gradient_check_n(weights, biases, gradsDW,gradsDB, X, Y, epsilon
41 = 1e-7,
42
43 outputActivationFunc="softmax",numClasses=layersDimensions(size(layersDimensi
44 ons)(2)));
45 1.0986
46 The implementation works perfectly! 2.0021e-005
47 Gradients from back propagation
48 {
49     [1,1] =
50         -7.1590e-005 4.1375e-005
51         -1.9494e-004 -5.2014e-005
52         -1.4554e-004 5.1699e-005
53     [1,2] =
54         3.3129e-004 1.9806e-004 -1.5662e-005
55         -4.9692e-004 -3.7756e-004 -8.2318e-005
56         1.6562e-004 1.7950e-004 9.7980e-005
57 }
58 {
59     [1,1] =
60         -3.0856e-005
61         -3.3321e-004
62         -3.8197e-004
63     [1,2] =
64         1.2046e-006
65         2.9259e-007
66         -1.4972e-006
67 }
68 Gradients from gradient check
69 {
70     [1,1] =
71         -7.1586e-005 4.1377e-005
72         -1.9494e-004 -5.2013e-005
73         -1.4554e-004 5.1695e-005
74 }
```

```

75      3.3129e-004 1.9806e-004 -1.5664e-005
76      -4.9692e-004 -3.7756e-004 -8.2316e-005
77      1.6562e-004 1.7950e-004 9.7979e-005
78  }
79  {
80  [1,1] =
81  -3.0852e-005
82  -3.3321e-004
83  -3.8197e-004
84  [1,2] =
85  1.1902e-006
86  2.8200e-007
87  -1.4644e-006
88  }
89

```

2. Tip for tuning hyperparameters

Deep Learning Networks come with a large number of hyper parameters which require tuning. The hyper parameters are

1. α -learning rate
2. Number of layers
3. Number of hidden units
4. Number of iterations
5. Momentum – β – 0.9
6. RMSProp – β_1 – 0.9
7. Adam – β_1, β_2 and ϵ
8. learning rate decay
9. mini batch size
10. Initialization method – He, Xavier
11. Regularization

- Among the above the most critical learning rate α . Rather than just trying out random values, it may help to try out values on a logarithmic scale. So, we could try out values -0.01,0.1,1.0,10 etc. If we find that the cost is between 0.01 and 0.1 we could use the bisection technique so we can try 0.05. If we need to be bigger than 0.01 and 0.05 we could try 0.03 and then keep halving the distance etc.
- The performance of Momentum and RMSProp are very good and work well with values 0.9. Even with this, it is better to try out values of $1-\beta$ in the logarithmic range. So, $1-\beta$ could 0.001,0.01,0.1 and hence β would be 0.999,0.99 or 0.9
- Increasing the number of hidden units or number of hidden layers needs to be done gradually. I have noticed that increasing number of hidden layers heavily does not improve performance and sometimes degrades it.
- Sometimes, I tend to increase the number of iterations if I think I see a steady decrease in the cost for a certain learning rate
- It may also help to add learning rate decay if you see there is an oscillation while it decreases.
- Xavier and He initializations also help in a fast convergence and are worth trying out.

3. Final thoughts

As I come to a close in this Deep Learning Series from first principles in Python, R and Octave, I must admit that I learnt a lot in the process.

- * Building a L-layer, vectorized Deep Learning Network in Python, R and Octave was extremely challenging but very rewarding
- * One benefit of building vectorized versions in Python, R and Octave was that I was looking at each function that I was implementing thrice, and hence I was able to fix any bugs in any of the languages
- * In addition since I built the generic L-Layer DL network with all the bells and whistles, layer by layer I further had an opportunity to look at all the functions in each successive post.
- * Each language has its advantages and disadvantages. From the performance perspective I think Python is the best, followed by Octave and then R
- * Interesting, I noticed that even if small bugs creep into your implementation, the DL network does learn and does generate a valid set of weights and biases, however this may not be an optimum solution. In one case of an inadvertent bug, I was not updating the weights in the final layer of the DL network. Yet, using all the other layers, the DL network was able to come with a reasonable solution (maybe like random dropout, remaining units can still learn the data!)
- * Having said that, the Gradient Check method discussed and implemented in this post can be very useful in ironing out bugs.

4. Conclusion

These months when I was writing the chapters and also churning up the code in Python, R and Octave were very hectic. There have been times when I found that implementations of some function to be extremely demanding and I almost felt like giving up. There have been other times when I had to spend quite some time on an intractable DL network which would not respond to changes in hyper-parameters. All in all, it was a great learning experience.

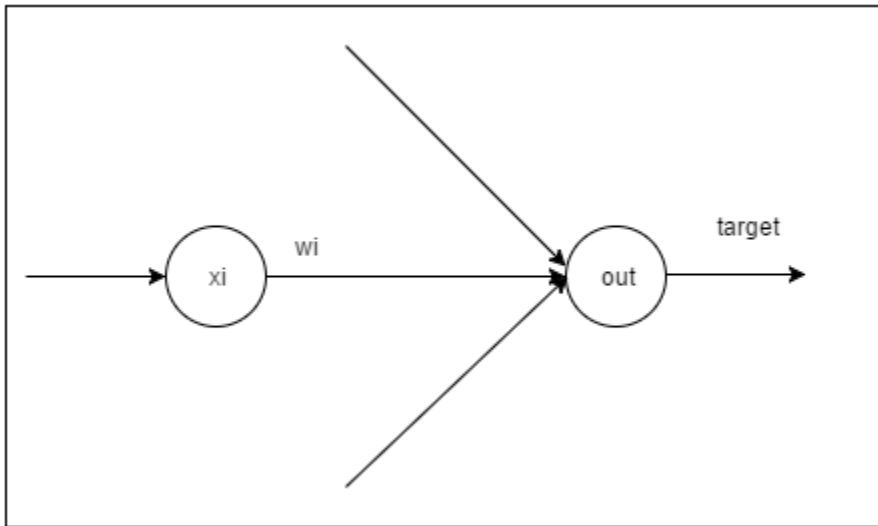
Hope you enjoyed this detailed derivations and the implementation!!!

1.Appendix A

Neural Networks The mechanics of backpropagation

The initial work in the ‘Backpropagation Algorithm’ started in the 1980’s and led to an explosion of interest in Neural Networks and the application of backpropagation

The ‘Backpropagation’ algorithm computes the minimum of an error function with respect to the weights in the Neural Network. It uses the method of gradient descent. The combination of weights in a multi-layered neural network, which minimizes the error/cost function, is considered a solution of the learning problem.



In the Neural Network above

$$out_{o1} = \sum_i w_i * x_i$$

$$E = 1/2(target - out)^2$$

$$\partial E / \partial out = 1/2 * 2 * (target - out) * -1 = -(target - out)$$

$$\partial E / \partial w_i = \partial E / \partial y * \partial y / \partial w_i$$

$$\partial E / \partial w_i = -(target - out) * x_i$$

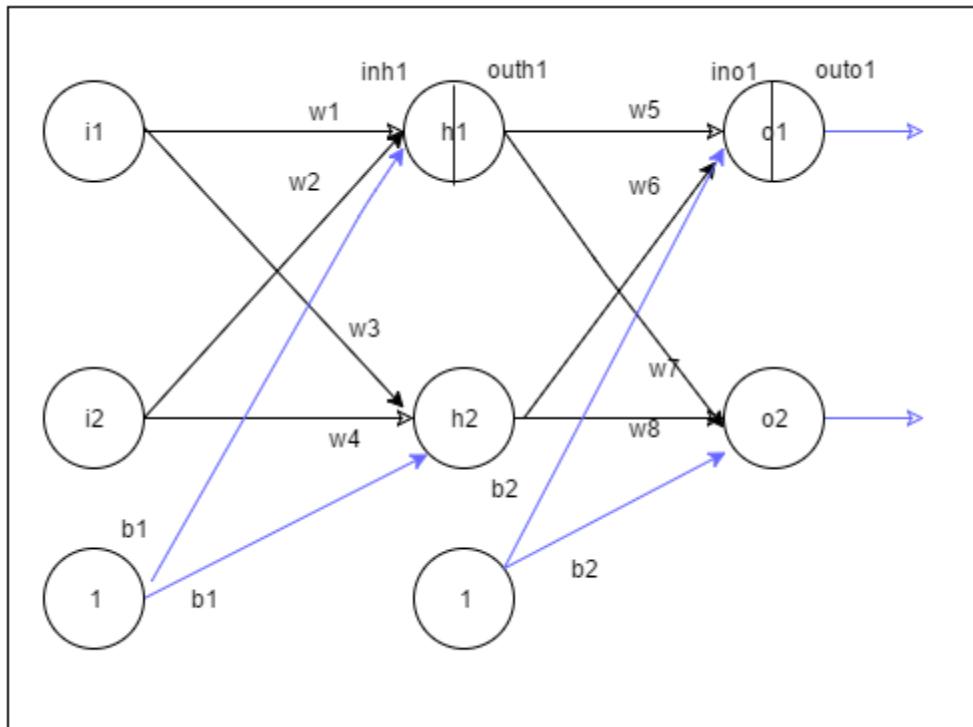
Perceptrons and single layered neural networks can only classify, if the sample space is linearly separable. For non-linear decision boundaries, a multi layered neural network with backpropagation is required to generate more complex boundaries. The backpropagation algorithm, computes the minimum of the error function in weight space using the method of gradient descent. This computation of the gradient requires the activation function to be both differentiable and continuous. Hence, the sigmoid or logistic function is typically chosen as the activation function at every layer.

This post looks at a 3 layer neural network with 1 input, 1 hidden and 1 output. To a large extent this post is based on Matt Mazur’s detailed “A step by step backpropagation example” (<https://www.coursera.org/learn/neural-networks/home>), and Prof Hinton’s “Neural Networks for

Machine Learning” (<https://www.coursera.org/learn/neural-networks/home/welcome>) at Coursera and a few other sources.

While Matt Mazur’s post uses example values, I generate the formulas for the gradient derivatives for each weight in the hidden and input layers. I intend to implement a vector version of backpropagation in Octave, R and Python. Therefore, this post is a prequel to that.

The 3-layer neural network is as below



Some basic derivations which are used in backpropagation

Chain rule of differentiation

Let $y=f(u)$
and $u=g(x)$ then
 $\partial y/\partial x = \partial y/\partial u * \partial u/\partial x$

An important result

$y = 1/(1 + e^{-z})$
Let $x = 1 + e^{-z}$ then
 $y = 1/x$
 $\partial y/\partial x = -1/x^2$
 $\partial x/\partial z = -e^{-z}$

Using the chain rule of differentiation we get
 $\partial y/\partial z = \partial y/\partial x * \partial x/\partial z$

$$= -1/(1 + e^{-z})^2 * -e^{-z} = e^{-z}/(1 + e^{-z})^2$$

Therefore $\partial y / \partial z = y(1 - y)$ -(A)

1) Feed forward network

The net output at the 1st hidden layer

$$in_{h1} = w_1 i_1 + w_2 i_2 + b_1$$

$$in_{h2} = w_3 i_1 + w_4 i_2 + b_1$$

The sigmoid/logistic function function is used to generate the activation outputs for each hidden layer. The sigmoid is chosen because it is continuous and has a continuous derivative

$$out_{h1} = 1/(1 + e^{-in_{h1}})$$

$$out_{h2} = 1/(1 + e^{-in_{h2}})$$

The net output at the output layer

$$in_{o1} = w_5 out_{h1} + w_6 out_{h2} + b_2$$

$$in_{o2} = w_7 out_{h1} + w_8 out_{h2} + b_2$$

Total error

$$E_{total} = 1/2 \sum (target - output)^2$$

$$E_{total} = E_{o1} + E_{o2}$$

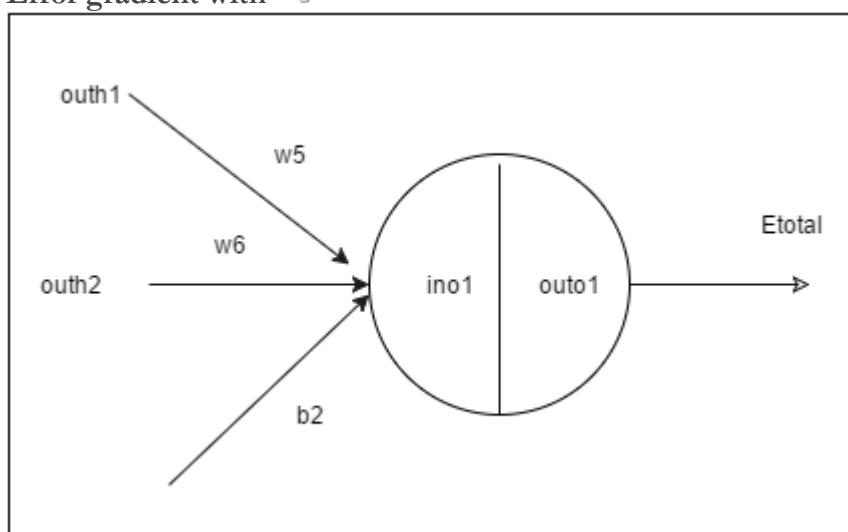
$$E_{total} = 1/2(target_{o1} - out_{o1})^2 + 1/2(target_{o2} - out_{o2})^2$$

2) The backwards pass

In the backward pass we need to compute how the squared error changes with changing weight. i.e we compute $\partial E_{total} / \partial w_i$ for each weight w_i . This is shown below

A squared error is assumed

Error gradient with w_5



$$\partial E_{total} / \partial w_5 = \partial E_{total} / \partial out_{o1} * \partial out_{o1} / \partial in_{o1} * \partial in_{o1} / \partial w_5 \text{ -(B)}$$

Since

$$E_{total} = 1/2 \sum (target - output)^2$$

$$E_{total} = 1/2(target_{o1} - out_{o1})^2 + 1/2(target_{o2} - out_{o2})^2$$

$$\partial E_{total} / \partial out_{o1} = \partial E_{o1} / \partial out_{o1} + \partial E_{o2} / \partial out_{o1}$$

$$\partial E_{total} / \partial out_{o1} = \partial / \partial out_{o1} [1/2(target_{o1} - out_{o1})^2 - 1/2(target_{o2} - out_{o2})^2]$$

$$\partial E_{total} / \partial out_{o1} = 2 * 1/2 * (target_{o1} - out_{o1}) * -1 + 0$$

Now considering the 2nd term in (B)

$$\partial out_{o1} / \partial in_{o1} = \partial / \partial in_{o1} [1/(1 + e^{-in_{o1}})]$$

Using result (A)

$$\partial out_{o1} / \partial in_{o1} = \partial / \partial in_{o1} [1/(1 + e^{-in_{o1}})] = out_{o1}(1 - out_{o1})$$

The 3rd term in (B)

$$\partial in_{o1} / \partial w_5 = \partial / \partial w_5 [w_5 * out_{h1} + w_6 * out_{h2}] = out_{h1}$$

$$\partial E_{total} / \partial w_5 = -(target_{o1} - out_{o1}) * out_{o1} * (1 - out_{o1}) * out_{h1}$$

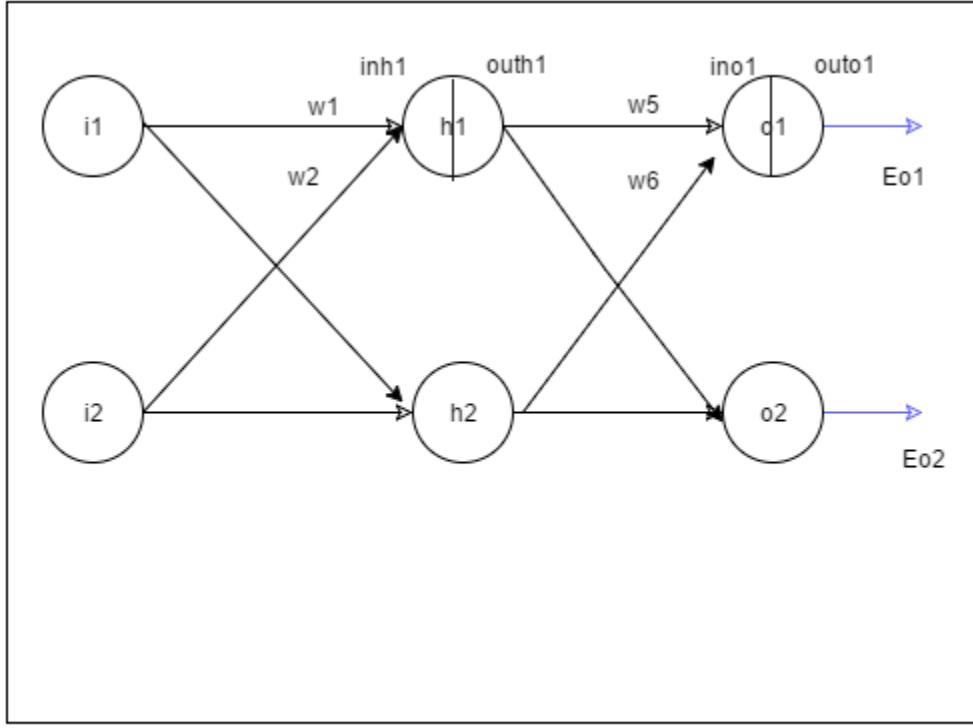
Having computed $\partial E_{total} / \partial w_5$, we now perform gradient descent, by computing a new weight, assuming a learning rate α

$$w_5^+ = w_5 - \alpha * \partial E_{total} / \partial w_5$$

If we do this for $\partial E_{total} / \partial w_6$ we would get

$$\partial E_{total} / \partial w_6 = -(target_{o2} - out_{o2}) * out_{o2} * (1 - out_{o2}) * out_{h2}$$

3) Hidden layer



Text

We now compute how the total error changes for a change in weight w_1
 $\partial E_{total}/\partial w_1 = \partial E_{total}/\partial out_{h1} * \partial out_{h1}/\partial in_{h1} * \partial in_{h1}/\partial w_1 - (C)$

Using

$E_{total} = E_{o1} + E_{o2}$ we get

$$\begin{aligned}\partial E_{total}/\partial w_1 &= (\partial E_{o1}/\partial out_{h1} + \partial E_{o2}/\partial out_{h1}) * \partial out_{h1}/\partial in_{h1} * \partial in_{h1}/\partial w_1 \\ \partial E_{total}/\partial w_1 &= (\partial E_{o1}/\partial out_{h1} + \partial E_{o2}/\partial out_{h1}) * out_{h1} * (1 - out_{h1}) * i_1 \partial w_1 - (D)\end{aligned}$$

Considering the 1st term in (C)

$$\partial E_{total}/\partial out_{h1} = \partial E_{o1}/\partial out_{h1} + \partial E_{o2}/\partial out_{h1}$$

Now

$$\partial E_{o1}/\partial out_{h1} = \partial E_{o1}/\partial out_{o1} * \partial out_{o1}/\partial in_{o1} * \partial in_{o1}/\partial out_{h1} - (E)$$

$$\partial E_{o2}/\partial out_{h1} = \partial E_{o2}/\partial out_{o2} * \partial out_{o2}/\partial in_{o2} * \partial in_{o2}/\partial out_{h1} - (F)$$

which gives the following

$$\partial E_{o1}/\partial out_{o1} * \partial out_{o1}/\partial in_{o1} * \partial in_{o1}/\partial out_{h1} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * w_5$$

$$\partial E_{o2}/\partial out_{o2} * \partial out_{o2}/\partial in_{o2} * \partial in_{o2}/\partial out_{h1} = -(target_{o2} - out_{o2}) * out_{o2}(1 - out_{o2}) * w_6$$

Combining (D), (E) & (F) we get

$$\partial E_{total} / \partial w_1 = -[(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * w_5 + (target_{o2} - out_{o2}) * out_{o2}(1 - out_{o2}) * w_6] * out_{h1} * (1 - out_{h1}) * i_1$$

This can be represented as

$$\partial E_{total} / \partial w_1 = -\sum_i [(target_{oi} - out_{oi}) * out_{oi}(1 - out_{oi}) * w_j] * out_{h1} * (1 - out_{h1}) * i_1$$

With this derivative a new value of w_1 is computed

$$w_1^+ = w_1 - \alpha * \partial E_{total} / \partial w_1$$

Hence there are 2 important results

At the output layer we have

a) $\partial E_{total} / \partial w_j = -(target_{oi} - out_{oi}) * out_{oi} * (1 - out_{oi}) * out_{hi}$

At each hidden layer we compute

b) $\partial E_{total} / \partial w_k = -\sum_i [(target_{oi} - out_{oi}) * out_{oi}(1 - out_{oi}) * w_j] * out_{hk} * (1 - out_{hk}) * i_k$

Backpropagation was very successful in the early years, but the algorithm does have its problems for e.g the issue of the ‘vanishing’ and ‘exploding’ gradient. Yet it is a key development in Neural Networks, and the issues with the backprop gradients have been addressed through techniques such as the momentum method and adaptive learning rate etc.

In this post, I derive the weights at the output layer and the hidden layer. As I already mentioned above, I intend to implement a vector version of the backpropagation algorithm in Octave, R and Python in the days to come.

2. Appendix 1 – Logistic Regression as a Neural Network

1.1 Python functions

```

1 ##########
2 #####
3 #
4 # Created by: Tinniam V Ganesh
5 # Date : 4 Jan 2018
6 # File: DLfunctions1.py
7 #
8 #####
9 #####
10
11 import numpy as np
12 import pandas as pd
13 import os
14 import matplotlib.pyplot as plt
15 from sklearn.model_selection import train_test_split
16
17 # Define the sigmoid function
18 def sigmoid(z):
19     a=1/(1+np.exp(-z))
20     return a
21
22 # Initialize weights and biases
23 def initialize(dim):
24     w = np.zeros(dim).reshape(dim,1)
25     b = 0
26     return w
27
28 # Compute the loss
29 # Inputs: numTraining
30 #          : Y
31 #          : A
32 # Outputs : loss
33 def computeLoss(numTraining,Y,A):
34     Loss=-1/numTraining *np.sum(Y*np.log(A) + (1-Y)*(np.log(1-A)))
35     return Loss
36
37 # Execute the forward propagation
38 # Inputs: w
39 #          : b
40 #          : X
41 #          : Y
42 # Outputs : gradients, loss (dict)
43 def forwardPropagation(w,b,X,Y):
44     # Compute Z
45     Z=np.dot(w.T,X)+b
46     # Determine the number of training samples
47     numTraining=float(len(X))
48     # Compute the output of the sigmoid activation function
49     A=sigmoid(Z)
50     #Compute the loss
51     loss = computeLoss(numTraining,Y,A)
52     # Compute the gradients dz, dw and db
53     dZ=A-Y
54     dw=1/numTraining*np.dot(X,dZ.T)

```

```

55         db=1/numTraining*np.sum(dZ)
56
57     # Return the results as a dictionary
58     gradients = {"dw": dw,
59                   "db": db}
60     loss = np.squeeze(loss)
61     return gradients,loss
62
63 # Compute Gradient Descent
64 # Inputs: w
65 #         : b
66 #         : X
67 #         : Y
68 #         : numIterations
69 #         : learningRate
70 # Outputs : params, grads, losses, idx
71 def gradientDescent(w, b, X, Y, numIterations, learningRate):
72     losses=[]
73     idx =[]
74     # Iterate
75     for i in range(numIterations):
76         gradients,loss=forwardPropagation(w,b,X,Y)
77         #Get the derivates
78         dw = gradients["dw"]
79         db = gradients["db"]
80         w = w-learningRate*dw
81         b = b-learningRate*db
82
83         # Store the loss
84         if i % 100 == 0:
85             idx.append(i)
86             losses.append(loss)
87         # Set params and grads
88         params = {"w": w,
89                   "b": b}
90         grads = {"dw": dw,
91                   "db": db}
92
93     return params, grads, losses,idx
94
95 # Predict the output for a training set
96 # Inputs: w
97 #         : b
98 #         : X
99 # Outputs : yPredicted
100 def predict(w,b,X):
101     size=X.shape[1]
102     yPredicted=np.zeros((1,size))
103     Z=np.dot(w.T,X)
104     # Compute the sigmoid
105     A=sigmoid(Z)
106     for i in range(A.shape[1]):
107         #If the value is > 0.5 then set as 1
108         if(A[0][i] > 0.5):
109             yPredicted[0][i]=1
110         else:
111             # Else set as 0
112             yPredicted[0][i]=0
113
114     return yPredicted
115
116 #Normalize the data
117 # Predict the output for a training set
118 # Inputs: x

```

```

119 # Outputs : x (normalized)
120 def normalize(x):
121     x_norm = None
122     x_norm = np.linalg.norm(x, axis=1, keepdims=True)
123     x= x/x_norm
124     return x

```

1.2 R

```

1 ##########
2 #####
3 #
4 # Created by: Tinniam V Ganesh
5 # Date : 4 Jan 2018
6 # File: DLfunctions1.py
7 #
8 #####
9 #####
10 source("RFunctions-1.R")
11
12 # Define the sigmoid function
13 sigmoid <- function(z){
14     a <- 1/(1+ exp(-z))
15     a
16 }
17
18
19 # Compute the loss
20 # Inputs: numTraining
21 #          : Y
22 #          : A
23 # Outputs : loss
24 computeLoss <- function(numTraining,Y,A){
25     loss <- -1/numTraining* sum(Y*log(A) + (1-Y)*log(1-A))
26     return(loss)
27 }
28
29 # Compute forward propagation
30 # Inputs: w
31 #          : b
32 #          : X
33 #          : Y
34 # Outputs : fwdProp (list)
35 forwardPropagation <- function(w,b,X,Y){
36     # Compute Z
37     Z <- t(w) %*% X +b
38     #Set the number of samples
39     numTraining <- ncol(X)
40     # Compute the activation function
41     A=sigmoid(Z)
42
43     #Compute the loss
44     loss <- computeLoss(numTraining,Y,A)
45
46     # Compute the gradients dz, dw and db
47     dz<-A-Y
48     dw<-1/numTraining * X %*% t(dz)
49     db<-1/numTraining*sum(dz)
50

```

```

51         fwdProp <- list("loss" = loss, "dw" = dw, "db" = db)
52     return(fwdProp)
53 }
54
55 # Perform one cycle of Gradient descent
56 # Compute Gradient Descent
57 # Inputs: w
58 #          : b
59 #          : X
60 #          : Y
61 #          : numIterations
62 #          : learningRate
63 # Outputs : gradDescnt (list)
64 gradientDescent <- function(w, b, X, Y, numIterations, learningRate){
65     losses <- NULL
66     idx <- NULL
67     # Loop through the number of iterations
68     for(i in 1:numIterations){
69         fwdProp <- forwardPropagation(w,b,X,Y)
70         #Get the derivatives
71         dw <- fwdProp$dw
72         db <- fwdProp$db
73         #Perform gradient descent
74         w = w-learningRate*dw
75         b = b-learningRate*db
76         l <- fwdProp$loss
77         # Store the loss
78         if(i %% 100 == 0){
79             idx <- c(idx,i)
80             losses <- c(losses,l)
81         }
82     }
83
84     # Return the weights and losses
85     gradDescnt <-
86     list("w"=w,"b"=b,"dw"=dw,"db"=db,"losses"=losses,"idx"=idx)
87     return(gradDescnt)
88 }
89
90 # Compute the predicted value for input
91 # Predict the output for a training set
92 # Inputs: w
93 #          : b
94 #          : X
95 # Outputs : yPredicted
96 predict <- function(w,b,X){
97     m=dim(X)[2]
98     # Create a vector of 0's
99     yPredicted=matrix(rep(0,m),nrow=1,ncol=m)
100    Z <- t(w) %*% X + b
101    # Compute sigmoid
102    A=sigmoid(Z)
103    for(i in 1:dim(A)[2]){
104        # If A > 0.5 set value as 1
105        if(A[1,i] > 0.5)
106            yPredicted[1,i]=1
107        else
108            # Else set as 0
109            yPredicted[1,i]=0
110    }
111
112    return(yPredicted)
113 }
114

```

```

115 # Normalize the matrix
116 # Predict the output for a training set
117 # Inputs: x
118 # Outputs : normalized
119 normalize <- function(x){
120   #Create the norm of the matrix.Perform the Frobenius norm of the
121   matrix
122   n<-as.matrix(sqrt(rowSums(x^2)))
123   #Sweep by rows by norm. Note '1' in the function which performing on
124   every row
125   normalized<-sweep(x, 1, n, FUN="/")
126   return(normalized)

```

1.3 Octave

```

1 #####
2 #####
3 #
4 # Created by: Tinniam V Ganesh
5 # Date : 4 Jan 2018
6 # File: DLfunctions.m
7 #
8 #####
9 #####
10 1;
11 # Define sigmoid function
12 function a = sigmoid(z)
13   a = 1 ./ (1+ exp(-z));
14 end
15
16 # Compute the loss
17 # Inputs: numTraining
18 #           : Y
19 #           : A
20 # Outputs : loss
21 function loss=computeLoss(numtraining,Y,A)
22   loss = -1/numtraining * sum((Y .* log(A)) + (1-Y) .* log(1-A));
23 end
24
25 # Perform forward propagation
26 # Inputs: w
27 #           : b
28 #           : x
29 #           : Y
30 # Outputs : loss,dw,db,dz
31 function [loss,dw,db,dz] = forwardPropagation(w,b,x,Y)
32   % Compute Z
33   Z = w' * x + b;
34   numtraining = size(x)(1,2);
35   # Compute sigmoid
36   A = sigmoid(Z);
37
38   #Compute loss. Note this is element wise product
39   loss =computeLoss(numtraining,Y,A);
40   # Compute the gradients dz, dw and db
41   dz = A-Y;
42   dw = 1/numtraining* x * dz';
43   db =1/numtraining*sum(dz);
44
45 end

```

```

46
47 # Compute Gradient Descent
48 # Inputs: w
49 #   : b
50 #   : X
51 #   : Y
52 #   : numIterations
53 #   : learningRate
54 # Outputs : w,b,dw,db,losses,index
55 function [w,b,dw,db,losses,index]=gradientDescent(w, b, X, Y,
56 numIterations, learningRate)
57     #Initialize losses and idx
58     losses=[];
59     index=[];
60     # Loop through the number of iterations
61     for i=1:numIterations,
62         [loss,dw,db,dZ] = forwardPropagation(w,b,X,Y);
63         # Perform Gradient descent
64         w = w - learningRate*dw;
65         b = b - learningRate*db;
66         if(mod(i,100) ==0)
67             # Append index and loss
68             index = [index i];
69             losses = [losses loss];
70         endif
71     end
72 end
73
74
75 # Determine the predicted value for dataset
76 # Inputs: w
77 #   : b
78 #   : X
79 # Outputs : yPredicted
80 function yPredicted = predict(w,b,X)
81     m = size(X)(1,2);
82     yPredicted=zeros(1,m);
83     # Compute Z
84     Z = w' * X + b;
85     # Compute sigmoid
86     A = sigmoid(Z);
87     for i=1:size(X)(1,2),
88         # Set predicted as 1 if A > 0.5
89         if(A(1,i) >= 0.5)
90             yPredicted(1,i)=1;
91         else
92             yPredicted(1,i)=0;
93         endif
94     end
95 end
96
97 # Normalize by dividing each value by the sum of squares
98 # Predict the output for a training set
99 # Inputs: x
100 # Outputs : normalized
101 function normalized = normalize(x)
102     # Compute Frobenius norm. Square the elements, sum rows and then find
103     # square root
104     a = sqrt(sum(x .^ 2,2));
105     # Perform element wise division
106     normalized = x ./ a;
107 end
108
109 # Split into train and test sets

```

```
110 # Predict the output for a training set
111 # Inputs: dataset
112 #           :trainPercent
113 # Outputs : x_train,y_train,x_test,y_test
114 function [x_train,y_train,x_test,y_test] =
115 trainTestSplit(dataset,trainPercent)
116     # Create a random index
117     ix = randperm(length(dataset));
118     # Split into training
119     trainSize = floor(trainPercent/100 * length(dataset));
120     train=dataset(ix(1:trainSize),:);
121     # And test
122     test=dataset(ix(trainSize+1:length(dataset)),:);
123     x_train = train(:,1:30);
124     y_train = train(:,31);
125     X_test = test(:,1:30);
126     y_test = test(:,31);
127 end
```

3.Appendix 2 - Implementing a simple Neural Network

2.1 Python

```
1 ######
2 #####
3 #
4 # Created by: Tinniam V Ganesh
5 # Date : 11 Jan 2018
6 # File: DLfunctions.py
7 #
8 #####
9 #####
10 import numpy as np
11 import matplotlib
12 import matplotlib.pyplot as plt
13
14 # Compute the sigmoid of a vector
15 # Inputs: z
16 # Outputs : a
17 def sigmoid(z):
18     a=1/(1+np.exp(-z))
19     return a
20
21 # Compute the model shape given the dataset
22 # Input : X
23 #          Y
24 # Returns: modelParams
25 def getModelShape(X,Y):
26     numTraining= X.shape[1] # No of training examples
27     numFeats=X.shape[0]      # No of input features
28     numHidden=4              # No of units in hidden layer
29     numOutput=Y.shape[0]    # No of output units
30     # Create a dcitionary of values
31
32 modelParams={"numTraining":numTraining,"numFeats":numFeats,"numHidden":numHidden,"numOutput":numOutput}
33     return(modelParams)
34
35
36
37 # Initialize the model
38 # Input : number of features
39 #          number of hidden units
40 #          number of units in output
41 # Returns: weight and bias matrices and vectors
42 def initializeModel(numFeats,numHidden,numOutput):
43     np.random.seed(2)
44     w1=np.random.randn(numHidden,numFeats)*0.01 # Multiply by .01
45     b1=np.zeros((numHidden,1))
46     w2=np.random.randn(numOutput,numHidden)*0.01
47     b2=np.zeros((numOutput,1))
48
49     # Create a dictionary of the neural network parameters
50     nnParameters={'w1':w1,'b1':b1,'w2':w2,'b2':b2}
51     return(nnParameters)
52
53 # Compute the forward propoagation through the neural network
54 # Input : X
```

```

55      # nnParameters
56      # Returns : The Activation of 2nd layer
57      #          : Output and activation of layer 1 & 2
58  def forwardPropagation(X,nnParameters):
59      # Get the parameters
60      w1=nnParameters["w1"]
61      b1=nnParameters["b1"]
62      w2=nnParameters["w2"]
63      b2=nnParameters["b2"]
64
65      # Compute z1 of the input layer
66      z1=np.dot(w1,X)+b1
67      # Compute the output A1 with the tanh activation function. The tanh
activation function
68      # performs better than the sigmoid function
69      A1=np.tanh(z1)
70
71      # Compute z2 of the 2nd layer
72      z2=np.dot(w2,A1)+b2
73      # Compute the output A2 with the tanh activation function. The tanh
activation function
74      # performs better than the sigmoid function
75      A2=sigmoid(z2)
76      cache={'z1':z1,'A1':A1,'z2':z2,'A2':A2}
77      return A2,cache
78
79
80
81      # Compute the cost
82      # Input : A2
83      #          :Y
84      # Output: cost
85  def computeCost(A2,Y):
86      m= float(Y.shape[1])
87      # Element wise multiply for logprobs
88      cost=-1/m *np.sum(Y*np.log(A2) + (1-Y)*(np.log(1-A2)))
89      cost = np.squeeze(cost)
90      return cost
91
92      # Compute the backpropagation for 1 cycle
93      # Input : Neural Network parameters - weights and biases
94      #          # Z and Activations of 2 layers
95      #          # Input features
96      #          # Output values Y
97      # Returns: Gradients
98  def backPropagation(nnParameters, cache, X, Y):
99      numtraining=float(X.shape[1])
100     # Get parameters
101     w1=nnParameters["w1"]
102     w2=nnParameters["w2"]
103
104     #Get the NN cache
105     A1=cache["A1"]
106     A2=cache["A2"]
107
108     # Compute gradients
109     dZ2 = A2 - Y
110     dW2 = 1/numtraining *np.dot(dZ2,A1.T)
111     dB2 = 1/numtraining *np.sum(dZ2, axis=1, keepdims=True)
112     dZ1 = np.multiply(np.dot(w2.T,dZ2), (1 - np.power(A1, 2)))
113     dW1 = 1/numtraining * np.dot(dZ1,X.T)
114     dB1 = 1/numtraining *np.sum(dZ1, axis=1, keepdims=True)
115
116     # Create a dictionary
117     gradients = {"dw1": dW1,
118                  "db1": dB1,

```

```

119         "dw2": dw2,
120         "db2": db2}
121     return gradients
122
123 # Perform Gradient Descent
124 # Input : Weights and biases
125 #           : gradients
126 #           : learning rate
127 #output : Updated weights after 1 iteration
128 def gradientDescent(nnParameters, gradients, learningRate):
129     w1 = nnParameters['w1']
130     b1 = nnParameters['b1']
131     w2 = nnParameters['w2']
132     b2 = nnParameters['b2']
133     dw1 = gradients["dw1"]
134     db1 = gradients["db1"]
135     dw2 = gradients["dw2"]
136     db2 = gradients["db2"]
137     w1 = w1-learningRate*dw1
138     b1 = b1-learningRate*db1
139     w2 = w2-learningRate*dw2
140     b2 = b2-learningRate*db2
141     # Update the Neural Network parameters
142     updatedNNParameters = {"w1": w1,
143                           "b1": b1,
144                           "w2": w2,
145                           "b2": b2}
146
147     return updatedNNParameters
148
149 # Compute the Neural Network by minimizing the cost
150 # Input : Input data X,
151 #           Output Y
152 #           No of hidden units in hidden layer
153 #           No of iterations
154 # Returns Updated weight and bias vectors of the neural network
155 def computeNN(X, Y, numHidden, learningRate, numIterations = 10000):
156     np.random.seed(3)
157     modelParams = getModelShape(X, Y)
158     numFeats=modelParams['numFeats']
159     numOutput=modelParams['numOutput']
160
161     costs=[]
162
163     nnParameters = initializeModel(numFeats,numHidden,numOutput)
164     w1 = nnParameters['w1']
165     b1 = nnParameters['b1']
166     w2 = nnParameters['w2']
167     b2 = nnParameters['b2']
168     # Perform gradient descent
169     for i in range(0, numIterations):
170         # Evaluate forward prop to compute activation at output layer
171         A2, cache = forwardPropagation(X, nnParameters)
172         # Compute cost from Activation at output and Y
173         cost = computeCost(A2, Y)
174         # Perform backprop to compute gradients
175         gradients = backPropagation(nnParameters, cache, X, Y)
176         # Use gradients to update the weights for each iteration.
177         nnParameters = gradientDescent(nnParameters, gradients,learningRate)
178         # Print the cost every 1000 iterations
179         if i % 1000 == 0:
180             costs.append(cost)
181             print ("Cost after iteration %i: %f" %(i, cost))
182     return nnParameters,costs

```

```

183
184 # Compute the predicted value for a given input
185 # Input : Neural Network parameters
186 #           : Input data
187 def predict(nnParameters, X):
188     A2, cache = forwardPropagation(X, nnParameters)
189     predictions = (A2>0.5)
190     return predictions
191
192 # Plot a decision boundary
193 # Input : Input Model,
194 #           X
195 #           Y
196 #           Fig to be saved as
197 # Returns Null
198 def plot_decision_boundary_n(model, X, y, fig):
199     # Set min and max values and give it some padding
200     x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
201     y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
202     colors=['black', 'gold']
203     cmap = matplotlib.colors.ListedColormap(colors)
204     h = 0.01
205     # Generate a grid of points with distance h between them
206     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,
207     h))
208     # Predict the function value for the whole grid
209     Z = model(np.c_[xx.ravel(), yy.ravel()])
210     Z = Z.reshape(xx.shape)
211     # Plot the contour and training examples
212     plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
213     plt.ylabel('x2')
214     plt.xlabel('x1')
215     plt.scatter(X[0, :], X[1, :], c=y, cmap=cmap)
216     plt.title("Decision Boundary for logistic regression")
217     plt.savefig(fig, bbox_inches='tight')
218
219 # Plot a decision boundary
220 # Input : Input Model,
221 #           X
222 #           Y
223 #           sz - Num of hidden units
224 #           lr - Learning rate
225 #           Fig to be saved as
226 # Returns Null
227 def plot_decision_boundary(model, X, y, sz, lr, fig):
228     # Set min and max values and give it some padding
229     x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
230     y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
231     colors=['black', 'gold']
232     cmap = matplotlib.colors.ListedColormap(colors)
233     h = 0.01
234     # Generate a grid of points with distance h between them
235     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,
236     h))
237     # Predict the function value for the whole grid
238     Z = model(np.c_[xx.ravel(), yy.ravel()])
239     Z = Z.reshape(xx.shape)
240     # Plot the contour and training examples
241     plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
242     plt.ylabel('x2')
243     plt.xlabel('x1')
244     plt.scatter(X[0, :], X[1, :], c=y, cmap=cmap)
245     plt.title("Decision Boundary for hidden layer size:" + sz + " and learning
246     rate:" + lr)

```

```
247     plt.savefig(fig, bbox_inches='tight')
248
```

2.2 R

```
1 ##########
2 #####
3 #
4 # Created by: Tinniam V Ganesh
5 # Date : 11 jan 2018
6 # File: DLfunctions2_1.R
7 #
8 #####
9 #####
10 library(ggplot2)
11
12 # Sigmoid function
13 sigmoid <- function(z){
14     a <- 1/(1+ exp(-z))
15     a
16 }
17
18 # Compute the model shape given the dataset
19 # Input : X - features
20 #          Y - output
21 #
22 # Returns: no of training samples, no features, no hidden, no output
23 getModelState <- function(X,Y){
24     numTraining <- dim(X)[2] # No of training examples
25     numFeats <- dim(X)[1]    # No of input features
26     numHidden<-4           # No of units in hidden layer
27     # If Y is a row vector set numOutput as 1
28     if(is.null(dim(Y)))
29         numOutput <-1      # No of output units
30     else
31         numOutput <- dim(Y)[1]
32     # Create a list of values
33     modelParams <-
34     list("numTraining"=numTraining,"numFeats"=numFeats,"numHidden"=numHidden,
35          "numOutput"=numOutput)
36     return(modelParams)
37 }
38
39
40 # Initialize the model
41 # Input : number of features
42 #          number of hidden units
43 #          number of units in output
44 # Returns: Weight and bias matrices and vectors
45 initializeModel <- function(numFeats,numHidden,numOutput){
46     set.seed(2)
47     w= rnorm(numHidden*numFeats)*0.01
48     w1<-matrix(w,nrow=numHidden,ncol=numFeats) # Multiply by .01
49     b1<-matrix(rep(0,numHidden),nrow=numHidden,ncol=1)
50     w= rnorm(numOutput*numHidden)
51     W2<-matrix(w,nrow=numOutput,ncol=numHidden)
52     b2<- matrix(rep(0,numOutput),nrow=numOutput,ncol=1)
53
54     # Create a list of the neural network parameters
55     nnParameters<- list('W1'=w1,'b1'=b1,'W2'=W2,'b2'=b2)
```

```

56     return(nnParameters)
57 }
58
59
60 #Compute the forward propagation through the neural network
61 # Input : Features
62 #          weight and bias matrices and vectors
63 # Returns : The Activation of 2nd layer
64 #          : Output and activation of layer 1 & 2
65
66 forwardPropagation <- function(X,nnParameters){
67   # Get the parameters
68   W1<-nnParameters$w1
69   b1<-nnParameters$b1
70   W2<-nnParameters$w2
71   b2<-nnParameters$b2
72
73   z <- W1 %*% X
74
75   # Broadcast the bias vector for each row. Use 'sweep' The value '1' for
76 MARGIN
77   # indicates sweep each row by this value( add a column vector to each
78 row).
79   # If we want to sweep by column use '2'for MARGIN. Here a row vector is
80 added to
81   # column (braodcasting!)
82
83   Z1 <-sweep(z,1,b1,'+')
84   # Compute the output A1 with the tanh activation function. The tanh
85 activation function
86   # performs better than the sigmoid function
87
88   A1<-tanh(Z1)
89
90   # Compute z2 of the 2nd layer
91   z <- W2 %*% A1
92   # Broadcast the bias vector for each row. Use 'sweep'
93   Z2 <- sweep(z,1,b2,'+')
94   # Compute the output A1 with the tanh activation function. The tanh
95 activation function
96   # performs better than the sigmoid function
97   A2<-sigmoid(Z2)
98   cache <- list('Z1'=Z1,'A1'=A1,'Z2'=Z2,'A2'=A2)
99   return(list('A2'=A2, 'cache'=cache))
100 }
101
102 # Compute the cost
103 # Input : Activation of 2nd layer
104 #          : Output from data
105 # Output: cost
106 computeCost <- function(A2,Y){
107   m= length(Y)
108   cost=-1/m*sum(Y*log(A2) + (1-Y)*log(1-A2))
109   #cost=-1/m*sum(a+b)
110   return(cost)
111 }
112
113
114 # Compute the backpropoagation for 1 cycle
115 # Input : Neural Network parameters - weights and biases
116 #          # Z and Activations of 2 layers
117 #          # Input features
118 #          # Output values Y
119 # Returns: Gradients

```

```

120 backPropagation <- function(nnParameters, cache, X, Y){
121   numtraining<- dim(X)[2]
122   # Get parameters
123   w1<-nnParameters$w1
124   w2<-nnParameters$w2
125
126   #Get the NN cache
127   A1<-cache$A1
128   A2<-cache$A2
129
130
131   dz2 <- A2 - Y
132   dw2 <- 1/numtraining * dz2 %*% t(A1)
133   db2 <- 1/numtraining * rowSums(dz2)
134   dz1 <- t(w2) %*% dz2 * (1 - A1^2)
135   dw1 = 1/numtraining* dz1 %*% t(X)
136   db1 = 1/numtraining * rowSums(dz1)
137
138   gradients <- list("dw1"= dw1, "db1"= db1, "dw2"= dw2, "db2"= db2)
139   return(gradients)
140 }
141
142
143
144 # Gradient descent
145 # Perform Gradient Descent
146 # Input : Weights and biases
147 #           : gradients
148 #           : learning rate
149 #output : Updated weights after 1 iteration
150 gradientDescent <- function(nnParameters, gradients, learningRate){
151   w1 <- nnParameters$w1
152   b1 <- nnParameters$b1
153   w2 <- nnParameters$w2
154   b2 <- nnParameters$b2
155   dw1<-gradients$dw1
156   db1 <- gradients$db1
157   dw2 <- gradients$dw2
158   db2 <-gradients$db2
159   w1 <- w1-learningRate*dw1
160   b1 <- b1-learningRate*db1
161   w2 <- w2-learningRate*dw2
162   b2 <- b2-learningRate*db2
163   updatedNNParameters <- list("w1"= w1, "b1"= b1, "w2"= w2, "b2"= b2)
164   return(updatedNNParameters)
165 }
166
167 # Compute the Neural Network by minimizing the cost
168 # Input : Input data X,
169 #           Output Y
170 #           No of hidden units in hidden layer
171 #           No of iterations
172 # Returns Updated weight and bias vectors of the neural network
173 computeNN <- function(X, Y, numHidden, learningRate, numIterations = 10000){
174
175   modelParams <- getModelShape(X, Y)
176   numFeats<-modelParams$numFeats
177   numOutput<-modelParams$numOutput
178   costs=NULL
179   nnParameters <- initializeModel(numFeats,numHidden,numOutput)
180   w1 <- nnParameters$w1
181   b1<-nnParameters$b1
182   w2<-nnParameters$w2
183   b2<-nnParameters$b2

```

```

184     # Perform gradient descent
185     for(i in 0: numIterations){
186
187         # Evaluate forward prop to compute activation at output layer
188         #print("Here")
189         fwdProp = forwardPropagation(x, nnParameters)
190         # Compute cost from Activation at output and Y
191         cost = computeCost(fwdProp$A2, Y)
192         # Perform backprop to compute gradients
193         gradients = backPropagation(nnParameters, fwdProp$cache, x, Y)
194         # Use gradients to update the weights for each iteration.
195         nnParameters = gradientDescent(nnParameters, gradients, learningRate)
196         # Print the cost every 1000 iterations
197         if(i%1000 == 0){
198             costs=c(costs,cost)
199             print(cost)
200         }
201     }
202
203     nnVals <- list("nnParameter"=nnParameters, "costs"=costs)
204     return(nnVals)
205 }
206 # Predict the output
207 predict <- function(parameters, x){
208
209     fwdProp <- forwardPropagation(x, parameters)
210     predictions <- fwdProp$A2>0.5
211
212     return (predictions)
213 }
214
215 # Plot a decision boundary
216 # This function uses the contour method
217 drawBoundary <- function(z,nn){
218     # Find the minimum and maximum of the 2 fatures
219     xmin<-min(z[,1])
220     xmax<-max(z[,1])
221     ymin<-min(z[,2])
222     ymax<-max(z[,2])
223
224     a=seq(xmin,xmax,length=100)
225     b=seq(ymin,ymax,length=100)
226     grid <- expand.grid(x=a, y=b)
227     grid1 <- t(grid)
228     q <-predict(nn$nnParameter,grid1)
229     # Works
230     contour(a, b, z=matrix(q, nrow=100), levels=0.5,
231             col="black", drawlabels=FALSE, lwd=2,xlim=range(2,10))
232     points(z[,1],z[,2],col=ifelse(z[,3]==1, "coral",
233     "cornflowerblue"),pch=18)
234 }
235
236 # Plot a decision boundary
237 # This function uses ggplot2
238 plotDecisionBoundary <- function(z,nn,sz,lr){
239     xmin<-min(z[,1])
240     xmax<-max(z[,1])
241     ymin<-min(z[,2])
242     ymax<-max(z[,2])
243
244     a=seq(xmin,xmax,length=100)
245     b=seq(ymin,ymax,length=100)
246     grid <- expand.grid(x=a, y=b)

```

```

248     colnames(grid) <- c('x1', 'x2')
249     grid1 <- t(grid)
250     q <- predict(nn$nnParameter,grid1)
251     q1 <- t(data.frame(q))
252     q2 <- as.numeric(q1)
253     grid2 <- cbind(grid,q2)
254     colnames(grid2) <- c('x1', 'x2','q2')
255
256     z1 <- data.frame(z)
257     names(z1) <- c("x1","x2","y")
258     atitle=paste("Decision boundary for hidden layer size:",sz,"learning
259 rate:",lr)
260     ggplot(z1) +
261       geom_point(data = z1, aes(x = x1, y = x2, color = y)) +
262       stat_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2),
263 alpha = 0.9)+
264       ggtile(atitle)
265   }
266
267 # Plot a decision boundary
268 # This function uses ggplot2 and stat_contour
269 plotBoundary <- function(z,nn){
270   xmin<-min(z[,1])
271   xmax<-max(z[,1])
272   ymin<-min(z[,2])
273   ymax<-max(z[,2])
274
275
276   a=seq(xmin,xmax,length=100)
277   b=seq(ymin,ymax,length=100)
278   grid <- expand.grid(x=a, y=b)
279   colnames(grid) <- c('x1', 'x2')
280   grid1 <- t(grid)
281   q <- predict(nn$nnParameter,grid1)
282   q1 <- t(data.frame(q))
283   q2 <- as.numeric(q1)
284   grid2 <- cbind(grid,q2)
285   colnames(grid2) <- c('x1', 'x2','q2')
286
287   z1 <- data.frame(z)
288   names(z1) <- c("x1","x2","y")
289   data.plot <- ggplot() +
290     geom_point(data = z1, aes(x = x1, y = x2, color = y)) +
291     coord_fixed() +
292
293     xlab('x1') +
294     ylab('x2')
295   print(data.plot)
296
297   data.plot + stat_contour(data = grid2, aes(x = x1, y = x2, z = q2), alpha
298 = 0.9)
299 }
300

```

2.3 Octave

```

1 #####
2 #####

```

```

3  #
4  # Created by: Tinniam V Ganesh
5  # Date : 11 Jan 2018
6  # File: DLfunctions2.m
7  #
8 #####
9 #####
10 1;
11 # Define sigmoid function
12 function a = sigmoid(z)
13     a = 1 ./ (1+ exp(-z));
14 end
15
16 # Compute the loss
17 # Inputs: numTraining
18 #           : Y
19 #           : A
20 # Outputs : loss
21 function loss=computeLoss(numtraining,Y,A)
22     loss = -1/numtraining * sum((Y .* log(A)) + (1-Y) .* log(1-A));
23 end
24
25 # Compute the model shape given the dataset
26 # Inputs: X
27 #           : Y
28 # Outputs : n_x,m,n_h,n_y
29 function [n_x,m,n_h,n_y] = getModelShape(X,Y)
30     m= size(X)(2);
31     n_x=size(X)(1);
32     n_h=4;
33     n_y=size(Y)(1);
34 end
35
36 # Initialize model
37 # Inputs: n_x
38 #           : n_h
39 #           : n_y
40 # Outputs : w1,b1,w2,b2
41 function [w1,b1,w2,b2] = modelInit(n_x,n_h,n_y)
42     rand ("seed", 2);
43     w1=rand(n_h,n_x)*0.01; # set the initial values to a small number
44     b1=zeros(n_h,1);
45     w2=rand(n_y,n_h)*0.01;
46     b2=zeros(n_y,1);
47
48 end
49
50 # Compute the forward propagation through the neural network
51 # Input : Features
52 #           weight and bias matrices and vectors
53 # Returns : The Activation of 2nd layer
54 #           : Output and activation of layer 1 & 2
55 function [Z1,A1,Z2,A2]= forwardPropagation(X,w1,b1,w2,b2)
56     # Get the parameters
57
58     # Determine the number of training samples
59     m=size(X)(2);
60     # Compute Z1 of the input layer
61     # Octave also handles broadcasting like Python!!
62     Z1=w1 * X +b1;
63     # Compute the output A1 with the tanh activation function. The tanh
64     # activation function
65     # performs better than the sigmoid function
66     A1=tanh(Z1);

```

```

67
68    # Compute Z2 of the input layer
69    Z2=W2 * A1+b2;
70    # Compute the output A1 with the tanh activation function. The tanh
71 activation function
72    # performs better than the sigmoid function
73    A2=sigmoid(Z2);
74
75 end
76
77 # Compute the cost
78 # Input : Activation of 2nd layer
79 #           : Output from data
80 # Output: cost
81 function [cost] = computeCost(A,Y)
82     numTraining= size(Y)(2);
83     # Element wise multiply for logprobs
84     cost = -1/numTraining * sum((Y .* log(A)) + (1-Y) .* log(1-A));
85 end
86
87
88 # Compute the backpropagation for 1 cycle
89 # Input : Neural Network parameters - weights and biases
90 #           # Z and Activations of 2 layers
91 #           # Input features
92 #           # Output values Y
93 # Returns: Gradients
94 function [dw1,db1,dw2,db2]= backPropagation(w1,w2,A1,A2, X, Y)
95     numTraining=size(X)(2);
96
97     dz2 = A2 - Y;
98     dw2 = 1/numTraining * dz2 * A1';
99     db2 = 1/numTraining * sum(dz2);
100
101    dz1 = w2' * dz2 .* (1 - power(A1, 2));
102    dw1 = 1/numTraining * dz1 * X';
103    # Note the '2' in the next statement indicates that a row sum has to done
104 , 2nd dimension
105    db1 = 1/numTraining * sum(dz1,2);
106
107 end
108
109 # Perform Gradient Descent
110 # Input : Weights and biases
111 #           : gradients
112 #           : learning rate
113 #output : Updated weights after 1 iteration
114 function [w1,b1,w2,b2]= gradientDescent(w1,b1,w2,b2, dw1,db1,dw2,db2,
115 learningRate)
116     w1 = w1-learningRate*dw1;
117     b1 = b1-learningRate*db1;
118     w2 = w2-learningRate*dw2;
119     b2 = b2-learningRate*db2;
120 end
121
122
123 # Compute the Neural Network by minimizing the cost
124 # Input : Input data X,
125 #           Output Y
126 #           No of hidden units in hidden layer
127 #           No of iterations
128 # Returns Updated weight and bias vectors of the neural network
129 #function [w1,b1,w2,b2,costs]= computeNN(X, Y,numHidden, learningRate,
130 numIterations = 10000)
```

```

131
132 [numFeats,numTraining,n_h,numOutput] = getModelShape(X, Y);
133
134 costs=[];
135
136 [w1,b1,w2,b2] = modelInit(numFeats,numHidden,numOutput) ;
137 #w1 =[-0.00416758, -0.00056267; -0.02136196, 0.01640271; -0.01793436, -
138 0.00841747; 0.00502881 -0.01245288];
139 #W2=[-0.01057952, -0.00909008, 0.00551454, 0.02292208];
140 #b1=[0;0;0;0];
141 #b2=[0];
142 # Perform gradient descent
143 for i =0:numIterations
144     # Evaluate forward prop to compute activation at output layer
145     [Z1,A1,Z2,A2] = forwardPropagation(X, w1,b1,w2,b2);
146     # Compute cost from Activation at output and Y
147     cost = computeCost(A2, Y);
148     # Perform backprop to compute gradients
149     [dw1,db1,dw2,db2] = backPropagation(w1,W2,A1,A2, X, Y);
150     # Use gradients to update the weights for each iteration.
151
152     [w1,b1,w2,b2] = gradientDescent(w1,b1,w2,b2,
153 dw1,db1,dw2,db2,learningRate);
154     # Print the cost every 1000 iterations
155     if ( mod(i,1000) == 0)
156         costs =[costs cost];
157         #disp ("Cost after iteration"), disp(i),disp(cost);
158     endif
159 endfor
160 end
161
162 # Compute the predicted value for a given input
163 # Input : Neural Network parameters
164 #          : Input data
165 # Output : predictions
166 function [predictions]= predict(w1,b1,w2,b2, x)
167     [Z1,A1,Z2,A2] = forwardPropagation(X, w1,b1,w2,b2);
168     predictions = (A2>0.5);
169 end
170
171 # Plot the decision boundary
172 function plotDecisionBoundary(data,w1,b1,w2,b2)
173     %Plot a non-linear decision boundary learned by the SVM
174     colormap ("default");
175
176     % Make classification predictions over a grid of values
177     x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)';
178     x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)';
179     [X1, X2] = meshgrid(x1plot, x2plot);
180     vals = zeros(size(X1));
181     # Plot the prediction for the grid
182     for i = 1:size(X1, 2)
183         gridPoints = [X1(:, i), X2(:, i)];
184         vals(:, i)=predict(w1,b1,w2,b2,gridPoints');
185     endfor
186
187     scatter(data(:,1),data(:,2),8,c=data(:,3),"filled");
188     % Plot the boundary
189     hold on
190     #contour(X1, X2, vals, [0 0], 'LineWidth', 2);
191     contour(X1, X2, vals);
192     title ({"3 layer Neural Network decision boundary"});
193     hold off;
194

```

```

195 end
196
197 # Plot the cost vs iterations
198 function plotLRCostVsIterations()
199     data=csvread("data.csv");
200
201     X=data(:,1:2);
202     Y=data(:,3);
203     lr=[0.5,1.2,3]
204     col='kmb'
205     for i=1:3
206         [W1,b1,W2,b2,costs]= computeNN(X', Y',4, learningRate=lr(i),
207 numIterations = 10000);
208         iterations = 1000*[0:10];
209         hold on;
210         plot(iterations,costs,color=col(i),"linewidth", 3);
211         hold off;
212         title ("Cost vs no of iterations for different learning rates");
213         xlabel("No of iterations")
214         ylabel("Cost")
215         legend('0.5','1.2','3.0')
216     endfor
217 end
218
219 # Plot the cost vs number of hidden units
220 function plotHiddenCostVsIterations()
221     data=csvread("data1.csv");
222
223     X=data(:,1:2);
224     Y=data(:,3);
225     hidden=[4,9,12]
226     col='kmb'
227     for i=1:3
228         [W1,b1,W2,b2,costs]= computeNN(X', Y',hidden(i), learningRate=1.5,
229 numIterations = 10000);
230         iterations = 1000*[0:10];
231         hold on;
232         plot(iterations,costs,color=col(i),"linewidth", 3);
233         hold off;
234         title ("Cost vs no of iterations for different number of hidden
235 units");
236         xlabel("No of iterations")
237         ylabel("Cost")
238         legend('4','9','12')
239     endfor
240 end

```

4.Appendix 3 - Building a L-Layer Deep Learning Network

3.1 Python

```
1 # -*- coding: utf-8 -*-
2 """
3 ######
4 ######
5 #
6 # File: DLfunctions34.py
7 # Developer: Tinniam V Ganesh
8 # Date : 30 Jan 2018
9 #
10#####
11#####
12@author: Ganesh
13"""
14
15 import numpy as np
16 import matplotlib.pyplot as plt
17 import matplotlib
18 import matplotlib.pyplot as plt
19 from matplotlib import cm
20
21
22 # Compute the sigmoid of a vector. Also return Z
23 def sigmoid(Z):
24     A=1/(1+np.exp(-Z))
25     cache=Z
26     return A,cache
27
28 # Compute the Relu of a vector
29 def relu(Z):
30     A = np.maximum(0,Z)
31     cache=Z
32     return A,cache
33
34 # Compute the tanh of a vector
35 def tanh(Z):
36     A = np.tanh(Z)
37     cache=Z
38     return A,cache
39
40 # Compute the derivative of Relu
41 # g'(z) = 1 if z >0 and 0 otherwise
42 def reluDerivative(dA, cache):
```

```

44     Z = cache
45     dZ = np.array(dA, copy=True) # just converting dz to a correct object.
46     # When z <= 0, you should set dz to 0 as well.
47     dZ[Z <= 0] = 0
48     return dZ
49
50 # Compute the derivative of sigmoid
51 # Derivative g'(z) = a*(1-a)
52 def sigmoidDerivative(dA, cache):
53     Z = cache
54     s = 1/(1+np.exp(-Z))
55     dZ = dA * s * (1-s)
56     return dZ
57
58 # Compute the derivative of tanh
59 # Derivative g'(z) = 1 - a^2
60 def tanhDerivative(dA, cache):
61     Z = cache
62     a = np.tanh(Z)
63     dZ = dA * (1 - np.power(a, 2))
64     return dZ
65
66
67 # Initialize the model
68 # Input : number of features
69 #          number of hidden units
70 #          number of units in output
71 # Returns: weight and bias matrices and vectors
72 def initializeModel(numFeats,numHidden,numOutput):
73     np.random.seed(1)
74     W1=np.random.randn(numHidden,numFeats)*0.01 # Multiply by .01
75     b1=np.zeros((numHidden,1))
76     W2=np.random.randn(numOutput,numHidden)*0.01
77     b2=np.zeros((numOutput,1))
78
79     # Create a dictionary of the neural network parameters
80     nnParameters={'W1':W1,'b1':b1,'W2':W2,'b2':b2}
81     return(nnParameters)
82
83
84 # Initialize model for L layers
85 # Input : List of units in each layer
86 # Returns: Initial weights and biases matrices for all layers
87 def initializeDeepModel(layerDimensions):
88     np.random.seed(3)
89     # note the weight matrix at layer '1' is a matrix of size (1,1-1)
90     # The Bias is a vectors of size (1,1)
91
92     # Loop through the layer dimension from 1.. L. Initialize an empty
93     # dictionary
94     layerParams = {}
95     for l in range(1,len(layerDimensions)):
96         # Append to dictionary
97         layerParams['w' + str(l)] =
98         np.random.randn(layerDimensions[l],layerDimensions[l-1])*0.01 # Multiply by
99         .01
100        layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1))
101
102    return(layerParams)
103
104
105 # Compute the activation at a layer 'l' for forward prop in a Deep Network
106 # Input : A_prec - Activation of previous layer
107 #          w,b - weight and bias matrices and vectors

```

```

108 # activationFunc - Activation function - sigmoid, tanh, relu etc
109 # Returns : The Activation of this layer
110 #
111 # Z = W * X + b
112 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
113 def layerActivationForward(A_prev, w, b, activationFunc):
114
115     # Compute Z
116     Z = np.dot(w,A_prev) + b
117     forward_cache = (A_prev, w, b)
118     # Compute the activation for sigmoid
119     if activationFunc == "sigmoid":
120         A, activation_cache = sigmoid(Z)
121     # Compute the activation for Relu
122     elif activationFunc == "relu":
123         A, activation_cache = relu(Z)
124     # Compute the activation for tanh
125     elif activationFunc == 'tanh':
126         A, activation_cache = tanh(Z)
127     # Cache the forward-cache and activation_cache
128     cache = (forward_cache, activation_cache)
129     return A, cache
130
131
132
133 # Compute the forward propagation for layers 1..L
134 # Input : X - Input Features
135 #           paramaters: Weights and biases
136 #           hiddenActivationFunc - Activation function at hidden layers
137 Relu/tanh
138 # Returns : AL
139 #           caches
140 # The forward propagation uses the Relu/tanh activation from layer 1..L-1 and
141 # sigmoid actiovation at layer L
142 def forwardPropagationDeep(X, parameters,hiddenActivationFunc='relu'):
143     caches = []
144     # Set A to X (A0)
145     A = X
146     L = int(len(parameters)/2) # number of layers in the neural network
147     # Loop through from layer 1 to upto layer L
148     for l in range(1, L):
149         A_prev = A
150         A, cache = layerActivationForward(A_prev, parameters['w'+str(l)],
151 parameters['b'+str(l)], activationFunc = hiddenActivationFunc)
152         caches.append(cache)
153
154     # Since this is binary classification use the sigmoid activation function
155     in
156     # last layer
157     AL, cache = layerActivationForward(A, parameters['w'+str(L)],
158 parameters['b'+str(L)], activationFunc = "sigmoid")
159     caches.append(cache)
160
161     return AL, caches
162
163
164 # Compute the cost
165 # Input : Activation of last layer
166 #           : Output from data
167 # Output: cost
168 def computeCost(AL,Y):
169     m= float(Y.shape[1])
170     # Element wise multiply for logprobs
171     cost=-1/m *np.sum(Y*np.log(AL) + (1-Y)*(np.log(1-AL)))

```

```

172     cost = np.squeeze(cost)
173     return cost
174
175 # Compute the backpropagation for through 1 layer
176 # Input : Neural Network parameters - dA
177 #         # cache - forward_cache & activation_cache
178 #         # Input features
179 #         # Output values Y
180 # Returns: Gradients
181 # dL/dwi= dL/dzi*A1-1
182 # dL/db1 = dL/dz1
183 # dL/dz_prev=dL/dz1*w
184 def layerActivationBackward(dA, cache, activationFunc):
185     forward_cache, activation_cache = cache
186
187     # Compute derivative based on activation function
188     if activationFunc == "relu":
189         dz = reluDerivative(dA, activation_cache)
190     elif activationFunc == "sigmoid":
191         dz = sigmoidDerivative(dA, activation_cache)
192     elif activationFunc == "tanh":
193         dz = tanhDerivative(dA, activation_cache)
194
195     # Compute gradients
196     A_prev, w, b = forward_cache
197     numtraining = float(A_prev.shape[1])
198     dw = 1/numtraining *(np.dot(dz,A_prev.T))
199     db = 1/numtraining * np.sum(dz, axis=1, keepdims=True)
200     dA_prev = np.dot(w.T,dz)
201     return dA_prev, dw, db
202
203 # Compute the backpropagation for 1 cycle, through all layers
204 # Input : AL: Output of L layer Network - weights
205 #         # Y Real output
206 #         # caches -- list of caches containing:
207 #             every cache of layerActivationForward() with "relu"/"tanh"
208 #             #(it's caches[], for l in range(L-1) i.e l = 0...L-2)
209 #             #the cache of layerActivationForward() with "sigmoid" (it's caches[L-
210 1])
211 #             hiddenActivationFunc - Activation function at hidden layers
212 #
213 # Returns:
214 #     gradients -- A dictionary with the gradients
215 #                 gradients["dA" + str(l)] = ...
216 #                 gradients["dw" + str(l)] = ...
217
218 def backwardPropagationDeep(AL, Y, caches,hiddenActivationFunc='relu'):
219     #initialize the gradients
220     gradients = {}
221
222     # Set the number of layers
223     L = len(caches)
224     m = float(AL.shape[1])
225     Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL
226
227     # Initializing the backpropagation
228     # dL/dAL= -(y/a + (1-y)/(1-a)) - At the output layer
229     dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
230
231     # Since this is a binary classification the activation at output is
232     sigmoid
233     # Get the gradients at the last layer
234     current_cache = caches[L-1]

```

```

235     gradients["dA" + str(L)], gradients["dw" + str(L)], gradients["db" +
236     str(L)] = layerActivationBackward(dAL, current_cache, activationFunc =
237     "sigmoid")
238
239     # Traverse in the reverse direction
240     for l in reversed(range(L-1)):
241         # Compute the gradients for L-1 to 1 for Relu/tanh
242         current_cache = caches[l]
243         dA_prev_temp, dw_temp, db_temp =
244     layerActivationBackward(gradients['dA'+str(l+2)], current_cache,
245     activationFunc = hiddenActivationFunc)
246         gradients["dA" + str(l + 1)] = dA_prev_temp
247         gradients["dw" + str(l + 1)] = dw_temp
248         gradients["db" + str(l + 1)] = db_temp
249
250
251     return gradients
252
253 # Perform Gradient Descent
254 # Input : Weights and biases
255 #       : gradients
256 #       : Learning rate
257 #output : Updated weights after 1 iteration
258 def gradientDescent(parameters, gradients, learningRate):
259
260     L = len(parameters) / 2
261
262     # Update rule for each parameter.
263     for l in range(L):
264         parameters["w" + str(l+1)] = parameters['w'+str(l+1)] -learningRate*
265     gradients['dw' + str(l+1)]
266         parameters["b" + str(l+1)] = parameters['b'+str(l+1)] -learningRate*
267     gradients['db' + str(l+1)]
268
269     return parameters
270
271
272 # Execute a L layer Deep learning model
273 # Input : X - Input features
274 #       : Y output
275 #       : layersDimensions - Dimension of layers
276 #       : hiddenActivationFunc - Activation function at hidden layer relu
277 /tanh
278 #       : learning rate
279 #       : num of iterations
280 #output : Updated weights and biases
281
282 def L_Layer_DeepModel(X, Y, layersDimensions, hiddenActivationFunc='relu',
283 learning_rate = .3, num_iterations = 10000, fig="figx.png"):#lr was 0.009
284
285     np.random.seed(1)
286     costs = []
287
288     #Initialize parameters
289     parameters = initializeDeepModel(layersDimensions)
290
291     # Perform gradient descent
292     for i in range(0, num_iterations):
293         # Perform one cycle of forward propagation
294         AL, caches = forwardPropagationDeep(X,
295     parameters,hiddenActivationFunc)
296
297         # Compute cost.
298         cost = computeCost(AL, Y)

```

```

299
300     # Compute gradients through 1 cycle of backprop
301     gradients = backwardPropagationDeep(AL, Y,
302     caches,hiddenActivationFunc)
303
304     # update parameters.
305     parameters = gradientDescent(parameters, gradients, learning_rate)
306
307     # Store the costs
308     if i % 100 == 0:
309         print ("Cost after iteration %i: %f" %(i, cost))
310     if i % 100 == 0:
311         costs.append(cost)
312
313     # plot the cost
314     fig1=plt.plot(np.squeeze(costs))
315     fig1=plt.ylabel('cost')
316     fig1=plt.xlabel('No of iterations(per 100)')
317     fig1=plt.title("Learning rate =" + str(learning_rate))
318     #plt.show()
319     fig1.figure.savefig(fig,bbox_inches='tight')
320     plt.clf()
321
322     return parameters
323
324
325 # Plot a decision boundary
326 # Input : Input Model,
327 #          X
328 #          Y
329 #          sz - Num of hiden units
330 #          lr - Learning rate
331 #          Fig to be saved as
332 # Returns Null
333 def plot_decision_boundary(model, x, y,lr,fig):
334     # Set min and max values and give it some padding
335     x_min, x_max = x[0, :].min() - 1, x[0, :].max() + 1
336     y_min, y_max = x[1, :].min() - 1, x[1, :].max() + 1
337     colors=['black','yellow']
338     cmap = matplotlib.colors.ListedColormap(colors)
339     h = 0.01
340     # Generate a grid of points with distance h between them
341     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,
342     h))
343     # Predict the function value for the whole grid
344     Z = model(np.c_[xx.ravel(), yy.ravel()])
345     Z = Z.reshape(xx.shape)
346     # Plot the contour and training examples
347     fig2=plt.contourf(xx, yy, Z, cmap="coolwarm")
348     fig2=plt.ylabel('x2')
349     fig2=plt.xlabel('x1')
350     fig2=plt.scatter(x[0, :], x[1, :], c=y, s=7,cmap=cmap)
351     fig2=plt.title("Decision Boundary for learning rate:"+lr)
352     fig2.figure.savefig(fig, bbox_inches='tight')
353     plt.clf()
354
355     # Predict the output for given input
356     # Input : parameters
357     #          : X
358     # Output: predictions
359     def predict(parameters, x):
360         A2, cache = forwardPropagationDeep(x, parameters)
361         predictions = (A2>0.5)
362         return predictions

```

```

363
364 # Predict the probability scores for given data
365 # Input : parameters
366 #      : X
367 # Output: probability of output
368 def predict_proba(parameters, X):
369     A2, cache = forwardPropagationDeep(X, parameters)
370     proba=A2
371     return proba
372
373
374 # Plot a decision boundary
375 # Input : Input Model,
376 #          X
377 #          Y
378 #          sz - Num of hiden units
379 #          lr - Learning rate
380 #          Fig to be saved as
381 # Returns Null
382 def plot_decision_surface(model, X, y,sz,lr,fig):
383     # Set min and max values and give it some padding
384     x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
385     y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
386     z_min, z_max = X[2, :].min() - 1, X[2, :].max() + 1
387     colors=['black','gold']
388     cmap = matplotlib.colors.ListedColormap(colors)
389     h = 3
390     # Generate a grid of points with distance h between them
391     xx, yy, zz = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,
392     y_max, h), np.arange(z_min, z_max, h))
393     # Predict the function value for the whole grid
394     a=np.c_[xx.ravel(), yy.ravel(), zz.ravel()]
395
396     Z = predict(parameters,a.T)
397     Z = Z.reshape(xx.shape)
398     # Plot the contour and training examples
399     #plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
400     ax = plt.axes(projection='3d')
401     ax.contour3D(xx, yy, Z, 50, cmap='binary')
402     #plt.ylabel('x2')
403     #plt.xlabel('x1')
404     plt.scatter(X[0, :], X[1, :], c=y, cmap=cmap)
405     plt.title("Decision Boundary for hidden layer size:" + sz + " and learning
406 rate:"+lr)
407     plt.show()
408
409 def plotSurface(X,parameters):
410
411     #xx, yy, zz = np.meshgrid(np.arange(10), np.arange(10), np.arange(10))
412     x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
413     y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
414     z_min, z_max = X[2, :].min() - 1, X[2, :].max() + 1
415     colors=['red']
416     cmap = matplotlib.colors.ListedColormap(colors)
417     h = 1
418     xx, yy, zz = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,
419     y_max, h),
420                             np.arange(z_min, z_max, h))
421     # For the meh grid values predict a model
422     a=np.c_[xx.ravel(), yy.ravel(), zz.ravel()]
423     Z = predict(parameters,a.T)
424     r=Z.T
425     r1=r.reshape(xx.shape)
426     # Find teh values for which the repdiction is 1

```

```

427     xx1=xx[r1]
428     yy1=yy[r1]
429     zz1=zz[r1]
430     # Plot these values
431     ax = plt.axes(projection='3d')
432     #ax.plot_trisurf(xx1, yy1, zz1, cmap='bone', edgecolor='none');
433     ax.scatter3D(xx1, yy1, zz1, c=zz1, s=10, cmap=cmap)
434     #ax.plot_surface(xx1, yy1, zz1, 'gray')

```

3.2 R

```

1 ##########
2 #####
3 #
4 # File   : DLfunctions33.R
5 # Author : Tinniam V Ganesh
6 # Date   : 30 Jan 2018
7 #
8 #####
9 #####
10 library(ggplot2)
11 library(PRROC)
12 library(dplyr)
13
14 # Compute the sigmoid of a vector
15 sigmoid <- function(z){
16   A <- 1/(1+ exp(-z))
17   cache<-z
18   retvals <- list("A"=A, "Z"=z)
19   return(retvals)
20 }
21
22 # Compute the Relu of a vector
23 relu   <-function(z){
24   A <- apply(z, 1:2, function(x) max(0,x))
25   cache<-z
26   retvals <- list("A"=A, "Z"=z)
27   return(retvals)
28 }
29
30 # Compute the tanh activation of a vector
31 tanhActivation <- function(z){
32   A <- tanh(z)
33   cache<-z
34   retvals <- list("A"=A, "Z"=z)
35   return(retvals)
36 }
37
38 # Compute the derivative of Relu
39 # g'(z) = 1 if z >0 and 0 otherwise
40 reluDerivative <-function(dA, cache){
41   Z <- cache
42   dZ <- dA
43   # Create a logical matrix of values > 0
44   a <- Z > 0
45   # when z <= 0, you should set dZ to 0 as well. Perform an element wise
46   multiple
47   dZ <- dZ * a
48   return(dZ)
49 }
50

```

```

51
52 # Compute the derivative of sigmoid
53 # Derivative g'(z) = a * (1-a)
54 sigmoidDerivative <- function(dA, cache){
55   z <- cache
56   s <- 1/(1+exp(-z))
57   dZ <- dA * s * (1-s)
58   return(dZ)
59 }
60
61 # Compute the derivative of tanh
62 # Derivative g'(z) = 1- a^2
63 tanhDerivative <- function(dA, cache){
64   z = cache
65   a = tanh(z)
66   dZ = dA * (1 - a^2)
67   return(dZ)
68 }
69
70 # Initialize model for L layers
71 # Input : List of units in each layer
72 # Returns: Initial weights and biases matrices for all layers
73 initializeDeepModel <- function(layerDimensions){
74   set.seed(2)
75
76   # Initialize empty list
77   layerParams <- list()
78
79   # Note the weight matrix at layer 'l' is a matrix of size (l,l-1)
80   # The Bias is a vectors of size (l,1)
81
82   # Loop through the layer dimension from 1.. L
83   # Indices in R start from 1
84   for(l in 2:length(layersDimensions)){
85     # Initialize a matrix of small random numbers of size l x l-1
86     # Create random numbers of size l x l-1
87     w=rnorm(layersDimensions[l]*layersDimensions[l-1])*0.01
88
89     # Create a weight matrix of size l x l-1 with this initial weights
90     and
91     # Add to list w1,w2..., WL
92     layerParams[[paste('w',l-1,sep="")]] =
93     matrix(w,nrow=layersDimensions[1],
94
95     ncol=layerDimensions[l-1])
96     layerParams[[paste('b',l-1,sep="")]] =
97     matrix(rep(0,layersDimensions[1]),
98
99     nrow=layersDimensions[1],ncol=1)
100   }
101   return(layerParams)
102 }
103
104
105 # Compute the activation at a layer 'l' for forward prop in a Deep Network
106 # Input : A_prev - Activation of previous layer
107 #           W,b - Weight and bias matrices and vectors
108 #           activationFunc - Activation function - sigmoid, tanh, relu etc
109 # Returns(list) : The Activation of this layer
110 #
111 # Z = W * X + b
112 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
113 layerActivationForward <- function(A_prev, W, b, activationFunc){
114

```

```

115      # Compute Z
116      z = w %*% A_prev
117      # Broadcast the bias 'b' by column
118      Z <- sweep(z,1,b,'+')
119
120      forward_cache <- list("A_prev"=A_prev, "w"=w, "b"=b)
121      # Compute the activation for sigmoid
122      if(activationFunc == "sigmoid"){
123          vals = sigmoid(Z)
124      } else if (activationFunc == "relu"){ # Compute the activation for relu
125          vals = relu(Z)
126      } else if(activationFunc == 'tanh'){ # Compute the activation for tanh
127          vals = tanhActivation(Z)
128      }
129      # Create a list of forward and activation cache
130      cache <- list("forward_cache"=forward_cache,
131      "activation_cache"=vals[['Z']])
132      retvals <- list("A"=vals[['A']], "cache"=cache)
133      return(retvals)
134  }
135
136  # Compute the forward propagation for layers 1..L
137  # Input : X - Input Features
138  #           paramaters: weights and biases
139  # Returns (list) : AL
140  #           caches
141  # The forward propagation uses the Relu/tanh activation from layer 1..L-1 and
142  # sigmoid activation at layer L
143  forwardPropagationDeep <- function(X,
144  parameters,hiddenActivationFunc='relu'){
145      caches <- list()
146      # Set A to X (A0)
147      A <- X
148      L <- length(parameters)/2 # number of layers in the neural network
149      # Loop through from layer 1 to upto layer L
150      for(l in 1:(L-1)){
151          A_prev <- A
152          # Zi = Wi x Ai-1 + bi and Ai = g(Zi)
153          # Set W and b for layer 'l'
154          # Loop through from w1,w2... ,WL-1
155          w <- parameters[[paste("w",l,sep="")]]
156          b <- parameters[[paste("b",l,sep="")]]
157
158          # Compute the forward propagation through layer 'l' using the
159          # activation function
160          actForward <- layerActivationForward(A_prev,
161                                              w,
162                                              b,
163                                              activationFunc =
164          hiddenActivationFunc)
165          A <- actForward[['A']]
166          # Append the cache A_prev,w,b, z
167          caches[[l]] <- actForward
168      }
169
170      # Since this is binary classification use the sigmoid activation function
171  in
172      # last layer
173      # Set the weights and biases for the last layer
174      w <- parameters[[paste("w",L,sep="")]]
175      b <- parameters[[paste("b",L,sep="")]]
176      # Compute the sigmoid activation
177      actForward = layerActivationForward(A, w, b, activationFunc = "sigmoid")
178      AL <- actForward[['A']]

```

```

179     # Append the output of this forward propagation through the last layer
180     caches[[L]] <- actForward
181     # Create a list of the final output and the caches
182     fwdPropDeep <- list("AL"=AL,"caches"=caches)
183     return(fwdPropDeep)
184
185 }
186
187
188 # Compute the cost
189 # Input : Activation of last layer
190 #           : Output from data
191 # Output: cost
192 computeCost <- function(AL,Y){
193     # Element wise multiply for logprobs
194     m= length(Y)
195     cost=-1/m*sum(Y*log(AL) + (1-Y)*log(1-AL))
196     #cost=-1/m*sum(a+b)
197     return(cost)
198 }
199
200
201 # Compute the backpropagation through a layer
202 # Input : Neural Network parameters - dA
203 #           # cache - forward_cache & activation_cache
204 #           # Input features
205 #           # Output values Y
206 # Returns: Gradients as list
207 # dL/dwi= dL/dzi*A_l-1
208 # dL/dbl = dL/dz_l
209 # dL/dz_prev=dL/dz_l*w
210 layerActivationBackward <- function(dA, cache, activationFunc){
211     # Get A_prev,w,b
212     forward_cache <-cache[['forward_cache']]
213     # Get Z
214     activation_cache <- cache[['activation_cache']]
215     if(activationFunc == "relu"){
216         dz <- reluDerivative(dA, activation_cache)
217     } else if(activationFunc == "sigmoid"){
218         dz <- sigmoidDerivative(dA, activation_cache)
219     } else if(activationFunc == "tanh"){
220         dz <- tanhDerivative(dA, activation_cache)
221     }
222     A_prev <- forward_cache[['A_prev']]
223     w <- forward_cache[['w']]
224     b <- forward_cache[['b']]
225     numtraining = dim(A_prev)[2]
226     dw = 1/numtraining * dz %*% t(A_prev)
227     db = 1/numtraining * rowSums(dz)
228     dA_prev = t(w) %*% dz
229     retvals <- list("dA_Prev"=dA_prev, "dw"=dw, "db"=db)
230     return(retvals)
231 }
232
233 # Compute the backpropagation for 1 cycle through all layers
234 # Input : AL: Output of L layer Network - weights
235 #           # Y Real output
236 #           # caches -- list of caches containing:
237 #           # every cache of layerActivationForward() with "relu"/"tanh"
238 #           # (it's caches[], for l in range(L-1) i.e l = 0...L-2)
239 #           # the cache of layerActivationForward() with "sigmoid" (it's caches[L-1])
240 #           # hiddenActivationFunc - Activation function at hidden layers
241 #

```

```

243 # Returns:
244 # gradients -- A list with the gradients
245 #           gradients["dA" + str(l)] = ...
246 #
247 backwardPropagationDeep <- function(AL, Y,
248 caches,hiddenActivationFunc='relu'){
249   #initialize the gradients
250   gradients = list()
251   # Set the number of layers
252   L = length(caches)
253   numTraining = dim(AL)[2]
254
255   # Initializing the backpropagation
256   #  $dL/dAL = -(y/a) - ((1-y)/(1-a))$  - At the output layer
257   dAL = -( (Y/AL) -(1 - Y)/(1 - AL))
258
259   # Since this is a binary classification the activation at output is
260   sigmoid
261   # Get the gradients at the last layer
262   # Inputs: "AL, Y, caches".
263   # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"]
264   # Start with Layer L
265   # Get the current cache
266   current_cache = caches[[L]]$cache
267   #gradients["dA" + str(L)], gradients["dw" + str(L)], gradients["db" +
268   str(L)] = layerActivationBackward(dAL, current_cache, activationFunc =
269   "sigmoid")
270   retvals <- layerActivationBackward(dAL, current_cache, activationFunc =
271   "sigmoid")
272   # Create gradients as lists
273   gradients[[paste("dA",L,sep="")]] <- retvals[['dA_Prev']]
274   gradients[[paste("dw",L,sep="")]] <- retvals[['dw']]
275   gradients[[paste("db",L,sep="")]] <- retvals[['db']]
276
277
278   # Traverse in the reverse direction
279   for(l in (L-1):1){
280     # Compute the gradients for L-1 to 1 for Relu/tanh
281     # Inputs: "gradients["dA" + str(l + 2)], caches".
282     # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l +
283     1)] , gradients["db" + str(l + 1)]
284     current_cache = caches[[l]]$cache
285     retvals =
286     layerActivationBackward(gradients[[paste('dA',l+1,sep="")]],
287                             current_cache,
288                             activationFunc =
289                             hiddenActivationFunc)
290
291     gradients[[paste("dA",l,sep="")]] <-retvals[['dA_Prev']]
292     gradients[[paste("dw",l,sep="")]] <- retvals[['dw']]
293     gradients[[paste("db",l,sep="")]] <- retvals[['db']]
294   }
295
296   return(gradients)
297 }
298
299
300 # Perform Gradient Descent
301 # Input : Weights and biases
302 #           : gradients
303 #           : Learning rate
304 #output : Updated weights after 1 iteration as a list
305 gradientDescent <- function(parameters, gradients, learningRate){
306

```

```

307     L = length(parameters)/2 # number of layers in the neural network
308
309     # Update rule for each parameter. Use a for loop.
310     for(l in 1:L){
311         parameters[[paste("w",l,sep="")]] = parameters[[paste("w",l,sep="")]]
312         -
313         learningRate* gradients[[paste("dw",l,sep="")]]
314         parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]]
315         -
316         learningRate* gradients[[paste("db",l,sep="")]]
317     }
318     return(parameters)
319 }
320
321
322 # Execute a L layer Deep learning model
323 # Input : X - Input features
324 #           : Y output
325 #           : layersDimensions - Dimension of layers
326 #           : hiddenActivationFunc - Activation function at hidden layer relu
327 /tanh
328 #           : learning rate
329 #           : num of iterations
330 #output : Updated weights and biases
331 L_Layer_DeepModel <- function(X, Y, layersDimensions,
332                               hiddenActivationFunc='relu',
333                               learningRate = .3,
334                               numIterations = 10000, print_cost=False){
335     #Initialize costs vector as NULL
336     costs <- NULL
337
338     # Parameters initialization.
339     parameters = initializeDeepModel(layersDimensions)
340
341     # Loop (gradient descent)
342     for( i in 0:numIterations){
343         # Forward propagation: [LINEAR -> RELU]^(L-1) -> LINEAR -> SIGMOID.
344         retvals = forwardPropagationDeep(X, parameters,hiddenActivationFunc)
345         AL <- retvals[['AL']]
346         caches <- retvals[['caches']]
347
348         # Compute cost.
349         cost <- computeCost(AL, Y)
350
351         # Backward propagation.
352         gradients = backwardPropagationDeep(AL, Y,
353         caches,hiddenActivationFunc)
354
355         # Update parameters.
356         parameters = gradientDescent(parameters, gradients, learningRate)
357
358
359         if(i%1000 == 0){
360             costs=c(costs,cost)
361             print(cost)
362         }
363     }
364
365     retvals <- list("parameters"=parameters,"costs"=costs)
366
367     return(retvals)
368 }
369
370 # Predict the output for given input

```

```

371 # Input : parameters
372 #      : x
373 # Output: predictions
374 predict <- function(parameters, x,hiddenActivationFunc='relu'){
375
376     fwdProp <- forwardPropagationDeep(x, parameters,hiddenActivationFunc)
377     predictions <- fwdProp$AL>0.5
378
379     return (predictions)
380 }
381
382 # Plot a decision boundary
383 # This function uses ggplot2
384 plotDecisionBoundary <- function(z,retvals,hiddenActivationFunc,lr){
385     # Find the minimum and maximum for the data
386     xmin<-min(z[,1])
387     xmax<-max(z[,1])
388     ymin<-min(z[,2])
389     ymax<-max(z[,2])
390
391     # Create a grid of values
392     a=seq(xmin,xmax,length=100)
393     b=seq(ymin,ymax,length=100)
394     grid <- expand.grid(x=a, y=b)
395     colnames(grid) <- c('x1', 'x2')
396     grid1 <- t(grid)
397
398     # Predict the output for this grid
399     q <- predict(retval$parameters,grid1,hiddenActivationFunc)
400     q1 <- t(data.frame(q))
401     q2 <- as.numeric(q1)
402     grid2 <- cbind(grid,q2)
403     colnames(grid2) <- c('x1', 'x2','q2')
404
405     z1 <- data.frame(z)
406     names(z1) <- c("x1","x2","y")
407     atitle=paste("Decision boundary for learning rate:",lr)
408     # Plot the contour of the boundary
409     ggplot(z1) +
410         geom_point(data = z1, aes(x = x1, y = x2, color = y)) +
411         stat_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2),
412         alpha = 0.9)+ gtitle(atitle)
413 }
414
415 # Predict the probability scores for given data set
416 # Input : parameters
417 #      : x
418 # Output: probability of output
419 computeScores <- function(parameters, x,hiddenActivationFunc='relu'){
420
421     fwdProp <- forwardPropagationDeep(x, parameters,hiddenActivationFunc)
422     scores <- fwdProp$AL
423
424     return (scores)
425 }
```

3.3 Octave

```

1 ##########
2 #####
3 #
4 # File: DLfunctions3.m
5 # Developer: Tinniam V Ganesh
6 # Date : 30 Jan 2018
7 #
8 #####
9 #####
10 1;
11 # Define sigmoid function
12 function [A,cache] = sigmoid(z)
13     A = 1 ./ (1+ exp(-z));
14     cache=z;
15 end
16
17 # Define Relu function
18 function [A,cache] = relu(z)
19     A = max(0,z);
20     cache=z;
21 end
22
23 # Define Relu function
24 function [A,cache] = tanhAct(z)
25     A = tanh(z);
26     cache=z;
27 end
28
29 # Define Relu Derivative
30 function [dZ] = reluDerivative(dA,cache)
31     Z = cache;
32     dZ = dA;
33     # Get elements that are greater than 0
34     a = (z > 0);
35     # Select only those elements where z > 0
36     dZ = dZ .* a;
37 end
38
39 # Define Sigmoid Derivative
40 function [dZ] = sigmoidDerivative(dA,cache)
41     Z = cache;
42     s = 1 ./ (1+ exp(-z));
43     dZ = dA .* s .* (1-s);
44 end
45
46 # Define Tanh Derivative
47 function [dZ] = tanhDerivative(dA,cache)
48     Z = cache;
49     a = tanh(Z);
50     dZ = dA .* (1 - a .^ 2);
51 end
52
53 # Initialize the model
54 # Input : number of features
55 #           number of hidden units
56 #           number of units in output
57 # Returns: Weight and bias matrices and vectors
58
59
60 # Initialize model for L layers
61 # Input : Vector of units in each layer
62 # Returns: Initial weights and biases matrices for all layers as a cell array
63 function [w b] = initializeDeepModel(layerDimensions)
64     rand ("seed", 3);

```

```

65      # note the weight matrix at layer 'l' is a matrix of size (1,1-1)
66      # The Bias is a vectors of size (1,1)
67
68      # Loop through the layer dimension from 1.. L
69      # Create cell arrays for weights and biases
70
71      for l =2:size(layerDimensions)(2)
72          w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*0.01; #
73      Multiply by .01
74          b{l-1} = zeros(layerDimensions(l),1);
75
76      endfor
77 end
78
79 # Compute the activation at a layer 'l' for forward prop in a Deep Network
80 # Input : A_prev - Activation of previous layer
81 #           w,b - Weight and bias matrices and vectors
82 #           activationFunc - Activation function - sigmoid, tanh, relu etc
83 # Returns : A, forward_cache, activation_cache
84 #           :
85 # Z = W * X + b
86 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
87 function [A forward_cache activation_cache] = layerActivationForward(A_prev,
88 w, b, activationFunc)
89
90     # Compute Z
91     Z = W * A_prev +b;
92     # Create a cell array
93     forward_cache = {A_prev w b};
94     # Compute the activation for sigmoid
95     if (strcmp(activationFunc,"sigmoid"))
96         [A activation_cache] = sigmoid(Z);
97     elseif (strcmp(activationFunc, "relu")) # Compute the activation for
98     Relu
99         [A activation_cache] = relu(Z);
100    elseif(strcmp(activationFunc,'tanh')) # Compute the activation for
101    tanh
102        [A activation_cache] = tanhAct(Z);
103    endif
104
105 end
106
107 # Compute the forward propagation for layers 1..L
108 # Input : X - Input Features
109 #           paramaters: Weights and biases
110 #           hiddenActivationFunc - Activation function at hidden layers
111 Relu/tanh
112 # Returns : AL, forward_caches, activation_caches as a cell array
113 # The forward propoagation uses the Relu/tanh activation from layer 1..L-1 and
114 sigmoid actiovation at layer L
115 function [AL forward_caches activation_caches] = forwardPropagationDeep(X,
116 weights,biases, hiddenActivationFunc='relu')
117     # Create an empty cell array
118     forward_caches = {};
119     activation_caches = {};
120     # Set A to X (A0)
121     A = X;
122     L = length(weights); # number of layers in the neural network
123     # Loop through from layer 1 to upto layer L
124     for l =1:L-1
125         A_prev = A;
126         # Zi = Wi x Ai-1 + bi and Ai = g(Zi)
127         w = weights{l};
128         b = biases{l};

```

```

129      [A forward_cache activation_cache] = layerActivationForward(A_prev,
130      w,b, activationFunc=hiddenActivationFunc);
131      forward_caches{1}=forward_cache;
132      activation_caches{1} = activation_cache;
133  endfor
134  # Since this is binary classification use the sigmoid activation function
135  in
136      # last layer
137      w = weights{L};
138      b = biases{L};
139      [AL, forward_cache activation_cache] = layerActivationForward(A, w,b,
140      activationFunc = "sigmoid");
141      forward_caches{L}=forward_cache;
142      activation_caches{L} = activation_cache;
143
144 end
145
146 # Compute the cost
147 # Input : Activation of last layer
148 #           : Output from data
149 # Output: cost
150 function [cost]=computeCost(AL,Y)
151     numTraining= size(Y)(2);
152     # Element wise multiply for logprobs
153     cost = -1/numTraining * sum((Y .* log(AL)) + (1-Y) .* log(1-AL));
154 end
155
156 # Compute the layerActivationBackward
157 # Input : Neural Network parameters - dA
158 #           # cache - forward_cache & activation_cache
159 # Returns: dA_prev, dw, db
160 # dL/dwi= dL/dzi*A1-1
161 # dL/dbl = dL/dz1
162 # dL/dz_prev=dL/dz1*w
163 function [dA_prev dw db] = layerActivationBackward(dA, forward_cache,
164 activation_cache, activationFunc)
165
166     if (strcmp(activationFunc,"relu"))
167         dz = reluDerivative(dA, activation_cache);
168     elseif (strcmp(activationFunc,"sigmoid"))
169         dz = sigmoidDerivative(dA, activation_cache);
170     elseif(strcmp(activationFunc, "tanh"))
171         dz = tanhDerivative(dA, activation_cache);
172     endif
173     A_prev = forward_cache{1};
174     w =forward_cache{2};
175     b = forward_cache{3};
176     numTraining = size(A_prev)(2);
177     dw = 1/numTraining * dz * A_prev';
178     db = 1/numTraining * sum(dz,2);
179     dA_prev = w'*dz;
180
181 end
182
183
184 # Compute the backpropagation for 1 cycle
185 # Input : AL: Output of L layer Network - weights
186 #           # Y Real output
187 #           # activation_caches
188 #           # forward_caches
189 #           every cache of layerActivationForward() with "relu"/"tanh"
190 #           #(it's caches[], for l in range(L-1) i.e l = 0...L-2)
191 #           #the cache of layerActivationForward() with "sigmoid" (it's caches[L-
192 1])

```

```

193 #     hiddenActivationFunc - Activation function at hidden layers
194 #
195 #   Returns (cell array): gradsDA,gradsDW, gradsDB
196 function [gradsDA gradsDW gradsDB]= backwardPropagationDeep(AL, Y,
197 activation_caches,forward_caches,hiddenActivationFunc='relu')
198
199
200     # Set the number of layers
201     L = length(activation_caches);
202     m = size(AL)(2);
203
204     # Initializing the backpropagation
205     #  $dL/dAL = -(y/a + (1-y)/(1-a))$  - At the output layer
206     dAL = -((Y ./ AL) - (1 - Y) ./ (1 - AL));
207
208     # Since this is a binary classification the activation at output is
209     sigmoid
210     # Get the gradients at the last layer
211     # Inputs: "AL, Y, caches".
212     # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"]
213     activation_cache = activation_caches{L};
214     forward_cache = forward_caches(L);
215     # Note the cell array includes an array of forward caches. To get to this
216     we need to include the index {1}
217     [dA dw db] = layerActivationBackward(dAL, forward_cache{1},
218     activation_cache, activationFunc = "sigmoid");
219     gradsDA{L}= dA;
220     gradsDW{L}= dw;
221     gradsDB{L}= db;
222
223     # Traverse in the reverse direction
224     for l =(L-1):-1:1
225         # Compute the gradients for L-1 to 1 for Relu/tanh
226         # Inputs: "gradients["dA" + str(l + 2)], caches".
227         # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l +
228         1)] , gradients["db" + str(l + 1)]
229         activation_cache = activation_caches{l};
230         forward_cache = forward_caches(l);
231
232         #dA_prev_temp, dw_temp, db_temp =
233         layerActivationBackward(gradients['dA'+str(l+1)], current_cache,
234         activationFunc = "relu")
235         # dAl the derivative of the activation of the lth layer, is the first
236         element
237         dAl= gradsDA{l+1};
238         [dA_prev_temp, dw_temp, db_temp] = layerActivationBackward(dA,
239         forward_cache{1}, activation_cache, activationFunc = hiddenActivationFunc);
240         gradsDA{l}= dA_prev_temp;
241         gradsDW{l}= dw_temp;
242         gradsDB{l}= db_temp;
243     endfor
244
245 end
246
247
248 # Perform Gradient Descent
249 # Input : Weights and biases
250 #           : gradients
251 #           : Learning rate
252 #Output : Updated weights and biases after 1 iteration
253 function [weights biases] = gradientDescent(weights, biases,gradsw,gradsB,
254 learningRate)
255
256     L = size(weights)(2); # number of layers in the neural network

```

```

257
258     # Update rule for each parameter.
259     for l=1:L
260         weights{l} = weights{l} - learningRate* gradsW{l};
261         biases{l} = biases{l} - learningRate* gradsB{l};
262     endfor
263 end
264
265
266 # Execute a L layer Deep learning model
267 # Input : X - Input features
268 #           : Y output
269 #           : layersDimensions - Dimension of layers
270 #           : hiddenActivationFunc - Activation function at hidden layer relu
271 /tanh
272 #           : learning rate
273 #           : num of iterations
274 #output : Updated weights and biases
275 function [weights biases costs] = L_Layer_DeepModel(X, Y, layersDimensions,
276 hiddenActivationFunc='relu', learning_rate = .3, num_iterations = 10000)#lr
277 was 0.009
278
279 rand ("seed", 1);
280 costs = [] ;
281
282 # Parameters initialization.
283 [weights biases] = initializeDeepModel(layersDimensions);
284
285 # Loop (gradient descent)
286 for i = 0:num_iterations
287     # Forward propagation: [LINEAR -> RELU]^(L-1) -> LINEAR -> SIGMOID.
288     [AL forward_caches activation_caches] = forwardPropagationDeep(X,
289 weights, biases,hiddenActivationFunc);
290
291     # Compute cost.
292     cost = computeCost(AL, Y);
293
294     # Backward propagation.
295     [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y,
296 activation_caches,forward_caches,hiddenActivationFunc);
297
298     # update parameters.
299     [weights biases] = gradientDescent(weights,biases,
300 gradsDW,gradsDB,learning_rate);
301
302
303     # Print the cost every 1000 iterations
304     if ( mod(i,1000) == 0)
305         costs =[costs cost];
306         #disp ("Cost after iteration"), disp(i),disp(cost);
307         printf("Cost after iteration i=%i cost=%d\n",i,cost);
308     endif
309     endfor
310
311 end
312
313 #Plot cost vs iterations
314 function plotCostVsIterations(maxIterations,costs)
315     iterations=[0:1000:maxIterations];
316     plot(iterations,costs);
317     title ("Cost vs no of iterations for different learning rates");
318     xlabel("No of iterations");
319     ylabel("Cost");
320 end;

```

```

321
322 # Compute the predicted value for a given input
323 # Input : Neural Network parameters
324 # : Input data
325 function [predictions]= predict(weights, biases,
326 x,hiddenActivationFunc="relu")
327 [AL forward_caches activation_caches] = forwardPropagationDeep(X,
328 weights, biases,hiddenActivationFunc);
329 predictions = (AL>0.5);
330 end
331
332 # Plot the decision boundary
333 function plotDecisionBoundary(data,weights,
334 biases,hiddenActivationFunc="relu")
335 %Plot a non-linear decision boundary learned by the SVM
336 colormap ("summer");
337
338 % Make classification predictions over a grid of values
339 x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)';
340 x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)';
341 [X1, X2] = meshgrid(x1plot, x2plot);
342 vals = zeros(size(X1));
343 # Plot the prediction for the grid
344 for i = 1:size(X1, 2)
345     gridPoints = [X1(:, i), X2(:, i)];
346     vals(:, i)=predict(weights,
347 biases,gridPoints',hiddenActivationFunc=hiddenActivationFunc);
348 endfor
349
350 scatter(data(:,1),data(:,2),8,c=data(:,3),"filled");
351 % Plot the boundary
352 hold on
353 #contour(X1, X2, vals, [0 0], 'LineWidth', 2);
354 contour(X1, X2, vals,"LineWidth",4);
355 title ({"3 layer Neural Network decision boundary"});
356 hold off;
357
358 end
359
360 # Compute scores
361 function [AL]= scores(weights, biases, x,hiddenActivationFunc="relu")
362 [AL forward_caches activation_caches] = forwardPropagationDeep(X,
363 weights, biases,hiddenActivationFunc);
364 end

```

5. Appendix 4 - Deep Learning network with the Softmax

4.1 Python

1 # -*- coding: utf-8 -*-

```

2 """
3 ######
4 ######
5 #
6 # File: DLfunctions41.py
7 # Developer: Tinniam V Ganesh
8 # Date : 26 Feb 2018
9 #
10 #####
11 #####
12 @author: Ganesh
13 """
14
15 import numpy as np
16 import matplotlib.pyplot as plt
17 import matplotlib
18 import matplotlib.pyplot as plt
19 from matplotlib import cm
20
21 # Compute the Relu of a vector
22 def relu(Z):
23     A = np.maximum(0,Z)
24     cache=Z
25     return A,cache
26
27 # Compute the softmax of a vector
28 def softmax(Z):
29     # get unnormalized probabilities
30     exp_scores = np.exp(Z.T)
31     # normalize them for each example
32     A = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
33     cache=Z
34     return A,cache
35
36 # Compute the derivative of Relu
37 def reluDerivative(dA, cache):
38     Z = cache
39     dZ = np.array(dA, copy=True) # just converting dz to a correct object.
40     # when z <= 0, you should set dz to 0 as well.
41     dZ[Z <= 0] = 0
42     return dZ
43
44
45 # Compute the derivative of softmax
46 def softmaxDerivative(dA, cache,y,numTraining):
47     # Note : dA not used. dL/dZ = dL/dA * dA/dZ = pi-yi
48     Z = cache
49     # Compute softmax
50     exp_scores = np.exp(Z.T)
51     # normalize them for each example
52     probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
53
54     # compute the gradient on scores
55     dZ = probs
56     # dZ = pi- yi
57     dZ[range(int(numTraining)),y] -= 1
58     return(dZ)
59
60
61 # Initialize the model
62 # Input : number of features
63 #           number of hidden units
64 #           number of units in output
65 # Returns: Weight and bias matrices and vectors

```

```

66 def initializeModel(numFeats,numHidden,numOutput):
67     np.random.seed(1)
68     w1=np.random.randn(numHidden,numFeats)*0.01 # Multiply by .01
69     b1=np.zeros((numHidden,1))
70     w2=np.random.randn(numOutput,numHidden)*0.01
71     b2=np.zeros((numOutput,1))
72
73     # Create a dictionary of the neural network parameters
74     nnParameters={'W1':w1,'b1':b1,'W2':w2,'b2':b2}
75     return(nnParameters)
76
77
78
79 # Compute the activation at a layer 'l' for forward prop in a Deep Network
80 # Input : A_prev - Activation of previous layer
81 #          W,b - Weight and bias matrices and vectors
82 #          activationFunc - Activation function - sigmoid, tanh, relu etc
83 # Returns : A, cache
84 #
85 # Z = W * X + b
86 # A = sigmoid(Z), A= Relu(z), A= tanh(z)
87 def layerActivationForward(A_prev, w, b, activationFunc):
88
89     # Compute Z
90     Z = np.dot(w,A_prev) + b
91     forward_cache = (A_prev, w, b)
92     # Compute the activation for sigmoid
93     if activationFunc == "sigmoid":
94         A, activation_cache = sigmoid(Z)
95     # Compute the activation for Relu
96     elif activationFunc == "relu":
97         A, activation_cache = relu(Z)
98     # Compute the activation for tanh
99     elif activationFunc == 'tanh':
100        A, activation_cache = tanh(Z)
101    # Compute the activation for softmax
102    elif activationFunc == 'softmax':
103        A, activation_cache = softmax(Z)
104    cache = (forward_cache, activation_cache)
105    return A, cache
106
107 # Compute the backpropagation for 1 cycle
108 # Input : Neural Network parameters - dA
109 #          # cache - forward_cache & activation_cache
110 #          # y
111 #          # activationFunc
112 # Returns: dA_prev, dw, db
113 # dL/dwi= dL/dzi*A_l-1
114 # dL/dbl = dL/dzl
115 # dL/dz_prev=dL/dzl*w
116 def layerActivationBackward(dA, cache, y, activationFunc):
117     forward_cache, activation_cache = cache
118     A_prev, w, b = forward_cache
119     numtraining = float(A_prev.shape[1])
120     if activationFunc == "relu":
121         dz = reluDerivative(dA, activation_cache)
122     elif activationFunc == "sigmoid":
123         dz = sigmoidDerivative(dA, activation_cache)
124     elif activationFunc == "tanh":
125         dz = tanhDerivative(dA, activation_cache)
126     elif activationFunc == "softmax":
127         dz = softmaxDerivative(dA, activation_cache,y,numtraining)
128
129     if activationFunc == 'softmax':

```

```

130         dw = 1/numtraining * np.dot(A_prev,dz)
131         db = np.sum(dz, axis=0, keepdims=True)
132         dA_prev = np.dot(dz,w)
133     else:
134         #print(numtraining)
135         dw = 1/numtraining *(np.dot(dZ,A_prev.T))
136         #print("dw=",dw)
137         db = 1/numtraining * np.sum(dZ, axis=1, keepdims=True)
138         #print("db=",db)
139         dA_prev = np.dot(w.T,dZ)
140     return dA_prev, dw, db
141
142
143 # Plot a decision boundary
144 # Input : Input Model,
145 #          X
146 #          Y
147 #          w1
148 #          b1
149 #          w2
150 #          fig
151 # Returns Null
152 def plot_decision_boundary(x, y,w1,b1,w2,b2,fig1):
153     #plot_decision_boundary(lambda x: predict(parameters, x.T),
154     x1,y1.T,str(0.3),"fig2.png")
155     h = 0.02
156     x_min, x_max = x[:, 0].min() - 1, x[:, 0].max() + 1
157     y_min, y_max = x[:, 1].min() - 1, x[:, 1].max() + 1
158     xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
159                           np.arange(y_min, y_max, h))
160     z = np.dot(np.maximum(0, np.dot(np.c_[xx.ravel(), yy.ravel()], w1.T) +
161 b1.T), w2.T) + b2.T
162     z = np.argmax(z, axis=1)
163     z = z.reshape(xx.shape)
164
165     fig = plt.figure()
166     plt.contourf(xx, yy, z, cmap=plt.cm.Spectral, alpha=0.8)
167     plt.scatter(x[:, 0], x[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
168     plt.xlim(xx.min(), xx.max())
169     plt.ylim(yy.min(), yy.max())
170

```

4.2 R

```

1 #####
2 #####
3 #
4 # File: DLfunctions41.R
5 # Developer: Tinniam V Ganesh
6 # Date : 26 Feb 2018
7 #
8 #####
9 #####
10 #####
11 # Compute the Relu of a vector
12 relu <-function(z){
13   A <- apply(z, 1:2, function(x) max(0,x))
14   cache<-z

```

```

15     retvals <- list("A"=A,"Z"=Z)
16     return(retvals)
17   }
18
19 # Compute the softmax of a vector
20 softmax <- function(Z){
21   # get unnormalized probabilities
22   exp_scores = exp(t(Z))
23   # normalize them for each example
24   A = exp_scores / rowSums(exp_scores)
25   retvals <- list("A"=A,"Z"=Z)
26   return(retvals)
27 }
28
29 # Compute the derivative of ReLU
30 reluDerivative <- function(dA, cache){
31   Z <- cache
32   dZ <- dA
33   # Create a logical matrix of values > 0
34   a <- Z > 0
35   # When z <= 0, you should set dz to 0 as well. Perform an element wise
36   multiple
37   dZ <- dZ * a
38   return(dZ)
39 }
40
41 # Compute the derivative of softmax
42 softmaxDerivative <- function(dA, cache ,y,numTraining){
43   # Note : dA not used. dL/dz = dL/dA * dA/dz = pi-yi
44   Z <- cache
45   # Compute softmax
46   exp_scores = exp(t(Z))
47   # normalize them for each example
48   probs = exp_scores / rowSums(exp_scores)
49   # Get the number of 0, 1 and 2 classes and store in a,b,c
50   a=sum(y==0)
51   b=sum(y==1)
52   c=sum(y==2)
53   # Create a yi matrix based on yi for each class
54   m= matrix(rep(c(1,0,0),a),nrow=a,ncol=3,byrow=T)
55   n= matrix(rep(c(0,1,0),b),nrow=b,ncol=3,byrow=T)
56   o= matrix(rep(c(0,0,1),c),nrow=c,ncol=3,byrow=T)
57   # Stack them vertically
58   yi=rbind(m,n,o)
59
60   dZ = probs-yi
61   return(dZ)
62 }
63
64 # Initialize the model
65 # Input : number of features
66 #          number of hidden units
67 #          number of units in output
68 # Returns: list of weight and bias matrices and vectors
69 initializeModel <- function(numFeats,numHidden,numOutput){
70   set.seed(2)
71   a<-rnorm(numHidden*numFeats)*0.01 # Multiply by .01
72   W1 <- matrix(a,nrow=numHidden,ncol=numFeats)
73   a<-rnorm(numHidden*1)
74   b1 <- matrix(a,nrow=numHidden,ncol=1)
75   a<-rnorm(numOutput*numHidden)*0.01
76   W2 <- matrix(a,nrow=numOutput,ncol=numHidden)
77   a<-rnorm(numOutput*1)
78   b2 <- matrix(a,nrow=numOutput,ncol=1)

```

```

79     parameters <- list("W1"=w1,"b1"=b1,"W2"=w2,"b2"=b2)
80     return(parameters)
81   }
82 
83 # Compute the activation at a layer 'l' for forward prop in a Deep Network
84 # Input : A_prev - Activation of previous layer
85 #           W,b - Weight and bias matrices and vectors
86 #           activationFunc - Activation function - sigmoid, tanh, relu etc
87 # Returns : A list of forward_cache, activation_cache, cache
88 # Z = W * X + b
89 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
90 layerActivationForward <- function(A_prev, W, b, activationFunc){
91 
92   # Compute Z
93   z = W %*% A_prev
94   Z <- sweep(z,1,b,'+')
95 
96   forward_cache <- list("A_prev"=A_prev, "W"=W, "b"=b)
97   # Compute the activation for sigmoid
98   if(activationFunc == "sigmoid"){
99     vals = sigmoid(Z)
100  } else if (activationFunc == "relu"){ # Compute the activation for relu
101    vals = relu(Z)
102  } else if(activationFunc == 'tanh'){ # Compute the activation for tanh
103    vals = tanhActivation(Z)
104  } else if(activationFunc == 'softmax'){
105    vals = softmax(Z)
106  }
107 
108  cache <- list("forward_cache"=forward_cache,
109 "activation_cache"=vals[['Z']])
110  retvals <- list("A"=vals[['A']], "cache"=cache)
111  return(retvals)
112 }
113 
114 
115 
116 # Compute the backpropagation for 1 cycle
117 # Input : Neural Network parameters - dA
118 #           # cache - forward_cache & activation_cache
119 #           # y
120 #           # activationFunc
121 # Returns: Gradients - a list of dA_prev, dw, db
122 # dL/dwi= dL/dzi*A_l-1
123 # dL/dbl = dL/dz_l
124 # dL/dz_prev=dL/dz_l*w
125 layerActivationBackward <- function(dA, cache, y, activationFunc){
126   # Get A_prev,W,b
127   forward_cache <- cache[['forward_cache']]
128   activation_cache <- cache[['activation_cache']]
129   A_prev <- forward_cache[['A_prev']]
130   numtraining = dim(A_prev)[2]
131   # Get Z
132 
133   if(activationFunc == "relu"){
134     dz <- reluDerivative(dA, activation_cache)
135   } else if(activationFunc == "sigmoid"){
136     dz <- sigmoidDerivative(dA, activation_cache)
137   } else if(activationFunc == "tanh"){
138     dz <- tanhDerivative(dA, activation_cache)
139   } else if(activationFunc == "softmax"){
140     dz <- softmaxDerivative(dA, activation_cache,y,numtraining)
141   }
142 }
```

```

143     # Check if softmax
144     if (activationFunc == 'softmax'){
145         w <- forward_cache[['w']]
146         b <- forward_cache[['b']]
147         dw = 1/numtraining * A_prev%*%dz
148         db = 1/numtraining* matrix(colSums(dz) ,nrow=1,ncol=3)
149         dA_prev = dz %*%w
150     } else {
151         w <- forward_cache[['w']]
152         b <- forward_cache[['b']]
153         numtraining = dim(A_prev)[2]
154         dw = 1/numtraining * dz %*% t(A_prev)
155         db = 1/numtraining * rowSums(dz)
156         dA_prev = t(w) %*% dz
157     }
158     retvals <- list("dA_prev"=dA_prev,"dw"=dw,"db"=db)
159     return(retvals)
160 }
161
162
163
164
165 # Plot a decision boundary for softmax output activation
166 # This function uses ggplot2
167 plotDecisionBoundary <- function(z,w1,b1,w2,b2){
168     xmin<-min(z[,1])
169     xmax<-max(z[,1])
170     ymin<-min(z[,2])
171     ymax<-max(z[,2])
172
173     # Create a grid of points
174     a=seq(xmin,xmax,length=100)
175     b=seq(ymin,ymax,length=100)
176     grid <- expand.grid(x=a, y=b)
177     colnames(grid) <- c('x1', 'x2')
178     grid1 <-t(grid)
179
180
181     # Predict the output based on the grid of points
182     retvals <- layerActivationForward(grid1,w1,b1,'relu')
183     A1 <- retvals[['A']]
184     cache1 <- retvals[['cache']]
185     forward_cache1 <- cache1[['forward_cache1']]
186     activation_cache <- cache1[['activation_cache']]
187
188     retvals = layerActivationForward(A1,w2,b2,'softmax')
189     A2 <- retvals[['A']]
190     cache2 <- retvals[['cache']]
191     forward_cache2 <- cache2[['forward_cache1']]
192     activation_cache2 <- cache2[['activation_cache']]
193
194     # From the softmax probabilities pick the one with the highest
195 probability
196     q= apply(A2,1,which.max)
197
198     q1 <- t(data.frame(q))
199     q2 <- as.numeric(q1)
200     grid2 <- cbind(grid,q2)
201     colnames(grid2) <- c('x1', 'x2','q2')
202
203     z1 <- data.frame(z)
204     names(z1) <- c("x1","x2","y")
205     atitle=paste("Decision boundary")
206     ggplot(z1) +

```

```

207     geom_point(data = z1, aes(x = x1, y = x2, color = y)) +
208     stat_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2),
209     alpha = 0.9) +
210     ggtitle(atitle) + scale_colour_gradientn(colours = brewer.pal(10,
211     "Spectral"))
212   }
213
214 # Predict the output
215 computeScores <- function(parameters, x,hiddenActivationFunc='relu'){
216
217   fwdProp <- forwardPropagationDeep(x, parameters,hiddenActivationFunc)
218   scores <- fwdProp$AL
219
220   return (scores)
221 }
```

4.3 Octave

```

1 #####
2 #####
3 #####
4 #
5 # File: DLfunctions41.m
6 # Developer: Tinniam V Ganesh
7 # Date : 26 Feb 2018
8 #
9 #####
10 #####
11 1;
12
13 # Define Relu function
14 function [A,cache] = relu(z)
15   A = max(0,z);
16   cache=z;
17 end
18
19
20 # Define Softmax function
21 function [A,cache] = softmax(z)
22   # get unnormalized probabilities
23   exp_scores = exp(z');
24   # normalize them for each example
25   A = exp_scores ./ sum(exp_scores,2);
26   cache=z;
27 end
28
29 # Define Relu Derivative
30 function [dz] = reluDerivative(dA,cache)
31   Z = cache;
32   dz = dA;
33   # Get elements that are greater than 0
34   a = (Z > 0);
35   # Select only those elements where z > 0
36   dz = dz .* a;
37 end
38
39
40 # Define Softmax Derivative
41 function [dz] = softmaxDerivative(dA,cache,Y)
```

```

42 z = cache;
43 # get unnormalized probabilities
44 exp_scores = exp(z');
45 # normalize them for each example
46 probs = exp_scores ./ sum(exp_scores,2);
47
48 # dz = pi - yi
49 a=sum(Y==0);
50 b=sum(Y==1);
51 c=sum(Y==2);
52 m= repmat([1 0 0],a,1);
53 n= repmat([0 1 0],b,1);
54 o= repmat([0 0 1],c,1);
55 yi=[m;n;o];
56 dz=probs-yi;
57
58 end
59
60 # Initialize the model
61 # Input : number of features
62 #           number of hidden units
63 #           number of units in output
64 # Returns: weight and bias matrices and vectors
65 function [w1 b1 w2 b2] = initializeModel(numFeats,numHidden,numOutput)
66 rand ("seed", 3);
67 w1=rand(numHidden,numFeats)*0.01; # Multiply by .01
68 b1=zeros(numHidden,1);
69 w2=rand(numOutput,numHidden)*0.01;
70 b2=zeros(numOutput,1);
71 end
72
73
74
75 # Compute the activation at a layer 'l' for forward prop in a Deep Network
76 # Input : A_prev - Activation of previous layer
77 #           w,b - weight and bias matrices and vectors
78 #           activationFunc - Activation function - sigmoid, tanh, relu etc
79 # Returns : A, forward_cache, activation_cache
80 # Z = W * X + b
81 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
82 function [A forward_cache activation_cache] = layerActivationForward(A_prev,
83 w, b, activationFunc)
84
85     # Compute Z
86     Z = w * A_prev +b;
87     # Create a cell array
88     forward_cache = {A_prev w b};
89     # Compute the activation for sigmoid
90     if (strcmp(activationFunc,"sigmoid"))
91         [A activation_cache] = sigmoid(Z);
92     elseif (strcmp(activationFunc, "relu")) # Compute the activation for
93     Relu
94         [A activation_cache] = relu(Z);
95     elseif(strcmp(activationFunc,'tanh')) # compute the activation for
96     tanh
97         [A activation_cache] = tanhAct(Z);
98     elseif(strcmp(activationFunc,'softmax')) # Compute the activation
99     for softmax
100        [A activation_cache] = softmax(Z);
101    endif
102
103
104
105 end

```

```

106
107
108 # Compute the backpropagation for 1 cycle
109 # Input : Neural Network parameters - dA
110 #          # cache - forward_cache & activation_cache
111 #          # Input features
112 #          # Output values Y
113 # Returns: dA_prev, dW, db
114 # dL/dwi= dL/dzi*A1-1
115 # dL/db1 = dL/dz1
116 # dL/dz_prev=dL/dz1*w
117 function [dA_prev dW dB] = layerActivationBackward(dA, forward_cache,
118 activation_cache, Y, activationFunc)
119
120     if (strcmp(activationFunc, "relu"))
121         dZ = reluDerivative(dA, activation_cache);
122     elseif (strcmp(activationFunc, "sigmoid"))
123         dZ = sigmoidDerivative(dA, activation_cache);
124     elseif(strcmp(activationFunc, "tanh"))
125         dZ = tanhDerivative(dA, activation_cache);
126     elseif(strcmp(activationFunc, "softmax"))
127         dZ = softmaxDerivative(dA, activation_cache,Y);
128     endif
129     A_prev = forward_cache{1};
130     numTraining = size(A_prev)(2);
131
132     # If activation is softmax
133     if(strcmp(activationFunc, "softmax"))
134         w =forward_cache{2};
135         b = forward_cache{3};
136         dW = 1/numTraining * A_prev * dZ;
137         dB = 1/numTraining * sum(dZ,1);
138         dA_prev = dZ*w;
139     else
140         w =forward_cache{2};
141         b = forward_cache{3};
142         dW = 1/numTraining * dZ * A_prev';
143         dB = 1/numTraining * sum(dZ,2);
144         dA_prev = w'*dZ;
145     endif
146 end
147
148 # Plot cost vs iterations
149 function plotCostVsIterations(iterations,costs)
150
151     plot(iterations,costs);
152     title ("Cost vs no of iterations for different learning rates");
153     xlabel("No of iterations");
154     ylabel("Cost");
155     print -dpng "figo2.png"
156 end;
157
158 # Plot softmax decision boundary
159 function plotDecisionBoundary( X,Y,w1,b1,w2,b2)
160     % Make classification predictions over a grid of values
161     x1plot = linspace(min(X(:,1)), max(X(:,1)), 400)';
162     x2plot = linspace(min(X(:,2)), max(X(:,2)), 400)';
163     [X1, X2] = meshgrid(x1plot, x2plot);
164     vals = zeros(size(X1));
165
166     for i = 1:size(X1, 2)
167         gridPoints = [X1(:, i), X2(:, i)];
168         [A1,cache1 activation_cache1]=
169         layerActivationForward(gridPoints',w1,b1,activationFunc ='relu');

```

```

170      [A2,cache2 activation_cache2] =
171 layerActivationForward(A1,w2,b2,activationFunc='softmax');
172      [1 m] = max(A2, [ ], 2);
173      vals(:, i)= m;
174 endfor
175
176 scatter(x(:,1),x(:,2),8,c=Y,"filled");
177 % Plot the boundary
178 hold on
179 contour(x1, x2, vals,"linewidth",4);
180 print -dpng "fig-o1.png"
181

```

6.Appendix 5 - MNIST classification with Softmax

5.1 Python

```

1  # -*- coding: utf-8 -*-
2  """
3  ######
4  ######
5  #
6  # File: DLfunctions41.py
7  # Developer: Tinniam V Ganesh
8  # Date : 23 Mar 2018
9  #
10 #####
11 #####
12 @author: Ganesh
13 """
14
15 import numpy as np
16 import matplotlib.pyplot as plt
17 import matplotlib
18 import matplotlib.pyplot as plt
19 from matplotlib import cm
20 import math
21
22 # Compute the sigmoid of a vector
23 def sigmoid(Z):
24     A=1/(1+np.exp(-Z))
25     cache=Z
26     return A,cache
27
28 # Compute the Relu of a vector
29 def relu(Z):
30     A = np.maximum(0,Z)
31     cache=Z
32     return A,cache
33
34 # Compute the tanh of a vector
35 def tanh(Z):
36     A = np.tanh(Z)
37     cache=Z
38     return A,cache
39
40 # Compute the softmax of a vector
41 def softmax(Z):
42     # get unnormalized probabilities
43     exp_scores = np.exp(Z.T)
44     # normalize them for each example
45     A = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
46     cache=Z
47     return A,cache
48
49 # Compute the Stable Softmax of a vector
50 def stableSoftmax(Z):
51     #Compute the softmax of vector x in a numerically stable way.
52     shiftz = Z.T - np.max(Z.T, axis=1).reshape(-1,1)
53     exp_scores = np.exp(shiftz)
54
55     # normalize them for each example
56     A = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
57     cache=Z
58     return A,cache
59
60 # Compute the derivative of Relu
61 def reluDerivative(dA, cache):
62
63     Z = cache
64     dZ = np.array(dA, copy=True) # just converting dz to a correct object.

```

```

65      # When z <= 0, you should set dz to 0 as well.
66      dz[z <= 0] = 0
67      return dz
68
69  # Compute the derivative of Sigmoid
70  def sigmoidDerivative(dA, cache):
71      z = cache
72      s = 1/(1+np.exp(-z))
73      dz = dA * s * (1-s)
74      return dz
75
76  # Compute the derivative of tanh
77  def tanhDerivative(dA, cache):
78      z = cache
79      a = np.tanh(z)
80      dz = dA * (1 - np.power(a, 2))
81      return dz
82
83  # Compute the derivative of Softmax
84  def softmaxDerivative(dA, cache,y,numTraining):
85      # Note : dA not used. dL/dZ = dL/dA * dA/dz = pi-yi
86      z = cache
87      # Compute softmax
88      exp_scores = np.exp(z.T)
89      # normalize them for each example
90      probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
91
92      # compute the gradient on scores
93      dz = probs
94
95      # dz = pi- yi
96      dz[range(int(numTraining)),y[:,0]] -= 1
97      return(dz)
98
99  # Compute the derivative of Stable Softmax
100 def stableSoftmaxDerivative(dA, cache,y,numTraining):
101     # Note : dA not used. dL/dZ = dL/dA * dA/dz = pi-yi
102     z = cache
103     # Compute stable softmax
104     shiftZ = Z.T - np.max(Z.T, axis=1).reshape(-1,1)
105     exp_scores = np.exp(shiftZ)
106     # normalize them for each example
107     probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
108     #print(probs)
109     # compute the gradient on scores
110     dz = probs
111
112     # dz = pi- yi
113     dz[range(int(numTraining)),y[:,0]] -= 1
114     return(dz)
115
116
117  # Initialize the model
118  # Input : number of features
119  #           number of hidden units
120  #           number of units in output
121  # Returns: nnParameters dict
122  def initializeModel(numFeats,numHidden,numOutput):
123      np.random.seed(1)
124      W1=np.random.randn(numHidden,numFeats)*0.01 # Multiply by .01
125      b1=np.zeros((numHidden,1))
126      W2=np.random.randn(numOutput,numHidden)*0.01
127      b2=np.zeros((numOutput,1))
128

```

```

129     # Create a dictionary of the neural network parameters
130     nnParameters={'w1':w1,'b1':b1,'w2':w2,'b2':b2}
131     return(nnParameters)
132
133
134 # Initialize model for L layers
135 # Input : List of units in each layer
136 # Returns: Z, cache
137 def initializeDeepModel(layerDimensions):
138     np.random.seed(3)
139     # note the weight matrix at layer 'l' is a matrix of size (l,l-1)
140     # The Bias is a vectors of size (l,1)
141
142     # Loop through the layer dimension from 1.. L
143     layerParams = {}
144     for l in range(1,len(layerDimensions)):
145         layerParams['w' + str(l)] =
146             np.random.randn(layerDimensions[l],layerDimensions[l-1])*0.01 # Multiply by
147             .01
148         layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1))
149
150     return(layerParams)
151 return Z, cache
152
153 # Compute the activation at a layer 'l' for forward prop in a Deep Network
154 # Input : A_prev - Activation of previous layer
155 #          W,b - Weight and bias matrices and vectors
156 #          activationFunc - Activation function - sigmoid, tanh, relu etc
157 # Returns : A, cache
158 #
159 # Z = W * X + b
160 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
161 def layerActivationForward(A_prev, w, b, activationFunc):
162
163     # Compute z
164     z = np.dot(w,A_prev) + b
165     forward_cache = (A_prev, w, b)
166     # Compute the activation for sigmoid
167     if activationFunc == "sigmoid":
168         A, activation_cache = sigmoid(z)
169     # Compute the activation for Relu
170     elif activationFunc == "relu":
171         A, activation_cache = relu(z)
172     # Compute the activation for tanh
173     elif activationFunc == 'tanh':
174         A, activation_cache = tanh(z)
175     elif activationFunc == 'softmax':
176         A, activation_cache = stablesoftmax(z)
177
178     cache = (forward_cache, activation_cache)
179     return A, cache
180
181 # Compute the forward propagation for layers 1..L
182 # Input : X - Input Features
183 #          parameters: Weights and biases
184 #          hiddenActivationFunc - Activation function at hidden layers
185 Relu/tanh
186 #          outputActivationFunc - Activation function at output -
187 sigmoid/softmax
188 # Returns : AL
189 #          caches
190 # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and
191 sigmoid actiovation at layer L

```

```

192 def forwardPropagationDeep(X,
193 parameters,hiddenActivationFunc='relu',outputActivationFunc='sigmoid'):
194     caches = []
195     # Set A to X (A0)
196     A = X
197     L = int(len(parameters)/2) # number of layers in the neural network
198     # Loop through from layer 1 to upto layer L
199     for l in range(1, L):
200         A_prev = A
201         # Zi = Wi x Ai-1 + bi and Ai = g(Zi)
202         #A, cache = layerActivationForward(A_prev, parameters['w'+str(l)],
203 parameters['b'+str(l)], activationFunc = "relu")
204         A, cache = layerActivationForward(A_prev, parameters['w'+str(l)],
205 parameters['b'+str(l)], activationFunc = hiddenActivationFunc)
206         caches.append(cache)
207         #print("l=",l)
208         #print(A)
209
210     # Since this is binary classification use the sigmoid activation function
211     in
212     # last layer
213     AL, cache = layerActivationForward(A, parameters['w'+str(L)],
214 parameters['b'+str(L)], activationFunc = outputActivationFunc)
215     caches.append(cache)
216
217     return AL, caches
218
219
220 # Compute the cost
221 # Input : AL-Activation of last layer
222 #          : Y
223 #          :outputActivationFunc - Activation function at output -
224 sigmoid/softmax
225 # Output: cost
226 def computeCost(AL,Y,outputActivationFunc="sigmoid"):
227
228     if outputActivationFunc=="sigmoid":
229         m= float(Y.shape[1])
230         # Element wise multiply for logprobs
231         cost=-1/m *np.sum(Y*np.log(AL) + (1-Y)*(np.log(1-AL)))
232         cost = np.squeeze(cost)
233     elif outputActivationFunc=="softmax":
234         # Note:Take transpose of Y for softmax
235         Y=Y.T
236         m= float(len(Y))
237         # Compute log probs. Take the log prob of correct class based on
238 output y
239         correct_logprobs = -np.log(AL[range(int(m)),Y.T])
240         # Compute loss
241         cost = np.sum(correct_logprobs)/m
242     return cost
243
244 # Compute the backpropagation for 1 cycle
245 # Input : Neural Network parameters - dA
246 #          # cache - forward_cache & activation_cache
247 #          # Y
248 #          # activationFunc # relu, tanh, sigmoid,softmax
249 # Returns: da_prev, dw, db
250 # dL/dwi= dL/dzi*A1-1
251 # dL/dbl = dL/dz1
252 # dL/dz_prev=dL/dz1*w
253 def layerActivationBackward(dA, cache, Y, activationFunc):
254     forward_cache, activation_cache = cache
255     A_prev, w, b = forward_cache

```

```

256     numtraining = float(A_prev.shape[1])
257     #print("n=",numtraining)
258     #print("no=",numtraining)
259     if activationFunc == "relu":
260         dZ = reluDerivative(dA, activation_cache)
261     elif activationFunc == "sigmoid":
262         dZ = sigmoidDerivative(dA, activation_cache)
263     elif activationFunc == "tanh":
264         dZ = tanhDerivative(dA, activation_cache)
265     elif activationFunc == "softmax":
266         dZ = stableSoftmaxDerivative(dA, activation_cache,Y,numtraining)
267
268     if activationFunc == 'softmax':
269         dw = 1/numtraining * np.dot(A_prev,dZ)
270         db = 1/numtraining * np.sum(dZ, axis=0, keepdims=True)
271         dA_prev = np.dot(dZ,W)
272     else:
273         #print(numtraining)
274         dw = 1/numtraining *(np.dot(dZ,A_prev.T))
275         #print("dw=",dw)
276         db = 1/numtraining * np.sum(dZ, axis=1, keepdims=True)
277         #print("db=",db)
278         dA_prev = np.dot(W.T,dZ)
279
280
281     return dA_prev, dw, db
282
283 # Compute the backpropoagation for 1 cycle
284 # Input : AL: Output of L layer Network - weights
285 #          # Y Real output
286 #          # caches -- list of caches containing:
287 #          # every cache of layerActivationForward() with "relu"/"tanh"
288 #          # #(it's caches[], for l in range(L-1) i.e l = 0...L-2)
289 #          # the cache of layerActivationForward() with "sigmoid" (it's caches[L-1])
290 #          hiddenActivationFunc - Activation function at hidden layers -
291 #          relu/sigmoid/tanh
292 #          outputActivationFunc - Activation function at output -
293 #          sigmoid/softmax
294 #
295 #      Returns:
296 #          gradients -- A dictionary with the gradients
297 #                      gradients["dA" + str(l)] = ...
298 #                      gradients["dw" + str(l)] = ...
299 #                      gradients["db" + str(l)]
300 def backwardPropagationDeep(AL, Y,
301 caches,hiddenActivationFunc='relu',outputActivationFunc="sigmoid"):
302     #initialize the gradients
303     gradients = {}
304     # Set the number of layers
305     L = len(caches)
306     m = float(AL.shape[1])
307
308     if outputActivationFunc == "sigmoid":
309         Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL
310         # Initializing the backpropagation
311         # dL/dAL= -(y/a + (1-y)/(1-a)) - At the output layer
312         dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
313     else:
314         dAL =0
315         Y=Y.T
316
317     # Since this is a binary classification the activation at output is
318     sigmoid

```

```

320     # Get the gradients at the last layer
321     # Inputs: "AL, Y, caches".
322     # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dB"]
323     current_cache = caches[L-1]
324     gradients["dA" + str(L)], gradients["dW" + str(L)], gradients["db" +
325     str(L)] = layerActivationBackward(dAL, current_cache, Y, activationFunc =
326     outputActivationFunc)
327
328     # Note dA for softmax is the transpose
329     if outputActivationFunc == "softmax":
330         gradients["dA" + str(L)] = gradients["dA" + str(L)].T
331     # Traverse in the reverse direction
332     for l in reversed(range(L-1)):
333         # Compute the gradients for L-1 to 1 for Relu/tanh
334         # Inputs: "gradients["dA" + str(l + 2)], caches".
335         # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l +
336         1)] , gradients["db" + str(l + 1)]
337         current_cache = caches[l]
338
339         #dA_prev_temp, dw_temp, db_temp =
340         layerActivationBackward(gradients['dA'+str(l+2)], current_cache,
341         activationFunc = "relu")
342         dA_prev_temp, dw_temp, db_temp =
343         layerActivationBackward(gradients['dA'+str(l+2)], current_cache, Y,
344         activationFunc = hiddenActivationFunc)
345         gradients["dA" + str(l + 1)] = dA_prev_temp
346         gradients["dw" + str(l + 1)] = dw_temp
347         gradients["db" + str(l + 1)] = db_temp
348
349
350     return gradients
351
352     # Perform Gradient Descent
353     # Input : weights and biases
354     #          : gradients
355     #          : learning rate
356     #          : outputActivationFunc - Activation function at output -
357     #sigmoid/softmax
358     #return : parameters
359     def gradientDescent(parameters, gradients,
360     learningRate, outputActivationFunc="sigmoid"):
361
362         L = int(len(parameters) / 2)
363         # Update rule for each parameter.
364         for l in range(L-1):
365             parameters["w" + str(l+1)] = parameters['w'+str(l+1)] -learningRate*
366             gradients['dw' + str(l+1)]
367             parameters["b" + str(l+1)] = parameters['b'+str(l+1)] -learningRate*
368             gradients['db' + str(l+1)]
369
370             if outputActivationFunc=="sigmoid":
371                 parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
372                 gradients['dw' + str(L)]
373                 parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
374                 gradients['db' + str(L)]
375             elif outputActivationFunc=="softmax":
376                 parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
377                 gradients['dw' + str(L)].T
378                 parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
379                 gradients['db' + str(L)].T
380
381
382
383     return parameters

```

```

384
385 # Execute a L layer Deep learning model
386 # Input : X1 - Input features
387 #       : Y1 output
388 #       : layersDimensions - Dimension of layers
389 #       : hiddenActivationFunc - Activation function at hidden layer relu
390 /tanh/sigmoid
391 #       : outputActivationFunc - sigmoid/softmax
392 #       : learning rate
393 #       : num of iteration
394 # output : parameters
395
396 def L_Layer_DeepModel(x1, Y1, layersDimensions, hiddenActivationFunc='relu',
397 outputActivationFunc="sigmoid", learningRate = .3, num_iterations = 10000,
398 print_cost=False):#lr was 0.009
399
400     np.random.seed(1)
401     costs = []
402
403     # Parameters initialization.
404     parameters = initializeDeepModel(layersDimensions)
405
406     # Loop (gradient descent)
407     for i in range(0, num_iterations):
408         # Forward propagation: [LINEAR -> RELU]**(L-1) -> LINEAR -> SIGMOID.
409         #AL, caches = forwardPropagationDeep(x,
410 parameters,hiddenActivationFunc)
411
412         # Compute cost.
413         #cost = computeCost(AL, Y)
414
415         # Backward propagation.
416         #gradients = backwardPropagationDeep(AL, Y,
417 caches,hiddenActivationFunc)
418
419         ## Update parameters.
420         #parameters = gradientDescent(parameters, gradients, learning_rate)
421
422         AL, caches = forwardPropagationDeep(x1,
423 parameters,hiddenActivationFunc="relu",outputActivationFunc=outputActivationF
424 unc)
425
426         # Compute cost
427         cost = computeCost(AL, Y1,outputActivationFunc=outputActivationFunc)
428         #print("Y1=",Y1.shape)
429         # Backward propagation.
430         gradients = backwardPropagationDeep(AL, Y1,
431 caches,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
432
433         # update parameters.
434         parameters = gradientDescent(parameters, gradients,
435 learningRate=learningRate,outputActivationFunc=outputActivationFunc)
436
437
438         # Print the cost every 100 training example
439         if print_cost and i % 1000 == 0:
440             print ("Cost after iteration %i: %f" %(i, cost))
441         if print_cost and i % 1000 == 0:
442             costs.append(cost)
443
444         # plot the cost
445         plt.plot(np.squeeze(costs))
446         plt.ylabel('cost')
447         plt.xlabel('No of iterations (x100)')

```

```

448     plt.title("Learning rate =" + str(learningRate))
449     #plt.show()
450     plt.savefig("fig1",bbox_inches='tight')
451
452     return parameters
453
454 # Execute a L layer Deep learning model Stoachastic Gradient Descent
455 # Input : X1 - Input features
456 #           : Y1- output
457 #           : layersDimensions - Dimension of layers
458 #           : hiddenActivationFunc - Activation function at hidden layer relu
459 #           : outputActivationFunc - Activation function at output -
460 #           : sigmoid/tanh
461 #           : learning rate
462 #           : mini_batch_size
463 #           : num_epochs
464 #output : parameters
465 def L_Layer_DeepModel_SGD(X1, Y1, layersDimensions,
466 hiddenActivationFunc='relu', outputActivationFunc="sigmoid", learningRate =
467 .3, mini_batch_size = 64, num_epochs = 2500, print_cost=False):#lr was 0.009
468
469     np.random.seed(1)
470     costs = []
471
472     # Parameters initialization.
473     parameters = initializeDeepModel(layersDimensions)
474     seed=10
475
476     # Loop for number of epochs
477     for i in range(num_epochs):
478         # Define the random minibatches. We increment the seed to reshuffle
479         # differently the dataset after each epoch
480         seed = seed + 1
481         minibatches = random_mini_batches(X1, Y1, mini_batch_size, seed)
482
483         batch=0
484         # Loop through each mini batch
485         for minibatch in minibatches:
486             #print("batch=",batch)
487             batch=batch+1
488             # Select a minibatch
489             (minibatch_X, minibatch_Y) = minibatch
490
491             # Perfrom forward propagation
492             AL, caches = forwardPropagationDeep(minibatch_X,
493 parameters,hiddenActivationFunc="relu",outputActivationFunc=outputActivationF
494 unc)
495
496             # Compute cost
497             cost = computeCost(AL,
498 minibatch_Y,outputActivationFunc=outputActivationFunc)
499             #print("minibatch_Y=",minibatch_Y.shape)
500             # Backward propagation.
501             gradients = backwardPropagationDeep(AL, minibatch_Y,
502 caches,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
503
504             # Update parameters.
505             parameters = gradientDescent(parameters, gradients,
506 learningRate=learningRate,outputActivationFunc=outputActivationFunc)
507
508             # Print the cost every 1000 epoch
509             if print_cost and i % 100 == 0:
510                 print ("Cost after epoch %i: %f" %(i, cost))
511             if print_cost and i % 100 == 0:

```

```

512         costs.append(cost)
513
514         # plot the cost
515         plt.plot(np.squeeze(costs))
516         plt.ylabel('cost')
517         plt.xlabel('No of iterations')
518         plt.title("Learning rate = " + str(learningRate))
519         #plt.show()
520         plt.savefig("fig1",bbox_inches='tight')
521
522     return parameters
523
524
525 # Create random mini batches
526 # Input : X - Input features
527 #           : Y- output
528 #           : miniBatchSizes
529 #           : seed
530 #output : mini_batches
531 def random_mini_batches(X, Y, miniBatchSize = 64, seed = 0):
532
533     np.random.seed(seed)
534     # Get number of training samples
535     m = X.shape[1]
536     # Initialize mini batches
537     mini_batches = []
538
539     # Create a list of random numbers < m
540     permutation = list(np.random.permutation(m))
541     # Randomly shuffle the training data
542     shuffled_X = X[:, permutation]
543     shuffled_Y = Y[:, permutation].reshape((1,m))
544
545     # Compute number of mini batches
546     numCompleteMinibatches = math.floor(m/miniBatchsize)
547
548     # For the number of mini batches
549     for k in range(0, numCompleteMinibatches):
550
551         # Set the start and end of each mini batch
552         mini_batch_X = shuffled_X[:, k*miniBatchSize : (k+1) * miniBatchsize]
553         mini_batch_Y = shuffled_Y[:, k*miniBatchSize : (k+1) * miniBatchsize]
554
555         mini_batch = (mini_batch_X, mini_batch_Y)
556         mini_batches.append(mini_batch)
557
558
559     #if m % miniBatchSize != 0:. The batch does not evenly divide by the mini
560     batch
561     if m % miniBatchSize != 0:
562         l=math.floor(m/miniBatchsize)*miniBatchsize
563         # Set the start and end of last mini batch
564         m=l+m % miniBatchSize
565         mini_batch_X = shuffled_X[:,l:m]
566         mini_batch_Y = shuffled_Y[:,l:m]
567
568         mini_batch = (mini_batch_X, mini_batch_Y)
569         mini_batches.append(mini_batch)
570
571     return mini_batches
572
573 # Plot a decision boundary
574 # Input : Input Model,
575 #           X

```

```

576 #           Y
577 #           lr - Learning rate
578 #           Fig to be saved as
579 # Returns Null
580 def plot_decision_boundary(model, x, y,lr,fig):
581     # Set min and max values and give it some padding
582     x_min, x_max = x[0, :].min() - 1, x[0, :].max() + 1
583     y_min, y_max = x[1, :].min() - 1, x[1, :].max() + 1
584     colors=['black','yellow']
585     cmap = matplotlib.colors.ListedColormap(colors)
586     h = 0.01
587     # Generate a grid of points with distance h between them
588     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,
589     h))
590     # Predict the function value for the whole grid
591     Z = model(np.c_[xx.ravel(), yy.ravel()])
592     Z = Z.reshape(xx.shape)
593     # Plot the contour and training examples
594     plt.contourf(xx, yy, Z, cmap="coolwarm")
595     plt.ylabel('x2')
596     plt.xlabel('x1')
597     plt.scatter(x[0, :], x[1, :], c=y, s=7,cmap=cmap)
598     plt.title("Decision Boundary for learning rate:"+str(lr))
599     #plt.show()
600     plt.savefig(fig, bbox_inches='tight')
601
602 def predict(parameters, x):
603     A2, cache = forwardPropagationDeep(x, parameters)
604     predictions = (A2>0.5)
605     return predictions
606
607 def predict_proba(parameters, x, outputActivationFunc="sigmoid"):
608     A2, cache = forwardPropagationDeep(x, parameters)
609     if outputActivationFunc=="sigmoid":
610         proba=A2
611     elif outputActivationFunc=="softmax":
612         proba=np.argmax(A2, axis=0).reshape(-1,1)
613         print("A2=",A2.shape)
614     return proba
615
616
617 # Plot a decision boundary
618 # Input : Input Model,
619 #           X
620 #           Y
621 #           sz - Num of hidden units
622 #           lr - Learning rate
623 #           Fig to be saved as
624 # Returns Null
625 def plot_decision_surface(model, x, y,sz,lr,fig):
626     # Set min and max values and give it some padding
627     x_min, x_max = x[0, :].min() - 1, x[0, :].max() + 1
628     y_min, y_max = x[1, :].min() - 1, x[1, :].max() + 1
629     z_min, z_max = x[2, :].min() - 1, x[2, :].max() + 1
630     colors=['black','gold']
631     cmap = matplotlib.colors.ListedColormap(colors)
632     h = 3
633     # Generate a grid of points with distance h between them
634     xx, yy, zz = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h),
635     np.arange(z_min, z_max, h))
636     # Predict the function value for the whole grid
637     a=np.c_[xx.ravel(), yy.ravel(), zz.ravel()]
638
639     Z = predict(parameters,a.T)

```

```

640     Z = Z.reshape(xx.shape)
641     # Plot the contour and training examples
642     #plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
643     fig = plt.figure()
644     ax = plt.axes(projection='3d')
645     ax.contour3D(xx, yy, Z, 50, cmap='binary')
646     #plt.ylabel('x2')
647     #plt.xlabel('x1')
648     plt.scatter(X[0, :], X[1, :], c=y, cmap=cmap)
649     plt.title("Decision Boundary for hidden layer size:" + sz + " and learning
rate:" + lr)
650     plt.show()
652
653 def plotSurface(X,parameters):
654
655     #xx, yy, zz = np.meshgrid(np.arange(10), np.arange(10), np.arange(10))
656     x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
657     y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
658     z_min, z_max = X[2, :].min() - 1, X[2, :].max() + 1
659     colors=['red']
660     cmap = matplotlib.colors.ListedColormap(colors)
661     h = 1
662     xx, yy, zz = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,
y_max, h),
663                                     np.arange(z_min, z_max, h))
664     # For the mesh grid values predict a model
665     a=np.c_[xx.ravel(), yy.ravel(), zz.ravel()]
666     Z = predict(parameters,a.T)
667     r=Z.T
668     r1=r.reshape(xx.shape)
669     # Find the values for which the prediction is 1
670     xx1=xx[r1]
671     yy1=yy[r1]
672     zz1=zz[r1]
673     # Plot these values
674     ax = plt.axes(projection='3d')
675     #ax.plot_trisurf(xx1, yy1, zz1, cmap='bone', edgecolor='none');
676     ax.scatter3D(xx1, yy1, zz1, c=zz1,s=10,cmap=cmap)
677     #ax.plot_surface(xx1, yy1, zz1, 'gray')
678

```

5.2 R

```

1 ##########
2 #####
3 #
4 # File   : DLfunctions5.R
5 # Author : Tinniam V Ganesh
6 # Date   : 22 Mar 2018
7 #
8 #####
9 #####
10 library(ggplot2)
11 library(PRROC)
12 library(dplyr)
13
14 # Compute the sigmoid of a vector
15 sigmoid <- function(z){

```

```

16 A <- 1/(1+ exp(-Z))
17 cache<-Z
18 retvals <- list("A"=A, "Z"=Z)
19 return(retvals)
20
21 }
22
23 # Compute the Relu(old) of a vector (performance hog!)
24 reluOld <-function(Z){
25   A <- apply(Z, 1:2, function(x) max(0,x))
26   cache<-Z
27   retvals <- list("A"=A, "Z"=Z)
28   return(retvals)
29 }
30
31 # Compute the Relu (current) of a vector (much better performance!)
32 relu <-function(Z){
33   # Perform relu. Set values less than equal to 0 as 0
34   Z[Z<0]=0
35   A=Z
36   cache<-Z
37   retvals <- list("A"=A, "Z"=Z)
38   return(retvals)
39 }
40
41 # Compute the tanh activation of a vector
42 tanhActivation <- function(Z){
43   A <- tanh(Z)
44   cache<-Z
45   retvals <- list("A"=A, "Z"=Z)
46   return(retvals)
47 }
48
49 # Compute the softmax of a vector
50 softmax <- function(Z){
51   # get unnormalized probabilities
52   exp_scores = exp(t(Z))
53   # normalize them for each example
54   A = exp_scores / rowSums(exp_scores)
55   retvals <- list("A"=A, "Z"=Z)
56   return(retvals)
57 }
58
59 # Compute the derivative of Relu
60 # g'(z) = 1 if z >0 and 0 otherwise
61 reluDerivative <-function(dA, cache){
62   Z <- cache
63   dZ <- dA
64   # Create a logical matrix of values > 0
65   a <- Z > 0
66   # When z <= 0, you should set dz to 0 as well. Perform an element wise
67   multiple
68   dZ <- dZ * a
69   return(dZ)
70 }
71
72 # Compute the derivative of sigmoid
73 # Derivative g'(z) = a*(1-a)
74 sigmoidDerivative <- function(dA, cache){
75   Z <- cache
76   s <- 1/(1+exp(-Z))
77   dZ <- dA * s * (1-s)
78   return(dZ)
79 }
```

```

80
81 # Compute the derivative of tanh
82 # Derivative g'(z) = 1- a^2
83 tanhDerivative <- function(dA, cache){
84   z = cache
85   a = tanh(z)
86   dz = dA * (1 - a^2)
87   return(dz)
88 }
89
90 # This function is used in computing the softmax derivative
91 # Populate a matrix of 1s in rows where Y==1
92 # This may need to be extended for K classes. Currently
93 # supports K=3 & K=10
94 popMatrix <- function(Y,numClasses){
95   a=rep(0,times=length(Y))
96   Y1=matrix(a,nrow=length(Y),ncol=numClasses)
97   #Set the rows and columns as 1's where Y is the class value
98   if(numClasses==3){
99     Y1[Y==0,1]=1
100    Y1[Y==1,2]=1
101    Y1[Y==2,3]=1
102  } else if (numClasses==10){
103    Y1[Y==0,1]=1
104    Y1[Y==1,2]=1
105    Y1[Y==2,3]=1
106    Y1[Y==3,4]=1
107    Y1[Y==4,5]=1
108    Y1[Y==5,6]=1
109    Y1[Y==6,7]=1
110    Y1[Y==7,8]=1
111    Y1[Y==8,9]=1
112    Y1[Y==9,0]=1
113  }
114  return(Y1)
115 }
116
117 # Compute the softmax derivative
118 softmaxDerivative <- function(dA, cache ,y,numTraining,numClasses){
119   # Note : dA not used. dL/dz = dL/dA * dA/dZ = pi-yi
120   z <- cache
121   # Compute softmax
122   exp_scores = exp(t(z))
123   # normalize them for each example
124   probs = exp_scores / rowSums(exp_scores)
125   # Create a matrix of zeros
126   Y1=popMatrix(y,numClasses)
127   dZ = probs-Y1
128   return(dZ)
129 }
130
131
132 # Initialize model for L layers
133 # Input : Vector of units in each layer
134 # Returns: Initial weights and biases matrices for all layers
135 initializeDeepModel <- function(layerDimensions){
136   set.seed(2)
137
138   # Initialize empty list
139   layerParams <- list()
140
141   # Note the weight matrix at layer '1' is a matrix of size (1,1-1)
142   # The Bias is a vectors of size (1,1)
143 }
```

```

144 # Loop through the layer dimension from 1.. L
145 # Indices in R start from 1
146 for(l in 2:length(layersDimensions)){
147     # Initialize a matrix of small random numbers of size 1 x 1-1
148     # Create random numbers of size 1 x 1-1
149     w=rnorm(layersDimensions[1]*layersDimensions[1-1])*0.01
150
151     # Create a weight matrix of size 1 x 1-1 with this initial weights and
152     # Add to list W1,W2,.. WL
153     layerParams[[paste('w',l-1,sep="")]] = matrix(w,nrow=layersDimensions[1],
154                                                 ncol=layersDimensions[1-1])
155     layerParams[[paste('b',l-1,sep="")]] = matrix(rep(0,layersDimensions[1]),
156                                                 nrow=layersDimensions[1],ncol=1)
157 }
158 return(layerParams)
159 }
160
161
162 # Compute the activation at a layer 'l' for forward prop in a Deep Network
163 # Input : A_prev - Activation of previous layer
164 #          W,b - Weight and bias matrices and vectors
165 #          activationFunc - Activation function - sigmoid, tanh, relu etc
166 # Returns : forward_cache,activation_cache,cache
167 # :
168 #
169 # Z = W * X + b
170 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
171 layerActivationForward <- function(A_prev, W, b, activationFunc){
172
173     # Compute Z
174     z = W %*% A_prev
175     # Broadcast the bias 'b' by column
176     z <- sweep(z,1,b,'+')
177
178     forward_cache <- list("A_prev"=A_prev, "W"=W, "b"=b)
179     # Compute the activation for sigmoid
180     if(activationFunc == "sigmoid"){
181         vals = sigmoid(z)
182     } else if (activationFunc == "relu"){ # Compute the activation for relu
183         vals = relu(z)
184     } else if(activationFunc == 'tanh'){ # Compute the activation for tanh
185         vals = tanhActivation(z)
186     } else if(activationFunc == 'softmax'){
187         vals = softmax(z)
188     }
189     # Create a list of forward and activation cache
190     cache <- list("forward_cache"=forward_cache,
191 "activation_cache"=vals[['Z']])
192     retvals <- list("A"=vals[['A']], "cache"=cache)
193     return(retvals)
194 }
195
196 # Compute the forward propagation for layers 1..L
197 # Input : X - Input Features
198 #          parameters: Weights and biases
199 #          hiddenActivationFunc - relu/sigmoid/tanh
200 #          outputActivationFunc - Activation function at hidden layer
201 #          sigmoid/softmax
202 # Returns : AL
203 #          caches
204 # The forward propagation uses the Relu/tanh activation from layer 1..L-1 and
205 # sigmoid activation at layer L
206 forwardPropagationDeep <- function(X, parameters,hiddenActivationFunc='relu',
207                                     outputActivationFunc='sigmoid'){


```

```

208 caches <- list()
209 # Set A to X (A0)
210 A <- X
211 L <- length(parameters)/2 # number of layers in the neural network
212 # Loop through from layer 1 to upto layer L
213 for(l in 1:(L-1)){
214   A_prev <- A
215   # Zi = Wi x Ai-1 + bi and Ai = g(Zi)
216   # Set w and b for layer 'l'
217   # Loop throug from w1,w2... WL-1
218   w <- parameters[[paste("w",l,sep="")]]
219   b <- parameters[[paste("b",l,sep="")]]
220   # Compute the forward propagation through layer 'l' using the activation
221   function
222   actForward <- layerActivationForward(A_prev,
223                                         w,
224                                         b,
225                                         activationFunc =
226                                         hiddenActivationFunc)
227   A <- actForward[['A']]
228   # Append the cache A_prev,w,b, z
229   caches[[l]] <- actForward
230 }
231
232 # Since this is binary classification use the sigmoid activation function
233 in
234 # last layer
235 # Set the weights and biases for the last layer
236 w <- parameters[[paste("w",L,sep="")]]
237 b <- parameters[[paste("b",L,sep="")]]
238 # Compute the sigmoid activation
239 actForward = layerActivationForward(A, w, b, activationFunc =
240 outputActivationFunc)
241 AL <- actForward[['A']]
242 # Append the output of this forward propagation through the last layer
243 caches[[L]] <- actForward
244 # Create a list of the final output and the caches
245 fwdPropDeep <- list("AL"=AL,"caches"=caches)
246 return(fwdPropDeep)
247 }
248
249
250 # Function pickColumns(). This function is in computeCost()
251 # Pick columns
252 # Input : AL
253 #           : Y
254 #           : numClasses
255 # Output: a
256 pickColumns <- function(AL,Y,numClasses){
257   if(numClasses==3){
258     # Select the elements where the y values are 0, 1 or 2 and make a
259     vector
260     a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3])
261   }
262   else if (numClasses==10){
263     # Select the elements where the y values are 0,1,2...,9
264     a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3],AL[Y==3,4],AL[Y==4,5],
265          AL[Y==5,6],AL[Y==6,7],AL[Y==7,8],AL[Y==8,9],AL[Y==9,10])
266   }
267   return(a)
268 }
269
270
271 # Compute the cost

```

```

272 # Input : AL
273 #      : Y
274 #      : outputActivationFunc - Activation function at hidden layer
275 sigmoid/softmax
276 #      : numClasses
277 # Output: cost
278 computeCost <- function(AL,Y,outputActivationFunc="sigmoid",numClasses=3){
279   if(outputActivationFunc=="sigmoid"){
280     m= length(Y)
281     cost=-1/m*sum(Y*log(AL) + (1-Y)*log(1-AL))
282   }else if (outputActivationFunc=="softmax"){
283     # Pick columns
284     m= length(Y)
285     a =pickColumns(AL,Y,numClasses)
286     # Take log
287     correct_probs = -log(a)
288     # Compute loss
289     cost= sum(correct_probs)/m
290   }
291   #cost=-1/m*sum(a+b)
292   return(cost)
293 }
294
295
296 # Compute the backpropagation through a layer
297 # Input : Neural Network parameters - dA
298 #      # cache - forward_cache & activation_cache
299 #      # Output values Y
300 #      # activationFunc
301 #      # numClasses
302 # Returns: Gradients - dA_prev, dw,db
303 # dL/dwi= dL/dzi*A1-1
304 # dL/db1 = dL/dz1
305 # dL/dz_prev=dL/dz1*w
306
307 layerActivationBackward <- function(dA, cache, Y,
308 activationFunc,numClasses){
309   # Get A_prev,w,b
310   forward_cache <-cache[['forward_cache']]
311   activation_cache <- cache[['activation_cache']]
312   A_prev <- forward_cache[['A_prev']]
313   numtraining = dim(A_prev)[2]
314   # Get Z
315   activation_cache <- cache[['activation_cache']]
316   if(activationFunc == "relu"){
317     dz <- reluDerivative(dA, activation_cache)
318   } else if(activationFunc == "sigmoid"){
319     dz <- sigmoidDerivative(dA, activation_cache)
320   } else if(activationFunc == "tanh"){
321     dz <- tanhDerivative(dA, activation_cache)
322   } else if(activationFunc == "softmax"){
323     dz <- softmaxDerivative(dA, activation_cache,Y,numtraining,numClasses)
324   }
325   # Check if softmax
326   if (activationFunc == 'softmax'){
327     w <- forward_cache[['w']]
328     b <- forward_cache[['b']]
329     dw = 1/numtraining * A_prev%*%dz
330     db = 1/numtraining* matrix(colSums(dz),nrow=1,ncol=numClasses)
331     dA_prev = dz %*%w
332   } else {
333     w <- forward_cache[['w']]
334     b <- forward_cache[['b']]
335     numtraining = dim(A_prev)[2]

```

```

336     dw = 1/numtraining * dz %*% t(A_prev)
337     db = 1/numtraining * rowSums(dz)
338     dA_prev = t(W) %*% dz
339   }
340   retvals <- list("dA_prev"=dA_prev,"dw"=dw,"db"=db)
341   return(retvals)
342 }
343
344 # Compute the backpropagation for 1 cycle through all layers
345 # Input : AL: Output of L layer Network - weights
346 #           # Y Real output
347 #           # caches -- list of caches containing:
348 #           # every cache of layerActivationForward() with "relu"/"tanh"
349 #           # (it's caches[], for l in range(L-1) i.e l = 0...L-2)
350 #           # the cache of layerActivationForward() with "sigmoid" (it's caches[L-
351 1])
352 #           hiddenActivationFunc - Activation function at hidden layers -
353 relu/tanh/sigmoid
354 #           outputActivationFunc - Activation function at hidden layer
355 sigmoid/softmax
356 #           numClasses
357 #   Returns:
358 #       gradients -- listwith the gradients
359 #           gradients["dA" + str(l)]
360 #           gradients["dw" + str(l)]
361 #           gradients["db" + str(l)]
362 backwardPropagationDeep <- function(AL, Y,
363 caches,hiddenActivationFunc='relu',
364                                         outputActivationFunc="sigmoid",numClasses){
365   #initialize the gradients
366   gradients = list()
367   # Set the number of layers
368   L = length(caches)
369   numTraining = dim(AL)[2]
370
371   if(outputActivationFunc == "sigmoid")
372     # Initializing the backpropagation
373     #  $dL/dAL = -(y/a) - ((1-y)/(1-a))$  - At the output layer
374     dAL = -(Y/AL) -(1 - Y)/(1 - AL)
375   else if(outputActivationFunc == "softmax"){
376     dAL=0
377     Y=t(Y)
378   }
379
380   # Get the gradients at the last layer
381   # Inputs: "AL, Y, caches".
382   # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"]
383   # Start with Layer L
384   # Get the current cache
385   current_cache = caches[[L]]$cache
386   #gradients["dA" + str(L)], gradients["dw" + str(L)], gradients["db" +
387   str(L)] = layerActivationBackward(dAL, current_cache, activationFunc =
388   "sigmoid")
389   retvals <- layerActivationBackward(dAL, current_cache, Y, activationFunc =
390   outputActivationFunc,numClasses)
391   # Create gradients as lists
392   #Note: Take the transpose of dA
393   if(outputActivationFunc == "sigmoid")
394     gradients[[paste("dA",L,sep="")]] <- retvals[['dA_prev']]
395   else if(outputActivationFunc == "softmax")
396     gradients[[paste("dA",L,sep="")]] <- t(retvals[['dA_prev']])
397     gradients[[paste("dw",L,sep="")]] <- retvals[['dw']]
398     gradients[[paste("db",L,sep="")]] <- retvals[['db']]
399

```

```

400 # Traverse in the reverse direction
401 for(l in (L-1):1){
402     # Compute the gradients for L-1 to 1 for Relu/tanh
403     # Inputs: "gradients["dA" + str(l + 2)], caches".
404     # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l + 1)] ,
405     gradients["db" + str(l + 1)]
406     current_cache = caches[[l]]$cache
407
408     retvals = layerActivationBackward(gradients[[paste('dA',l+1,sep="")]], 
409                                         current_cache,
410                                         activationFunc = hiddenActivationFunc)
411
412     gradients[[paste("dA",l,sep="")]] <- retvals[['dA_prev']]
413     gradients[[paste("dw",l,sep="")]] <- retvals[['dw']]
414     gradients[[paste("db",l,sep="")]] <- retvals[['db']]
415 }
416
417 return(gradients)
418 }

420
421 # Perform Gradient Descent
422 # Input : Weights and biases
423 #       : gradients
424 #       : Learning rate
425 #       : outputActivationFunc - Activation function at hidden layer
426 sigmoid/softmax
427 # output : parameters
428 gradientDescent <- function(parameters, gradients,
429 learningRate,outputActivationFunc="sigmoid"){
430
431 L = length(parameters)/2 # number of layers in the neural network
432
433 # Update rule for each parameter. Use a for loop.
434 for(l in 1:(L-1)){
435     parameters[[paste("w",l,sep="")]] = parameters[[paste("w",l,sep="")]] - 
436         learningRate* gradients[[paste("dw",l,sep="")]]
437     parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] - 
438         learningRate* gradients[[paste("db",l,sep="")]]
439 }
440 if(outputActivationFunc=="sigmoid"){
441     parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]] - 
442         learningRate* gradients[[paste("dw",L,sep="")]]
443     parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - 
444         learningRate* gradients[[paste("db",L,sep="")]]
445
446 }else if (outputActivationFunc=="softmax"){
447     parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]] - 
448         learningRate* gradients[[paste("dw",L,sep="")]]
449     parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - 
450         learningRate* gradients[[paste("db",L,sep="")]]
451 }
452 return(parameters)
453 }

455
456 # Execute a L layer Deep Learning model
457 # Input : X - Input features
458 #       : Y output
459 #       : layersDimensions - Dimension of layers
460 #       : hiddenActivationFunc - Activation function at hidden layer relu
461 /tanh/sigmoid
462 #       : outputActivationFunc - Activation function at hidden layer
463 sigmoid/softmax

```

```

464 #      : learning rate
465 #      : num of iterations
466 #output : Updated weights after each iteration
467
468 L_Layer_DeepModel <- function(X, Y, layersDimensions,
469                         hiddenActivationFunc='relu',
470                         outputActivationFunc= 'sigmoid',
471                         learningRate = 0.5,
472                         numIterations = 10000,
473                         print_cost=False){
474     #Initialize costs vector as NULL
475     costs <- NULL
476
477     # Parameters initialization.
478     parameters = initializeDeepModel(layersDimensions)
479
480
481     # Loop (gradient descent)
482     for( i in 0:numIterations){
483         # Forward propagation: [LINEAR -> RELU]* (L-1) -> LINEAR ->
484         # SIGMOID/SOFTMAX.
485         retvals = forwardPropagationDeep(X, parameters,hiddenActivationFunc,
486
487         outputActivationFunc=outputActivationFunc)
488         AL <- retvals[['AL']]
489         caches <- retvals[['caches']]
490
491         # Compute cost.
492         cost <- computeCost(AL,
493             Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[3])
494
495         # Backward propagation.
496         gradients = backwardPropagationDeep(AL, Y, caches,hiddenActivationFunc,
497
498         outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[3])
499
500         # Update parameters.
501         parameters = gradientDescent(parameters, gradients, learningRate,
502                                         outputActivationFunc=outputActivationFunc)
503
504
505         if(i%%1000 == 0){
506             costs=c(costs,cost)
507             print(cost)
508         }
509     }
510
511     retvals <- list("parameters"=parameters,"costs"=costs)
512
513     return(retvals)
514 }
515
516 # Execute a L layer Deep learning model with Stochastic Gradient descent
517 # Input : X - Input features
518 #      : Y output
519 #      : layersDimensions - Dimension of layers
520 #      : hiddenActivationFunc - Activation function at hidden layer relu
521 # /tanh/sigmoid
522 #      : outputActivationFunc - Activation function at hidden layer
523 # sigmoid/softmax
524 #      : learning rate
525 #      : mini_batch_size
526 #      : num of epochs
527 #output : Updated weights after each iteration

```

```

528 L_Layer_DeepModel_SGD <- function(X, Y, layersDimensions,
529   hiddenActivationFunc='relu',
530   outputActivationFunc= 'sigmoid',
531   learningRate = .3,
532   mini_batch_size = 64,
533   num_epochs = 2500,
534   print_cost=False){
535
536   set.seed(1)
537   #Initialize costs vector as NULL
538   costs <- NULL
539
540   # Parameters initialization.
541   parameters = initializeDeepModel(layersDimensions)
542   seed=10
543
544   # Loop for number of epochs
545   for( i in 0:num_epochs){
546     seed=seed+1
547     minibatches = random_mini_batches(X, Y, mini_batch_size, seed)
548
549     for(batch in 1:length(minibatches)){
550
551       mini_batch_X=minibatches[[batch]][['mini_batch_X']]
552       mini_batch_Y=minibatches[[batch]][['mini_batch_Y']]
553       # Forward propagation:
554       retvals = forwardPropagationDeep(mini_batch_X,
555   parameters,hiddenActivationFunc,
556
557   outputActivationFunc=outputActivationFunc)
558     AL <- retvals[['AL']]
559     caches <- retvals[['caches']]
560
561     # Compute cost.
562     cost <- computeCost(AL,
563   mini_batch_Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)])
564
565     # Backward propagation.
566     gradients = backwardPropagationDeep(AL, mini_batch_Y,
567   caches,hiddenActivationFunc,
568
569   outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
570   layersDimensions)])
571
572     # Update parameters.
573     parameters = gradientDescent(parameters, gradients, learningRate,
574
575   outputActivationFunc=outputActivationFunc)
576     }
577
578     if(i%%100 == 0){
579       costs=c(costs,cost)
580       print(cost)
581     }
582   }
583
584   retvals <- list("parameters"=parameters,"costs"=costs)
585
586   return(retvals)
587 }
588
589 # Predict the output for given input
590 # Input : parameters

```

```

592 #      : X
593 #      : hiddenActivationFunc
594 # Output: predictions
595 predict <- function(parameters, x,hiddenActivationFunc='relu'){
596
597   fwdProp <- forwardPropagationDeep(x, parameters,hiddenActivationFunc)
598   predictions <- fwdProp$AL>0.5
599
600   return (predictions)
601 }
602
603 # Predict the output
604 predictProba <- function(parameters, x,hiddenActivationFunc,
605                           outputActivationFunc){
606   retvals = forwardPropagationDeep(X, parameters,hiddenActivationFunc,
607                                   outputActivationFunc="softmax")
608   if(outputActivationFunc == "sigmoid")
609     predictions <- retvals$AL>0.5
610   else if (outputActivationFunc == "softmax")
611     predictions <- apply(retvals$AL, 1,which.max) -1
612
613   return (predictions)
614 }
615
616 # Plot a decision boundary
617 # This function uses ggplot2
618 plotDecisionBoundary <- function(z,retvals,hiddenActivationFunc,lr){
619   # Find the minimum and maximum for the data
620   xmin<-min(z[,1])
621   xmax<-max(z[,1])
622   ymin<-min(z[,2])
623   ymax<-max(z[,2])
624
625   # Create a grid of values
626   a=seq(xmin,xmax,length=100)
627   b=seq(ymin,ymax,length=100)
628   grid <- expand.grid(x=a, y=b)
629   colnames(grid) <- c('x1', 'x2')
630   grid1 <- t(grid)
631
632   # Predict the output for this grid
633   q <- predict(retvals$parameters,grid1,hiddenActivationFunc)
634   q1 <- t(data.frame(q))
635   q2 <- as.numeric(q1)
636   grid2 <- cbind(grid,q2)
637   colnames(grid2) <- c('x1', 'x2','q2')
638
639   z1 <- data.frame(z)
640   names(z1) <- c("x1", "x2", "y")
641   atitle=paste("Decision boundary for learning rate:",lr)
642   # Plot the contour of the boundary
643   ggplot(z1) +
644     geom_point(data = z1, aes(x = x1, y = x2, color = y)) +
645     stat_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2), alpha =
646     0.9)+ ggtitle(atitle)
647 }
648
649 # Predict the probability scores for given data set
650 # Input : parameters
651 #      : X
652 # Output: probability of output
653 computeScores <- function(parameters, x,hiddenActivationFunc='relu'){
654
655   fwdProp <- forwardPropagationDeep(x, parameters,hiddenActivationFunc)

```

```

656 scores <- fwdProp$AL
657
658   return (scores)
659 }
660
661 # Create random mini batches
662 # Input : X - Input features
663 #           : Y- output
664 #           : miniBatchSize
665 #           : seed
666 #output : mini_batches
667 random_mini_batches <- function(X, Y, miniBatchSize = 64, seed = 0){
668
669
670   set.seed(seed)
671   # Get number of training samples
672   m = dim(X)[2]
673   # Initialize mini batches
674   mini_batches = list()
675
676   # Create a list of random numbers < m
677   permutation = c(sample(m))
678   # Randomly shuffle the training data
679   shuffled_X = X[, permutation]
680   shuffled_Y = Y[1, permutation]
681
682   # Compute number of mini batches
683   numCompleteMinibatches = floor(m/miniBatchSize)
684   batch=0
685   for(k in 0:(numCompleteMinibatches-1)){
686     batch=batch+1
687     # Set the lower and upper bound of the mini batches
688     lower=(k*miniBatchSize)+1
689     upper=((k+1) * miniBatchSize)
690     mini_batch_X = shuffled_X[, lower:upper]
691     mini_batch_Y = shuffled_Y[lower:upper]
692     # Add it to the list of mini batches
693     mini_batch =
694     list("mini_batch_X"=mini_batch_X,"mini_batch_Y"=mini_batch_Y)
695     mini_batches[[batch]] =mini_batch
696
697   }
698
699   # If the batch size does not divide evenly with mini batch size
700   if(m % miniBatchSize != 0){
701     p=floor(m/miniBatchSize)*miniBatchSize
702     # Set the start and end of last batch
703     q=p+m %% miniBatchSize
704     mini_batch_X = shuffled_X[, (p+1):q]
705     mini_batch_Y = shuffled_Y[(p+1):q]
706   }
707   # Return the list of mini batches
708   mini_batch =
709   list("mini_batch_X"=mini_batch_X,"mini_batch_Y"=mini_batch_Y)
710   mini_batches[[batch]]=mini_batch
711
712   return(mini_batches)
713 }
```

5.3 Octave

```

1 ##########
2 #####
3 #
4 # File: DLfunctions5.m
5 # Developer: Tinniam V Ganesh
6 # Date : 23 Mar 2018
7 #
8 #####
9 #####
10 1;
11 # Define sigmoid function
12 function [A,cache] = sigmoid(z)
13     A = 1 ./ (1+ exp(-z));
14     cache=z;
15 end
16
17 # Define Relu function
18 function [A,cache] = relu(z)
19     A = max(0,z);
20     cache=z;
21 end
22
23 # Define Relu function
24 function [A,cache] = tanhAct(z)
25     A = tanh(z);
26     cache=z;
27 end
28
29 # Define Softmax function
30 function [A,cache] = softmax(z)
31     # get unnormalized probabilities
32     exp_scores = exp(z');
33     # normalize them for each example
34     A = exp_scores ./ sum(exp_scores,2);
35     cache=z;
36 end
37
38 # Define Stable Softmax function
39 function [A,cache] = stableSoftmax(z)
40     # Normalize by max value in each row
41     shiftz = z' - max(z',[],2);
42     exp_scores = exp(shiftz);
43     # normalize them for each example
44     A = exp_scores ./ sum(exp_scores,2);
45     #disp("sm")
46     #disp(A);
47     cache=z;
48 end
49
50 # Define Relu Derivative
51 function [dZ] = reluDerivative(dA,cache)
52     Z = cache;
53     dZ = dA;
54     # Get elements that are greater than 0
55     a = (Z > 0);
56     # Select only those elements where Z > 0
57     dZ = dZ .* a;
58 end
59
60 # Define Sigmoid Derivative
61 function [dZ] = sigmoidDerivative(dA,cache)
62     Z = cache;
63     s = 1 ./ (1+ exp(-Z));
64     dZ = dA .* s .* (1-s);

```

```

65 end
66
67 # Define Tanh Derivative
68 function [dz] = tanhDerivative(dA,cache)
69     z = cache;
70     a = tanh(z);
71     dz = dA .* (1 - a .^ 2);
72 end
73
74 # Populate a matrix with 1s in rows where Y=1
75 # This function may need to be modified if K is not 3, 10
76 # This function is used in computing the softmax derivative
77 function [Y1] = popMatrix(Y,numClasses)
78     Y1=zeros(length(Y),numClasses);
79     if(numClasses==3) # For 3 output classes
80         Y1(Y==0,1)=1;
81         Y1(Y==1,2)=1;
82         Y1(Y==2,3)=1;
83     elseif(numClasses==10) # For 10 output classes
84         Y1(Y==0,1)=1;
85         Y1(Y==1,2)=1;
86         Y1(Y==2,3)=1;
87         Y1(Y==3,4)=1;
88         Y1(Y==4,5)=1;
89         Y1(Y==5,6)=1;
90         Y1(Y==6,7)=1;
91         Y1(Y==7,8)=1;
92         Y1(Y==8,9)=1;
93         Y1(Y==9,10)=1;
94     endif
95 end
96
97 # Define Softmax Derivative
98 function [dz] = softmaxDerivative(dA,cache,Y, numClasses)
99     Z = cache;
100    # get unnormalized probabilities
101    shiftZ = Z' - max(Z',[],2);
102    exp_scores = exp(shiftZ);
103
104    # normalize them for each example
105    probs = exp_scores ./ sum(exp_scores,2);
106    # dz = pi- yi
107    yi=popMatrix(Y,numClasses);
108    dz=probs-yi;
109
110 end
111
112 # Define Stable Softmax Derivative
113 function [dZ] = stableSoftmaxDerivative(dA,cache,Y, numClasses)
114     Z = cache;
115     # get unnormalized probabilities
116     exp_scores = exp(Z');
117     # normalize them for each example
118     probs = exp_scores ./ sum(exp_scores,2);
119     # dz = pi- yi
120     yi=popMatrix(Y,numClasses);
121     dZ=probs-yi;
122
123 end
124
125 # Initialize model for L layers
126 # Input : List of units in each layer
127 # Returns: Initial weights and biases matrices for all layers

```

```

129 function [w b] = initializeDeepModel(layerDimensions)
130     rand ("seed", 3);
131     # note the weight matrix at layer 'l' is a matrix of size (1,1-1)
132     # The Bias is a vectors of size (1,1)
133
134     # Loop through the layer dimension from 1.. L
135     # Create cell arrays for weights and biases
136
137     for l =2:size(layerDimensions)(2)
138         w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*0.01; #
139         Multiply by .01
140         b{l-1} = zeros(layerDimensions(l),1);
141
142     endfor
143 end
144
145 # Compute the activation at a layer 'l' for forward prop in a Deep Network
146 # Input : A_prev - Activation of previous layer
147 #           W,b - Weight and bias matrices and vectors
148 #           activationFunc - Activation function - sigmoid, tanh, relu etc
149 # Returns : A, forward_cache, activation_cache
150 #
151 # Z = W * X + b
152 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
153 function [A forward_cache activation_cache] = layerActivationForward(A_prev,
154 w, b, activationFunc)
155
156     # Compute Z
157     Z = w * A_prev +b;
158     # Create a cell array
159     forward_cache = {A_prev w b};
160     # Compute the activation for sigmoid
161     if (strcmp(activationFunc,"sigmoid"))
162         [A activation_cache] = sigmoid(Z);
163     elseif (strcmp(activationFunc, "relu")) # Compute the activation for
164     Relu
165         [A activation_cache] = relu(Z);
166     elseif(strcmp(activationFunc,'tanh')) # Compute the activation for
167     tanh
168         [A activation_cache] = tanhAct(Z);
169     elseif(strcmp(activationFunc,'softmax')) # Compute the activation for
170     tanh
171         #[A activation_cache] = softmax(Z);
172         [A activation_cache] = stableSoftmax(Z);
173     endif
174
175 end
176
177 # Compute the forward propagation for layers 1..L
178 # Input : X - Input Features
179 #           paramaters: Weights and biases
180 #           hiddenActivationFunc - Activation function at hidden layers
181 Relu/tanh
182     # outputActivationFunc- sigmoid/softmax
183 # Returns : AL, forward_caches, activation_caches
184 # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and
185 # sigmoid actiovation at layer L
186 function [AL forward_caches activation_caches] = forwardPropagationDeep(X,
187 weights,biases,
188                                         hiddenActivationFunc='relu',
189 outputActivationFunc='sigmoid')
190     # Create an empty cell array
191     forward_caches = {};
192     activation_caches = {};

```

```

193     # Set A to X (A0)
194     A = X;
195     L = length(weights); # number of layers in the neural network
196     # Loop through from layer 1 to upto layer L
197     for l =1:L-1
198         A_prev = A;
199         # Zi = Wi x Ai-1 + bi and Ai = g(z)
200         W = weights{l};
201         b = biases{l};
202         [A forward_cache activation_cache] = layerActivationForward(A_prev,
203 w,b, activationFunc=hiddenActivationFunc);
204         forward_caches{l}=forward_cache;
205         activation_caches{l} = activation_cache;
206     endfor
207     # Since this is binary classification use the sigmoid activation function
208 in
209     # last layer
210     W = weights{L};
211     b = biases{L};
212     [AL, forward_cache activation_cache] = layerActivationForward(A, w,b,
213 activationFunc = outputActivationFunc);
214     forward_caches{L}=forward_cache;
215     activation_caches{L} = activation_cache;
216
217 end
218
219 # Pick columns where Y==1
220 # This function is used in computeCost
221 function [a] = pickColumns(AL,Y,numClasses)
222     if(numClasses==3)
223         a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)];
224     elseif (numClasses==10)
225         a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3);AL(Y==3,4);AL(Y==4,5);
226             AL(Y==5,6); AL(Y==6,7);AL(Y==7,8);AL(Y==8,9);AL(Y==9,10)];
227     endif
228 end
229
230
231 # Compute the cost
232 # Input : AL
233 #       : Y
234 #       : outputActivationFunc- sigmoid/softmax
235 #       : numClasses
236 # Output: cost
237 function [cost]= computeCost(AL, Y,
238 outputActivationFunc="sigmoid",numClasses)
239     if(strcmp(outputActivationFunc,"sigmoid"))
240         numTraining= size(Y)(2);
241         # Element wise multiply for logprobs
242         cost = -1/numTraining * sum((Y .* log(AL)) + (1-Y) .* log(1-AL));
243     elseif(strcmp(outputActivationFunc,'softmax'))
244         numTraining = size(Y)(2);
245         Y=Y';
246         # Select rows where Y=0,1, and 2 and concatenate to a long vector
247         #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)];
248         a =pickColumns(AL,Y,numClasses);
249
250         #Select the correct column for log prob
251         correct_probs = -log(a);
252         #Compute log loss
253         cost= sum(correct_probs)/numTraining;
254     endif
255 end
256

```

```

257 # Compute the backpropagation for 1 cycle
258 # Input : Neural Network parameters - dA
259 #         # cache - forward_cache & activation_cache
260 #         # Input features
261 #         # Output values Y
262 #         # activationFunc- sigmoid/softmax/tanh
263 #         # numClasses
264 # Returns: Gradients
265 # dL/dwi= dL/dzi*A1-1
266 # dL/dbl = dL/dz1
267 # dL/dz_prev=dL/dz1*w
268 function [dA_prev dw db] = layerActivationBackward(dA, forward_cache,
269 activation_cache, Y, activationFunc,numClasses)
270
271 A_prev = forward_cache{1};
272 w =forward_cache{2};
273 b = forward_cache{3};
274 numTraining = size(A_prev)(2);
275 if (strcmp(activationFunc,"relu"))
276     dz = reluDerivative(dA, activation_cache);
277 elseif (strcmp(activationFunc,"sigmoid"))
278     dz = sigmoidDerivative(dA, activation_cache);
279 elseif(strcmp(activationFunc, "tanh"))
280     dz = tanhDerivative(dA, activation_cache);
281 elseif(strcmp(activationFunc, "softmax"))
282     #dz = softmaxDerivative(dA, activation_cache,Y,numClasses);
283     dz = stableSoftmaxDerivative(dA, activation_cache,Y,numClasses);
284 endif
285
286 # Check if softmax
287 if (strcmp(activationFunc,"softmax"))
288     w =forward_cache{2};
289     b = forward_cache{3};
290     dw = 1/numTraining * A_prev * dz;
291     db = 1/numTraining * sum(dz,1);
292     dA_prev = dz*w;
293 else
294     w =forward_cache{2};
295     b = forward_cache{3};
296     dw = 1/numTraining * dz * A_prev';
297     db = 1/numTraining * sum(dz,2);
298     dA_prev = w'*dz;
299 endif
300
301 end
302
303
304 # Compute the backpropagation for 1 cycle
305 # Input : AL: Output of L layer Network - weights
306 #         # Y Real output
307 #         # caches -- list of caches containing:
308 #             every cache of layerActivationForward() with "relu"/"tanh"
309 #             #(it's caches[], for l in range(L-1) i.e l = 0...L-2)
310 #             #the cache of layerActivationForward() with "sigmoid" (it's caches[L-
311 ])
312 #             # hiddenActivationFunc - Activation function at hidden layers
313 #             # outputActivationFunc- sigmoid/softmax
314 #             # numClasses
315 #
316 # Returns:
317 #     gradients -- A dictionary with the gradients
318 #                 gradients["dA" + str(l)] = ...
319 #                 gradients["dw" + str(l)] = ...
320 #                 gradients["db" + str(l)] = ...

```

```

321
322 function [gradsDA gradsDW gradsDB]= backwardPropagationDeep(AL, Y,
323 activation_caches,forward_caches,
324
325 hiddenActivationFunc='relu',outputActivationFunc="sigmoid",numClasses)
326
327
328 # Set the number of layers
329 L = length(activation_caches);
330 m = size(AL)(2);
331
332 if (strcmp(outputActivationFunc,"sigmoid"))
333     # Initializing the backpropagation
334     #  $dL/dAL = -(y/a + (1-y)/(1-a))$  - At the output layer
335     dAL = -(Y ./ AL) - (1 - Y) ./ (1 - AL));
336 elseif (strcmp(outputActivationFunc,"softmax"))
337     dAL=0;
338     Y=Y';
339 endif
340
341
342 # Since this is a binary classification the activation at output is
343 sigmoid
344 # Get the gradients at the last layer
345 # Inputs: "AL, Y, caches".
346 # Outputs: "gradients["dAL"], gradients["dwL"], gradients["dbL"]"
347 activation_cache = activation_caches{L};
348 forward_cache = forward_caches(L);
349 # Note the cell array includes an array of forward caches. To get to this
350 we need to include the index {1}
351 [dA dw db] = layerActivationBackward(dAL, forward_cache{1},
352 activation_cache, Y, activationFunc = outputActivationFunc,numClasses);
353 if (strcmp(outputActivationFunc,"sigmoid"))
354     gradsDA{L}= dA;
355 elseif (strcmp(outputActivationFunc,"softmax"))
356     gradsDA{L}= dA';#Note the transpose
357 endif
358 gradsDW{L}= dw;
359 gradsDB{L}= db;
360
361 # Traverse in the reverse direction
362 for l =(L-1):-1:1
363     # Compute the gradients for L-1 to 1 for Relu/tanh
364     # Inputs: "gradients["dA" + str(l + 2)], caches".
365     # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l +
366 1)] , gradients["db" + str(l + 1)]
367     activation_cache = activation_caches{l};
368     forward_cache = forward_caches(l);
369
370     #dA_prev_temp, dw_temp, db_temp =
371 layerActivationBackward(gradients['dA'+str(l+1)], current_cache,
372 activationFunc = "relu")
373     # dAl the derivative of the activation of the lth layer,is the first
374 element
375     dAl= gradsDA{l+1};
376     [dA_prev_temp, dw_temp, db_temp] = layerActivationBackward(dAl,
377 forward_cache{1}, activation_cache, Y, activationFunc =
378 hiddenActivationFunc,numClasses);
379     gradsDA{l}= dA_prev_temp;
380     gradsDW{l}= dw_temp;
381     gradsDB{l}= db_temp;
382
383 endfor
384

```

```

385 end
386
387
388 # Perform Gradient Descent
389 # Input : weights and biases
390 #           : gradients
391 #           : learning rate
392 #           : outputActivationFunc
393 #output : weights, biases
394 function [weights biases] = gradientDescent(weights, biases,gradsw,gradsB,
395 learningRate,outputActivationFunc="sigmoid")
396
397 L = size(weights)(2); # number of layers in the neural network
398
399 # Update rule for each parameter.
400 for l=1:(L-1)
401     weights{l} = weights{l} -learningRate* gradsw{l};
402     biases{l} = biases{l} -learningRate* gradsB{l};
403 endfor
404
405
406 if (strcmp(outputActivationFunc,"sigmoid"))
407     weights{L} = weights{L} -learningRate* gradsw{L};
408     biases{L} = biases{L} -learningRate* gradsB{L};
409 elseif (strcmp(outputActivationFunc,"softmax"))
410     weights{L} = weights{L} -learningRate* gradsw{L}';
411     biases{L} = biases{L} -learningRate* gradsB{L}';
412 endif
413
414
415 end
416
417
418 # Execute a L layer Deep learning model
419 # Input : X - Input features
420 #           : Y output
421 #           : layersDimensions - Dimension of layers
422 #           : hiddenActivationFunc - Activation function at hidden layer relu
423 /tanh
424 #           : outputActivationFunc - Activation function at hidden layer
425 sigmoid/softmax
426 #           : learning rate
427 #           : num of iterations
428 #output : Updated weights and biases after each iteration
429 function [weights biases costs] = L_Layer_DeepModel(X, Y, layersDimensions,
430 hiddenActivationFunc='relu', outputActivationFunc="sigmoid",learning_rate =
431 .3, num_iterations = 10000)#lr was 0.009
432
433 rand ("seed", 1);
434 costs = [] ;
435
436 # Parameters initialization.
437 [weights biases] = initializeDeepModel(layersDimensions);
438
439 # Loop (gradient descent)
440 for i = 0:num_iterations
441     # Forward propagation: [LINEAR -> RELU]^(L-1) -> LINEAR -> SIGMOID.
442     [AL forward_caches activation_caches] = forwardPropagationDeep(X,
443 weights, biases,hiddenActivationFunc,
444 outputActivationFunc=outputActivationFunc);
445
446     # Compute cost.

```

```

447         cost = computeCost(AL,
448 Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(
449 layersDimensions)(2)));
450
451         # Backward propagation.
452         [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y,
453 activation_caches,forward_caches,hiddenActivationFunc,
454 outputActivationFunc=outputActivationFunc,
455
456 numClasses=layersDimensions(size(layersDimensions)(2)));
457         # Update parameters.
458         [weights biases] = gradientDescent(weights,biases,
459 gradsDW,gradsDB,learning_rate,outputActivationFunc=outputActivationFunc);
460
461
462         # Print the cost every 1000 iterations
463 if ( mod(i,1000) == 0)
464     costs =[costs cost];
465     #disp ("Cost after iteration"), disp(i),disp(cost);
466     printf("Cost after iteration i=%i cost=%d\n",i,cost);
467     endif
468 endfor
469
470 end
471
472 # Execute a L layer Deep learning model with Stochastic Gradient descent
473 # Input : X - Input features
474 #           : Y output
475 #           : layersDimensions - Dimension of layers
476 #           : hiddenActivationFunc - Activation function at hidden layer relu
477 /tanh
478 #           : outputActivationFunc - Activation function at hidden layer
479 sigmoid/softmax
480 #           : learning rate
481 #           : mini_batch_size
482 #           : num of epochs
483 #output : Updated weights and biases after each iteration
484 function [weights biases costs] = L_Layer_DeepModel_SGD(X, Y,
485 layersDimensions, hiddenActivationFunc='relu',
486 outputActivationFunc="sigmoid",learning_rate = .3,
487 mini_batch_size = 64, num_epochs = 2500)#lr was 0.009
488
489 rand ("seed", 1);
490 costs = [] ;
491
492 # Parameters initialization.
493 [weights biases] = initializeDeepModel(layersDimensions);
494 seed=10;
495 # Loop (gradient descent)
496 for i = 0:num_epochs
497     seed = seed + 1;
498     [mini_batches_X mini_batches_Y] = random_mini_batches(X, Y,
499 mini_batch_size, seed);
500
501     minibatches=length(mini_batches_X);
502     for batch=1:minibatches
503         X=mini_batches_X{batch};
504         Y=mini_batches_Y{batch};
505         # Forward propagation: [LINEAR -> RELU]^(L-1) -> LINEAR ->
506 SIGMOID/SOFTMAX.
507         [AL forward_caches activation_caches] =
508 forwardPropagationDeep(X, weights, biases,hiddenActivationFunc,
509 outputActivationFunc=outputActivationFunc);
510         #disp(batch);

```

```

511         #disp(size(X));
512         #disp(size(Y));
513
514         # Compute cost.
515         cost = computeCost(AL,
516 Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(
517 layersDimensions)(2)));
518
519         #disp(cost);
520         # Backward propagation.
521         [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y,
522 activation_caches,forward_caches,hiddenActivationFunc,
523 outputActivationFunc=outputActivationFunc,
524
525 numClasses=layersDimensions(size(layersDimensions)(2)));
526         # Update parameters.
527         [weights biases] = gradientDescent(weights,biases,
528 gradsDW,gradsDB,learning_rate,outputActivationFunc=outputActivationFunc);
529
530         endfor
531         # Print the cost every 1000 iterations
532         if ( mod(i,1000) == 0)
533             costs =[costs cost];
534             #disp ("Cost after iteration"), disp(i),disp(cost);
535             printf("Cost after iteration i=%i cost=%d\n",i,cost);
536         endif
537     endfor
538
539 end
540
541 # Plot cost vs iterations
542 function plotCostVsIterations(maxIterations,costs)
543     iterations=[0:1000:maxIterations];
544     plot(iterations,costs);
545     title ("Cost vs no of iterations ");
546     xlabel("No of iterations");
547     ylabel("Cost");
548     print -dpng figure23.jpg
549 end;
550
551 # Compute the predicted value for a given input
552 # Input : Neural Network parameters
553 #           : Input data
554 function [predictions]= predict(weights, biases,
555 X,hiddenActivationFunc="relu")
556     [AL forward_caches activation_caches] = forwardPropagationDeep(X,
557 weights, biases,hiddenActivationFunc);
558     predictions = (AL>0.5);
559 end
560
561 # Plot the decision boundary
562 function plotDecisionBoundary(data,weights,
563 biases,hiddenActivationFunc="relu")
564     %Plot a non-linear decision boundary learned by the SVM
565     colormap ("summer");
566
567     % Make classification predictions over a grid of values
568     x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)';
569     x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)';
570     [X1, X2] = meshgrid(x1plot, x2plot);
571     vals = zeros(size(X1));
572     # Plot the prediction for the grid
573     for i = 1:size(X1, 2)
574         gridPoints = [X1(:, i), X2(:, i)];
```

```

575         vals(:, i)=predict(weights,
576 biases,gridPoints',hiddenActivationFunc=hiddenActivationFunc);
577     endfor
578
579     scatter(data(:,1),data(:,2),8,c=data(:,3),"filled");
580     % Plot the boundary
581     hold on
582     #contour(X1, X2, vals, [0 0], 'LineWidth', 2);
583     contour(X1, X2, vals,"LineWidth",4);
584     title ({"3 layer Neural Network decision boundary"});
585     hold off;
586     print -dpng figure32.jpg
587
588 end
589
590 # Compute scores
591 function [AL]= scores(weights, biases, x,hiddenActivationFunc="relu")
592     [AL forward_caches activation_caches] = forwardPropagationDeep(X,
593 weights, biases,hiddenActivationFunc);
594 end
595
596 # Create Random mini batches. Return cell arrays with the mini batches
597 # Input : X, Y
598 #           : Size of minibatch
599 #           : seed
600 #Output : mini batches X & Y
601 function [mini_batches_X mini_batches_Y]= random_mini_batches(X, Y,
602 miniBatchSize = 64, seed = 0)
603
604 rand ("seed", seed);
605 # Get number of training samples
606 m = size(X)(2);
607
608
609 # Create a list of random numbers < m
610 permutation = randperm(m);
611 # Randomly shuffle the training data
612 shuffled_X = X(:, permutation);
613 shuffled_Y = Y(:, permutation);
614
615 # Compute number of mini batches
616 numCompleteMinibatches = floor(m/miniBatchSize);
617 batch=0;
618 for k = 0:(numCompleteMinibatches-1)
619     #Set the start and end of each mini batch
620     batch=batch+1;
621     lower=(k*miniBatchSize)+1;
622     upper=(k+1) * miniBatchSize;
623     mini_batch_X = shuffled_X(:, lower:upper);
624     mini_batch_Y = shuffled_Y(:, lower:upper);
625
626     # Create cell arrays
627     mini_batches_X{batch} = mini_batch_X;
628     mini_batches_Y{batch} = mini_batch_Y;
629 endfor
630
631 # If the batc size does not cleanly divide with number of mini batches
632 if mod(m ,miniBatchSize) != 0
633     # Set the start and end of the last mini batch
634     l=floor(m/miniBatchSize)*miniBatchSize;
635     m=l+ mod(m,miniBatchSize);
636     mini_batch_X = shuffled_X(:,(l+1):m);
637     mini_batch_Y = shuffled_Y(:,(l+1):m);
638

```

```

639         batch=batch+1;
640         mini_batches_X{batch} = mini_batch_X;
641         mini_batches_Y{batch} = mini_batch_Y;
642     endif
643 end

```

7.Appendix 6 - Initialization, regularization in Deep Learning

6.1 Python

```

1 # -*- coding: utf-8 -*-
2 ######
3 #####
4 #
5 # File: DLfunctions61.py
6 # Developer: Tinniam V Ganesh
7 # Date : 16 Apr 2018
8 #
9 #####
10 #####
11 import numpy as np
12 import matplotlib.pyplot as plt
13 import matplotlib
14 import matplotlib.pyplot as plt
15 from matplotlib import cm
16 import math
17 import sklearn
18 import sklearn.datasets
19
20 # Compute the sigmoid of a vector
21 def sigmoid(Z):
22     A=1/(1+np.exp(-Z))
23     cache=Z
24     return A,cache
25
26 # Compute the Relu of a vector
27 def relu(Z):
28     A = np.maximum(0,Z)
29     cache=Z
30     return A,cache
31
32 # Compute the tanh of a vector
33 def tanh(Z):
34     A = np.tanh(Z)
35     cache=Z
36     return A,cache
37
38 # Compute the softmax of a vector
39 def softmax(Z):
40     # get unnormalized probabilities
41     exp_scores = np.exp(Z.T)
42     # normalize them for each example
43     A = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
44     cache=Z
45     return A,cache
46
47 # Compute the Stable Softmax of a vector

```

```

48 def stablesoftmax(Z):
49     #Compute the softmax of vector x in a numerically stable way.
50     shiftZ = Z.T - np.max(Z.T, axis=1).reshape(-1,1)
51     exp_scores = np.exp(shiftZ)
52
53     # normalize them for each example
54     A = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
55     cache=Z
56     return A,cache
57
58 # Compute the derivative of Relu
59 def reluDerivative(dA, cache):
60
61     Z = cache
62     dZ = np.array(dA, copy=True) # just converting dZ to a correct object.
63     # When z <= 0, you should set dZ to 0 as well.
64     dZ[Z <= 0] = 0
65     return dZ
66
67 # Compute the derivative of sigmoid
68 def sigmoidDerivative(dA, cache):
69     Z = cache
70     s = 1/(1+np.exp(-Z))
71     dZ = dA * s * (1-s)
72     return dZ
73
74 # Compute the derivative of tanh
75 def tanhDerivative(dA, cache):
76     Z = cache
77     a = np.tanh(Z)
78     dZ = dA * (1 - np.power(a, 2))
79     return dZ
80
81 # Compute the derivative of softmax
82 def softmaxDerivative(dA, cache,y,numTraining):
83     # Note : dA not used. dL/dZ = dL/dA * dA/dZ = pi-yi
84     Z = cache
85     # Compute softmax
86     exp_scores = np.exp(Z.T)
87     # normalize them for each example
88     probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
89
90     # compute the gradient on scores
91     dZ = probs
92
93     # dZ = pi- yi
94     dZ[range(int(numTraining)),y[:,0]] -= 1
95     return(dZ)
96
97 # Compute the derivative of Stable Softmax
98 def stableSoftmaxDerivative(dA, cache,y,numTraining):
99     # Note : dA not used. dL/dZ = dL/dA * dA/dZ = pi-yi
100    Z = cache
101    # Compute stable softmax
102    shiftZ = Z.T - np.max(Z.T, axis=1).reshape(-1,1)
103    exp_scores = np.exp(shiftZ)
104    # normalize them for each example
105    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
106    #print(probs)
107    # compute the gradient on scores
108    dZ = probs
109
110    # dZ = pi- yi
111    dZ[range(int(numTraining)),y[:,0]] -= 1

```

```

112     return(dZ)
113
114
115 # Initialize the model
116 # Input : number of features
117 #           number of hidden units
118 #           number of units in output
119 # Returns: weight and bias matrices and vectors
120 def initializeModel(numFeats,numHidden,numOutput):
121     np.random.seed(1)
122     w1=np.random.randn(numHidden,numFeats)*0.01 # Multiply by .01
123     b1=np.zeros((numHidden,1))
124     w2=np.random.randn(numOutput,numHidden)*0.01
125     b2=np.zeros((numOutput,1))
126
127     # Create a dictionary of the neural network parameters
128     nnParameters={'w1':w1,'b1':b1,'w2':w2,'b2':b2}
129     return(nnParameters)
130
131
132 # Initialize model for L layers
133 # Input : List of units in each layer
134 # Returns: Initial weights and biases matrices for all layers
135 def initializeDeepModel(layerDimensions):
136     np.random.seed(3)
137     # note the weight matrix at layer '1' is a matrix of size (1,1-1)
138     # The Bias is a vectors of size (1,1)
139
140     # Loop through the layer dimension from 1.. L
141     layerParams = {}
142     for l in range(1,len(layerDimensions)):
143         layerParams['w' + str(l)] =
144             np.random.randn(layerDimensions[l],layerDimensions[l-1])*0.01 # Multiply by
145             .01
146         layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1))
147
148     return(layerParams)
149     return Z, cache
150
151 # He Initialization model for L layers
152 # Input : List of units in each layer
153 # Returns: Initial weights and biases matrices for all layers
154 # He initialization multiplies the random numbers with
155 sqrt(2/layerDimensions[1-1])
156 def HeInitializeDeepModel(layerDimensions):
157     np.random.seed(3)
158     # note the weight matrix at layer '1' is a matrix of size (1,1-1)
159     # The Bias is a vectors of size (1,1)
160
161     # Loop through the layer dimension from 1.. L
162     layerParams = {}
163     for l in range(1,len(layerDimensions)):
164         layerParams['w' + str(l)] = np.random.randn(layerDimensions[l],
165             layerDimensions[l-1])*np.sqrt(2/layerDimensions[l-1])
166         layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1))
167
168     return(layerParams)
169     return Z, cache
170
171 # Xavier Initialization model for L layers
172 # Input : List of units in each layer
173 # Returns: Initial weights and biases matrices for all layers
174 # Xavier initialization multiplies the random numbers with
175 sqrt(1/layerDimensions[1-1])

```

```

176 def XavInitializeDeepModel(layerDimensions):
177     np.random.seed(3)
178     # note the weight matrix at layer 'l' is a matrix of size (l,l-1)
179     # The Bias is a vectors of size (l,1)
180
181     # Loop through the layer dimension from 1.. L
182     layerParams = {}
183     for l in range(1,len(layerDimensions)):
184         layerParams['w' + str(l)] = np.random.randn(layerDimensions[l],
185                                         layerDimensions[l-1])*np.sqrt(1/layerDimensions[l-1])
186         layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1))
187
188     return(layerParams)
189     return Z, cache
190
191 # Compute the activation at a layer 'l' for forward prop in a Deep Network
192 # Input : A_prev - Activation of previous layer
193 #          w,b - Weight and bias matrices and vectors
194 #          keep_prob
195 #          activationFunc - Activation function - sigmoid, tanh, relu etc
196 # Returns : The Activation of this layer
197 #
198 # Z = w * x + b
199 # A = sigmoid(z), A= Relu(z), A= tanh(z)
200 def layerActivationForward(A_prev, w, b, keep_prob=1, activationFunc="relu"):
201
202     # Compute Z
203     Z = np.dot(w,A_prev) + b
204     forward_cache = (A_prev, w, b)
205     # Compute the activation for sigmoid
206     if activationFunc == "sigmoid":
207         A, activation_cache = sigmoid(Z)
208     # Compute the activation for Relu
209     elif activationFunc == "relu":
210         A, activation_cache = relu(Z)
211     # Compute the activation for tanh
212     elif activationFunc == 'tanh':
213         A, activation_cache = tanh(Z)
214     elif activationFunc == 'softmax':
215         A, activation_cache = stableSoftmax(Z)
216
217     cache = (forward_cache, activation_cache)
218     return A, cache
219
220 # Compute the forward propagation for layers 1..L
221 # Input : X - Input Features
222 #          parameters: Weights and biases
223 #          keep_prob
224 #          hiddenActivationFunc - Activation function at hidden layers
225 Relu/tanh
226 #          outputActivationFunc - Activation function at output -
227 sigmoid/softmax
228 # Returns : AL
229 #          caches
230 #          dropoutMat
231 # The forward propoagation uses the Relu/tanh activation from layer 1..L-1 and
232 # sigmoid actiovation at layer L
233 def forwardPropagationDeep(X, parameters,keep_prob=1,
234 hiddenActivationFunc='relu',outputActivationFunc='sigmoid'):
235     caches = []
236     #initialize the dropout matrix
237     dropoutMat = {}
238     # Set A to X (A0)
239     A = X

```

```

240     L = int(len(parameters)/2) # number of layers in the neural network
241     # Loop through from layer 1 to upto layer L
242     for l in range(1, L):
243         A_prev = A
244         # Zi = Wi x Ai-1 + bi and Ai = g(z)
245         A, cache = layerActivationForward(A_prev, parameters['w'+str(l)],
246 parameters['b'+str(l)], keep_prob, activationFunc = hiddenActivationFunc)
247
248         # Randomly drop some activation units
249         # Create a matrix as the same shape as A
250         D = np.random.rand(A.shape[0],A.shape[1])
251         D = (D < keep_prob)
252         # We need to use the same 'dropout' matrix in backward propagation
253         # Save the dropout matrix for use in backprop
254         dropoutMat["D" + str(l)] =D
255         A= np.multiply(A,D)
256         A = np.divide(A,keep_prob)
257
258         caches.append(cache)
259
260
261     # Since this is binary classification use the sigmoid activation function
262     in
263         # last layer
264         AL, cache = layerActivationForward(A, parameters['w'+str(L)],
265 parameters['b'+str(L)], activationFunc = outputActivationFunc)
266         caches.append(cache)
267
268     return AL, caches, dropoutMat
269
270
271 # Compute the cost
272 # Input : parameters
273 #      : AL
274 #      : Y
275 #      :outputActivationFunc - Activation function at output -
276 sigmoid/softmax/tanh
277 # Output: cost
278 def computeCost(parameters,AL,Y,outputActivationFunc="sigmoid"):
279     if outputActivationFunc=="sigmoid":
280         m= float(Y.shape[1])
281         # Element wise multiply for logprobs
282         cost=-1/m *np.sum(Y*np.log(AL) + (1-Y)*(np.log(1-AL)))
283         cost = np.squeeze(cost)
284     elif outputActivationFunc=="softmax":
285         # Take transpose of Y for softmax
286         Y=Y.T
287         m= float(len(Y))
288         # Compute log probs. Take the log prob of correct class based on
289 output y
290         correct_logprobs = -np.log(AL[range(int(m)),Y.T])
291         # Compute Loss
292         cost = np.sum(correct_logprobs)/m
293     return cost
294
295
296 # Compute the cost with regularization
297 # Input : parameters
298 #      : AL
299 #      : Y
300 #      : lambd
301 #      :outputActivationFunc - Activation function at output -
302 sigmoid/softmax/tanh
303 # Output: cost

```

```

304 def computeCostWithReg(parameters, AL, Y, lambd,
305 outputActivationFunc="sigmoid"):
306     if outputActivationFunc=="sigmoid":
307         m= float(Y.shape[1])
308         # Element wise multiply for logprobs
309         cost=-1/m *np.sum(Y*np.log(AL) + (1-Y)*(np.log(1-AL)))
310         cost = np.squeeze(cost)
311
312         # Regularization cost
313         L= int(len(parameters)/2)
314         L2RegularizationCost=0
315         for l in range(L):
316             L2RegularizationCost+=np.sum(np.square(parameters['w'+str(l+1)]))
317
318         L2RegularizationCost = (lambd/(2*m))*L2RegularizationCost
319         cost = cost + L2RegularizationCost
320
321     elif outputActivationFunc=="softmax":
322         # Take transpose of Y for softmax
323         Y=Y.T
324         m= float(len(Y))
325         # Compute log probs. Take the log prob of correct class based on
326         output y
327             correct_logprobs = -np.log(AL[range(int(m)),Y.T])
328             # Compute Loss
329             cost = np.sum(correct_logprobs)/m
330
331             # Regularization cost
332             L= int(len(parameters)/2)
333             L2RegularizationCost=0
334             for l in range(L):
335                 L2RegularizationCost+=np.sum(np.square(parameters['w'+str(l+1)]))
336
337             L2RegularizationCost = (lambd/(2*m))*L2RegularizationCost
338             cost = cost + L2RegularizationCost
339
340     return cost
341
342 # Compute the backpropagation for 1 cycle with dropout included
343 # Input : Neural Network parameters - dA
344 #          # cache - forward_cache & activation_cache
345 #          # Input features
346 #          # keep_prob
347 #          # Output values Y
348 # Returns: Gradients
349 # dL/dwi= dL/dzi*A1-1
350 # dL/db1 = dL/dz1
351 # dL/dz_prev=dL/dz1*w
352 def layerActivationBackward(dA, cache, Y, keep_prob=1,
353 activationFunc="relu"):
354     forward_cache, activation_cache = cache
355     A_prev, w, b = forward_cache
356     numtraining = float(A_prev.shape[1])
357     #print("n=",numtraining)
358     #print("no=",numtraining)
359     if activationFunc == "relu":
360         dz = reluDerivative(dA, activation_cache)
361     elif activationFunc == "sigmoid":
362         dz = sigmoidDerivative(dA, activation_cache)
363     elif activationFunc == "tanh":
364         dz = tanhDerivative(dA, activation_cache)
365     elif activationFunc == "softmax":
366         dz = stableSoftmaxDerivative(dA, activation_cache,Y,numtraining)
367

```

```

368     if activationFunc == 'softmax':
369         dw = 1/numtraining * np.dot(A_prev,dz)
370         db = 1/numtraining * np.sum(dZ, axis=0, keepdims=True)
371         dA_prev = np.dot(dz,w)
372     else:
373         #print(numtraining)
374         dw = 1/numtraining *(np.dot(dZ,A_prev.T))
375         #print("dw=",dw)
376         db = 1/numtraining * np.sum(dZ, axis=1, keepdims=True)
377         #print("db=",db)
378         dA_prev = np.dot(w.T,dz)
379
380     return dA_prev, dw, db
381
382
383 # Compute the backpropagation with regularization for 1 cycle
384 # Input : dA-Neural Network parameters
385 #         # cache - forward_cache & activation_cache
386 #         # Output values Y
387 #         # lambd
388 #         # activationFunc
389 # Returns dA_prev, dw, db
390 # Returns: Gradients
391 # dL/dwi= dL/dzi*A_l-1
392 # dL/dbl = dL/dz_l
393 # dL/dz_prev=dL/dz_l*w
394 def layerActivationBackwardwithReg(dA, cache, Y, lambd, activationFunc):
395     forward_cache, activation_cache = cache
396     A_prev, w, b = forward_cache
397     numtraining = float(A_prev.shape[1])
398     #print("n=",numtraining)
399     #print("no=",numtraining)
400     if activationFunc == "relu":
401         dz = reluDerivative(dA, activation_cache)
402     elif activationFunc == "sigmoid":
403         dz = sigmoidDerivative(dA, activation_cache)
404     elif activationFunc == "tanh":
405         dz = tanhDerivative(dA, activation_cache)
406     elif activationFunc == "softmax":
407         dz = stableSoftmaxDerivative(dA, activation_cache,Y,numtraining)
408
409     if activationFunc == 'softmax':
410         # Add the regularization factor
411         dw = 1/numtraining * np.dot(A_prev,dz) + (lambd/numtraining) * w.T
412         db = 1/numtraining * np.sum(dZ, axis=0, keepdims=True)
413         dA_prev = np.dot(dz,w)
414     else:
415         # Add the regularization factor
416         dw = 1/numtraining *(np.dot(dZ,A_prev.T)) + (lambd/numtraining) * w
417         #print("dw=",dw)
418         db = 1/numtraining * np.sum(dZ, axis=1, keepdims=True)
419         #print("db=",db)
420         dA_prev = np.dot(w.T,dz)
421
422     return dA_prev, dw, db
423
424 # Compute the backpropagation for 1 cycle
425 # Input : AL: Output of L layer Network - weights
426 #         # Y Real output
427 #         # caches -- list of caches containing:
428 #         # dropoutMat
429 #         # lambd
430 #         # keep_prob
431 #         # every cache of layerActivationForward() with "relu"/"tanh"
432 #         #(it's caches[l], for l in range(L-1) i.e l = 0...L-2)

```

```

432     # the cache of layerActivationForward() with "sigmoid" (it's caches[L-1])
433     # hiddenActivationFunc - Activation function at hidden layers - relu/sigmoid/tanh
434     # outputActivationFunc - Activation function at output - sigmoid/softmax
435     #
436     # Returns:
437     # gradients -- A dictionary with the gradients
438     #           gradients["dA" + str(l)] = ...
439     #           gradients["dw" + str(l)] = ...
440     #
441
442
443
444 def backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd=0, keep_prob=1,
445 hiddenActivationFunc='relu',outputActivationFunc="sigmoid"):
446     #initialize the gradients
447     gradients = {}
448     # Set the number of layers
449     L = len(caches)
450     m = float(AL.shape[1])
451
452     if outputActivationFunc == "sigmoid":
453         Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL
454         # Initializing the backpropagation
455         # dL/dAL= -(y/a + (1-y)/(1-a)) - At the output layer
456         dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
457     else:
458         dAL = 0
459         Y=Y.T
460
461     # Since this is a binary classification the activation at output is
462     # sigmoid
463     # Get the gradients at the last layer
464     # Inputs: "AL, Y, caches".
465     # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"]
466     current_cache = caches[L-1]
467     if lambd==0:
468         gradients["dA" + str(L)], gradients["dw" + str(L)], gradients["db" +
469         str(L)] = layerActivationBackward(dAL, current_cache,
470                                         Y, activationFunc =
471                                         outputActivationFunc)
472     else: #Regularization
473         gradients["dA" + str(L)], gradients["dw" + str(L)], gradients["db" +
474         str(L)] = layerActivationBackwardWithReg(dAL, current_cache,
475                                         Y, lambd, activationFunc =
476                                         outputActivationFunc)
477
478     # Note dA for softmax is the transpose
479     if outputActivationFunc == "softmax":
480         gradients["dA" + str(L)] = gradients["dA" + str(L)].T
481     # Traverse in the reverse direction
482     for l in reversed(range(L-1)):
483         # Compute the gradients for L-1 to 1 for Relu/tanh
484         # Inputs: "gradients["dA" + str(l + 2)], caches".
485         # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l +
486         1)], gradients["db" + str(l + 1)]
487         current_cache = caches[l]
488
489         #dA_prev_temp, dw_temp, db_temp =
490         layerActivationBackward(gradients['dA'+str(l+2)], current_cache,
491         activationFunc = "relu")
492         if lambd==0:
493
494             # In the reverse direction use the same dropout matrix
495             # Random dropout

```

```

496         # Multiply da'1' with the dropoutMat and divide to keep the
497     expected value same
498         D = dropoutMat["D" + str(l+1)]
499         # Drop some dA1's
500         gradients['dA'+str(l+2)]= np.multiply(gradients['dA'+str(l+2)],D)
501         # Divide by keep_prob to keep expected value same
502         gradients['dA'+str(l+2)] =
503     np.divide(gradients['dA'+str(l+2)],keep_prob)
504
505         dA_prev_temp, dw_temp, db_temp =
506     layerActivationBackward(gradients['dA'+str(l+2)], current_cache, Y,
507     keep_prob=1, activationFunc = hiddenActivationFunc)
508
509     else:
510         dA_prev_temp, dw_temp, db_temp =
511     layerActivationBackwardWithReg(gradients['dA'+str(l+2)], current_cache, Y,
512     lambd, activationFunc = hiddenActivationFunc)
513
514         gradients["dA" + str(l + 1)] = dA_prev_temp
515         gradients["dw" + str(l + 1)] = dw_temp
516         gradients["db" + str(l + 1)] = db_temp
517
518
519     return gradients
520
521 # Perform Gradient Descent
522 # Input : Weights and biases
523 #           : gradients
524 #           : learning rate
525 #           : outputActivationFunc - Activation function at output -
526 sigmoid/softmax
527 #output : Updated weights after 1 iteration
528 def gradientDescent(parameters, gradients,
529 learningRate,outputActivationFunc="sigmoid"):
530
531     L = int(len(parameters) / 2)
532     # Update rule for each parameter.
533     for l in range(L-1):
534         parameters["w" + str(l+1)] = parameters['w'+str(l+1)] -learningRate*
535 gradients['dw' + str(l+1)]
536         parameters["b" + str(l+1)] = parameters['b'+str(l+1)] -learningRate*
537 gradients['db' + str(l+1)]
538
539     if outputActivationFunc=="sigmoid":
540         parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
541 gradients['dw' + str(L)]
542         parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
543 gradients['db' + str(L)]
544     elif outputActivationFunc=="softmax":
545         parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
546 gradients['dw' + str(L)].T
547         parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
548 gradients['db' + str(L)].T
549     return parameters
550
551
552
553
554
555 # Execute a L layer Deep learning model
556 # Input : X - Input features
557 #           : Y output
558 #           : layersDimensions - Dimension of layers

```

```

559     : hiddenActivationFunc - Activation function at hidden layer relu
560     /tanh/sigmoid
561     : outputActivationFunc - Activation function at output layer
562     sigmoid/softmax
563     : learning rate
564     # : lambd
565     # : keep_prob
566     # : num of iteration
567     # : initType
568     #output : parameters
569 def L_Layer_DeepModel(X1, Y1, layersDimensions, hiddenActivationFunc='relu',
570     outputActivationFunc="sigmoid",
571     learningRate = .3, lambd=0, keep_prob=1,
572     num_iterations = 10000,initType="default",
573     print_cost=False,figure="figa.png"):
574
575     np.random.seed(1)
576     costs = []
577
578     # Parameters initialization.
579     if initType == "He":
580         parameters = HeInitializeDeepModel(layersDimensions)
581     elif initType == "Xavier":
582         parameters = XavInitializeDeepModel(layersDimensions)
583     else: #Default
584         parameters = initializeDeepModel(layersDimensions)
585
586     # Loop (gradient descent)
587     for i in range(0, num_iterations):
588
589         AL, caches, dropoutMat = forwardPropagationDeep(X1, parameters,
590         keep_prob,
591         hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
592
593         # Regularization parameter is 0
594         if lambd==0:
595             # Compute cost
596             cost = computeCost(parameters,AL, Y1,
597         outputActivationFunc=outputActivationFunc)
598             # Include L2 regularization
599         else:
600             # Compute cost
601             cost = computeCostWithReg(parameters,AL, Y1, lambd,
602         outputActivationFunc=outputActivationFunc)
603
604         # Backward propagation.
605         gradients = backwardPropagationDeep(AL, Y1, caches, dropoutMat,
606         lambd, keep_prob,
607         hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
608
609         # Update parameters.
610         parameters = gradientDescent(parameters, gradients,
611         learningRate=learningRate,outputActivationFunc=outputActivationFunc)
612
613
614         # Print the cost every 100 training example
615         if print_cost and i % 1000 == 0:
616             print ("Cost after iteration %i: %f" %(i, cost))
617             if print_cost and i % 1000 == 0:
618                 costs.append(cost)
619
620         # plot the cost
621         plt.plot(np.squeeze(costs))
622         plt.ylabel('Cost')
623         plt.xlabel('No of iterations (x1000)')

```

```

623     plt.title("Learning rate =" + str(learningRate))
624     plt.savefig(figure,bbox_inches='tight')
625     #plt.show()
626     plt.clf()
627     plt.close()
628
629
630
631     return parameters
632
633 # Execute a L layer Deep learning model Stoachastic Gradient Descent
634 # Input : X1 - Input features
635 #           : Y1- output
636 #           : layersDimensions - Dimension of layers
637 #           : hiddenActivationFunc - Activation function at hidden layer relu
638#/tanh/sigmoid
639 #           : outputActivationFunc - Activation function at output -
640 sigmoid/softmax
641 #           : learning rate
642 #           : mini_batch_size
643 #           : num_epochs
644 #output : parameters
645
646 def L_Layer_DeepModel_SGD(X1, Y1, layersDimensions,
647 hiddenActivationFunc='relu', outputActivationFunc="sigmoid", learningRate =
648 .3, mini_batch_size = 64, num_epochs = 2500, print_cost=False):#lr was 0.009
649
650     np.random.seed(1)
651     costs = []
652
653     # Parameters initialization.
654     parameters = initializeDeepModel(layersDimensions)
655     seed=10
656     # Loop for number of epochs
657     for i in range(num_epochs):
658         # Define the random minibatches. We increment the seed to reshuffle
659         differently the dataset after each epoch
660         seed = seed + 1
661         minibatches = random_mini_batches(X1, Y1, mini_batch_size, seed)
662
663         batch=0
664         # Loop through each mini batch
665         for minibatch in minibatches:
666             #print("batch=",batch)
667             batch=batch+1
668             # Select a minibatch
669             (minibatch_X, minibatch_Y) = minibatch
670
671             # Perfrom forward propagation
672             AL, caches = forwardPropagationDeep(minibatch_X,
673 parameters,hiddenActivationFunc="relu",outputActivationFunc=outputActivationF
674 unc)
675
676             # Compute cost
677             cost = computeCost(AL,
678 minibatch_Y,outputActivationFunc=outputActivationFunc)
679             #print("minibatch_Y=",minibatch_Y.shape)
680             # Backward propagation.
681             gradients = backwardPropagationDeep(AL, minibatch_Y,
682 caches,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
683
684             # Update parameters.
685             parameters = gradientDescent(parameters, gradients,
686 learningRate=learningRate,outputActivationFunc=outputActivationFunc)

```

```

687
688     # Print the cost every 1000 epoch
689     if print_cost and i % 100 == 0:
690         print ("Cost after epoch %i: %f" %(i, cost))
691     if print_cost and i % 100 == 0:
692         costs.append(cost)
693
694     # plot the cost
695     plt.plot(np.squeeze(costs))
696     plt.ylabel('cost')
697     plt.xlabel('No of iterations')
698     plt.title("Learning rate =" + str(learningRate))
699     #plt.show()
700     plt.savefig("fig1",bbox_inches='tight')
701     plt.close()
702     return parameters
703
704
705 # Create random mini batches
706 # Input : X - Input features
707 #           : Y- output
708 #           : miniBatchsizes
709 #           : seed
710 #output : mini_batches
711 def random_mini_batches(x, Y, miniBatchsize = 64, seed = 0):
712
713     np.random.seed(seed)
714     # Get number of training samples
715     m = X.shape[1]
716     # Initialize mini batches
717     mini_batches = []
718
719     # Create a list of random numbers < m
720     permutation = list(np.random.permutation(m))
721     # Randomly shuffle the training data
722     shuffled_X = X[:, permutation]
723     shuffled_Y = Y[:, permutation].reshape((1,m))
724
725     # Compute number of mini batches
726     numCompleteMinibatches = math.floor(m/miniBatchsize)
727
728     # For the number of mini batches
729     for k in range(0, numCompleteMinibatches):
730
731         # Set the start and end of each mini batch
732         mini_batch_X = shuffled_X[:, k*miniBatchSize : (k+1) * miniBatchsize]
733         mini_batch_Y = shuffled_Y[:, k*miniBatchSize : (k+1) * miniBatchsize]
734
735         mini_batch = (mini_batch_X, mini_batch_Y)
736         mini_batches.append(mini_batch)
737
738
739     #if m % miniBatchSize != 0:. The batch does not evenly divide by the mini
740     batch
741     if m % miniBatchSize != 0:
742         l=math.floor(m/miniBatchSize)*miniBatchSize
743         # Set the start and end of last mini batch
744         m=l+m % miniBatchSize
745         mini_batch_X = shuffled_X[:,l:m]
746         mini_batch_Y = shuffled_Y[:,l:m]
747
748         mini_batch = (mini_batch_X, mini_batch_Y)
749         mini_batches.append(mini_batch)
750

```

```

751     return mini_batches
752
753 # Plot a decision boundary
754 # Input : Input Model,
755 #           X
756 #           Y
757 #           sz - Num of hiden units
758 #           lr - Learning rate
759 #           Fig to be saved as
760 # Returns Null
761 def plot_decision_boundary(model, x, y,lr,figure1="figb.png"):
762     print("plot")
763     # Set min and max values and give it some padding
764     x_min, x_max = x[0, :].min() - 1, x[0, :].max() + 1
765     y_min, y_max = x[1, :].min() - 1, x[1, :].max() + 1
766     colors=['black','gold']
767     cmap = matplotlib.colors.ListedColormap(colors)
768     h = 0.01
769     # Generate a grid of points with distance h between them
770     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,
771 h))
772     # Predict the function value for the whole grid
773     Z = model(np.c_[xx.ravel(), yy.ravel()])
774     Z = Z.reshape(xx.shape)
775     # Plot the contour and training examples
776     plt.contourf(xx, yy, Z, cmap="coolwarm")
777     plt.ylabel('x2')
778     plt.xlabel('x1')
779     x=x.T
780     y=y.T.reshape(300, )
781     plt.scatter(x[:, 0], x[:, 1], c=y, s=20);
782     print(x.shape)
783     plt.title("Decision Boundary for learning rate:"+str(lr))
784     plt.savefig(figure1, bbox_inches='tight')
785     #plt.show()
786
787
788 def predict(parameters,
789 x,keep_prob=1,hiddenActivationFunc="relu",outputActivationFunc="sigmoid"):
790     A2, cache,dropoutMat = forwardPropagationDeep(X, parameters, keep_prob=1,
791 hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
792     predictions = (A2>0.5)
793     return predictions
794
795 def predict_proba(parameters, x,outputActivationFunc="sigmoid"):
796     A2, cache = forwardPropagationDeep(X, parameters)
797     if outputActivationFunc=="sigmoid":
798         proba=A2
799     elif outputActivationFunc=="softmax":
800         proba=np.argmax(A2, axis=0).reshape(-1,1)
801         print("A2=",A2.shape)
802     return proba
803
804 # Plot a decision boundary
805 # Input : Input Model,
806 #           X
807 #           Y
808 #           sz - Num of hiden units
809 #           lr - Learning rate
810 #           Fig to be saved as
811 # Returns Null
812 def plot_decision_boundary1(x, y,w1,b1,w2,b2,figure2="figc.png"):
813     #plot_decision_boundary(lambda x: predict(parameters, x.T),
814     x1,y1.T,str(0.3),"fig2.png")

```

```

815     h = 0.02
816     x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
817     y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
818     xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
819                           np.arange(y_min, y_max, h))
820     Z = np.dot(np.maximum(0, np.dot(np.c_[xx.ravel()], w1.T) +
821                         b1.T), w2.T) + b2.T
822     Z = np.argmax(Z, axis=1)
823     Z = Z.reshape(xx.shape)
824
825     fig = plt.figure()
826     plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8)
827     print(x.shape)
828     y1=y.reshape(300,)
829     plt.scatter(X[:, 0], X[:, 1], c=y1, s=40, cmap=plt.cm.Spectral)
830     plt.xlim(xx.min(), xx.max())
831     plt.ylim(yy.min(), yy.max())
832     plt.savefig('figure2', bbox_inches='tight')
833
834 # Load the circles data set
835 def load_dataset():
836     np.random.seed(1)
837     train_X, train_Y = sklearn.datasets.make_circles(n_samples=300,
838     noise=.05)
839     np.random.seed(2)
840     test_X, test_Y = sklearn.datasets.make_circles(n_samples=100, noise=.05)
841     # Visualize the data
842     print(train_X.shape)
843     print(train_Y.shape)
844     print("load")
845     #plt.scatter(train_X[:, 0], train_X[:, 1], c=train_Y, s=40,
846     cmap=plt.cm.Spectral);
847     train_X = train_X.T
848     train_Y = train_Y.reshape((1, train_Y.shape[0]))
849     test_X = test_X.T
850     test_Y = test_Y.reshape((1, test_Y.shape[0]))
851     return train_X, train_Y, test_X, test_Y

```

6.2 R

```

1 #####
2 #####
3 #
4 # File   : DLfunctions6.R
5 # Author : Tinniam V Ganesh
6 # Date   : 16 Apr 2018
7 #
8 #####
9 #####
10 library(ggplot2)
11 library(PRROC)
12 library(dplyr)
13
14 # Compute the sigmoid of a vector
15 sigmoid <- function(z){
16   A <- 1/(1+ exp(-z))
17   cache<-z
18   retvals <- list("A"=A,"Z"=z)

```

```

19 return(retvals)
20 }
21
22 # Compute the Relu(old) of a vector (performance hog!)
23 reluOld <- function(z){
24   A <- apply(z, 1:2, function(x) max(0,x))
25   cache<-z
26   retvals <- list("A"=A,"Z"=z)
27   return(retvals)
28 }
29
30 # Compute the Relu of a vector (performs better!)
31 relu <- function(z){
32   # Perform relu. Set values less than equal to 0 as 0
33   z[z<0]=0
34   A=z
35   cache<-z
36   retvals <- list("A"=A,"Z"=z)
37   return(retvals)
38 }
39
40 # Compute the tanh activation of a vector
41 tanhActivation <- function(z){
42   A <- tanh(z)
43   cache<-z
44   retvals <- list("A"=A,"Z"=z)
45   return(retvals)
46 }
47
48 # Compute the softmax of a vector
49 softmax <- function(z){
50   # get unnormalized probabilities
51   exp_scores = exp(t(z))
52   # normalize them for each example
53   A = exp_scores / rowSums(exp_scores)
54   retvals <- list("A"=A,"Z"=z)
55   return(retvals)
56 }
57
58 # Compute the derivative of Relu
59 # g'(z) = 1 if z > 0 and 0 otherwise
60 reluDerivative <- function(dA, cache){
61   Z <- cache
62   dZ <- dA
63   # Create a logical matrix of values > 0
64   a <- Z > 0
65   # When z <= 0, you should set dz to 0 as well. Perform an element wise
66   multiple
67   dZ <- dZ * a
68   return(dZ)
69 }
70
71 # Compute the derivative of sigmoid
72 # Derivative g'(z) = a * (1-a)
73 sigmoidDerivative <- function(dA, cache){
74   Z <- cache
75   s <- 1/(1+exp(-Z))
76   dZ <- dA * s * (1-s)
77   return(dZ)
78 }
79
80 # Compute the derivative of tanh
81 # Derivative g'(z) = 1 - a^2
82

```

```

83 tanhDerivative <- function(dA, cache){
84   Z = cache
85   a = tanh(Z)
86   dZ = dA * (1 - a^2)
87   return(dZ)
88 }
89
90 # This function is used in computing the softmax derivative
91 # Populate a matrix of 1s in rows where Y==1
92 # This may need to be extended for K classes. Currently
93 # supports K=3 & K=10
94 popMatrix <- function(Y,numClasses){
95   a=rep(0,times=length(Y))
96   Y1=matrix(a,nrow=length(Y),ncol=numClasses)
97   #Set the rows and columns as 1's where Y is the class value
98   if(numClasses==3){
99     Y1[Y==0,1]=1
100    Y1[Y==1,2]=1
101    Y1[Y==2,3]=1
102  } else if (numClasses==10){
103    Y1[Y==0,1]=1
104    Y1[Y==1,2]=1
105    Y1[Y==2,3]=1
106    Y1[Y==3,4]=1
107    Y1[Y==4,5]=1
108    Y1[Y==5,6]=1
109    Y1[Y==6,7]=1
110    Y1[Y==7,8]=1
111    Y1[Y==8,9]=1
112    Y1[Y==9,0]=1
113  }
114  return(Y1)
115 }
116
117 # Compute the softmax derivative
118 softmaxDerivative <- function(dA, cache ,y,numTraining,numClasses){
119   # Note : dA not used. dL/dZ = dL/dA * dA/dZ = pi-yi
120   Z <- cache
121   # Compute softmax
122   exp_scores = exp(t(Z))
123   # normalize them for each example
124   probs = exp_scores / rowSums(exp_scores)
125   # Create a matrix of zeros
126   Y1=popMatrix(y,numClasses)
127   #a=rep(0,times=length(Y))
128   #Y1=matrix(a,nrow=length(Y),ncol=numClasses)
129   #Set the rows and columns as 1's where Y is the class value
130   dZ = probs-Y1
131   return(dZ)
132 }
133
134 # Initialize model for L layers
135 # Input : List of units in each layer
136 # Returns: Initial weights and biases matrices for all layers
137 initializeDeepModel <- function(layerDimensions){
138   set.seed(2)
139
140   # Initialize empty list
141   layerParams <- list()
142
143   # Note the weight matrix at layer '1' is a matrix of size (1,1-1)
144   # The Bias is a vectors of size (1,1)
145
146   # Loop through the layer dimension from 1.. L

```

```

147 # Indices in R start from 1
148 for(l in 2:length(layersDimensions)){
149     # Initialize a matrix of small random numbers of size 1 x 1-1
150     # Create random numbers of size 1 x 1-1
151     w=rnorm(layersDimensions[1]*layersDimensions[1-1])*0.01
152     # Create a weight matrix of size 1 x 1-1 with this initial weights and
153     # Add to list w1,w2... WL
154     layerParams[[paste('w',l-1,sep="")]] = matrix(w,nrow=layersDimensions[1],
155                                                 ncol=layersDimensions[1-1])
156     layerParams[[paste('b',l-1,sep="")]] = matrix(rep(0,layersDimensions[1]),
157                                                 nrow=layersDimensions[1],ncol=1)
158 }
159 return(layerParams)
160 }
161 }
162
163 # He Initialization model for L layers
164 # Input : Vector of units in each layer
165 # Returns: Initial weights and biases matrices for all layers
166 # He initialization multiplies the random numbers with
167 # sqrt(2/layerDimensions[previouslayer])
168 HeInitializeDeepModel <- function(layerDimensions){
169     set.seed(2)
170
171     # Initialize empty list
172     layerParams <- list()
173
174     # Note the weight matrix at layer 'l' is a matrix of size (1,1-1)
175     # The Bias is a vectors of size (1,1)
176
177     # Loop through the layer dimension from 1.. L
178     # Indices in R start from 1
179     for(l in 2:length(layersDimensions)){
180         # Initialize a matrix of small random numbers of size 1 x 1-1
181         # Create random numbers of size 1 x 1-1
182         w=rnorm(layersDimensions[1]*layersDimensions[1-1])
183
184         # Create a weight matrix of size 1 x 1-1 with this initial weights
185         and
186         # Add to list w1,w2... WL
187         # He initialization - Divide by sqrt(2/layerDimensions[previous
188         layer])
189         layerParams[[paste('w',l-1,sep="")]] =
190         matrix(w,nrow=layersDimensions[1],
191             ncol=layersDimensions[1-1])*sqrt(2/layersDimensions[1-1])
192         layerParams[[paste('b',l-1,sep="")]] =
193         matrix(rep(0,layersDimensions[1]),
194             nrow=layersDimensions[1],ncol=1)
195     }
196     return(layerParams)
197 }
198
199
200 # XavInitializeDeepModel Initialization model for L layers
201 # Input : Vrctor of units in each layer
202 # Returns: Initial weights and biases matrices for all layers
203 # He initialization multiplies the random numbers with
204 # sqrt(1/layerDimensions[previouslayer])
205 xavInitializeDeepModel <- function(layerDimensions){
206     set.seed(2)
207
208     # Initialize empty list
209     layerParams <- list()

```

```

211
212     # Note the weight matrix at layer 'l' is a matrix of size (1,1-1)
213     # The Bias is a vectors of size (1,1)
214
215     # Loop through the layer dimension from 1.. L
216     # Indices in R start from 1
217     for(l in 2:length(layersDimensions)){
218         # Initialize a matrix of small random numbers of size 1 x 1-1
219         # Create random numbers of size 1 x 1-1
220         w=rnorm(layersDimensions[l]*layersDimensions[l-1])
221
222         # Create a weight matrix of size 1 x 1-1 with this initial weights
223         and
224         # Add to list w1,w2... WL
225         # He initialization - Divide by sqrt(2/layersDimensions[previous
226         layer])
227         layerParams[[paste('w',l-1,sep="")]] =
228         matrix(w,nrow=layersDimensions[l],
229
230         ncol=layersDimensions[l-1])*sqrt(1/layersDimensions[l-1])
231         layerParams[[paste('b',l-1,sep="")]] =
232         matrix(rep(0,layersDimensions[l]),
233
234         nrow=layersDimensions[l],ncol=1)
235     }
236     return(layerParams)
237 }
238
239
240 # Compute the activation at a layer 'l' for forward prop in a Deep Network
241 # Input : A_prev - Activation of previous layer
242 #          w,b - Weight and bias matrices and vectors
243 #          activationFunc - Activation function - sigmoid, tanh, relu etc
244 # Returns : The Activation of this layer
245 #
246 # Z = W * X + b
247 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
248 layerActivationForward <- function(A_prev, w, b, activationFunc){
249
250     # Compute Z
251     z = w %*% A_prev
252     # Broadcast the bias 'b' by column
253     z <- sweep(z,1,b,'+')
254
255     forward_cache <- list("A_prev"=A_prev, "w"=w, "b"=b)
256     # Compute the activation for sigmoid
257     if(activationFunc == "sigmoid"){
258         vals = sigmoid(z)
259     } else if (activationFunc == "relu"){ # Compute the activation for relu
260         vals = relu(z)
261     } else if(activationFunc == 'tanh'){ # Compute the activation for tanh
262         vals = tanhActivation(z)
263     } else if(activationFunc == 'softmax'){
264         vals = softmax(z)
265     }
266     # Create a list of forward and activation cache
267     cache <- list("forward_cache"=forward_cache,
268     "activation_cache"=vals[['z']])
269     retvals <- list("A"=vals[['A']], "cache"=cache)
270     return(retvals)
271 }
272
273
274 # Compute the forward propagation for layers 1..L

```

```

275 # Input : X - Input Features
276 #           parameters: Weights and biases
277 #           keep_prob
278 #           hiddenActivationFunc - relu/sigmoid/tanh
279 #           outputActivationFunc - Activation function at hidden layer
280 sigmoid/softmax
281 # Returns : AL
282 #           caches
283 #           dropoutMat
284 # The forward propagation uses the Relu/tanh activation from layer 1..L-1 and
285 # sigmoid activation at layer L
286 forwardPropagationDeep <- function(x, parameters, keep_prob=1,
287 hiddenActivationFunc='relu',
288                                     outputActivationFunc='sigmoid'){
289   caches <- list()
290   dropoutMat <- list()
291   # Set A to X (A0)
292   A <- x
293   L <- length(parameters)/2 # number of layers in the neural network
294   # Loop through from layer 1 to upto layer L
295   for(l in 1:(L-1)){
296     A_prev <- A
297     # Zi = Wi x Ai-1 + bi and Ai = g(z)
298     # Set w and b for layer 'l'
299     # Loop through from w1,w2... wL-1
300     w <- parameters[[paste("w",l,sep="")]]
301     b <- parameters[[paste("b",l,sep="")]]
302     # Compute the forward propagation through layer 'l' using the activation
303     # function
304     actForward <- layerActivationForward(A_prev,
305                                           w,
306                                           b,
307                                           activationFunc =
308   hiddenActivationFunc)
309   A <- actForward[['A']]
310   # Append the cache A_prev,w,b, z
311   caches[[l]] <- actForward
312
313   # Randomly drop some activation units
314   # Create a matrix as the same shape as A
315   set.seed(1)
316   i=dim(A)[1]
317   j=dim(A)[2]
318   a<-rnorm(i*j)
319   # Normalize a between 0 and 1
320   a = (a - min(a))/(max(a) - min(a))
321   # Create a matrix of D
322   D <- matrix(a,nrow=i, ncol=j)
323   # Find D which is less than equal to keep_prob
324   D <- D < keep_prob
325   # Remove some A's
326   A <- A * D
327   # Divide by keep_prob to keep expected value same
328   A <- A/keep_prob
329   dropoutMat[[paste("D",l,sep="")]] <- D
330 }
331
332 # Since this is binary classification use the sigmoid activation function
333 in
334 # last layer
335 # Set the weights and biases for the last layer
336 w <- parameters[[paste("w",L,sep="")]]
337 b <- parameters[[paste("b",L,sep="")]]
338 # Compute the sigmoid activation

```

```

339 actForward = layerActivationForward(A, w, b, activationFunc =
340 outputActivationFunc)
341 AL <- actForward[['A']]
342 # Append the output of this forward propagation through the last layer
343 caches[[L]] <- actForward
344 # Create a list of the final output and the caches
345 fwdPropDeep <- list("AL"=AL,"caches"=caches,"dropoutMat"=dropoutMat)
346 return(fwdPropDeep)
347
348 }
349
350 # Function pickColumns(). This function is in computeCost()
351 # Pick columns
352 # Input : AL
353 #           : Y
354 #           : numClasses
355 # Output: a
356 pickColumns <- function(AL,Y,numClasses){
357   if(numClasses==3){
358     a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3])
359   }
360   else if (numClasses==10){
361     a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3],AL[Y==3,4],AL[Y==4,5],
362          AL[Y==5,6],AL[Y==6,7],AL[Y==7,8],AL[Y==8,9],AL[Y==9,10])
363   }
364   return(a)
365 }
366
367
368 # Compute the cost
369 # Input : AL-Activation of last layer
370 #           : Y-Output from data
371 #           : outputActivationFunc - Activation function at hidden layer
372 sigmoid/softmax
373 #           : numClasses
374 # Output: cost
375 computeCost <- function(AL,Y,outputActivationFunc="sigmoid",numClasses=3){
376   if(outputActivationFunc=="sigmoid"){
377     m= length(Y)
378     cost=-1/m*sum(Y*log(AL) + (1-Y)*log(1-AL))
379
380
381 }else if (outputActivationFunc=="softmax"){
382   # Select the elements where the y values are 0, 1 or 2 and make a vector
383   # Pick columns
384   #a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3])
385   m= length(Y)
386   a =pickColumns(AL,Y,numClasses)
387   #a = c(A2[y=k,k+1])
388   # Take log
389   correct_probs = -log(a)
390   # Compute loss
391   cost= sum(correct_probs)/m
392 }
393 #cost=-1/m*sum(a+b)
394 return(cost)
395 }
396
397
398 # Compute the cost with Regularization
399 # Input : parameters
400 #           : AL-Activation of last layer
401 #           : Y-Output from data
402 #           : lambd

```

```

403 #           : outputActivationFunc - Activation function at hidden layer
404 sigmoid/softmax
405 #           : numClasses
406 # Output: cost
407 computeCostWithReg <- function(parameters, AL,Y,lambd,
408 outputActivationFunc="sigmoid",numClasses=3){
409
410     if(outputActivationFunc=="sigmoid"){
411         m= length(Y)
412         cost=-1/m*sum(Y*log(AL) + (1-Y)*log(1-AL))
413
414         # Regularization cost
415         L <- length(parameters)/2
416         L2RegularizationCost=0
417         for(l in 1:L){
418             L2RegularizationCost = L2RegularizationCost +
419                         sum(parameters[[paste("w",l,sep="")]]^2)
420         }
421         L2RegularizationCost = (lambd/(2*m))*L2RegularizationCost
422         cost = cost +  L2RegularizationCost
423
424     }else if (outputActivationFunc=="softmax"){
425         # Select the elements where the y values are 0, 1 or 2 and make a
426         vector
427         # Pick columns
428         #a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3])
429         m= length(Y)
430         a =pickColumns(AL,Y,numClasses)
431         #a = c(A2[y=k,k+1])
432         # Take log
433         correct_probs = -log(a)
434         # Compute loss
435         cost= sum(correct_probs)/m
436
437         # Regularization cost
438         L <- length(parameters)/2
439         L2RegularizationCost=0
440         # Add L2 norm
441         for(l in 1:L){
442             L2RegularizationCost = L2RegularizationCost +
443                         sum(parameters[[paste("w",l,sep="")]]^2)
444         }
445         L2RegularizationCost = (lambd/(2*m))*L2RegularizationCost
446         cost = cost +  L2RegularizationCost
447     }
448     return(cost)
449 }
450
451 # Compute the backpropagation through a layer
452 # Input : Neural Network parameters - dA
453 #           # cache - forward_cache & activation_cache
454 #           # Output values Y
455 #           # activationFunc
456 #           # numClasses
457 # Returns: Gradients
458 # dL/dwi= dL/dzi*A1-1
459 # dL/db1 = dL/dz1
460 # dL/dz_prev=dL/dz1*w
461
462 layerActivationBackward <- function(dA, cache, Y,
463 activationFunc,numClasses){
464     # Get A_prev,W,b
465     forward_cache <-cache[['forward_cache']]
466     activation_cache <- cache[['activation_cache']]

```

```

467 A_prev <- forward_cache[['A_prev']]
468 numtraining = dim(A_prev)[2]
469 # Get Z
470 activation_cache <- cache[['activation_cache']]
471 if(activationFunc == "relu"){
472   dz <- reluDerivative(dA, activation_cache)
473 } else if(activationFunc == "sigmoid"){
474   dz <- sigmoidDerivative(dA, activation_cache)
475 } else if(activationFunc == "tanh"){
476   dz <- tanhDerivative(dA, activation_cache)
477 } else if(activationFunc == "softmax"){
478   dz <- softmaxDerivative(dA, activation_cache, Y, numtraining, numClasses)
479 }
480
481 if (activationFunc == 'softmax'){
482   W <- forward_cache[['W']]
483   b <- forward_cache[['b']]
484   dw = 1/numtraining * A_prev%*%dz
485   db = 1/numtraining* matrix(colSums(dz), nrow=1, ncol=numClasses)
486   dA_prev = dz %*% W
487 } else {
488   W <- forward_cache[['W']]
489   b <- forward_cache[['b']]
490   numtraining = dim(A_prev)[2]
491
492   dw = 1/numtraining * dz %*% t(A_prev)
493   db = 1/numtraining * rowSums(dz)
494   dA_prev = t(W) %*% dz
495 }
496 retvals <- list("dA_prev"=dA_prev, "dw"=dw, "db"=db)
497 return(retvals)
498 }
499
500 # Compute the backpropagation through a layer with Regularization
501 # Input : dA-Neural Network parameters
502 #          # cache - forward_cache & activation_cache
503 #          # Output values Y
504 #          # lambd
505 #          # activationFunc
506 #          # numClasses
507 # Returns: Gradients
508 # dL/dwi= dL/dzi*A_l-1
509 # dL/dbl = dL/dz_l
510 # dL/dz_prev=dL/dz_l*w
511 layerActivationBackwardWithReg <- function(dA, cache, Y, lambd,
512 activationFunc, numClasses){
513   # Get A_prev,w,b
514   forward_cache <-cache[['forward_cache']]
515   activation_cache <- cache[['activation_cache']]
516   A_prev <- forward_cache[['A_prev']]
517   numtraining = dim(A_prev)[2]
518   # Get Z
519   activation_cache <- cache[['activation_cache']]
520   if(activationFunc == "relu"){
521     dz <- reluDerivative(dA, activation_cache)
522   } else if(activationFunc == "sigmoid"){
523     dz <- sigmoidDerivative(dA, activation_cache)
524   } else if(activationFunc == "tanh"){
525     dz <- tanhDerivative(dA, activation_cache)
526   } else if(activationFunc == "softmax"){
527     dz <- softmaxDerivative(dA,
528 activation_cache, Y, numtraining, numClasses)
529   }

```

```

531     if (activationFunc == 'softmax'){
532         w <- forward_cache[['w']]
533         b <- forward_cache[['b']]
534         # Add the regularization factor
535         dw = 1/numtraining * A_prev%*%dz + (lambd/numtraining) * t(w)
536         db = 1/numtraining* matrix(colSums(dz), nrow=1, ncol=numClasses)
537         dA_prev = dz %*% w
538     } else {
539         w <- forward_cache[['w']]
540         b <- forward_cache[['b']]
541         numtraining = dim(A_prev)[2]
542         # Add the regularization factor
543         dw = 1/numtraining * dz %*% t(A_prev) + (lambd/numtraining) * w
544         db = 1/numtraining * rowSums(dz)
545         dA_prev = t(w) %*% dz
546     }
547     retvals <- list("dA_prev"=dA_prev,"dw"=dw,"db"=db)
548     return(retvals)
549 }
550
551 # Compute the backpropagation for 1 cycle through all layers
552 # Input : AL: Output of L layer Network - weights
553 #          Y Real output
554 #          caches -- list of caches containing:
555 #          every cache of layerActivationForward() with "relu"/"tanh"
556 #          #(it's caches[], for l in range(L-1) i.e l = 0...L-2)
557 #          #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1])
558 #
559 #          dropoutMat
560 #          lambd
561 #          keep_prob
562 #          hiddenActivationFunc - Activation function at hidden layers -
563 #          relu/tanh/sigmoid
564 #          outputActivationFunc - Activation function at hidden layer
565 #          sigmoid/softmax
566 #          numClasses
567 #          Returns:
568 #          gradients -- A dictionary with the gradients
569 #                      gradients["dA" + str(l)]
570 #                      gradients["dw" + str(l)]
571 #                      gradients["db" + str(l)]
572 backwardPropagationDeep <- function(AL, Y, caches, dropoutMat, lambd=0,
573 keep_prob=0, hiddenActivationFunc='relu',
574 outputActivationFunc="sigmoid", numClasses){
575     #initialize the gradients
576     gradients = list()
577     # Set the number of layers
578     L = length(caches)
579     numTraining = dim(AL)[2]
580
581     if(outputActivationFunc == "sigmoid")
582         # Initializing the backpropagation
583         # d1/dAL= -(y/a) - ((1-y)/(1-a)) - At the output layer
584         dAL = -( (Y/AL) -(1 - Y)/(1 - AL))
585     else if(outputActivationFunc == "softmax"){
586         dAL=0
587         Y=t(Y)
588     }
589
590     # Get the gradients at the last layer
591     # Inputs: "AL, Y, caches".
592     # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"]
593     # Start with Layer L
594     # Get the current cache

```

```

595 current_cache = caches[[L]]$cache
596 if (lambd==0){
597     retvals <- layerActivationBackward(dAL, current_cache, Y,
598                                         activationFunc =
599                                         outputActivationFunc, numClasses)
600 } else {
601     retvals = layerActivationBackwardWithReg(dAL, current_cache, Y, lambd,
602                                         activationFunc =
603                                         outputActivationFunc, numClasses)
604 }
605
606
607
608 #Note: Take the transpose of dA
609 if(outputActivationFunc == "sigmoid")
610     gradients[[paste("dA", L, sep="")]] <- retvals[['dA_prev']]
611 else if(outputActivationFunc == "softmax")
612     gradients[[paste("dA", L, sep="")]] <- t(retvals[['dA_prev']])
613     gradients[[paste("dw", L, sep="")]] <- retvals[['dw']]
614     gradients[[paste("db", L, sep="")]] <- retvals[['db']]
615
616 # Traverse in the reverse direction
617 for(l in (L-1):1){
618     # Compute the gradients for L-1 to 1 for Relu/tanh
619     # Inputs: "gradients["dA" + str(l + 2)]", caches".
620     # Outputs: "gradients["dA" + str(l + 1)]", gradients["dw" + str(l + 1)]",
621     gradients["db" + str(l + 1)]
622     current_cache = caches[[l]]$cache
623     if (lambd==0){
624         # Get the dropout matrix
625         D <- dropoutMat[[paste("D", l, sep="")]]
626         # Multiply gradient with dropout matrix
627         gradients[[paste('dA', l+1, sep="")]] =
628         gradients[[paste('dA', l+1, sep="")]] * D
629         # Divide by keep_prob to keep expected value same
630         gradients[[paste('dA', l+1, sep="")]] =
631         gradients[[paste('dA', l+1, sep="")]]/keep_prob
632         retvals =
633         layerActivationBackward(gradients[[paste('dA', l+1, sep="")]],
634                                 current_cache, Y,
635                                 activationFunc = hiddenActivationFunc)
636     } else {
637         retvals =
638         layerActivationBackwardWithReg(gradients[[paste('dA', l+1, sep="")]],
639                                         current_cache, Y, lambd,
640                                         activationFunc =
641                                         hiddenActivationFunc)
642     }
643
644     gradients[[paste("dA", l, sep="")]] <- retvals[['dA_prev']]
645     gradients[[paste("dw", l, sep="")]] <- retvals[['dw']]
646     gradients[[paste("db", l, sep="")]] <- retvals[['db']]
647 }
648
649
650
651     return(gradients)
652 }
653
654
655 # Perform Gradient Descent
656 # Input : weights and biases
657 #       : gradients
658 #       : Learning rate

```

```

659 # : outputActivationFunc - Activation function at hidden layer
660 sigmoid/softmax
661 #output : Updated weights after 1 iteration
662 gradientDescent <- function(parameters, gradients,
663 learningRate,outputActivationFunc="sigmoid"){
664
665 L = length(parameters)/2 # number of layers in the neural network
666
667 # Update rule for each parameter. Use a for loop.
668 for(l in 1:(L-1)){
669   parameters[[paste("w",l,sep="")]] = parameters[[paste("w",l,sep="")]] -
670     learningRate* gradients[[paste("dw",l,sep="")]]
671   parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] -
672     learningRate* gradients[[paste("db",l,sep="")]]
673 }
674 if(outputActivationFunc=="sigmoid"){
675   parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]] -
676     learningRate* gradients[[paste("dw",L,sep="")]]
677   parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] -
678     learningRate* gradients[[paste("db",L,sep="")]]
679 }
680 }else if (outputActivationFunc=="softmax"){
681   parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]] -
682     learningRate* t(gradients[[paste("dw",L,sep="")]])
683   parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] -
684     learningRate* t(gradients[[paste("db",L,sep="")]])
685 }
686 return(parameters)
687 }
688
689
690 # Execute a L layer Deep learning model
691 # Input : X - Input features
692 # : Y output
693 # : layersDimensions - Dimension of layers
694 # : hiddenActivationFunc - Activation function at hidden layer relu
695 /tanh
696 # : outputActivationFunc - Activation function at hidden layer
697 sigmoid/softmax
698 # : learning rate
699 # : lambd
700 # : keep_prob
701 # : learning rate
702 # : num of iterations
703 # : initType
704 #output : Updated weights
705 L_Layer_DeepModel <- function(x, y, layersDimensions,
706                               hiddenActivationFunc='relu',
707                               outputActivationFunc= 'sigmoid',
708                               learningRate = 0.5,
709                               lambd=0,
710                               keep_prob=1,
711                               numIterations = 10000,
712                               initType="default",
713                               print_cost=False){
714   #Initialize costs vector as NULL
715   costs <- NULL
716
717   # Parameters initialization.
718   if (initType=="He"){
719     parameters =HeInitializeDeepModel(layersDimensions)
720   } else if (initType=="Xav"){
721     parameters =XavInitializeDeepModel(layersDimensions)
722   }

```

```

723     else{
724         print("Here")
725         parameters = initializeDeepModel(layersDimensions)
726     }
727
728     # Loop (gradient descent)
729     for( i in 0:numIterations){
730         # Forward propagation: [LINEAR -> RELU]^(L-1) -> LINEAR ->
731         SIGMOID/SOFTMAX.
732         retvals = forwardPropagationDeep(X, parameters, keep_prob,
733         hiddenActivationFunc,
734
735         outputActivationFunc=outputActivationFunc)
736         AL <- retvals[['AL']]
737         caches <- retvals[['caches']]
738         dropoutMat <- retvals[['dropoutMat']]
739
740         # Compute cost.
741         if(lambd==0){
742             cost <- computeCost(AL,
743             Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
744             layersDimensions)])
745         } else {
746             cost <- computeCostWithReg(parameters, AL, Y,lambd,
747             outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
748             layersDimensions)])
749         }
750         # Backward propagation.
751         gradients = backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd,
752         keep_prob, hiddenActivationFunc,
753
754         outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
755             layersDimensions)])
756
757         # Update parameters.
758         parameters = gradientDescent(parameters, gradients, learningRate,
759             outputActivationFunc=outputActivationFunc)
760
761
762         if(i%1000 == 0){
763             costs=c(costs,cost)
764             print(cost)
765         }
766     }
767
768     retvals <- list("parameters"=parameters,"costs"=costs)
769
770     return(retvals)
771 }
772
773 # Execute a L layer Deep learning model with stochastic Gradient descent
774 # Input : X - Input features
775 #           : Y output
776 #           : layersDimensions - Dimension of layers
777 #           : hiddenActivationFunc - Activation function at hidden layer relu
778 /tanh
779 #           : outputActivationFunc - Activation function at hidden layer
780 sigmoid/softmax
781 #           : learning rate
782 #           : mini_batch_size
783 #           : num of epochs
784 #output : Updated weights after each iteration
785 L_Layer_DeepModel_SGD <- function(X, Y, layersDimensions,
786                                     hiddenActivationFunc='relu',

```

```

787                               outputActivationFunc= 'sigmoid',
788                               learningRate = .3,
789                               mini_batch_size = 64,
790                               num_epochs = 2500,
791                               print_cost=False){
792
793     set.seed(1)
794     #Initialize costs vector as NULL
795     costs <- NULL
796
797     # Parameters initialization.
798     parameters = initializeDeepModel(layersDimensions)
799     seed=10
800
801     # Loop for number of epochs
802     for( i in 0:num_epochs){
803         seed=seed+1
804         minibatches = random_mini_batches(X, Y, mini_batch_size, seed)
805
806         for(batch in 1:length(minibatches)){
807
808             mini_batch_X=minibatches[[batch]][['mini_batch_X']]
809             mini_batch_Y=minibatches[[batch]][['mini_batch_Y']]
810             # Forward propagation:
811             retvals = forwardPropagationDeep(mini_batch_X,
812 parameters,hiddenActivationFunc,
813
814             outputActivationFunc=outputActivationFunc)
815             AL <- retvals[['AL']]
816             caches <- retvals[['caches']]
817
818             # Compute cost.
819             cost <- computeCost(AL,
820             mini_batch_Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)])
821
822             # Backward propagation.
823             gradients = backwardPropagationDeep(AL, mini_batch_Y,
824             caches,hiddenActivationFunc,
825
826             outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
827             layersDimensions)])
828
829             # Update parameters.
830             parameters = gradientDescent(parameters, gradients, learningRate,
831
832             outputActivationFunc=outputActivationFunc)
833             }
834
835             if(i%%100 == 0){
836                 costs=c(costs,cost)
837                 print(cost)
838             }
839         }
840
841         retvals <- list("parameters"=parameters,"costs"=costs)
842
843         return(retvals)
844     }
845
846     # Predict the output for given input
847     # Input : parameters
848     #           : X
849     # Output: predictions
850

```

```

851 predict <- function(parameters, x, keep_prob=1, hiddenActivationFunc='relu'){
852
853   fwdProp <- forwardPropagationDeep(x, parameters, keep_prob,
854   hiddenActivationFunc)
855   predictions <- fwdProp$AL>0.5
856
857   return (predictions)
858 }
859
860 # Plot a decision boundary
861 # This function uses ggplot2
862 plotDecisionBoundary <-
863 function(z, retvals, keep_prob=1, hiddenActivationFunc="sigmoid", lr=0.5){
864   # Find the minimum and maximum for the data
865   xmin<-min(z[,1])
866   xmax<-max(z[,1])
867   ymin<-min(z[,2])
868   ymax<-max(z[,2])
869
870   # Create a grid of values
871   a=seq(xmin,xmax,length=100)
872   b=seq(ymin,ymax,length=100)
873   grid <- expand.grid(x=a, y=b)
874   colnames(grid) <- c('x1', 'x2')
875   grid1 <- t(grid)
876
877   # Predict the output for this grid
878   q <- predict(retvals$parameters, grid1, keep_prob=1, hiddenActivationFunc)
879   q1 <- t(data.frame(q))
880   q2 <- as.numeric(q1)
881   grid2 <- cbind(grid, q2)
882   colnames(grid2) <- c('x1', 'x2', 'q2')
883
884   z1 <- data.frame(z)
885   names(z1) <- c("x1", "x2", "y")
886   atitle=paste("Decision boundary for learning rate:", lr)
887
888   # Plot the contour of the boundary
889   ggplot(z1) +
890     geom_point(data = z1, aes(x = x1, y = x2, color = y)) +
891     stat_contour(data = grid2, aes(x = x1, y = x2, z = q2, color=q2), alpha =
892 0.9) +
893     ggtitle(atitle) + scale_colour_gradientn(colours = brewer.pal(10,
894 "Spectral"))
895
896   # Predict the probability scores for given data set
897   # Input : parameters
898   #       : X
899   # Output: probability of output
900 computeScores <- function(parameters, x, hiddenActivationFunc='relu'){
901
902   fwdProp <- forwardPropagationDeep(x, parameters, hiddenActivationFunc)
903   scores <- fwdProp$AL
904
905   return (scores)
906 }
907
908 # Create random mini batches
909 # Input : X - Input features
910 #       : Y- output
911 #       : miniBatchSize
912 #       : seed
913 #output : mini_batches
914 random_mini_batches <- function(X, Y, miniBatchSize = 64, seed = 0){

```

```

915 set.seed(seed)
916 # Get number of training samples
917 m = dim(X)[2]
918 # Initialize mini batches
919 mini_batches = list()
920
921 # Create a list of random numbers < m
922 permutation = c(sample(m))
923 # Randomly shuffle the training data
924 shuffled_X = X[, permutation]
925 shuffled_Y = Y[1, permutation]
926
927 # Compute number of mini batches
928 numCompleteMinibatches = floor(m/miniBatchSize)
929 batch=0
930 for(k in 0:(numCompleteMinibatches-1)){
931     batch=batch+1
932     # Set the lower and upper bound of the mini batches
933     lower=(k*miniBatchSize)+1
934     upper=((k+1) * miniBatchSize)
935     mini_batch_X = shuffled_X[, lower:upper]
936     mini_batch_Y = shuffled_Y[lower:upper]
937     # Add it to the list of mini batches
938     mini_batch =
939     list("mini_batch_X"=mini_batch_X,"mini_batch_Y"=mini_batch_Y)
940     mini_batches[[batch]] =mini_batch
941 }
942
943
944 # If the batch size does not divide evenly with mini batch size
945 if(m %% miniBatchSize != 0){
946     p=floor(m/miniBatchSize)*miniBatchSize
947     # Set the start and end of last batch
948     q=p+m %% miniBatchSize
949     mini_batch_X = shuffled_X[, (p+1):q]
950     mini_batch_Y = shuffled_Y[(p+1):q]
951 }
952 # Return the list of mini batches
953 mini_batch =
954 list("mini_batch_X"=mini_batch_X,"mini_batch_Y"=mini_batch_Y)
955 mini_batches[[batch]]=mini_batch
956
957 return(mini_batches)
958 }
959
960 # Plot a decision boundary
961 # This function uses ggplot2
962 plotDecisionBoundary1 <- function(z,parameters,keep_prob=1){
963     xmin<-min(z[,1])
964     xmax<-max(z[,1])
965     ymin<-min(z[,2])
966     ymax<-max(z[,2])
967
968     # Create a grid of points
969     a=seq(xmin,xmax,length=100)
970     b=seq(ymin,ymax,length=100)
971     grid <- expand.grid(x=a, y=b)
972     colnames(grid) <- c('x1', 'x2')
973     grid1 <-t(grid)
974
975     retvals = forwardPropagationDeep(grid1, parameters,keep_prob, "relu",
976                                         outputActivationFunc="softmax")
977
978

```

```

979     AL <- retvals$AL
980     # From the softmax probabilities pick the one with the highest
981 probability
982     q= apply(AL,1,which.max)
983
984     q1 <- t(data.frame(q))
985     q2 <- as.numeric(q1)
986     grid2 <- cbind(grid,q2)
987     colnames(grid2) <- c('x1', 'x2','q2')
988
989     Z1 <- data.frame(Z)
990     names(Z1) <- c("x1","x2","y")
991     atitle=paste("Decision boundary")
992     ggplot(Z1) +
993         geom_point(data = Z1, aes(x = x1, y = x2, color = y)) +
994         stat_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2),
995 alpha = 0.9)+  

996         ggtile(atitle) + scale_colour_gradientn(colours = brewer.pal(10,
997 "Spectral"))
998 }
```

6.3 Octave

```

1 #####  

2 #####  

3 #  

4 # File : DLfunctions61.m  

5 # Author : Tinniam V Ganesh  

6 # Date : 16 Apr 2018  

7 #  

8 #####  

9 #####  

10 1;  

11 # Define sigmoid function  

12 function [A,cache] = sigmoid(z)  

13     A = 1 ./ (1+ exp(-z));  

14     cache=z;  

15 end  

16  

17 # Define Relu function  

18 function [A,cache] = relu(z)  

19     A = max(0,z);  

20     cache=z;  

21 end  

22  

23 # Define Tanh function  

24 function [A,cache] = tanhAct(z)  

25     A = tanh(z);  

26     cache=z;  

27 end  

28  

29 # Define Softmax function  

30 function [A,cache] = softmax(z)  

31     # get unnormalized probabilities  

32     exp_scores = exp(z');  

33     # normalize them for each example  

34     A = exp_scores ./ sum(exp_scores,2);  

35     cache=z;  

36 end  

37  

38 # Define Stable Softmax function
```

```

39 function [A,cache] = stableSoftmax(Z)
40     # Normalize by max value in each row
41     shiftZ = Z' - max(Z',[],2);
42     exp_scores = exp(shiftZ);
43     # normalize them for each example
44     A = exp_scores ./ sum(exp_scores,2);
45     #disp("sm")
46     #disp(A);
47     cache=Z;
48 end
49
50 # Define Relu Derivative
51 function [dZ] = reluDerivative(dA,cache)
52     Z = cache;
53     dZ = dA;
54     # Get elements that are greater than 0
55     a = (Z > 0);
56     # Select only those elements where Z > 0
57     dZ = dZ .* a;
58 end
59
60 # Define Sigmoid Derivative
61 function [dZ] = sigmoidDerivative(dA,cache)
62     Z = cache;
63     s = 1 ./ (1+ exp(-Z));
64     dZ = dA .* s .* (1-s);
65 end
66
67 # Define Tanh Derivative
68 function [dZ] = tanhDerivative(dA,cache)
69     Z = cache;
70     a = tanh(Z);
71     dZ = dA .* (1 - a .^ 2);
72 end
73
74 # Populate a matrix with 1s in rows where Y=1
75 # This function may need to be modified if K is not 3, 10
76 # This function is used in computing the softmax derivative
77 function [Y1] = popMatrix(Y,numClasses)
78     Y1=zeros(length(Y),numClasses);
79     if(numClasses==3) # For 3 output classes
80         Y1(Y==0,1)=1;
81         Y1(Y==1,2)=1;
82         Y1(Y==2,3)=1;
83     elseif(numClasses==10) # For 10 output classes
84         Y1(Y==0,1)=1;
85         Y1(Y==1,2)=1;
86         Y1(Y==2,3)=1;
87         Y1(Y==3,4)=1;
88         Y1(Y==4,5)=1;
89         Y1(Y==5,6)=1;
90         Y1(Y==6,7)=1;
91         Y1(Y==7,8)=1;
92         Y1(Y==8,9)=1;
93         Y1(Y==9,10)=1;
94     endif
95 end
96
97 # Define Softmax Derivative
98 function [dZ] = softmaxDerivative(dA,cache,Y, numClasses)
99     Z = cache;
100    # get unnormalized probabilities
101    shiftZ = Z' - max(Z',[],2);

```

```

103 exp_scores = exp(shiftz);
104
105 # normalize them for each example
106 probs = exp_scores ./ sum(exp_scores,2);
107 # dz = pi- yi
108 yi=popMatrix(Y,numClasses);
109 dz=probs-yi;
110
111 end
112
113 # Define Stable Softmax Derivative
114 function [dZ] = stableSoftmaxDerivative(dA,cache,Y, numClasses)
115 z = cache;
116 # get unnormalized probabilities
117 exp_scores = exp(z');
118 # normalize them for each example
119 probs = exp_scores ./ sum(exp_scores,2);
120 # dz = pi- yi
121 yi=popMatrix(Y,numClasses);
122 dz=probs-yi;
123
124 end
125
126 # Initialize model for L layers
127 # Input : List of units in each layer
128 # Returns: Initial weights and biases matrices for all layers
129 function [w b] = initializeDeepModel(layerDimensions)
130 rand ("seed", 3);
131 # note the weight matrix at layer '1' is a matrix of size (1,1-1)
132 # The Bias is a vectors of size (1,1)
133
134 # Loop through the layer dimension from 1.. L
135 # Create cell arrays for weights and biases
136
137 for l =2:size(layerDimensions)(2)
138 w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*0.01; #
139 Multiply by .01
140 b{l-1} = zeros(layerDimensions(l),1);
141
142 endfor
143 end
144
145 # He Initialization for L layers
146 # Input : vector of units in each layer
147 # Returns: Initial weights and biases matrices for all layers
148 function [w b] = HeInitializeDeepModel(layerDimensions)
149 rand ("seed", 3);
150 # note the weight matrix at layer '1' is a matrix of size (1,1-1)
151 # The Bias is a vectors of size (1,1)
152
153 # Loop through the layer dimension from 1.. L
154 # Create cell arrays for weights and biases
155
156 for l =2:size(layerDimensions)(2)
157 w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*sqrt(2/layerDimensions(l-1)); # Multiply by .01
158 b{l-1} = zeros(layerDimensions(l),1);
159
160 endfor
161 end
162
163 # Xavier Initialization for L layers
164 # Input : vector of units in each layer
165 # Returns: Initial weights and biases matrices for all layers

```

```

167 function [w b] = XavInitializeDeepModel(layerDimensions)
168     rand ("seed", 3);
169     # note the weight matrix at layer 'l' is a matrix of size (l,l-1)
170     # The Bias is a vectors of size (l,1)
171
172     # Loop through the layer dimension from 1.. L
173     # Create cell arrays for weights and biases
174
175     for l =2:size(layerDimensions)(2)
176         w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*%
177         sqrt(1/layerDimensions(l-1)); # Multiply by .01
178         b{l-1} = zeros(layerDimensions(l),1);
179
180     endfor
181 end
182
183
184 # Compute the activation at a layer 'l' for forward prop in a Deep Network
185 # Input : A_prev - Activation of previous layer
186 #           w,b - Weight and bias matrices and vectors
187 #           activationFunc - Activation function - sigmoid, tanh, relu etc
188 # Returns : The Activation of this layer
189 #
190 # Z = W * X + b
191 # A = sigmoid(z), A= Relu(z), A= tanh(z)
192 function [A forward_cache activation_cache] = layerActivationForward(A_prev,
193 w, b, activationFunc)
194
195     # Compute Z
196     Z = w * A_prev +b;
197     # Create a cell array
198     forward_cache = {A_prev w b};
199     # Compute the activation for sigmoid
200     if (strcmp(activationFunc,"sigmoid"))
201         [A activation_cache] = sigmoid(Z);
202     elseif (strcmp(activationFunc, "relu")) # Compute the activation for
203     Relu
204         [A activation_cache] = relu(Z);
205     elseif(strcmp(activationFunc,'tanh')) # Compute the activation for
206     tanh
207         [A activation_cache] = tanhAct(Z);
208     elseif(strcmp(activationFunc,'softmax')) # Compute the activation for
209     tanh
210         #[A activation_cache] = softmax(Z);
211         [A activation_cache] = stableSoftmax(Z);
212     endif
213
214 end
215
216 # Compute the forward propagation for layers 1..L
217 # Input : X - Input Features
218 #           paramaters: Weights and biases
219 #           keep_prob
220 #           hiddenActivationFunc - Activation function at hidden layers
221 Relu/tanh/sigmoid
222 #           outputActivationFunc- sigmoid/softmax
223 # Returns : AL
224 #           caches
225 # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and
226 # sigmoid actiovation at layer L
227 function [AL forward_caches activation_caches dropoutMat] =
228         forwardPropagationDeep(X, weights,biases, keep_prob=1,
229                                     hiddenActivationFunc='relu',
230                                     outputActivationFunc='sigmoid')

```

```

231     # Create an empty cell array
232     forward_caches = {};
233     activation_caches = {};
234     dropoutMat = {};
235     # Set A to X (A0)
236     A = X;
237     L = length(weights); # number of layers in the neural network
238     # Loop through from layer 1 to upto layer L
239     for l =1:L-1
240         A_prev = A;
241         # Zi = Wi x Ai-1 + bi and Ai = g(Zi)
242         W = weights{l};
243         b = biases{l};
244         [A forward_cache activation_cache] = layerActivationForward(A_prev,
245 w,b, activationFunc=hiddenActivationFunc);
246         D=rand(size(A)(1),size(A)(2));
247         D = (D < keep_prob) ;
248         # Multiply by DropoutMat
249         A=A.*D;
250         # Divide by keep_prob to keep expected value same
251         A = A ./ keep_prob;
252         # Store D
253         dropoutMat{l}=D;
254         forward_caches{l}=forward_cache;
255         activation_caches{l} = activation_cache;
256     endfor
257     # Since this is binary classification use the sigmoid activation function
258     in
259         # last layer
260         W = weights{L};
261         b = biases{L};
262         [AL, forward_cache activation_cache] = layerActivationForward(A, w,b,
263 activationFunc = outputActivationFunc);
264         forward_caches{L}=forward_cache;
265         activation_caches{L} = activation_cache;
266     end
267
268
269     # Pick columns where Y==1
270     # This function is used in computeCost
271     function [a] = pickColumns(AL,Y,numClasses)
272         if(numClasses==3)
273             a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)];
274         elseif (numClasses==10)
275             a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3);AL(Y==3,4);AL(Y==4,5);
276             AL(Y==5,6); AL(Y==6,7);AL(Y==7,8);AL(Y==8,9);AL(Y==9,10)];
277         endif
278     end
279
280
281     # Compute the cost
282     # Input : Activation of last layer
283     #          : Output from data
284     #          : outputActivationFunc- sigmoid/softmax
285     #          : numClasses
286     # Output: cost
287     function [cost]=computeCost(AL, Y,
288     outputActivationFunc="sigmoid",numClasses)
289         if(strcmp(outputActivationFunc,"sigmoid"))
290             numTraining= size(Y)(2);
291             # Element wise multiply for logprobs
292             cost = -1/numTraining * sum((Y .* log(AL)) + (1-Y) .* log(1-AL));
293
294

```

```

295 elseif(strcmp(outputActivationFunc, 'softmax'))
296     numTraining = size(Y)(2);
297     Y=Y';
298     # Select rows where Y=0,1, and 2 and concatenate to a long vector
299     #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)];
300     a =pickColumns(AL,Y,numClasses);
301
302     #Select the correct column for log prob
303     correct_probs = -log(a);
304     #Compute log loss
305     cost= sum(correct_probs)/numTraining;
306 endif
307 end
308
309 # Compute the cost with regularization
310 # Input : weights
311 #           : AL - Activation of last layer
312 #           : Output from data
313 #           : lambd
314 #           : outputActivationFunc- sigmoid/softmax
315 #           : numClasses
316 # Output: cost
317 function [cost]= computeCostWithReg(weights, AL, Y, lambd,
318 outputActivationFunc="sigmoid",numClasses)
319     if(strcmp(outputActivationFunc,"sigmoid"))
320         numTraining= size(Y)(2);
321         # Element wise multiply for logprobs
322         cost = -1/numTraining * sum((Y .* log(AL)) + (1-Y) .* log(1-AL));
323
324         # Regularization cost
325         L = size(weights)(2);
326         L2RegularizationCost=0;
327         for l=1:L
328             wtSqr = weights{l} .* weights{l};
329             #disp(sum(sum(wtSqr,1)));
330             L2RegularizationCost+=sum(sum(wtSqr,1));
331         endfor
332         L2RegularizationCost = (lambd/(2*numTraining))*L2RegularizationCost;
333         cost = cost + L2RegularizationCost ;
334     elseif(strcmp(outputActivationFunc,'softmax'))
335         numTraining = size(Y)(2);
336         Y=Y';
337         # Select rows where Y=0,1, and 2 and concatenate to a long vector
338         #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)];
339         a =pickColumns(AL,Y,numClasses);
340
341         #Select the correct column for log prob
342         correct_probs = -log(a);
343         #Compute log loss
344         cost= sum(correct_probs)/numTraining;
345         # Regularization cost
346         L = size(weights)(2);
347         L2RegularizationCost=0;
348         for l=1:L
349             # Compute L2 Norm
350             wtSqr = weights{l} .* weights{l};
351             #disp(sum(sum(wtSqr,1)));
352             L2RegularizationCost+=sum(sum(wtSqr,1));
353         endfor
354         L2RegularizationCost = (lambd/(2*numTraining))*L2RegularizationCost;
355         cost = cost + L2RegularizationCost ;
356     endif
357 end
358

```

```

359
360
361 # Compute the backpropagation for 1 cycle
362 # Input : dA- Neural Network parameters
363 #          # cache - forward_cache & activation_cache
364 #          # Y-Output values
365 #          # outputActivationFunc- sigmoid/softmax
366 #          # numClasses
367 # Returns: Gradients
368 # dL/dwi= dL/dzi*A1-1
369 # dL/dbl = dL/dz1
370 # dL/dz_prev=dL/dz1*w
371 function [dA_prev dw db] = layerActivationBackward(dA, forward_cache,
372 activation_cache, Y, activationFunc,numClasses)
373     A_prev = forward_cache{1};
374     w =forward_cache{2};
375     b = forward_cache{3};
376     numTraining = size(A_prev)(2);
377     if (strcmp(activationFunc,"relu"))
378         dz = reluDerivative(dA, activation_cache);
379     elseif (strcmp(activationFunc,"sigmoid"))
380         dz = sigmoidDerivative(dA, activation_cache);
381     elseif(strcmp(activationFunc, "tanh"))
382         dz = tanhDerivative(dA, activation_cache);
383     elseif(strcmp(activationFunc, "softmax"))
384         #dz = softmaxDerivative(dA, activation_cache,Y,numClasses);
385         dz = stableSoftmaxDerivative(dA, activation_cache,Y,numClasses);
386     endif
387     if (strcmp(activationFunc,"softmax"))
388         w =forward_cache{2};
389         b = forward_cache{3};
390         # Add the regularization factor
391         dw = 1/numTraining * A_prev * dz;
392         db = 1/numTraining * sum(dz,1);
393         dA_prev = dz*w;
394     else
395         w =forward_cache{2};
396         b = forward_cache{3};
397         # Add the regularization factor
398         dw = 1/numTraining * dz * A_prev';
399         db = 1/numTraining * sum(dz,2);
400         dA_prev = w'*dz;
401     endif
402
403 end
404
405 # Compute the backpropagation with regularization for 1 cycle
406 # Input : dA-Neural Network parameters
407 #          # cache - forward_cache & activation_cache
408 #          # Y-Output values
409 #          # lambd
410 #          # outputActivationFunc- sigmoid/softmax
411 #          # numClasses
412 # Returns: Gradients
413 # dL/dwi= dL/dzi*A1-1
414 # dL/dbl = dL/dz1
415 # dL/dz_prev=dL/dz1*w
416 function [dA_prev dw db] = layerActivationBackwardWithReg(dA, forward_cache,
417 activation_cache, Y, lambd=0, activationFunc,numClasses)
418     A_prev = forward_cache{1};
419     w =forward_cache{2};
420     b = forward_cache{3};
421     numTraining = size(A_prev)(2);
422     if (strcmp(activationFunc,"relu"))

```

```

423     dz = reluDerivative(dA, activation_cache);
424 elseif (strcmp(activationFunc,"sigmoid"))
425     dz = sigmoidDerivative(dA, activation_cache);
426 elseif(strcmp(activationFunc, "tanh"))
427     dz = tanhDerivative(dA, activation_cache);
428 elseif(strcmp(activationFunc, "softmax"))
429     #dz = softmaxDerivative(dA, activation_cache,Y,numClasses);
430     dz = stableSoftmaxDerivative(dA, activation_cache,Y,numClasses);
431 endif
432 if (strcmp(activationFunc,"softmax"))
433     w =forward_cache{2};
434     b = forward_cache{3};
435     # Add the regularization factor
436     dw = 1/numTraining * A_prev * dz + (lambd/numTraining) * w';
437     db = 1/numTraining * sum(dz,1);
438     dA_prev = dz*w;
439 else
440     w =forward_cache{2};
441     b = forward_cache{3};
442     # Add the regularization factor
443     dw = 1/numTraining * dz * A_prev' + (lambd/numTraining) * w;
444     db = 1/numTraining * sum(dz,2);
445     dA_prev = w'*dz;
446 endif
447
448 end
449
450
451 # Compute the backpropagation for 1 cycle
452 # Input : AL: Output of L layer Network - weights
453 #          Y Real output
454 #          caches -- list of caches containing:
455 #          every cache of layerActivationForward() with "relu"/"tanh"
456 #          #(it's caches[1], for l in range(L-1) i.e l = 0...L-2)
457 #          #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1])
458 #
459 #          dropoutMat
460 #          lambd
461 #          keep_prob
462 #          hiddenActivationFunc - Activation function at hidden layers
463 sigmoid/tanh/relu
464 #          outputActivationFunc- sigmoid/softmax
465 #          numClasses
466 #
467 # Returns:
468 #     gradients -- A dictionary with the gradients
469 #                 gradients["dA" + str(l)] = ...
470 #                 gradients["dw" + str(l)] = ...
471 function [gradsDA gradsDW gradsDB]= backwardPropagationDeep(AL, Y,
472 activation_caches,forward_caches,
473                                     dropoutMat, lambd=0, keep_prob=1,
474 hiddenActivationFunc='relu',outputActivationFunc="sigmoid",numClasses)
475
476     # Set the number of layers
477     L = length(activation_caches);
478     m = size(AL)(2);
479
480     if (strcmp(outputActivationFunc, "sigmoid"))
481         # Initializing the backpropagation
482         # d1/dAL= -(y/a + (1-y)/(1-a)) - At the output layer
483         dAL = -((Y ./ AL) - (1 - Y) ./ (1 - AL));
484     elseif (strcmp(outputActivationFunc, "softmax"))
485         dAL=0;
486         Y=Y';

```

```

487     endif
488
489
490     # Since this is a binary classification the activation at output is
491 sigmoid
492     # Get the gradients at the last layer
493     # Inputs: "AL, Y, caches".
494     # Outputs: "gradients["dAL"], gradients["dwL"], gradients["dbL"]
495     activation_cache = activation_caches{L};
496     forward_cache = forward_caches(L);
497     # Note the cell array includes an array of forward caches. To get to this
498 we need to include the index {1}
499     if (lambd==0)
500         [dA dw db] = layerActivationBackward(dAL, forward_cache{1},
501 activation_cache, Y, activationFunc = outputActivationFunc,numClasses);
502     else
503         [dA dw db] = layerActivationBackwardWithReg(dAL, forward_cache{1},
504 activation_cache, Y, lambd, activationFunc =
505 outputActivationFunc,numClasses);
506     endif
507     if (strcmp(outputActivationFunc,"sigmoid"))
508         gradsDA{L}= dA;
509     elseif (strcmp(outputActivationFunc,"softmax"))
510         gradsDA{L}= dA';#Note the transpose
511     endif
512     gradsDW{L}= dw;
513     gradsDB{L}= db;
514
515     # Traverse in the reverse direction
516     for l =(L-1):-1:1
517         # Compute the gradients for L-1 to 1 for Relu/tanh
518         # Inputs: "gradients["dA" + str(l + 2)], caches".
519         # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l +
520 1)], gradients["db" + str(l + 1)]
521         activation_cache = activation_caches{l};
522         forward_cache = forward_caches(l);
523
524         #dA_prev_temp, dw_temp, db_temp =
525         layerActivationBackward(gradients['dA'+str(l+1)], current_cache,
526 activationFunc = "relu")
527         # dAl the derivative of the activation of the lth layer,is the first
528 element
529         dAl= gradsDA{l+1};
530         if(lambd == 0)
531             # Get the dropout mat
532             D = dropoutMat{l};
533             #Multiply by the dropoutMat
534             dAl= dAl .* D;
535             # Divide by keep_prob to keep expected value same
536             dAl = dAl ./ keep_prob;
537             [dA_prev_temp, dw_temp, db_temp] = layerActivationBackward(dAl,
538 forward_cache{1}, activation_cache, Y, activationFunc =
539 hiddenActivationFunc,numClasses);
540         else
541             [dA_prev_temp, dw_temp, db_temp] =
542             layerActivationBackwardWithReg(dAl, forward_cache{1}, activation_cache, Y,
543 lambd, activationFunc = hiddenActivationFunc,numClasses);
544         endif
545         gradsDA{l}= dA_prev_temp;
546         gradsDW{l}= dw_temp;
547         gradsDB{l}= db_temp;
548
549     endfor
550

```

```

551 end
552
553
554 # Perform Gradient Descent
555 # Input : weights and biases
556 #       : gradients -gradsW,gradsB
557 #       : learning rate
558 #       : outputActivationFunc
559 #output : Updated weights after 1 iteration
560 function [weights biases] = gradientDescent(weights, biases,gradsW,gradsB,
561 learningRate,outputActivationFunc="sigmoid")
562
563 L = size(weights)(2); # number of layers in the neural network
564
565 # Update rule for each parameter.
566 for l=1:(L-1)
567     weights{l} = weights{l} -learningRate* gradsW{l};
568     biases{l} = biases{l} -learningRate* gradsB{l};
569 endfor
570
571 if (strcmp(outputActivationFunc,"sigmoid"))
572     weights{L} = weights{L} -learningRate* gradsW{L};
573     biases{L} = biases{L} -learningRate* gradsB{L};
574 elseif (strcmp(outputActivationFunc,"softmax"))
575     weights{L} = weights{L} -learningRate* gradsW{L}';
576     biases{L} = biases{L} -learningRate* gradsB{L}';
577 endif
578
579
580
581 end
582
583
584 # Execute a L layer Deep learning model
585 # Input : X - Input features
586 #       : Y output
587 #       : layersDimensions - Dimension of layers
588 #       : hiddenActivationFunc - Activation function at hidden layer relu
589 /tanh
590 #       : outputActivationFunc - Activation function at hidden layer
591 sigmoid/softmax
592 #       : learning rate
593 #       : lambd
594 #       : keep_prob
595 #       : num of iterations
596 #output : Updated weights and biases after each iteration
597 function [weights biases costs] = L_Layer_DeepModel(X, Y, layersDimensions,
598 hiddenActivationFunc='relu',
599             outputActivationFunc="sigmoid", learning_rate = .3, lambd=0,
600 keep_prob=1, num_iterations = 10000, initType="default")#lr was 0.009
601
602 rand ("seed", 1);
603 costs = [] ;
604 if (strcmp(initType,"He"))
605     # He Initialization
606     [weights biases] = HeInitializeDeepModel(layersDimensions);
607 elseif (strcmp(initType,"Xav"))
608     # Xavier Initialization
609     [weights biases] = XavInitializeDeepModel(layersDimensions);
610 else
611     # Default initialization.
612     [weights biases] = initializeDeepModel(layersDimensions);
613 endif
614
```

```

615     # Loop (gradient descent)
616     for i = 0:num_iterations
617         # Forward propagation: [LINEAR -> RELU]^(L-1) -> LINEAR -> SIGMOID.
618         [AL forward_caches activation_caches dropoutMat] =
619         forwardPropagationDeep(X, weights, biases, keep_prob, hiddenActivationFunc,
620         outputActivationFunc=outputActivationFunc);
621
622             # Regularization parameter is 0
623             if (lambd==0)
624                 # Compute cost.
625                 cost = computeCost(AL,
626 Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(
627 layersDimensions)(2)));
628             else
629                 # Compute cost with regularization
630                 cost = computeCostWithReg(weights, AL, Y, lambd,
631 outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(la
632 yersDimensions)(2)));
633             endif
634             # Backward propagation.
635             [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y,
636 activation_caches,forward_caches, dropoutMat, lambd, keep_prob,
637 hiddenActivationFunc, outputActivationFunc=outputActivationFunc,
638
639 numClasses=layersDimensions(size(layersDimensions)(2)));
640             # Update parameters.
641             [weights biases] = gradientDescent(weights,biases,
642 gradsDW,gradsDB,learning_rate,outputActivationFunc=outputActivationFunc);
643
644
645             # Print the cost every 1000 iterations
646             if ( mod(i,1000) == 0)
647                 costs =[costs cost];
648                 #disp ("Cost after iteration"),
649 L2RegularizationCost(i),disp(cost);
650                 printf("Cost after iteration i=%i cost=%d\n",i,cost);
651             endif
652         endfor
653
654     end
655
656 # Execute a L layer Deep learning model with Stochastic Gradient descent
657 # Input : X - Input features
658 #           : Y output
659 #           : layersDimensions - Dimension of layers
660 #           : hiddenActivationFunc - Activation function at hidden layer relu
661 /tanh
662 #           : outputActivationFunc - Activation function at hidden layer
663 sigmoid/softmax
664 #           : learning rate
665 #           : mini_batch_size
666 #           : num of epochs
667 #output : Updated weights and biases after each iteration
668 function [weights biases costs] = L_Layer_DeepModel_SGD(X, Y,
669 layersDimensions, hiddenActivationFunc='relu',
670 outputActivationFunc="sigmoid",learning_rate = .3,
671                         mini_batch_size = 64, num_epochs = 2500)#lr was 0.009
672
673 rand ("seed", 1);
674 costs = [] ;
675
676     # Parameters initialization.
677     [weights biases] = initializeDeepModel(layersDimensions);
678     seed=10;

```

```

679     # Loop (gradient descent)
680     for i = 0:num_epochs
681         seed = seed + 1;
682         [mini_batches_X mini_batches_Y] = random_mini_batches(X, Y,
683         mini_batch_size, seed);
684
685         minibatches=length(mini_batches_X);
686         for batch=1:minibatches
687             X=mini_batches_X{batch};
688             Y=mini_batches_Y{batch};
689             # Forward propagation: [LINEAR -> RELU]^(L-1) -> LINEAR ->
690             SIGMOID/SOFTMAX.
691             [AL forward_caches activation_caches] =
692             forwardPropagationDeep(X, weights, biases,hiddenActivationFunc,
693             outputActivationFunc=outputActivationFunc);
694                 #disp(batch);
695                 #disp(size(X));
696                 #disp(size(Y));
697
698                 # Compute cost.
699                 cost = computeCost(AL,
700                 Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(
701                 layersDimensions)(2)));
702
703                 #disp(cost);
704                 # Backward propagation.
705                 [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y,
706                 activation_caches,forward_caches,hiddenActivationFunc,
707                 outputActivationFunc=outputActivationFunc,
708
709                 numClasses=layersDimensions(size(layersDimensions)(2)));
710                 # Update parameters.
711                 [weights biases] = gradientDescent(weights,biases,
712                 gradsDW,gradsDB,learning_rate,outputActivationFunc=outputActivationFunc);
713
714             endfor
715             # Print the cost every 1000 iterations
716             if ( mod(i,1000) == 0)
717                 costs =[costs cost];
718                 #disp ("Cost after iteration"), disp(i),disp(cost);
719                 printf("Cost after iteration i=%i cost=%d\n",i,cost);
720             endif
721         endfor
722
723     end
724
725     # Plot cost vs iterations
726     function plotCostVsIterations(maxIterations,costs,fig1)
727         iterations=[0:1000:maxIterations];
728         plot(iterations,costs);
729         title ("Cost vs no of iterations ");
730         xlabel("No of iterations");
731         ylabel("Cost");
732         print -dpng figReg2-o
733     end;
734
735     # Compute the predicted value for a given input
736     # Input : Neural Network parameters
737     #           : Input data
738     function [predictions]= predict(weights, biases,
739     x,keep_prob=1,hiddenActivationFunc="relu")
740         [AL forward_caches activation_caches] = forwardPropagationDeep(x,
741         weights, biases,keep_prob,hiddenActivationFunc);
742         predictions = (AL>0.5);

```

```

743 end
744
745 # Plot the decision boundary
746 function plotDecisionBoundary(data,weights,
747 biases,keep_prob=1,hiddenActivationFunc="relu",fig2)
748 %Plot a non-linear decision boundary learned by the SVM
749 colormap ("summer");
750
751 % Make classification predictions over a grid of values
752 x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)';
753 x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)';
754 [X1, X2] = meshgrid(x1plot, x2plot);
755 vals = zeros(size(X1));
756 # Plot the prediction for the grid
757 for i = 1:size(X1, 2)
758     gridPoints = [X1(:, i), X2(:, i)];
759     vals(:, i)=predict(weights, biases,gridPoints',keep_prob,
760 hiddenActivationFunc=hiddenActivationFunc);
761 endfor
762
763 scatter(data(:,1),data(:,2),8,c=data(:,3),"filled");
764 % Plot the boundary
765 hold on
766 #contour(X1, X2, vals, [0 0], 'LineWidth', 2);
767 contour(X1, X2, vals,"LineWidth",4);
768 title ({"3 layer Neural Network decision boundary"});
769 hold off;
770 print -dpng figReg22-o
771
772 end
773
774 #Compute scores
775 function [AL]= scores(weights, biases, x,hiddenActivationFunc="relu")
776     [AL forward_caches activation_caches] = forwardPropagationDeep(X,
777 weights, biases,hiddenActivationFunc);
778 end
779
780 # Create Random mini batches. Return cell arrays with the mini batches
781 # Input : X, Y
782 #          : Size of minibatch
783 #Output : mini batches X & Y
784 function [mini_batches_X mini_batches_Y]= random_mini_batches(X, Y,
785 miniBatchSize = 64, seed = 0)
786
787     rand ("seed", seed);
788     # Get number of training samples
789     m = size(X)(2);
790
791
792     # Create a list of random numbers < m
793     permutation = randperm(m);
794     # Randomly shuffle the training data
795     shuffled_X = X(:, permutation);
796     shuffled_Y = Y(:, permutation);
797
798     # Compute number of mini batches
799     numCompleteMinibatches = floor(m/miniBatchSize);
800     batch=0;
801     for k = 0:(numCompleteMinibatches-1)
802         #Set the start and end of each mini batch
803         batch=batch+1;
804         lower=(k*miniBatchSize)+1;
805         upper=(k+1) * miniBatchSize;
806         mini_batch_X = shuffled_X(:, lower:upper);

```

```

807     mini_batch_Y = shuffled_Y(:, lower:upper);
808
809     # Create cell arrays
810     mini_batches_X{batch} = mini_batch_X;
811     mini_batches_Y{batch} = mini_batch_Y;
812 endfor
813
814 # If the batch size does not cleanly divide with number of mini batches
815 if mod(m ,miniBatchSize) != 0
816     # Set the start and end of the last mini batch
817     l=floor(m/minibatchsize)*minibatchsize;
818     m=l+ mod(m,minibatchsize);
819     mini_batch_X = shuffled_X(:,(l+1):m);
820     mini_batch_Y = shuffled_Y(:,(l+1):m);
821
822     batch=batch+1;
823     mini_batches_X{batch} = mini_batch_X;
824     mini_batches_Y{batch} = mini_batch_Y;
825 endif
826 end
827
828 # Plot decision boundary
829 function plotDecisionBoundary1( data,weights, biases,keep_prob=1,
830 hiddenActivationFunc="relu")
831     % Make classification predictions over a grid of values
832     x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)';
833     x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)';
834     [X1, X2] = meshgrid(x1plot, x2plot);
835     vals = zeros(size(X1));
836     for i = 1:size(X1, 2)
837         gridPoints = [X1(:, i), X2(:, i)];
838         [AL forward_caches activation_caches] =
839         forwardPropagationDeep(gridPoints', weights,
840         biases,keep_prob,hiddenActivationFunc, outputActivationFunc="softmax");
841         [l m] = max(AL, [ ], 2);
842         vals(:, i)= m;
843     endfor
844
845     scatter(data(:,1),data(:,2),8,c=data(:,3),"filled");
846     % Plot the boundary
847     hold on
848     contour(X1, X2, vals,"linewidth",4);
849     print -dpng "fig-01.png"
850 end

```

8.Appendix 7 - Gradient Descent Optimization techniques

7.1 Python

```
1 # -*- coding: utf-8 -*-
2 ######
3 ######
4 #
5 # File: DLfunctions7.py
6 # Developer: Tinniam V Ganesh
7 # Date : 16 Apr 2018
8 #
9 #####
10 #####
11 import numpy as np
12 import matplotlib.pyplot as plt
13 import matplotlib
14 import matplotlib.pyplot as plt
15 from matplotlib import cm
16 import math
17 import sklearn
18 import sklearn.datasets
19
20 # Compute the sigmoid of a vector
21 def sigmoid(Z):
22     A=1/(1+np.exp(-Z))
23     cache=Z
24     return A,cache
25
26 # Compute the Relu of a vector
27 def relu(Z):
28     A = np.maximum(0,Z)
29     cache=Z
30     return A,cache
31
32 # Compute the tanh of a vector
33 def tanh(Z):
34     A = np.tanh(Z)
35     cache=Z
36     return A,cache
37
38 # Compute the softmax of a vector
39 def softmax(Z):
40     # get unnormalized probabilities
41     exp_scores = np.exp(Z.T)
42     # normalize them for each example
43     A = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
44     cache=Z
45     return A,cache
46
47 # Compute the stable softmax of a vector
48 def stableSoftmax(Z):
49     #Compute the softmax of vector x in a numerically stable way.
50     shiftZ = Z.T - np.max(Z.T, axis=1).reshape(-1,1)
51     exp_scores = np.exp(shiftZ)
52
53     # normalize them for each example
54     A = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
55     cache=Z
```

```

56     return A,cache
57
58 # Compute the derivative of Relu
59 def reluDerivative(dA, cache):
60
61     Z = cache
62     dZ = np.array(dA, copy=True) # just converting dZ to a correct object.
63     # When z <= 0, you should set dZ to 0 as well.
64     dZ[Z <= 0] = 0
65     return dZ
66
67 # Compute the derivative of sigmoid
68 def sigmoidDerivative(dA, cache):
69     Z = cache
70     s = 1/(1+np.exp(-Z))
71     dZ = dA * s * (1-s)
72     return dZ
73
74 # Compute the derivative of tanh
75 def tanhDerivative(dA, cache):
76     Z = cache
77     a = np.tanh(Z)
78     dZ = dA * (1 - np.power(a, 2))
79     return dZ
80
81 # Compute the derivative of softmax
82 def softmaxDerivative(dA, cache,y,numTraining):
83     # Note : dA not used. dL/dZ = dL/dA * dA/dZ = pi-yi
84     Z = cache
85     # Compute softmax
86     exp_scores = np.exp(Z.T)
87     # normalize them for each example
88     probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
89
90     # compute the gradient on scores
91     dZ = probs
92
93     # dZ = pi- yi
94     dZ[range(int(numTraining)),y[:,0]] -= 1
95     return(dZ)
96
97 # Compute the derivative of stable softmax
98 def stableSoftmaxDerivative(dA, cache,y,numTraining):
99     # Note : dA not used. dL/dZ = dL/dA * dA/dZ = pi-yi
100    Z = cache
101    # Compute stable softmax
102    shiftZ = Z.T - np.max(Z.T, axis=1).reshape(-1,1)
103    exp_scores = np.exp(shiftZ)
104    # normalize them for each example
105    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
106    #print(probs)
107    # compute the gradient on scores
108    dZ = probs
109
110    # dZ = pi- yi
111    dZ[range(int(numTraining)),y[:,0]] -= 1
112    return(dZ)
113
114
115 # Initialize the model
116 # Input : number of features
117 #           number of hidden units
118 #           number of units in output
119 # Returns: weight and bias matrices and vectors

```

```

120 def initializeModel(numFeats,numHidden,numOutput):
121     np.random.seed(1)
122     w1=np.random.randn(numHidden,numFeats)*0.01 # Multiply by .01
123     b1=np.zeros((numHidden,1))
124     w2=np.random.randn(numOutput,numHidden)*0.01
125     b2=np.zeros((numOutput,1))
126
127     # Create a dictionary of the neural network parameters
128     nnParameters={'w1':w1,'b1':b1,'w2':w2,'b2':b2}
129     return(nnParameters)
130
131
132 # Initialize model for L layers
133 # Input : List of units in each layer
134 # Returns: Initial weights and biases matrices for all layers
135 def initializeDeepModel(layerDimensions):
136     np.random.seed(3)
137     # note the weight matrix at layer '1' is a matrix of size (1,1-1)
138     # The Bias is a vectors of size (1,1)
139
140     # Loop through the layer dimension from 1.. L
141     layerParams = {}
142     for l in range(1,len(layerDimensions)):
143         layerParams['w' + str(l)] =
144             np.random.randn(layerDimensions[l],layerDimensions[l-1])*0.01 # Multiply by
145             .01
146         layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1))
147         np.savetxt('w' + str(l)+'.csv',layerParams['w' +
148             str(l)],delimiter=',')
149         np.savetxt('b' + str(l)+'.csv',layerParams['b' +
150             str(l)],delimiter=',')
151     return(layerParams)
152     return Z, cache
153
154 # He Initialization model for L layers
155 # Input : List of units in each layer
156 # Returns: Initial weights and biases matrices for all layers
157 # He initialization multiplies the random numbers with
158 # sqrt(2/layerDimensions[l-1])
159 def HeInitializeDeepModel(layerDimensions):
160     np.random.seed(3)
161     # note the weight matrix at layer '1' is a matrix of size (1,1-1)
162     # The Bias is a vectors of size (1,1)
163
164     # Loop through the layer dimension from 1.. L
165     layerParams = {}
166     for l in range(1,len(layerDimensions)):
167         layerParams['w' + str(l)] = np.random.randn(layerDimensions[l],
168             layerDimensions[l-1])*np.sqrt(2/layerDimensions[l-1])
169         layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1))
170
171     return(layerParams)
172     return Z, cache
173
174 # Xavier Initialization model for L layers
175 # Input : List of units in each layer
176 # Returns: Initial weights and biases matrices for all layers
177 # Xavier initialization multiplies the random numbers with
178 # sqrt(1/layerDimensions[l-1])
179 def XavInitializeDeepModel(layerDimensions):
180     np.random.seed(3)
181     # note the weight matrix at layer '1' is a matrix of size (1,1-1)
182     # The Bias is a vectors of size (1,1)
183

```

```

184     # Loop through the layer dimension from 1.. L
185     layerParams = {}
186     for l in range(1,len(layerDimensions)):
187         layerParams['w' + str(l)] = np.random.randn(layerDimensions[l],
188                                         layerDimensions[l-1])*np.sqrt(1/layerDimensions[l-1])
189         layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1))
190
191     return(layerParams)
192     return Z, cache
193
194 # Initialize velocity of
195 # Input : parameters
196 # Returns: v - Initial velocity
197 def initializeVelocity(parameters):
198
199     L = len(parameters)//2 # Create an integer
200     v = {}
201
202     # Initialize velocity with the same dimensions as w
203     for l in range(L):
204         v["dw" + str(l+1)] = np.zeros((parameters['w' + str(l+1)].shape[0],
205                                         parameters['w' + str(l+1)].shape[1]))
206         v["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0],
207                                         parameters['b' + str(l+1)].shape[1]))
208
209     return v
210
211 # Initialize RMSProp param
212 # Input : List of units in each layer
213 # Returns: s - Initial RMSProp
214 def initializeRMSProp(parameters):
215
216     L = len(parameters)//2 # Create an integer
217     s = {}
218
219     # Initialize velocity with the same dimensions as w
220     for l in range(L):
221         s["dw" + str(l+1)] = np.zeros((parameters['w' + str(l+1)].shape[0],
222                                         parameters['w' + str(l+1)].shape[1]))
223         s["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0],
224                                         parameters['b' + str(l+1)].shape[1]))
225
226     return s
227
228 # Initialize Adam param
229 # Input : List of units in each layer
230 # Returns: v and s - Adam paramaters
231 def initializeAdam(parameters) :
232
233     L = len(parameters) // 2 # number of layers in the neural networks
234     v = {}
235     s = {}
236
237     # Initialize v, s.
238     for l in range(L):
239
240         v["dw" + str(l+1)] = np.zeros((parameters['w' + str(l+1)].shape[0],
241                                         parameters['w' + str(l+1)].shape[1]))
242         v["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0],
243                                         parameters['b' + str(l+1)].shape[1]))
244         s["dw" + str(l+1)] = np.zeros((parameters['w' + str(l+1)].shape[0],
245                                         parameters['w' + str(l+1)].shape[1]))
246         s["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0],
247                                         parameters['b' + str(l+1)].shape[1]))

```

```

248     return v, s
249
250
251 # Compute the activation at a layer 'l' for forward prop in a Deep Network
252 # Input : A_prev - Activation of previous layer
253 #           W,b - Weight and bias matrices and vectors
254 #           keep_prob
255 #           activationFunc - Activation function - sigmoid, tanh, relu etc
256 # Returns : A, cache
257 #
258 # Z = W * X + b
259 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
260 def layerActivationForward(A_prev, w, b, keep_prob=1, activationFunc="relu"):
261
262     # Compute Z
263     Z = np.dot(w,A_prev) + b
264     forward_cache = (A_prev, w, b)
265     # Compute the activation for sigmoid
266     if activationFunc == "sigmoid":
267         A, activation_cache = sigmoid(Z)
268     # Compute the activation for Relu
269     elif activationFunc == "relu":
270         A, activation_cache = relu(Z)
271     # Compute the activation for tanh
272     elif activationFunc == 'tanh':
273         A, activation_cache = tanh(Z)
274     elif activationFunc == 'softmax':
275         A, activation_cache = stableSoftmax(Z)
276
277     cache = (forward_cache, activation_cache)
278     return A, cache
279
280 # Compute the forward propagation for layers 1..L
281 # Input : X - Input Features
282 #           parameters: Weights and biases
283 #           keep_prob
284 #           hiddenActivationFunc - Activation function at hidden layers
285 Relu/tanh
286 #           outputActivationFunc - Activation function at output -
287 sigmoid/softmax
288 # Returns : AL
289 #           caches
290 #           dropoutMat
291 # The forward propagation uses the Relu/tanh activation from layer 1..L-1 and
292 # sigmoid activation at layer L
293 def forwardPropagationDeep(X, parameters,keep_prob=1,
294 hiddenActivationFunc='relu',outputActivationFunc='sigmoid'):
295     caches = []
296     #initialize the dropout matrix
297     dropoutMat = {}
298     # Set A to X (A0)
299     A = X
300     L = len(parameters)//2 # number of layers in the neural network
301     # Loop through from layer 1 to upto layer L
302     for l in range(1, L):
303         A_prev = A
304         # Zi = Wi x Ai-1 + bi and Ai = g(Zi)
305         A, cache = layerActivationForward(A_prev, parameters['w'+str(l)],
306 parameters['b'+str(l)], keep_prob, activationFunc = hiddenActivationFunc)
307
308         # Randomly drop some activation units
309         # Create a matrix as the same shape as A
310         D = np.random.rand(A.shape[0],A.shape[1])
311         D = (D < keep_prob)

```

```

312     # We need to use the same 'dropout' matrix in backward propagation
313     # Save the dropout matrix for use in backprop
314     dropoutMat["D" + str(l)] = D
315     A= np.multiply(A,D)
316     A = np.divide(A,keep_prob)
317
318     caches.append(cache)
319
320     # last layer
321     AL, cache = layerActivationForward(A, parameters['w'+str(L)],
322                                         parameters['b'+str(L)], activationFunc = outputActivationFunc)
323     caches.append(cache)
324
325     return AL, caches, dropoutMat
326
327
328 # Compute the cost
329 # Input : parameters
330 #       : AL
331 #       : Y
332 #       :outputActivationFunc - Activation function at output -
333 # sigmoid/softmax/tanh
334 # Output: cost
335 def computeCost(parameters,AL,Y,outputActivationFunc="sigmoid"):
336     if outputActivationFunc=="sigmoid":
337         m= float(Y.shape[1])
338         # Element wise multiply for logprobs
339         cost=-1/m *np.sum(Y*np.log(AL) + (1-Y)*(np.log(1-AL)))
340         cost = np.squeeze(cost)
341     elif outputActivationFunc=="softmax":
342         # Take transpose of Y for softmax
343         Y=Y.T
344         m= float(len(Y))
345         # Compute log probs. Take the log prob of correct class based on
346         output y
347         correct_logprobs = -np.log(AL[range(int(m)),Y.T])
348         # Compute Loss
349         cost = np.sum(correct_logprobs)/m
350     return cost
351
352
353 # Compute the cost with regularization
354 # Input : parameters
355 #       : AL
356 #       : Y
357 #       : lambd
358 #       :outputActivationFunc - Activation function at output -
359 # sigmoid/softmax/tanh
360 # Output: cost
361 def computeCostWithReg(parameters,AL,Y,lambd,
362 outputActivationFunc="sigmoid"):
363
364     if outputActivationFunc=="sigmoid":
365         m= float(Y.shape[1])
366         # Element wise multiply for logprobs
367         cost=-1/m *np.sum(Y*np.log(AL) + (1-Y)*(np.log(1-AL)))
368         cost = np.squeeze(cost)
369
370         # Regularization cost
371         L= int(len(parameters)/2)
372         L2RegularizationCost=0
373         for l in range(L):
374             L2RegularizationCost+=np.sum(np.square(parameters['w'+str(l+1)]))

```

```

376
377     L2RegularizationCost = (lambd/(2*m))*L2RegularizationCost
378     cost = cost +  L2RegularizationCost
379
380
381     elif outputActivationFunc=="softmax":
382         # Take transpose of Y for softmax
383         Y=Y.T
384         m= float(len(Y))
385         # Compute log probs. Take the log prob of correct class based on
386         output y
387         correct_logprobs = -np.log(AL[range(int(m)),Y.T])
388         # Compute Loss
389         cost = np.sum(correct_logprobs)/m
390
391         # Regularization cost
392         L= int(len(parameters)/2)
393         L2RegularizationCost=0
394         for l in range(L):
395             L2RegularizationCost+=np.sum(np.square(parameters['w'+str(l+1)]))
396
397         L2RegularizationCost = (lambd/(2*m))*L2RegularizationCost
398         cost = cost +  L2RegularizationCost
399
400
401     return cost
402
403 # Compute the backpropagation for 1 cycle with dropout included
404 # Input : Neural Network parameters - dA
405 #          # cache - forward_cache & activation_cache
406 #          # Input features
407 #          # keep_prob
408 #          # Output values Y
409 # Returns: Gradients
410 # dL/dwi= dL/dzi*A1-1
411 # dL/db1 = dL/dz1
412 # dL/dz_prev=dL/dz1*w
413 def layerActivationBackward(dA, cache, Y, keep_prob=1,
414 activationFunc="relu"):
415     forward_cache, activation_cache = cache
416     A_prev, W, b = forward_cache
417     numtraining = float(A_prev.shape[1])
418     #print("n=",numtraining)
419     #print("no=",numtraining)
420     if activationFunc == "relu":
421         dz = reluDerivative(dA, activation_cache)
422     elif activationFunc == "sigmoid":
423         dz = sigmoidDerivative(dA, activation_cache)
424     elif activationFunc == "tanh":
425         dz = tanhDerivative(dA, activation_cache)
426     elif activationFunc == "softmax":
427         dz = stableSoftmaxDerivative(dA, activation_cache,Y,numtraining)
428
429     if activationFunc == 'softmax':
430         dw = 1/numtraining * np.dot(A_prev,dz)
431         db = 1/numtraining * np.sum(dz, axis=0, keepdims=True)
432         dA_prev = np.dot(dz,W)
433     else:
434         #print(numtraining)
435         dw = 1/numtraining *(np.dot(dz,A_prev.T))
436         #print("dw=",dw)
437         db = 1/numtraining * np.sum(dz, axis=1, keepdims=True)
438         #print("db=",db)
439         dA_prev = np.dot(W.T,dz)

```

```

440     return dA_prev, dw, db
441
442
443 # Compute the backpropagation with regularization for 1 cycle
444 # Input : dA- Neural Network parameters
445 #         # cache - forward_cache & activation_cache
446 #         # Output values Y
447 #         # lambd
448 #         # activationFunc
449 # Returns dA_prev, dw, db
450 # Returns: Gradients
451 # dL/dwi= dL/dzi*A1-1
452 # dL/dbl = dL/dz1
453 # dL/dz_prev=dL/dz1*w
454 def layerActivationBackwardWithReg(dA, cache, Y, lambd, activationFunc):
455     forward_cache, activation_cache = cache
456     A_prev, w, b = forward_cache
457     numtraining = float(A_prev.shape[1])
458     #print("n=",numtraining)
459     #print("no=",numtraining)
460     if activationFunc == "relu":
461         dz = reluDerivative(dA, activation_cache)
462     elif activationFunc == "sigmoid":
463         dz = sigmoidDerivative(dA, activation_cache)
464     elif activationFunc == "tanh":
465         dz = tanhDerivative(dA, activation_cache)
466     elif activationFunc == "softmax":
467         dz = stableSoftmaxDerivative(dA, activation_cache,Y,numtraining)
468
469     if activationFunc == 'softmax':
470         # Add the regularization factor
471         dw = 1/numtraining * np.dot(A_prev,dz) + (lambd/numtraining) * w.T
472         db = 1/numtraining * np.sum(dz, axis=0, keepdims=True)
473         dA_prev = np.dot(dz,w)
474     else:
475         # Add the regularization factor
476         dw = 1/numtraining *(np.dot(dz,A_prev.T)) + (lambd/numtraining) * w
477         #print("dw=",dw)
478         db = 1/numtraining * np.sum(dz, axis=1, keepdims=True)
479         #print("db=",db)
480         dA_prev = np.dot(w.T,dz)
481
482
483     return dA_prev, dw, db
484
485 # Compute the backpropagation for 1 cycle
486 # Input : AL: Output of L layer Network - weights
487 #         # Y Real output
488 #         # caches -- list of caches containing:
489 #         # dropoutMat
490 #         # lambd
491 #         # keep_prob
492 #         every cache of layerActivationForward() with "relu"/"tanh"
493 #         #(it's caches[], for l in range(L-1) i.e l = 0...L-2)
494 #         #the cache of layerActivationForward() with "sigmoid" (it's caches[L-
495 1])
496 #         # hiddenActivationFunc - Activation function at hidden layers -
497 #         # relu/sigmoid/tanh
498 #         #outputActivationFunc - Activation function at output -
499 #         # sigmoid/softmax
500 #
501 #     Returns:
502 #     gradients -- A dictionary with the gradients
503 #                 gradients["dA" + str(l)] = ...

```

```

504 # gradients["dw" + str(l)] = ...
505 # gradients["db" + str(l)] = ...
506 def backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd=0, keep_prob=1,
507 hiddenActivationFunc='relu',outputActivationFunc="sigmoid"):
508     #initialize the gradients
509     gradients = {}
510     # Set the number of layers
511     L = len(caches)
512     m = float(AL.shape[1])
513
514     if outputActivationFunc == "sigmoid":
515         Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL
516         # Initializing the backpropagation
517         #  $dL/dAL = -(y/a + (1-y)/(1-a))$  - At the output layer
518         dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
519     else:
520         dAL = 0
521         Y=Y.T
522
523     # Since this is a binary classification the activation at output is
524     sigmoid
525     # Get the gradients at the last layer
526     # Inputs: "AL, Y, caches".
527     # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dBW"]"
528     current_cache = caches[L-1]
529     if lambd==0:
530         gradients["dA" + str(L)], gradients["dw" + str(L)], gradients["db" +
531         str(L)] = layerActivationBackward(dAL, current_cache,
532                                         Y, activationFunc =
533                                         outputActivationFunc)
534     else: #Regularization
535         gradients["dA" + str(L)], gradients["dw" + str(L)], gradients["db" +
536         str(L)] = layerActivationBackwardWithReg(dAL, current_cache,
537                                         Y, lambd, activationFunc =
538                                         outputActivationFunc)
539
540     # Note dA for softmax is the transpose
541     if outputActivationFunc == "softmax":
542         gradients["dA" + str(L)] = gradients["dA" + str(L)].T
543     # Traverse in the reverse direction
544     for l in reversed(range(L-1)):
545         # Compute the gradients for L-1 to 1 for Relu/tanh
546         # Inputs: "gradients["dA" + str(l + 2)], caches".
547         # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l +
548         1)] , gradients["db" + str(l + 1)]
549         current_cache = caches[l]
550
551         #dA_prev_temp, dw_temp, db_temp =
552         layerActivationBackward(gradients['dA'+str(l+2)], current_cache,
553         activationFunc = "relu")
554         if lambd==0:
555
556             # In the reverse direction use the same dropout matrix
557             # Random dropout
558             # Multiply dA'1' with the dropoutMat and divide to keep the
559             expected value same
560             D = dropoutMat["D" + str(l+1)]
561             # Drop some dA1's
562             gradients['dA'+str(l+2)]= np.multiply(gradients['dA'+str(l+2)],D)
563             # Divide by keep_prob to keep expected value same
564             gradients['dA'+str(l+2)] =
565             np.divide(gradients['dA'+str(l+2)],keep_prob)
566

```

```

567         dA_prev_temp, dw_temp, db_temp =
568     layerActivationBackward(gradients['dA'+str(l+2)], current_cache, Y,
569     keep_prob=1, activationFunc = hiddenActivationFunc)
570
571     else:
572         dA_prev_temp, dw_temp, db_temp =
573     layerActivationBackwardWithReg(gradients['dA'+str(l+2)], current_cache, Y,
574     lambd, activationFunc = hiddenActivationFunc)
575
576     gradients["dA" + str(l + 1)] = dA_prev_temp
577     gradients["dw" + str(l + 1)] = dw_temp
578     gradients["db" + str(l + 1)] = db_temp
579
580
581     return gradients
582
583 # Perform Gradient Descent
584 # Input : Weights and biases
585 #           : gradients
586 #           : learning rate
587 #           : outputActivationFunc - Activation function at output -
588 # sigmoid/softmax
589 #output : Updated weights after 1 iteration
590 def gradientDescent(parameters, gradients,
591 learningRate,outputActivationFunc="sigmoid"):
592
593     L = int(len(parameters) / 2)
594     # Update rule for each parameter.
595     for l in range(L-1):
596         parameters["w" + str(l+1)] = parameters['w'+str(l+1)] -learningRate*
597     gradients['dw' + str(l+1)]
598         parameters["b" + str(l+1)] = parameters['b'+str(l+1)] -learningRate*
599     gradients['db' + str(l+1)]
600
601     if outputActivationFunc=="sigmoid":
602         parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
603     gradients['dw' + str(L)]
604         parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
605     gradients['db' + str(L)]
606     elif outputActivationFunc=="softmax":
607         parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
608     gradients['dw' + str(L)].T
609         parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
610     gradients['db' + str(L)].T
611
612     return parameters
613
614 # Update parameters with momentum
615 # Input : parameters
616 #           : gradients
617 #           : v
618 #           : beta
619 #           : learningRate
620 #           : outputActivationFunc - softmax/sigmoid
621 #output : Updated parameters and velocity
622 def gradientDescentWithMomentum(parameters, gradients, v, beta, learningRate,
623 outputActivationFunc="sigmoid"):
624
625     L = len(parameters) // 2 # number of layers in the neural networks
626     # Momentum update for each parameter
627     for l in range(L-1):
628
629         # Compute velocities
630         # v['dwk'] = beta *v['dwk'] + (1-beta)*dwk

```

```

631     v["dw" + str(l+1)] = beta*v["dw" + str(l+1)] + (1-beta) *
632     gradients['dw' + str(l+1)] = beta*v["db" + str(l+1)] + (1-beta) *
633     v["db" + str(l+1)] = beta*v["db" + str(l+1)] + (1-beta) *
634     gradients['db' + str(l+1)]
635     # Update parameters with velocities
636     parameters["w" + str(l+1)] = parameters['w' + str(l+1)] -
637     learningRate* v["dw" + str(l+1)]
638     parameters["b" + str(l+1)] = parameters['b' + str(l+1)] -
639     learningRate* v["db" + str(l+1)]
640
641     if outputActivationFunc=="sigmoid":
642         v["dw" + str(L)] = beta*v["dw" + str(L)] + (1-beta) * gradients['dw'
643 + str(L)]
644         v["db" + str(L)] = beta*v["db" + str(L)] + (1-beta) * gradients['db'
645 + str(L)]
646         parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
647     gradients['dw' + str(L)]
648         parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
649     gradients['db' + str(L)]
650     elif outputActivationFunc=="softmax":
651         v["dw" + str(L)] = beta*v["dw" + str(L)] + (1-beta) * gradients['dw'
652 + str(L)].T
653         v["db" + str(L)] = beta*v["db" + str(L)] + (1-beta) * gradients['db'
654 + str(L)].T
655         parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
656     gradients['dw' + str(L)].T
657         parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
658     gradients['db' + str(L)].T
659
660     return parameters, v
661
662
663 # Update parameters with RMSProp
664 # Input : parameters
665 #       : gradients
666 #       : s
667 #       : beta1
668 #       : learningRate
669 #       : outputActivationFunc - sigmoid/softmax
670 # output : Updated parameters and RMSProp
671 def gradientDescentWithRMSProp(parameters, gradients, s, beta1, epsilon,
672 learningRate, outputActivationFunc="sigmoid"):
673
674     L = len(parameters) // 2 # number of layers in the neural networks
675     # Momentum update for each parameter
676     for l in range(L-1):
677
678         # Compute RMSProp
679         # s['dwk'] = beta1 *s['dwk'] + (1-beta1)*dwk**2/sqrt(s['dwk'])
680         s["dw" + str(l+1)] = beta1*s["dw" + str(l+1)] + (1-beta1) * \
681             np.multiply(gradients['dw' + str(l+1)],gradients['dw' +
682 str(l+1)])
683         s["db" + str(l+1)] = beta1*s["db" + str(l+1)] + (1-beta1) * \
684             np.multiply(gradients['db' + str(l+1)],gradients['db' +
685 str(l+1)])
686         # Update parameters with RMSProp
687         parameters["w" + str(l+1)] = parameters['w' + str(l+1)] - \
688             learningRate* gradients['dw' + str(l+1)]/np.sqrt(s["dw" +
689 str(l+1)] + epsilon)
690         parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - \
691             learningRate* gradients['db' + str(l+1)]/np.sqrt(s["db" +
692 str(l+1)] + epsilon)
693
694     if outputActivationFunc=="sigmoid":

```

```

695     s["dw" + str(L)] = beta1*s["dw" + str(L)] + (1-beta1) * \
696         np.multiply(gradients['dw' + str(L)],gradients['dw' + str(L)])
697     s["db" + str(L)] = beta1*s["db" + str(L)] + (1-beta1) * \
698         np.multiply(gradients['db' + str(L)],gradients['db' + str(L)])
699     parameters["w" + str(L)] = parameters['w'+str(L)] - \
700         learningRate* gradients['dw' + str(L)]/np.sqrt(s["dw" + str(L)]
701 + epsilon)
702     parameters["b" + str(L)] = parameters['b'+str(L)] - \
703         learningRate* gradients['db' + str(L)]/np.sqrt(s["db" + str(L)]
704 + epsilon)
705     elif outputActivationFunc=="softmax":
706         s["dw" + str(L)] = beta1*s["dw" + str(L)] + (1-beta1) * \
707             np.multiply(gradients['dw' + str(L)].T,gradients['dw' +
708             str(L)].T)
709         s["db" + str(L)] = beta1*s["db" + str(L)] + (1-beta1) * \
710             np.multiply(gradients['db' + str(L)].T,gradients['db' +
711             str(L)].T)
712         parameters["w" + str(L)] = parameters['w'+str(L)] - \
713             learningRate* gradients['dw' + str(L)].T/np.sqrt(s["dw" +
714             str(L)] + epsilon)
715         parameters["b" + str(L)] = parameters['b'+str(L)] - \
716             learningRate* gradients['db' + str(L)].T/np.sqrt(s["db" +
717             str(L)] + epsilon)
718
719     return parameters, s
720
721 # Update parameters with Adam
722 # Input : parameters
723 #           : gradients
724 #           : v
725 #           : s
726 #           : t
727 #           : beta1
728 #           : beta2
729 #           : epsilon
730 #           : learningRate
731 #           : outputActivationFunc - sigmoid/softmax
732 # output : Updated parameters and RMSProp
733 def gradientDescentWithAdam(parameters, gradients, v, s, t,
734                             beta1=0.9, beta2=0.999, epsilon=1e-8,
735                             learningRate=0.1,
736                             outputActivationFunc="sigmoid"):
737
738     L = len(parameters) // 2
739     # Initializing first moment estimate, python dictionary
740     v_corrected = {}
741     # Initializing second moment estimate, python dictionary
742     s_corrected = {}
743
744     # Perform Adam upto L-1
745     for l in range(L-1):
746
747         # Compute momentum
748         v["dw" + str(l+1)] = beta1*v["dw" + str(l+1)] + \
749             (1-beta1) * gradients['dw' + str(l+1)]
750         v["db" + str(l+1)] = beta1*v["db" + str(l+1)] + \
751             (1-beta1) * gradients['db' + str(l+1)]
752
753         # Compute bias-corrected first moment estimate.
754         v_corrected["dw" + str(l+1)] = v["dw" + str(l+1)]/(1-
755             np.power(beta1,t))
756         v_corrected["db" + str(l+1)] = v["db" + str(l+1)]/(1-
757             np.power(beta1,t))

```

```

759
760
761     # Moving average of the squared gradients like RMSProp
762     s["dw" + str(l+1)] = beta2*s["dw" + str(l+1)] + \
763         (1-beta2) * np.multiply(gradients['dw' +
764             str(l+1)],gradients['dw' + str(l+1)])
765     s["db" + str(l+1)] = beta2*s["db" + str(l+1)] + \
766         (1-beta2) * np.multiply(gradients['db' +
767             str(l+1)],gradients['db' + str(l+1)])
768
769
770     # Compute bias-corrected second raw moment estimate.
771     s_corrected["dw" + str(l+1)] = s["dw" + str(l+1)]/(1-
772     np.power(beta2,t))
773     s_corrected["db" + str(l+1)] = s["db" + str(l+1)]/(1-
774     np.power(beta2,t))
775
776     # Update parameters.
777     d1=np.sqrt(s_corrected["dw" + str(l+1)]+epsilon)
778     d2=np.sqrt(s_corrected["db" + str(l+1)]+epsilon)
779     parameters["w" + str(l+1)] = parameters['w' + str(l+1)] - \
780         (learningRate* v_corrected["dw" + str(l+1)]/d1)
781     parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - \
782         (learningRate* v_corrected["db" + str(l+1)]/d2)
783
784     if outputActivationFunc=="sigmoid":
785         #Compute 1st moment for L
786         v["dw" + str(L)] = beta1*v["dw" + str(L)] + (1-beta1) *
787     gradients['dw' + str(L)]
788         v["db" + str(L)] = beta1*v["db" + str(L)] + (1-beta1) *
789     gradients['db' + str(L)]
790         # Compute bias-corrected first moment estimate.
791         v_corrected["dw" + str(L)] = v["dw" + str(L)]/(1-
792     np.power(beta1,t))
793         v_corrected["db" + str(L)] = v["db" + str(L)]/(1-
794     np.power(beta1,t))
795
796         # Compute 2nd moment for L
797         s["dw" + str(L)] = beta2*s["dw" + str(L)] + (1-beta2) * \
798             np.multiply(gradients['dw' + str(L)],gradients['dw' +
799             str(L)])
800         s["db" + str(L)] = beta2*s["db" + str(L)] + (1-beta2) * \
801             np.multiply(gradients['db' + str(L)],gradients['db' +
802             str(L)])
803
804         # Compute bias-corrected second raw moment estimate.
805         s_corrected["dw" + str(L)] = s["dw" + str(L)]/(1-
806     np.power(beta2,t))
807         s_corrected["db" + str(L)] = s["db" + str(L)]/(1-
808     np.power(beta2,t))
809
810         # Update parameters.
811         d1=np.sqrt(s_corrected["dw" + str(L)]+epsilon)
812         d2=np.sqrt(s_corrected["db" + str(L)]+epsilon)
813         parameters["w" + str(L)] = parameters['w' + str(L)] - \
814             (learningRate* v_corrected["dw" + str(L)]/d1)
815         parameters["b" + str(L)] = parameters['b' + str(L)] - \
816             (learningRate* v_corrected["db" + str(L)]/d2)
817
818     elif outputActivationFunc=="softmax":
819         # Compute 1st moment
820         v["dw" + str(L)] = beta1*v["dw" + str(L)] + (1-beta1) *
821     gradients['dw' + str(L)].T

```

```

822         v["db" + str(L)] = beta1*v["db" + str(L)] + (1-beta1) *
823     gradients['db' + str(L)].T
824         # Compute bias-corrected first moment estimate.
825         v_corrected["dw" + str(L)] = v["dw" + str(L)]/(1-
826 np.power(beta1,t))
827         v_corrected["db" + str(L)] = v["db" + str(L)]/(1-
828 np.power(beta1,t))
829
830         #Compute 2nd moment
831         s["dw" + str(L)] = beta2*s["dw" + str(L)] + (1-beta2) *
832 np.multiply(gradients['dw' + str(L)].T,gradients['dw' + str(L)].T)
833         s["db" + str(L)] = beta2*s["db" + str(L)] + (1-beta2) *
834 np.multiply(gradients['db' + str(L)].T,gradients['db' + str(L)].T)
835         # Compute bias-corrected second raw moment estimate.
836         s_corrected["dw" + str(L)] = s["dw" + str(L)]/(1-
837 np.power(beta2,t))
838         s_corrected["db" + str(L)] = s["db" + str(L)]/(1-
839 np.power(beta2,t))
840
841         # Update parameters.
842         d1=np.sqrt(s_corrected["dw" + str(L)]+epsilon)
843         d2=np.sqrt(s_corrected["db" + str(L)]+epsilon)
844         parameters["w" + str(L)] = parameters['w' + str(L)] - \
845             (learningRate* v_corrected["dw" + str(L)]/d1)
846         parameters["b" + str(L)] = parameters['b' + str(L)] - \
847             (learningRate* v_corrected["db" + str(L)]/d2)
848
849
850     return parameters, v, s
851
852
853 # Execute a L layer Deep learning model
854 # Input : X - Input features
855 #           : Y output
856 #           : layersDimensions - Dimension of layers
857 #           : hiddenActivationFunc - Activation function at hidden layer relu
858 /tanh/sigmoid
859 #           : outputActivationFunc - Activation function at output layer
860 sigmoid/softmax
861 #           : learning rate
862 #           : lambd
863 #           : keep_prob
864 #           : num of iteration
865 #           : initType
866 #output : parameters
867 def L_Layer_DeepModel(X1, Y1, layersDimensions, hiddenActivationFunc='relu',
868 outputActivationFunc="sigmoid",
869             learningRate = .3, lambd=0, keep_prob=1,
870 num_iterations = 10000,initType="default",
871 print_cost=False,figure="figa.png"):
872
873     np.random.seed(1)
874     costs = []
875
876     # Parameters initialization.
877     if initType == "He":
878         parameters = HeInitializeDeepModel(layersDimensions)
879     elif initType == "Xavier" :
880         parameters = XavInitializeDeepModel(layersDimensions)
881     else: #Default
882         parameters = initializeDeepModel(layersDimensions)
883     # Loop (gradient descent)
884     for i in range(0, num_iterations):

```

```

886     AL, caches, dropoutMat = forwardPropagationDeep(X1, parameters,
887     keep_prob,
888     hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
889
890         # Regularization parameter is 0
891         if lambd==0:
892             # Compute cost
893             cost = computeCost(parameters,AL, Y1,
894             outputActivationFunc=outputActivationFunc)
895             # Include L2 regularization
896         else:
897             # Compute cost
898             cost = computeCostWithReg(parameters,AL, Y1, lambd,
899             outputActivationFunc=outputActivationFunc)
900
901         # Backward propagation.
902         gradients = backwardPropagationDeep(AL, Y1, caches, dropoutMat,
903         lambd, keep_prob,
904         hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
905
906         # Update parameters.
907         parameters = gradientDescent(parameters, gradients,
908         learningRate=learningRate,outputActivationFunc=outputActivationFunc)
909
910
911         # Print the cost every 100 training example
912         if print_cost and i % 1000 == 0:
913             print ("Cost after iteration %i: %f" %(i, cost))
914         if print_cost and i % 1000 == 0:
915             costs.append(cost)
916
917         # plot the cost
918         plt.plot(np.squeeze(costs))
919         plt.ylabel('Cost')
920         plt.xlabel('No of iterations (x1000)')
921         plt.title("Learning rate =" + str(learningRate))
922         plt.savefig(figure,bbox_inches='tight')
923         #plt.show()
924         plt.clf()
925         plt.close()
926
927     return parameters
928
929 # Execute a L layer Deep learning model Stoachastic Gradient Descent
930 # Input : X - Input features
931 #
932 #       : Y output
933 #       : layersDimensions - Dimension of layers
934 #       : hiddenActivationFunc - Activation function at hidden layer relu
935 #       : outputActivationFunc - Activation function at output -
936 #           sigmoid/softmax
937 #       : learning rate
938 #       : lrDecay
939 #       : lambd
940 #       : keep_prob
941 #       : optimizer
942 #       : beta
943 #       : beta1
944 #       : beta2
945 #       : epsilon
946 #       : mini_batch_size
947 #       : num_epochs
948 #
949 #output : Updated weights and biases

```

```

950 def L_Layer_DeepModel_SGD(X1, Y1, layersDimensions,
951     hiddenActivationFunc='relu', outputActivationFunc="sigmoid",
952             learningRate = .3, lrDecay=False, decayRate=1,
953             lambd=0, keep_prob=1,
954     optimizer="gd", beta=0.9,beta1=0.9, beta2=0.999,
955             epsilon = 1e-8,mini_batch_size = 64, num_epochs =
956     2500, print_cost=False, figure="figa.png"):
957
958     print("lr=",learningRate)
959     print("lrDecay=",lrDecay)
960     print("decayRate=",decayRate)
961     print("lambd=",lambd)
962     print("keep_prob=",keep_prob)
963     print("optimizer=",optimizer)
964     print("beta=",beta)
965
966     print("beta1=",beta1)
967     print("beta2=",beta2)
968     print("epsilon=",epsilon)
969
970     print("mini_batch_size=",mini_batch_size)
971     print("num_epochs=",num_epochs)
972     print("epsilon=",epsilon)
973
974
975     t =0 # Adam counter
976     np.random.seed(1)
977     costs = []
978
979     # Parameters initialization.
980     parameters = initializeDeepModel(layersDimensions)
981
982     #Initialize the optimizer
983     if optimizer == "gd":
984         pass # no initialization required for gradient descent
985     elif optimizer == "momentum":
986         v = initializeVelocity(parameters)
987     elif optimizer == "rmsprop":
988         s = initializeRMSProp(parameters)
989     elif optimizer == "adam":
990         v,s = initializeAdam(parameters)
991
992     seed=10
993     # Loop for number of epochs
994     for i in range(num_epochs):
995         # Define the random minibatches. we increment the seed to reshuffle
996         # differently the dataset after each epoch
997         seed = seed + 1
998         minibatches = random_mini_batches(X1, Y1, mini_batch_size, seed)
999
1000        batch=0
1001        # Loop through each mini batch
1002        for minibatch in minibatches:
1003            #print("batch=",batch)
1004            batch=batch+1
1005            # Select a minibatch
1006            (minibatch_X, minibatch_Y) = minibatch
1007
1008            # Perfrom forward propagation
1009            AL, caches, dropoutMat = forwardPropagationDeep(minibatch_X,
1010 parameters, keep_prob,
1011 hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
1012
1013            # Compute cost

```

```

1014         # Regularization parameter is 0
1015         if lambd==0:
1016             # Compute cost
1017             cost = computeCost(parameters, AL, minibatch_Y,
1018 outputActivationFunc=outputActivationFunc)
1019             else: # Include L2 regularization
1020                 # Compute cost
1021                 cost = computeCostWithReg(parameters, AL, minibatch_Y, lambd,
1022 outputActivationFunc=outputActivationFunc)
1023
1024             # Backward propagation.
1025             gradients = backwardPropagationDeep(AL, minibatch_Y,
1026 caches, dropoutMat, lambd,
1027 keep_prob,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFu
1028 nc)
1029
1030             if optimizer == "gd":
1031                 # Update parameters normal gradient descent
1032                 parameters = gradientDescent(parameters, gradients,
1033 learningRate=learningRate,outputActivationFunc=outputActivationFunc)
1034             elif optimizer == "momentum":
1035                 # Update parameters for gradient descent with momentum
1036                 parameters, v = gradientDescentWithMomentum(parameters,
1037 gradients, v, beta, \
1038
1039 learningRate=learningRate,outputActivationFunc=outputActivationFunc)
1040             elif optimizer == "rmsprop":
1041                 # Update parameters for gradient descent with RMSProp
1042                 parameters, s = gradientDescentWithRMSProp(parameters,
1043 gradients, s, beta1, epsilon, \
1044
1045 learningRate=learningRate,outputActivationFunc=outputActivationFunc)
1046             elif optimizer == "adam":
1047                 t = t + 1 # Adam counter
1048                 parameters, v, s = gradientDescentWithAdam(parameters,
1049 gradients, v, s,
1050                                         t, beta1,
1051 beta2, epsilon,
1052
1053 learningRate=learningRate,outputActivationFunc=outputActivationFunc)
1054
1055             # Print the cost every 1000 epoch
1056             if print_cost and i % 100 == 0:
1057                 print ("Cost after epoch %i: %f" %(i, cost))
1058             if print_cost and i % 100 == 0:
1059                 costs.append(cost)
1060             if lrDecay == True:
1061                 learningRate = np.power(decayRate,(num_epochs/1000)) *
1062 learningRate
1063
1064
1065             # plot the cost
1066             plt.plot(np.squeeze(costs))
1067             plt.ylabel('Cost')
1068             plt.xlabel('No of epochs(x100)')
1069             plt.title("Learning rate =" + str(learningRate))
1070             plt.savefig('figure',bbox_inches='tight')
1071             #plt.show()
1072             plt.clf()
1073             plt.close()
1074
1075
1076 # Create random mini batches
1077 # Input : X - Input features

```

```

1078 #      : Y- output
1079 #      : miniBatchSizes
1080 #      : seed
1081 #output : mini_batches
1082 def random_mini_batches(X, Y, miniBatchSize = 64, seed = 0):
1083
1084     np.random.seed(seed)
1085     # Get number of training samples
1086     m = X.shape[1]
1087     # Initialize mini batches
1088     mini_batches = []
1089
1090     # Create a list of random numbers < m
1091     permutation = list(np.random.permutation(m))
1092     # Randomly shuffle the training data
1093     shuffled_X = X[:, permutation]
1094     shuffled_Y = Y[:, permutation].reshape((1,m))
1095
1096     # Compute number of mini batches
1097     numCompleteMinibatches = math.floor(m/miniBatchSize)
1098
1099     # For the number of mini batches
1100     for k in range(0, numCompleteMinibatches):
1101
1102         # Set the start and end of each mini batch
1103         mini_batch_X = shuffled_X[:, k*miniBatchSize : (k+1) * miniBatchSize]
1104         mini_batch_Y = shuffled_Y[:, k*miniBatchSize : (k+1) * miniBatchSize]
1105
1106         mini_batch = (mini_batch_X, mini_batch_Y)
1107         mini_batches.append(mini_batch)
1108
1109
1110     #if m % miniBatchSize != 0:. The batch does not evenly divide by the mini
1111 batch
1112     if m % miniBatchSize != 0:
1113         l=math.floor(m/miniBatchSize)*miniBatchSize
1114         # Set the start and end of last mini batch
1115         m=l+m % miniBatchSize
1116         mini_batch_X = shuffled_X[:,l:m]
1117         mini_batch_Y = shuffled_Y[:,l:m]
1118
1119         mini_batch = (mini_batch_X, mini_batch_Y)
1120         mini_batches.append(mini_batch)
1121
1122     return mini_batches
1123
1124
1125 # Plot a decision boundary
1126 # Input : Input Model,
1127 #
1128 #      X
1129 #      Y
1130 #      sz - Num of hiden units
1131 #      lr - Learning rate
1132 #      Fig to be saved as
1133 # Returns Null
1134 def plot_decision_boundary(model, X, y,lr,figure1="figb.png"):
1135     print("plot")
1136     # Set min and max values and give it some padding
1137     x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
1138     y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
1139     colors=['black','gold']
1140     cmap = matplotlib.colors.ListedColormap(colors)
1141     h = 0.01
1142     # Generate a grid of points with distance h between them

```

```

1142     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,
1143                           h))
1144     # Predict the function value for the whole grid
1145     Z = model(np.c_[xx.ravel(), yy.ravel()])
1146     Z = Z.reshape(xx.shape)
1147     # Plot the contour and training examples
1148     plt.contourf(xx, yy, Z, cmap="coolwarm")
1149     plt.ylabel('x2')
1150     plt.xlabel('x1')
1151     x=X.T
1152     y=y.T.reshape(300, )
1153     plt.scatter(x[:, 0], x[:, 1], c=y, s=20);
1154     print(x.shape)
1155     plt.title("Decision Boundary for learning rate:"+lr)
1156     plt.savefig(figure1, bbox_inches='tight')
1157     #plt.show()
1158
1159 # Predict output
1160 def predict(parameters,
1161             X, keep_prob=1, hiddenActivationFunc="relu", outputActivationFunc="sigmoid"):
1162     A2, cache, dropoutMat = forwardPropagationDeep(X, parameters, keep_prob=1,
1163             hiddenActivationFunc=hiddenActivationFunc, outputActivationFunc=outputActivationFunc)
1164     predictions = (A2>0.5)
1165     return predictions
1166
1167 # Predict probabilities
1168 def predict_proba(parameters, X, outputActivationFunc="sigmoid"):
1169     A2, cache = forwardPropagationDeep(X, parameters)
1170     if outputActivationFunc=="sigmoid":
1171         proba=A2
1172     elif outputActivationFunc=="softmax":
1173         proba=np.argmax(A2, axis=0).reshape(-1,1)
1174         print("A2=",A2.shape)
1175     return proba
1176
1177 # Plot a decision boundary
1178 # Input : Input Model,
1179 #          X
1180 #          Y
1181 #          sz - Num of hiden units
1182 #          lr - Learning rate
1183 #          Fig to be saved as
1184 # Returns Null
1185 def plot_decision_boundary1(x, y,w1,b1,w2,b2,figure2="figc.png"):
1186     #plot_decision_boundary(lambda x: predict(parameters, x.T),
1187     x1,y1.T,str(0.3),"fig2.png")
1188     h = 0.02
1189     x_min, x_max = x[:, 0].min() - 1, x[:, 0].max() + 1
1190     y_min, y_max = x[:, 1].min() - 1, x[:, 1].max() + 1
1191     xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
1192                           np.arange(y_min, y_max, h))
1193     z = np.dot(np.maximum(0, np.dot(np.c_[xx.ravel()], yy.ravel()), w1.T) +
1194     b1.T), w2.T) + b2.T
1195     z = np.argmax(z, axis=1)
1196     z = z.reshape(xx.shape)
1197
1198     fig = plt.figure()
1199     plt.contourf(xx, yy, z, cmap=plt.cm.Spectral, alpha=0.8)
1200     print(x.shape)
1201     y1=y.reshape(300, )
1202     plt.scatter(x[:, 0], x[:, 1], c=y1, s=40, cmap=plt.cm.Spectral)
1203     plt.xlim(xx.min(), xx.max())
1204     plt.ylim(yy.min(), yy.max())
1205     plt.savefig(figure2, bbox_inches='tight')

```

```

1206
1207
1208 # Load the data set
1209 def load_dataset():
1210     np.random.seed(1)
1211     train_X, train_Y = sklearn.datasets.make_circles(n_samples=300,
1212 noise=.05)
1213     np.random.seed(2)
1214     test_X, test_Y = sklearn.datasets.make_circles(n_samples=100, noise=.05)
1215     # visualize the data
1216     print(train_X.shape)
1217     print(train_Y.shape)
1218     print("load")
1219     #plt.scatter(train_X[:, 0], train_X[:, 1], c=train_Y, s=40,
1220 cmap=plt.cm.Spectral);
1221     train_X = train_X.T
1222     train_Y = train_Y.reshape((1, train_Y.shape[0]))
1223     test_X = test_X.T
1224     test_Y = test_Y.reshape((1, test_Y.shape[0]))
1225     return train_X, train_Y, test_X, test_Y

```

7.2 R

```

1 #####
2 #####
3 #
4 # File   : DLfunctions7.R
5 # Author : Tinniam V Ganesh
6 # Date   : 16 Apr 2018
7 #
8 #####
9 #####
10 library(ggplot2)
11 library(PRROC)
12 library(dplyr)
13
14 # Compute the sigmoid of a vector
15 sigmoid <- function(z){
16     A <- 1/(1+ exp(-z))
17     cache<-z
18     retvals <- list("A"=A, "Z"=z)
19     return(retvals)
20 }
21
22 # This is the older version. Very performance intensive
23 # Compute relu
24 reluOld <-function(z){
25     A <- apply(z, 1:2, function(x) max(0,x))
26     cache<-z
27     retvals <- list("A"=A, "Z"=z)
28     return(retvals)
29 }
30
31
32 # Compute the Relu of a vector (current version)
33 relu   <-function(z){
34     # Perform relu. Set values less than equal to 0 as 0
35     z[z<0]=0
36     A=z
37     cache<-z

```

```

38     retvals <- list("A"=A, "Z"=Z)
39     return(retvals)
40   }
41
42 # Compute the tanh activation of a vector
43 tanhActivation <- function(Z){
44   A <- tanh(Z)
45   cache<-Z
46   retvals <- list("A"=A, "Z"=Z)
47   return(retvals)
48 }
49
50 # Compute the softmax of a vector
51 softmax <- function(Z){
52   # get unnormalized probabilities
53   exp_scores = exp(t(Z))
54   # normalize them for each example
55   A = exp_scores / rowSums(exp_scores)
56   retvals <- list("A"=A, "Z"=Z)
57   return(retvals)
58 }
59
60 # Compute the derivative of ReLU
61 # g'(z) = 1 if z > 0 and 0 otherwise
62 reluDerivative <- function(dA, cache){
63   Z <- cache
64   dZ <- dA
65   # Create a logical matrix of values > 0
66   a <- Z > 0
67   # When z <= 0, you should set dz to 0 as well. Perform an element wise
68   multiply
69   dZ <- dZ * a
70   return(dZ)
71 }
72
73 # Compute the derivative of sigmoid
74 # Derivative g'(z) = a * (1-a)
75 sigmoidDerivative <- function(dA, cache){
76   Z <- cache
77   s <- 1/(1+exp(-Z))
78   dZ <- dA * s * (1-s)
79   return(dZ)
80 }
81
82 # Compute the derivative of tanh
83 # Derivative g'(z) = 1 - a^2
84 tanhDerivative <- function(dA, cache){
85   Z = cache
86   a = tanh(Z)
87   dZ = dA * (1 - a^2)
88   return(dZ)
89 }
90
91 # This function is used in computing the softmax derivative
92 # Populate a matrix of 1s in rows where Y==1
93 # This may need to be extended for K classes. Currently
94 # supports K=3 & K=10
95 popMatrix <- function(Y,numClasses){
96   a=rep(0,times=length(Y))
97   Y1=matrix(a,nrow=length(Y),ncol=numClasses)
98   #Set the rows and columns as 1's where Y is the class value
99   if(numClasses==3){
100     Y1[Y==0,1]=1
101     Y1[Y==1,2]=1

```

```

102     Y1[Y==2,3]=1
103 } else if (numClasses==10){
104     Y1[Y==0,1]=1
105     Y1[Y==1,2]=1
106     Y1[Y==2,3]=1
107     Y1[Y==3,4]=1
108     Y1[Y==4,5]=1
109     Y1[Y==5,6]=1
110     Y1[Y==6,7]=1
111     Y1[Y==7,8]=1
112     Y1[Y==8,9]=1
113     Y1[Y==9,0]=1
114 }
115 return(Y1)
116 }
117
118 # Compute the softmax derivative
119 softmaxDerivative <- function(dA, cache ,y,numTraining,numClasses){
120 # Note : dA not used. dL/dZ = dL/dA * dA/dZ = pi-yi
121 z <- cache
122 # Compute softmax
123 exp_scores = exp(t(z))
124 # normalize them for each example
125 probs = exp_scores / rowSums(exp_scores)
126 # Create a matrix of zeros
127 Y1=popMatrix(y,numClasses)
128 #a=rep(0,times=length(Y))
129 #Y1=matrix(a,nrow=length(Y),ncol=numClasses)
130 #Set the rows and columns as 1's where Y is the class value
131 dZ = probs-Y1
132 return(dZ)
133 }
134
135 # Initialize model for L layers
136 # Input : Vector of units in each layer
137 # Returns: Initial weights and biases matrices for all layers
138 initializeDeepModel <- function(layerDimensions){
139   set.seed(2)
140
141   # Initialize empty list
142   layerParams <- list()
143
144   # Note the weight matrix at layer '1' is a matrix of size (1,1-1)
145   # The Bias is a vectors of size (1,1)
146
147   # Loop through the layer dimension from 1.. L
148   # Indices in R start from 1
149   for(l in 2:length(layersDimensions)){
150     # Initialize a matrix of small random numbers of size 1 x 1-1
151     # Create random numbers of size 1 x 1-1
152     w=rnorm(layersDimensions[1]*layersDimensions[1-1])*0.01
153     # Create a weight matrix of size 1 x 1-1 with this initial weights and
154     # Add to list w1,w2... WL
155     layerParams[[paste('w',l-1,sep="")]] = matrix(w,nrow=layersDimensions[1],
156                                                 ncol=layersDimensions[1-1])
156     layerParams[[paste('b',l-1,sep="")]] = matrix(rep(0,layersDimensions[1]),
157
158     nrow=layersDimensions[1],ncol=1)
159   }
160   return(layerParams)
161 }
162
163
164
165

```

```

166 # He Initialization model for L layers
167 # Input : Vector of units in each layer
168 # Returns: Initial weights and biases matrices for all layers
169 # He initialization multiplies the random numbers with
170 sqrt(2/layerDimensions[previouslayer])
171 HeInitializeDeepModel <- function(layerDimensions){
172   set.seed(2)
173
174   # Initialize empty list
175   layerParams <- list()
176
177   # Note the weight matrix at layer 'l' is a matrix of size (1,1-1)
178   # The Bias is a vectors of size (1,1)
179
180   # Loop through the layer dimension from 1.. L
181   # Indices in R start from 1
182   for(l in 2:length(layersDimensions)){
183     # Initialize a matrix of small random numbers of size 1 x 1-1
184     # Create random numbers of size 1 x 1-1
185     w=rnorm(layersDimensions[l]*layersDimensions[l-1])
186
187     # Create a weight matrix of size 1 x 1-1 with this initial weights
188     and
189     # Add to list w1,w2... WL
190     # He initialization - Divide by sqrt(2/layerDimensions[previous
191     layer])
192     layerParams[[paste('w',l-1,sep="")]] =
193     matrix(w,nrow=layersDimensions[l],
194
195     ncol=layersDimensions[l-1])*sqrt(2/layersDimensions[l-1])
196     layerParams[[paste('b',l-1,sep="")]] =
197     matrix(rep(0,layersDimensions[l]),
198
199     nrow=layersDimensions[l],ncol=1)
200   }
201   return(layerParams)
202 }
203
204 # XavInitializeDeepModel Initialization model for L layers
205 # Input : Vector of units in each layer
206 # Returns: Initial weights and biases matrices for all layers
207 # He initialization multiplies the random numbers with
208 # sqrt(1/layerDimensions[previouslayer])
209 XavInitializeDeepModel <- function(layerDimensions){
210   set.seed(2)
211
212   # Initialize empty list
213   layerParams <- list()
214
215   # Note the weight matrix at layer 'l' is a matrix of size (1,1-1)
216   # The Bias is a vectors of size (1,1)
217
218   # Loop through the layer dimension from 1.. L
219   # Indices in R start from 1
220   for(l in 2:length(layersDimensions)){
221     # Initialize a matrix of small random numbers of size 1 x 1-1
222     # Create random numbers of size 1 x 1-1
223     w=rnorm(layersDimensions[l]*layersDimensions[l-1])
224
225     # Create a weight matrix of size 1 x 1-1 with this initial weights
226     and
227     # Add to list w1,w2... WL
228     # He initialization - Divide by sqrt(2/layerDimensions[previous
229     layer])
```

```

230     layerParams[[paste('w', l-1, sep="")]] =
231     matrix(w, nrow=layersDimensions[1],
232     ncol=layersDimensions[l-1])*sqrt(1/layersDimensions[l-1])
234     layerParams[[paste('b', l-1, sep="")]] =
235     matrix(rep(0, layersDimensions[1]),
236     nrow=layersDimensions[1], ncol=1)
238   }
239   return(layerParams)
240 }
241
242 # Initialize velocity
243 # Input : parameters
244 # Returns: v -Initial velocity
245 initializeVelocity <- function(parameters){
246
247   L <- length(parameters)/2
248   v <- list()
249
250   # Initialize velocity with the same dimensions as w
251   for(l in 1:L){
252     # Get the size of weight matrix
253     sz <- dim(parameters[[paste('w', l, sep="")]])
254     v[[paste('dw', l, sep="")]] = matrix(rep(0, sz[1]*sz[2]),
255                                             nrow=sz[1], ncol=sz[2])
256     #Get the size of bias matrix
257     sz <- dim(parameters[[paste('b', l, sep="")]])
258     v[[paste('db', l, sep="")]] = matrix(rep(0, sz[1]*sz[2]),
259                                             nrow=sz[1], ncol=sz[2])
260   }
261
262   return(v)
263 }
264
265
266 # Initialize RMSProp
267 # Input : parameters
268 # Returns: s - Initial RMSProp
269 initializeRMSProp <- function(parameters){
270
271   L <- length(parameters)/2
272   s <- list()
273
274   # Initialize velocity with the same dimensions as w
275   for(l in 1:L){
276     # Get the size of weight matrix
277     sz <- dim(parameters[[paste('w', l, sep="")]])
278     s[[paste('dw', l, sep="")]] = matrix(rep(0, sz[1]*sz[2]),
279                                             nrow=sz[1], ncol=sz[2])
280     #Get the size of bias matrix
281     sz <- dim(parameters[[paste('b', l, sep="")]])
282     s[[paste('db', l, sep="")]] = matrix(rep(0, sz[1]*sz[2]),
283                                             nrow=sz[1], ncol=sz[2])
284   }
285
286   return(s)
287 }
288
289 # Initialize Adam
290 # Input : parameters
291 # Returns: (v,s) - Initial Adam parameters
292 initializeAdam <- function(parameters){
293

```

```

294     L <- length(parameters)/2
295     v <- list()
296     s <- list()
297
298     # Initialize velocity with the same dimensions as w
299     for(l in 1:L){
300         # Get the size of weight matrix
301         sz <- dim(parameters[[paste('W',l,sep="")]])
302         v[[paste('dw',l,sep="")]] = matrix(rep(0,sz[1]*sz[2]),
303                                         nrow=sz[1],ncol=sz[2])
304         s[[paste('dw',l,sep="")]] = matrix(rep(0,sz[1]*sz[2]),
305                                         nrow=sz[1],ncol=sz[2])
306         #Get the size of bias matrix
307         sz <- dim(parameters[[paste('b',l,sep="")]])
308         v[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]*sz[2]),
309                                         nrow=sz[1],ncol=sz[2])
310         s[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]*sz[2]),
311                                         nrow=sz[1],ncol=sz[2])
312     }
313     retvals <- list("v"=v,"s"=s)
314     return(retvals)
315 }
316
317 # Compute the activation at a layer 'l' for forward prop in a Deep Network
318 # Input : A_prev - Activation of previous layer
319 #          w,b - Weight and bias matrices and vectors
320 #          activationFunc - Activation function - sigmoid, tanh, relu etc
321 # Returns : The Activation of this layer
322 #
323 # Z = W * X + b
324 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
325 layerActivationForward <- function(A_prev, w, b, activationFunc){
326
327     # Compute Z
328     z = w %*% A_prev
329     # Broadcast the bias 'b' by column
330     Z <- sweep(z,1,b,'+')
331
332     forward_cache <- list("A_prev"=A_prev, "w"=w, "b"=b)
333     # Compute the activation for sigmoid
334     if(activationFunc == "sigmoid"){
335         vals = sigmoid(Z)
336     } else if (activationFunc == "relu"){ # Compute the activation for relu
337         vals = relu(Z)
338     } else if(activationFunc == 'tanh'){ # compute the activation for tanh
339         vals = tanhActivation(Z)
340     } else if(activationFunc == 'softmax'){
341         vals = softmax(Z)
342     }
343     # Create a list of forward and activation cache
344     cache <- list("forward_cache"=forward_cache,
345 "activation_cache"=vals[['z']])
346     retvals <- list("A"=vals[['A']], "cache"=cache)
347     return(retvals)
348 }
349
350 # Compute the forward propagation for layers 1..L
351 # Input : X - Input Features
352 #          parameters: Weights and biases
353 #          keep_prob
354 #          hiddenActivationFunc - relu/sigmoid/tanh
355 #          outputActivationFunc - Activation function at hidden layer
356 #          sigmoid/softmax
357 # Returns : AL

```

```

358 #           caches
359 #           dropoutMat
360 # The forward propagation uses the Relu/tanh activation from layer 1..L-1 and
361 sigmoid activation at layer L
362 forwardPropagationDeep <- function(x, parameters, keep_prob=1,
363 hiddenActivationFunc='relu',
364                                     outputActivationFunc='sigmoid'){
365   caches <- list()
366   dropoutMat <- list()
367   # Set A to X (A0)
368   A <- x
369   L <- length(parameters)/2 # number of layers in the neural network
370   # Loop through from layer 1 to upto layer L
371   for(l in 1:(L-1)){
372     A_prev <- A
373     # Zi = Wi x Ai-1 + bi and Ai = g(z)
374     # Set w and b for layer 'l'
375     # Loop through from W1,W2...WL-1
376     w <- parameters[[paste("w",l,sep="")]]
377     b <- parameters[[paste("b",l,sep="")]]
378     # Compute the forward propagation through layer 'l' using the activation
379     # function
380     actForward <- layerActivationForward(A_prev,
381                                           w,
382                                           b,
383                                           activationFunc =
384   hiddenActivationFunc)
385     A <- actForward[['A']]
386     # Append the cache A_prev,w,b, z
387     caches[[l]] <- actForward
388
389     # Randomly drop some activation units
390     # Create a matrix as the same shape as A
391     set.seed(1)
392     i=dim(A)[1]
393     j=dim(A)[2]
394     a<-rnorm(i*j)
395     # Normalize a between 0 and 1
396     a = (a - min(a))/(max(a) - min(a))
397     # Create a matrix of D
398     D <- matrix(a,nrow=i, ncol=j)
399     # Find D which is less than equal to keep_prob
400     D <- D < keep_prob
401     # Remove some A's
402     A <- A * D
403     # Divide by keep_prob to keep expected value same
404     A <- A/keep_prob
405     dropoutMat[[paste("D",l,sep="")]] <- D
406   }
407
408   # Since this is binary classification use the sigmoid activation function
409   in
410   # last layer
411   # Set the weights and biases for the last layer
412   w <- parameters[[paste("w",L,sep="")]]
413   b <- parameters[[paste("b",L,sep="")]]
414   # Last layer
415   actForward = layerActivationForward(A, w, b, activationFunc =
416   outputActivationFunc)
417   AL <- actForward[['A']]
418   # Append the output of this forward propagation through the last layer
419   caches[[L]] <- actForward
420   # Create a list of the final output and the caches
421   fwdPropDeep <- list("AL"=AL,"caches"=caches,"dropoutMat"=dropoutMat)

```

```

422 return(fwdPropDeep)
423 }
425
426 # Function pickColumns(). This function is in computeCost()
427 # Pick columns
428 # Input : AL
429 #      : Y
430 #      : numClasses
431 # Output: a
432 pickColumns <- function(AL,Y,numClasses){
433   if(numClasses==3){
434     a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3])
435   }
436   else if (numClasses==10){
437     a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3],AL[Y==3,4],AL[Y==4,5],
438          AL[Y==5,6],AL[Y==6,7],AL[Y==7,8],AL[Y==8,9],AL[Y==9,10])
439   }
440   return(a)
441 }
442
443
444 # Compute the cost
445 # Input : Activation of last layer
446 #      : Output from data
447 #      :outputActivationFunc - Activation function at hidden layer
448 sigmoid/softmax
449 #      : numClasses
450 # Output: cost
451 computeCost <- function(AL,Y,outputActivationFunc="sigmoid",numClasses=3){
452   if(outputActivationFunc=="sigmoid"){
453     m= length(Y)
454     cost=-1/m*sum(Y*log(AL) + (1-Y)*log(1-AL))
455
456
457   }else if (outputActivationFunc=="softmax"){
458     # Select the elements where the y values are 0, 1 or 2 and make a vector
459     # Pick columns
460     #a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3])
461     m= length(Y)
462     a =pickColumns(AL,Y,numclasses)
463     #a = c(A2[y=k,k+1])
464     # Take log
465     correct_probs = -log(a)
466
467     # Compute loss
468     cost= sum(correct_probs)/m
469   }
470   return(cost)
471 }
472
473
474 # Compute the cost with Regularization
475 # Input : parameters
476 #      : AL-Activation of last layer
477 #      : Y-Output from data
478 #      : lambd
479 #      : outputActivationFunc - Activation function at hidden layer
480 sigmoid/softmax
481 #      : numClasses
482 # Output: cost
483 computeCostWithReg <- function(parameters, AL,Y,lambd,
484 outputActivationFunc="sigmoid",numClasses=3){
485

```

```

486         if(outputActivationFunc=="sigmoid"){
487             m= length(Y)
488             cost=-1/m*sum(Y*log(AL) + (1-Y)*log(1-AL))
489
490             # Regularization cost
491             L <- length(parameters)/2
492             L2RegularizationCost=0
493             for(l in 1:L){
494                 L2RegularizationCost = L2RegularizationCost +
495                     sum(parameters[[paste("W",l,sep="")]]^2)
496             }
497             L2RegularizationCost = (lambda/(2*m))*L2RegularizationCost
498             cost = cost + L2RegularizationCost
499
500         }else if (outputActivationFunc=="softmax"){
501             # Select the elements where the y values are 0, 1 or 2 and make a
502             vector
503             # Pick columns
504             #a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3])
505             m= length(Y)
506             a =pickColumns(AL,Y,numClasses)
507             #a = c(A2[y=k,k+1])
508             # Take log
509             correct_probs = -log(a)
510             # Compute loss
511             cost= sum(correct_probs)/m
512
513             # Regularization cost
514             L <- length(parameters)/2
515             L2RegularizationCost=0
516             # Add L2 norm
517             for(l in 1:L){
518                 L2RegularizationCost = L2RegularizationCost +
519                     sum(parameters[[paste("W",l,sep="")]]^2)
520             }
521             L2RegularizationCost = (lambda/(2*m))*L2RegularizationCost
522             cost = cost + L2RegularizationCost
523
524         }
525     }
526
527     # Compute the backpropagation through a layer
528     # Input : Neural Network parameters - dA
529     #          # cache - forward_cache & activation_cache
530     #          # Input features
531     #          # Output values Y
532     #          # activationFunc
533     #          # numClasses
534     # Returns: Gradients
535     # dL/dwi= dL/dzi*A1-1
536     # dL/db1 = dL/dz1
537     # dL/dz_prev=dL/dz1*w
538
539     layerActivationBackward <- function(dA, cache, Y,
540     activationFunc,numClasses){
541         # Get A_prev,W,b
542         forward_cache <-cache[['forward_cache']]
543         activation_cache <- cache[['activation_cache']]
544         A_prev <- forward_cache[['A_prev']]
545         numtraining = dim(A_prev)[2]
546         # Get Z
547         activation_cache <- cache[['activation_cache']]
548         if(activationFunc == "relu"){
549             dz <- reluDerivative(dA, activation_cache)

```

```

550 } else if(activationFunc == "sigmoid"){
551     dz <- sigmoidDerivative(dA, activation_cache)
552 } else if(activationFunc == "tanh"){
553     dz <- tanhDerivative(dA, activation_cache)
554 } else if(activationFunc == "softmax"){
555     dz <- softmaxDerivative(dA, activation_cache, Y, numtraining, numClasses)
556 }
557
558 if (activationFunc == 'softmax'){
559     w <- forward_cache[['w']]
560     b <- forward_cache[['b']]
561     dw = 1/numtraining * A_prev%*%dz
562     db = 1/numtraining * matrix(colSums(dz), nrow=1, ncol=numClasses)
563     dA_prev = dz %*% w
564 } else {
565     w <- forward_cache[['w']]
566     b <- forward_cache[['b']]
567     numtraining = dim(A_prev)[2]
568
569     dw = 1/numtraining * dz %*% t(A_prev)
570     db = 1/numtraining * rowSums(dz)
571     dA_prev = t(w) %*% dz
572 }
573 retvals <- list("dA_prev"=dA_prev, "dw"=dw, "db"=db)
574 return(retvals)
575 }
576
577 # Compute the backpropagation through a layer with Regularization
578 # Input : dA-Neural Network parameters
579 #         # cache - forward_cache & activation_cache
580 #         # Output values Y
581 #         # lambd
582 #         # activationFunc
583 #         # numClasses
584 # Returns: Gradients
585 # dL/dwi= dL/dzi*A1-1
586 # dL/dbl = dL/dzl
587 # dL/dz_prev=dL/dzl*w
588
589 layerActivationBackwardWithReg <- function(dA, cache, Y, lambd,
590 activationFunc, numClasses){
591     # Get A_prev,w,b
592     forward_cache <- cache[['forward_cache']]
593     activation_cache <- cache[['activation_cache']]
594     A_prev <- forward_cache[['A_prev']]
595     numtraining = dim(A_prev)[2]
596
597     # Get Z
598     activation_cache <- cache[['activation_cache']]
599     if(activationFunc == "relu"){
600         dz <- reluDerivative(dA, activation_cache)
601     } else if(activationFunc == "sigmoid"){
602         dz <- sigmoidDerivative(dA, activation_cache)
603     } else if(activationFunc == "tanh"){
604         dz <- tanhDerivative(dA, activation_cache)
605     } else if(activationFunc == "softmax"){
606         dz <- softmaxDerivative(dA,
607         activation_cache, Y, numtraining, numClasses)
608     }
609
610     if (activationFunc == 'softmax'){
611         w <- forward_cache[['w']]
612         b <- forward_cache[['b']]
613         # Add the regularization factor
614         dw = 1/numtraining * A_prev%*%dz + (lambd/numtraining) * t(w)

```

```

614         db = 1/numtraining* matrix(colSums(dz), nrow=1, ncol=numClasses)
615         dA_prev = dz %*% w
616     } else {
617         w <- forward_cache[['w']]
618         b <- forward_cache[['b']]
619         numtraining = dim(A_prev)[2]
620         # Add the regularization factor
621         dw = 1/numtraining * dz %*% t(A_prev) + (lambd/numtraining) * w
622         db = 1/numtraining * rowSums(dz)
623         dA_prev = t(w) %*% dz
624     }
625     retvals <- list("dA_prev"=dA_prev, "dw"=dw, "db"=db)
626     return(retvals)
627 }
628
629 # Compute the backpropagation for 1 cycle through all layers
630 # Input : AL: Output of L layer Network - weights
631 #          Y Real output
632 #          caches -- List of caches containing:
633 #          every cache of layerActivationForward() with "relu"/"tanh"
634 #          #(it's caches[], for l in range(L-1) i.e l = 0...L-2)
635 #          #the cache of layerActivationForward() with "sigmoid" (it's caches[L-
636 1])
637 #          dropoutMat
638 #          lambd
639 #          keep_prob
640 #          hiddenActivationFunc - Activation function at hidden layers -
641 relu/tanh/sigmoid
642 #          outputActivationFunc - Activation function at hidden layer
643 sigmoid/softmax
644 #          numClasses
645 # Returns:
646 #     gradients -- A dictionary with the gradients
647 #                 gradients["dA" + str(l)]
648 #                 gradients["dw" + str(l)]
649 #                 gradients["db" + str(l)]
650 backwardPropagationDeep <- function(AL, Y, caches, dropoutMat, lambd=0,
651 keep_prob=0, hiddenActivationFunc='relu',
652                                     outputActivationFunc="sigmoid", numClasses){
653     #initialize the gradients
654     gradients = list()
655     # Set the number of layers
656     L = length(caches)
657     numTraining = dim(AL)[2]
658
659     if(outputActivationFunc == "sigmoid")
660         # Initializing the backpropagation
661         # d1/dAL= -(y/a) - ((1-y)/(1-a)) - At the output layer
662         dAL = -(Y/AL) -(1 - Y)/(1 - AL)
663     else if(outputActivationFunc == "softmax"){
664         dAL=0
665         Y=t(Y)
666     }
667
668     # Get the gradients at the last layer
669     # Inputs: "AL, Y, caches".
670     # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dB"]
671     # Start with Layer L
672     # Get the current cache
673     current_cache = caches[[L]]$cache
674     if (lambd==0){
675         retvals <- layerActivationBackward(dAL, current_cache, Y,
676                                         activationFunc =
677                                         outputActivationFunc, numClasses)

```

```

678 } else {
679     retvals = layerActivationBackwardWithReg(dAL, current_cache, Y, lambd,
680                                         activationFunc =
681 outputActivationFunc, numClasses)
682 }
683
684
685
686 #Note: Take the transpose of dA
687 if(outputActivationFunc == "sigmoid")
688     gradients[[paste("dA",L,sep="")]] <- retvals[['dA_prev']]
689 else if(outputActivationFunc == "softmax")
690     gradients[[paste("dA",L,sep="")]] <- t(retvals[['dA_prev']])
691
692 gradients[[paste("dw",L,sep="")]] <- retvals[['dw']]
693 gradients[[paste("db",L,sep="")]] <- retvals[['db']]
694
695
696
697 # Traverse in the reverse direction
698 for(l in (L-1):1){
699     # Compute the gradients for L-1 to 1 for Relu/tanh
700     # Inputs: "gradients["dA" + str(l + 2)], caches".
701     # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l + 1)] ,
702     gradients["db" + str(l + 1)]
703     current_cache = caches[[1]]$cache
704     if (lambd==0){
705         # Get the dropout matrix
706         D <- dropoutMat[[paste("D",l,sep="")]]
707         # Multiply gradient with dropout matrix
708         gradients[[paste('dA',l+1,sep="")]] =
709     gradients[[paste('dA',l+1,sep="")]] * D
710         # Divide by keep_prob to keep expected value same
711         gradients[[paste('dA',l+1,sep="")]] =
712     gradients[[paste('dA',l+1,sep="")]]/keep_prob
713         retvals =
714     layerActivationBackward(gradients[[paste('dA',l+1,sep="")]],
715                             current_cache, Y,
716                             activationFunc = hiddenActivationFunc)
717     } else {
718         retvals =
719     layerActivationBackwardWithReg(gradients[[paste('dA',l+1,sep="")]],
720                                     current_cache, Y, lambd,
721                                     activationFunc =
722     hiddenActivationFunc)
723     }
724
725     gradients[[paste("dA",l,sep="")]] <- retvals[['dA_prev']]
726     gradients[[paste("dw",l,sep="")]] <- retvals[['dw']]
727     gradients[[paste("db",l,sep="")]] <- retvals[['db']]
728 }
729
730 return(gradients)
731 }
732
733
734 # Perform Gradient Descent
735 # Input : Weights and biases
736 #       : gradients
737 #       : learning rate
738 #       : outputActivationFunc - Activation function at hidden layer
739 #       : sigmoid/softmax
740 #output : Updated weights after 1 iteration

```

```

741 gradientDescent <- function(parameters, gradients,
742 learningRate,outputActivationFunc="sigmoid"){
743
744     L = length(parameters)/2 # number of layers in the neural network
745     # Update rule for each parameter. Use a for loop.
746     for(l in 1:(L-1)){
747         parameters[[paste("w",l,sep="")]] = parameters[[paste("w",l,sep="")]] -
748             learningRate* gradients[[paste("dw",l,sep="")]]
749         parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] -
750             learningRate* gradients[[paste("db",l,sep="")]]
751     }
752     if(outputActivationFunc=="sigmoid"){
753         parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]] -
754             learningRate* gradients[[paste("dw",L,sep="")]]
755         parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] -
756             learningRate* gradients[[paste("db",L,sep="")]]
757
758     }else if (outputActivationFunc=="softmax"){
759         parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]] -
760             learningRate* t(gradients[[paste("dw",L,sep="")]])
761         parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] -
762             learningRate* t(gradients[[paste("db",L,sep="")]])
763
764     }
765     return(parameters)
766 }
767
768 # Perform Gradient Descent with momentum
769 # Input : Weights and biases
770 #          : beta
771 #          : gradients
772 #          : learning rate
773 #          : outputActivationFunc - Activation function at hidden layer
774 sigmoid/softmax
775 #output : Updated weights after 1 iteration
776 gradientDescentWithMomentum <- function(parameters, gradients,v, beta,
777 learningRate,outputActivationFunc="sigmoid"){
778
779     L = length(parameters)/2 # number of layers in the neural network
780
781     # Update rule for each parameter. Use a for loop.
782     for(l in 1:(L-1)){
783         # Compute velocities
784         # v['dwk'] = beta *v['dwk'] + (1-beta)*dwk
785         v[[paste("dw",l, sep="")]] = beta*v[[paste("dw",l, sep="")]] +
786             (1-beta) * gradients[[paste('dw',l,sep="")]]
787         v[[paste("db",l, sep="")]] = beta*v[[paste("db",l, sep="")]] +
788             (1-beta) * gradients[[paste('db',l,sep="")]]
789
790         parameters[[paste("w",l,sep="")]] = parameters[[paste("w",l,sep="")]] -
791             learningRate* v[[paste("dw",l, sep="")]]
792         parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] -
793             learningRate* v[[paste("db",l, sep="")]]
794     }
795
796     # Compute for the Lth layer
797     if(outputActivationFunc=="sigmoid"){
798         v[[paste("dw",L, sep="")]] = beta*v[[paste("dw",L, sep="")]] +
799             (1-beta) * gradients[[paste('dw',L,sep="")]]
800         v[[paste("db",L, sep="")]] = beta*v[[paste("db",L, sep="")]] +
801             (1-beta) * gradients[[paste('db',L,sep="")]]
802
803     }
804

```

```

805     parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
806     -
807     learningRate* v[[paste("dw",l, sep="")]]
808     parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
809     -
810     learningRate* v[[paste("db",l, sep="")]]
811
812 }else if (outputActivationFunc=="softmax"){
813   v[[paste("dw",L, sep="")]] = beta*v[[paste("dw",L, sep="")]] +
814   (1-beta) * t(gradients[[paste('dw',L,sep="")]])
815   v[[paste("db",L, sep="")]] = beta*v[[paste("db",L, sep="")]] +
816   (1-beta) * t(gradients[[paste('db',L,sep="")]])
817   parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
818   -
819   learningRate* t(gradients[[paste("dw",L,sep="")]])
820   parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
821   -
822   learningRate* t(gradients[[paste("db",L,sep="")]])
823 }
824 return(parameters)
825 }
826
827
828 # Perform Gradient Descent with RMSProp
829 # Input : parameters
830 #       : gradients
831 #       : s
832 #       : beta1
833 #       : epsilon
834 #       : learning rate
835 #       : outputActivationFunc - Activation function at hidden layer
836 # sigmoid/softmax
837 #output : Updated weights after 1 iteration
838 gradientDescentWithRMSProp <- function(parameters, gradients,s, beta1,
839 epsilon, learningRate,outputActivationFunc="sigmoid"){
840   L = length(parameters)/2 # number of layers in the neural network
841   # Update rule for each parameter. Use a for loop.
842   for(l in 1:(L-1)){
843     # Compute RMSProp
844     # s['dwk'] = beta1 *s['dwk'] + (1-beta1)*dwk**2/sqrt(s['dwk'])
845     # Element wise multiply of gradients
846     s[[paste("dw",l, sep="")]] = beta1*s[[paste("dw",l, sep="")]] +
847     (1-beta1) * gradients[[paste('dw',l,sep="")]] *
848 gradients[[paste('dw',l,sep="")]]
849     s[[paste("db",l, sep="")]] = beta1*s[[paste("db",l, sep="")]] +
850     (1-beta1) * gradients[[paste('db',l,sep="")]] *
851 gradients[[paste('db',l,sep="")]]
852
853     parameters[[paste("w",l,sep="")]] = parameters[[paste("w",l,sep="")]]
854   -
855     learningRate *
856 gradients[[paste('dw',l,sep="")]]/sqrt(s[[paste("dw",l, sep="")]]+epsilon)
857     parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]]
858   -
859
860 learningRate*gradients[[paste('db',l,sep="")]]/sqrt(s[[paste("db",l,
861 sep="")]]+epsilon)
862   }
863
864   # Compute for the Lth layer
865   if(outputActivationFunc=="sigmoid"){
866     s[[paste("dw",L, sep="")]] = beta1*s[[paste("dw",L, sep="")]] +
867     (1-beta1) * gradients[[paste('dw',L,sep="")]] *
868 gradients[[paste('dw',L,sep="")]]

```

```

869     s[[paste("db",L, sep="")]] = beta1*s[[paste("db",L, sep="")]] +
870         (1-beta1) * gradients[[paste('db',L,sep="")]] *
871     gradients[[paste('db',L,sep="")]]
872
873     parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
874 -
875         learningRate*
876     gradients[[paste('dw',l,sep="")]]/sqrt(s[[paste("dw",L, sep="")]]+epsilon)
877     parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
878 -
879         learningRate* gradients[[paste('db',l,sep="")]]/sqrt(
880     s[[paste("db",L, sep="")]]+epsilon)
881
882     }else if (outputActivationFunc=="softmax"){
883         s[[paste("dw",L, sep="")]] = beta1*s[[paste("dw",L, sep="")]] +
884             (1-beta1) * t(gradients[[paste('dw',L,sep="")]]) *
885         t(gradients[[paste('dw',L,sep="")]])
886         s[[paste("db",L, sep="")]] = beta1*s[[paste("db",L, sep="")]] +
887             (1-beta1) * t(gradients[[paste('db',L,sep="")]]) *
888         t(gradients[[paste('db',L,sep="")]])
889
890     parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
891 -
892         learningRate*
893     t(gradients[[paste("dw",L,sep="")]])/sqrt(s[[paste("dw",L, sep="")]]+epsilon)
894     parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
895 -
896         learningRate* t(gradients[[paste("db",L,sep="")]])/sqrt(
897     s[[paste("db",L, sep="")]]+epsilon)
898     }
899     return(parameters)
900 }
901
902
903 # Perform Gradient Descent with Adam
904 # Input : parameters
905 #       : gradients
906 #       : v
907 #       : s
908 #       : t
909 #       : beta1
910 #       : beta2
911 #       : epsilon
912 #       : learning rate
913 #       : outputActivationFunc - Activation function at hidden layer
914 # sigmoid/softmax
915 #output : Updated weights after 1 iteration
916 gradientDescentWithAdam <- function(parameters, gradients,v, s, t,
917                                     beta1=0.9, beta2=0.999, epsilon=10^-8,
918                                     learningRate=0.1,outputActivationFunc="sigmoid"){
919
920     L = length(parameters)/2 # number of layers in the neural network
921     v_corrected <- list()
922     s_corrected <- list()
923     # Update rule for each parameter. Use a for loop.
924     for(l in 1:(L-1)){
925         # v['dwb'] = beta *v['dwb'] + (1-beta)*dwb
926         v[[paste("dw",l, sep="")]] = beta1*v[[paste("dw",l, sep="")]] +
927             (1-beta1) * gradients[[paste('dw',l,sep="")]]
928         v[[paste("db",l, sep="")]] = beta1*v[[paste("db",l, sep="")]] +
929             (1-beta1) * gradients[[paste('db',l,sep="")]]
930
931         # Compute bias-corrected first moment estimate.

```

```

932     v_corrected[[paste("dw",1, sep="")]] = v[[paste("dw",1, sep="")]]/(1-
933 beta1^t)
934     v_corrected[[paste("db",1, sep="")]] = v[[paste("db",1, sep="")]]/(1-
935 beta1^t)
936
937
938     # Element wise multiply of gradients
939     s[[paste("dw",1, sep="")]] = beta2*s[[paste("dw",1, sep="")]] +
940         (1-beta2) * gradients[[paste('dw',1,sep="")]]
941     gradients[[paste('dw',1,sep="")]]
942     s[[paste("db",1, sep="")]] = beta2*s[[paste("db",1, sep="")]] +
943         (1-beta2) * gradients[[paste('db',1,sep="")]]
944     gradients[[paste('db',1,sep="")]]
945
946     # Compute bias-corrected second moment estimate.
947     s_corrected[[paste("dw",1, sep="")]] = s[[paste("dw",1, sep="")]]/(1-
948 beta2^t)
949     s_corrected[[paste("db",1, sep="")]] = s[[paste("db",1, sep="")]]/(1-
950 beta2^t)
951
952     # Update parameters.
953     d1=sqrt(s_corrected[[paste("dw",1, sep="")]]+epsilon)
954     d2=sqrt(s_corrected[[paste("db",1, sep="")]]+epsilon)
955
956     parameters[[paste("w",1,sep="")]] = parameters[[paste("w",1,sep="")]]
957
958     - learningRate * v_corrected[[paste("dw",1, sep="")]]/d1
959     parameters[[paste("b",1,sep="")]] = parameters[[paste("b",1,sep="")]]
960
961     - learningRate*v_corrected[[paste("db",1, sep="")]]/d2
962 }
963
964     # Compute for the Lth layer
965     if(outputActivationFunc=="sigmoid"){
966         v[[paste("dw",L, sep="")]] = beta1*v[[paste("dw",L, sep="")]] +
967             (1-beta1) * gradients[[paste('dw',L,sep="")]]
968         v[[paste("db",L, sep="")]] = beta1*v[[paste("db",L, sep="")]] +
969             (1-beta1) * gradients[[paste('db',L,sep="")]]
970
971
972     # Compute bias-corrected first moment estimate.
973     v_corrected[[paste("dw",L, sep="")]] = v[[paste("dw",L, sep="")]]/(1-
974 beta1^t)
975     v_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-
976 beta1^t)
977
978
979     # Element wise multiply of gradients
980     s[[paste("dw",L, sep="")]] = beta2*s[[paste("dw",L, sep="")]] +
981         (1-beta2) * gradients[[paste('dw',L,sep="")]]
982     gradients[[paste('dw',L,sep="")]]
983     s[[paste("db",L, sep="")]] = beta2*s[[paste("db",L, sep="")]] +
984         (1-beta2) * gradients[[paste('db',L,sep="")]]
985     gradients[[paste('db',L,sep="")]]
986
987     # Compute bias-corrected second moment estimate.
988     s_corrected[[paste("dw",L, sep="")]] = s[[paste("dw",L, sep="")]]/(1-
989 beta2^t)
990     s_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-
991 beta2^t)
992
993     # Update parameters.
994     d1=sqrt(s_corrected[[paste("dw",L, sep="")]]+epsilon)
995     d2=sqrt(s_corrected[[paste("db",L, sep="")]]+epsilon)

```

```

996     parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
997     -
998         learningRate * v_corrected[[paste("dw",L, sep="")]]/d1
999     parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
1000    -
1001        learningRate*v_corrected[[paste("db",L, sep="")]]/d2
1002
1003
1004 }else if (outputActivationFunc=="softmax"){
1005     v[[paste("dw",L, sep="")]] = beta1*v[[paste("dw",L, sep="")]] +
1006         (1-beta1) * t(gradients[[paste('dw',L,sep="")]])]
1007     v[[paste("db",L, sep="")]] = beta1*v[[paste("db",L, sep="")]] +
1008         (1-beta1) * t(gradients[[paste('db',L,sep="")]])]
1009
1010
1011     # Compute bias-corrected first moment estimate.
1012     v_corrected[[paste("dw",L, sep="")]] = v[[paste("dw",L, sep="")]]/(1-
1013 beta1^t)
1014     v_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-
1015 beta1^t)
1016
1017
1018     # Element wise multiply of gradients
1019     s[[paste("dw",L, sep="")]] = beta2*s[[paste("dw",L, sep="")]] +
1020         (1-beta2) * t(gradients[[paste('dw',L,sep="")]])] *
1021 t(gradients[[paste('dw',L,sep="")]])]
1022     s[[paste("db",L, sep="")]] = beta2*s[[paste("db",L, sep="")]] +
1023         (1-beta2) * t(gradients[[paste('db',L,sep="")]])] *
1024 t(gradients[[paste('db',L,sep="")]])]
1025
1026     # Compute bias-corrected second moment estimate.
1027     s_corrected[[paste("dw",L, sep="")]] = s[[paste("dw",L, sep="")]]/(1-
1028 beta2^t)
1029     s_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-
1030 beta2^t)
1031
1032     # Update parameters.
1033     d1=sqrt(s_corrected[[paste("dw",L, sep="")]]+epsilon)
1034     d2=sqrt(s_corrected[[paste("db",L, sep="")]]+epsilon)
1035
1036     parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
1037     -
1038         learningRate * v_corrected[[paste("dw",L, sep="")]]/d1
1039     parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
1040     -
1041         learningRate*v_corrected[[paste("db",L, sep="")]]/d2
1042     }
1043     return(parameters)
1044 }
1045
1046 # Execute a L layer Deep learning model
1047 # Input : X - Input features
1048 #           : Y output
1049 #           : layersDimensions - Dimension of layers
1050 #           : hiddenActivationFunc - Activation function at hidden layer relu
1051 /tanh
1052 #           : outputActivationFunc - Activation function at hidden layer
1053 sigmoid/softmax
1054 #           : learning rate
1055 #           : lambd
1056 #           : keep_prob
1057 #           : learning rate
1058 #           : num of iterations
1059 #           : initType

```

```

1060 #output : Updated weights
1061 L_Layer_DeepModel <- function(X, Y, layersDimensions,
1062                               hiddenActivationFunc='relu',
1063                               outputActivationFunc= 'sigmoid',
1064                               learningRate = 0.5,
1065                               lambd=0,
1066                               keep_prob=1,
1067                               numIterations = 10000,
1068                               initType="default",
1069                               print_cost=False){
1070     #Initialize costs vector as NULL
1071     costs <- NULL
1072
1073     # Parameters initialization.
1074     if (initType=="He"){
1075         parameters =HeInitializeDeepModel(layersDimensions)
1076     } else if (initType=="Xav"){
1077         parameters =XavInitializeDeepModel(layersDimensions)
1078     }
1079     else{
1080         parameters = initializeDeepModel(layersDimensions)
1081     }
1082
1083
1084     # Loop (gradient descent)
1085     for( i in 0:numIterations){
1086         # Forward propagation: [LINEAR -> RELU]^(L-1) -> LINEAR ->
1087         SIGMOID/SOFTMAX.
1088         retvals = forwardPropagationDeep(X, parameters,keep_prob,
1089                                         hiddenActivationFunc,
1090                                         outputActivationFunc=outputActivationFunc)
1091         AL <- retvals[['AL']]
1092         caches <- retvals[['caches']]
1093         dropoutMat <- retvals[['dropoutMat']]
1094
1095         # Compute cost.
1096         if(lambd==0){
1097             cost <- computeCost(AL,
1098                               Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
1099                               h(layersDimensions))])
1100         } else {
1101             cost <- computeCostWithReg(parameters, AL, Y,lambd,
1102                                         outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
1103                                         layersDimensions)])
1104         }
1105         # Backward propagation.
1106         gradients = backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd,
1107                                         keep_prob, hiddenActivationFunc,
1108                                         outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
1109                                         layersDimensions)])
1110
1111         # Update parameters.
1112         parameters = gradientDescent(parameters, gradients, learningRate,
1113                                         outputActivationFunc=outputActivationFunc)
1114
1115         if(i%%1000 == 0){
1116             costs=c(costs,cost)
1117             print(cost)
1118         }
1119     }
1120 }
```

```

1124     retvals <- list("parameters"=parameters,"costs"=costs)
1125
1126     return(retvals)
1127 }
1128
1129 # Execute a L layer Deep learning model with stochastic Gradient descent
1130 # Input : X - Input features
1131 #
1132 #           : Y output
1133 #           : layersDimensions - Dimension of layers
1134 #           : hiddenActivationFunc - Activation function at hidden layer relu
1135 #           : outputActivationFunc - Activation function at hidden layer
1136 #           : sigmoid/softmax
1137 #           : learning rate
1138 #           : lrDecay
1139 #           : decayRate
1140 #           : lambd
1141 #           : keep_prob
1142 #           : optimizer
1143 #           : beta
1144 #           : beta1
1145 #           : beta2
1146 #           : epsilon
1147 #           : mini_batch_size
1148 #           : num of epochs
1149 #output : Updated weights after each iteration
1150 L_Layer_DeepModel_SGD <- function(X, Y, layersDimensions,
1151                                     hiddenActivationFunc='relu',
1152                                     outputActivationFunc= 'sigmoid',
1153                                     learningRate = .3,
1154                                     lrDecay=FALSE,
1155                                     decayRate=1,
1156                                     lambd=0,
1157                                     keep_prob=1,
1158                                     optimizer="gd",
1159                                     beta=0.9,
1160                                     beta1=0.9,
1161                                     beta2=0.999,
1162                                     epsilon=10^-8,
1163                                     mini_batch_size = 64,
1164                                     num_epochs = 2500,
1165                                     print_cost=False){
1166
1167     # Check the values
1168     cat("learningRate= ",learningRate)
1169     cat("\n")
1170     cat("lambd=",lambd)
1171     cat("\n")
1172     cat("keep_prob=",keep_prob)
1173     cat("\n")
1174     cat("optimizer=",optimizer)
1175     cat("\n")
1176     cat("lrDecay=",lrDecay)
1177     cat("\n")
1178     cat("decayRate=",decayRate)
1179     cat("\n")
1180     cat("beta=",beta)
1181     cat("\n")
1182     cat("beta1=",beta1)
1183     cat("\n")
1184     cat("beta2=",beta2)
1185     cat("\n")
1186     cat("epsilon=",epsilon)
1187     cat("\n")
1188     cat("mini_batch_size=",mini_batch_size)

```

```

1188     cat("\n")
1189     cat("num_epochs=", num_epochs)
1190     cat("\n")
1191     set.seed(1)
1192     #Initialize costs vector as NULL
1193     costs <- NULL
1194     t <- 0
1195     # Parameters initialization.
1196     parameters = initializeDeepModel(layersDimensions)
1197
1198     #Initialize the optimizer
1199
1200     if(optimizer == "momentum"){
1201         v <- initializeVelocity(parameters)
1202     } else if(optimizer == "rmsprop"){
1203         s <- initializeRMSProp(parameters)
1204     } else if (optimizer == "adam"){
1205         adamVals <- initializeAdam(parameters)
1206     }
1207
1208     seed=10
1209
1210     # Loop for number of epochs
1211     for( i in 0:num_epochs){
1212         seed=seed+1
1213         minibatches = random_mini_batches(X, Y, mini_batch_size, seed)
1214
1215         for(batch in 1:length(minibatches)){
1216
1217             mini_batch_X=minibatches[[batch]][['mini_batch_X']]
1218             mini_batch_Y=minibatches[[batch]][['mini_batch_Y']]
1219             # Forward propagation:
1220             retvals = forwardPropagationDeep(mini_batch_X,
1221 parameters,keep_prob, hiddenActivationFunc,
1222
1223             outputActivationFunc=outputActivationFunc)
1224             AL <- retvals[['AL']]
1225             caches <- retvals[['caches']]
1226             dropoutMat <- retvals[['dropoutMat']]
1227
1228             # Compute cost.
1229             # Compute cost.
1230             if(lambd==0){
1231                 cost <- computeCost(AL,
1232             mini_batch_Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)])
1233             } else {
1234                 cost <- computeCostWithReg(parameters, AL, Y, lambd,
1235             outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)])
1236             }
1237             # Backward propagation.
1238             gradients = backwardPropagationDeep(AL, mini_batch_Y, caches,
1239             dropoutMat, lambd, keep_prob, hiddenActivationFunc,
1240
1241             outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)])
1242
1243             if(optimizer == "gd"){
1244                 # Update parameters.
1245                 parameters = gradientDescent(parameters, gradients,
1246             learningRate,
1247
1248             outputActivationFunc=outputActivationFunc)

```

```

1252         }else if(optimizer == "momentum"){
1253             # Update parameters with Momentum
1254             parameters = gradientDescentWithMomentum(parameters,
1255 gradients,v,beta, learningRate,
1256
1257             outputActivationFunc=outputActivationFunc)
1258
1259         } else if(optimizer == "rmsprop"){
1260             # Update parameters with RMSProp
1261             parameters = gradientDescentWithRMSProp(parameters,
1262 gradients,s,beta1, epsilon,learningRate,
1263
1264             outputActivationFunc=outputActivationFunc)
1265
1266         } else if(optimizer == "adam"){
1267             # Update parameters with Adam
1268             #Get v and s
1269             t <- t+1
1270             v <- adamVals[['v']]
1271             s <- adamVals[['s']]
1272             parameters = gradientDescentWithAdam(parameters, gradients,v,
1273 s,t, beta1,beta2, epsilon,learningRate,
1274
1275             outputActivationFunc=outputActivationFunc)
1276             }
1277         }
1278
1279         if(i%1000 == 0){
1280             costs=c(costs,cost)
1281             print(cost)
1282         }
1283         if(!rDecay==TRUE){
1284             learningRate = decayRate^(num_epochs/1000) * learningRate
1285         }
1286     }
1287
1288     retvals <- list("parameters"=parameters,"costs"=costs)
1289
1290     return(retvals)
1291 }
1292
1293 # Predict the output for given input
1294 # Input : parameters
1295 #           : X
1296 # Output: predictions
1297 predict <- function(parameters, X,keep_prob=1, hiddenActivationFunc='relu'){
1298
1299     fwdProp <- forwardPropagationDeep(X, parameters,keep_prob,
1300 hiddenActivationFunc)
1301     predictions <- fwdProp$AL>0.5
1302
1303     return (predictions)
1304 }
1305
1306 # Plot a decision boundary
1307 # This function uses ggplot2
1308 plotDecisionBoundary <-
1309 function(z,retvals,keep_prob=1,hiddenActivationFunc="sigmoid",lr=0.5){
1310     # Find the minimum and maximum for the data
1311     xmin<-min(z[,1])
1312     xmax<-max(z[,1])
1313     ymin<-min(z[,2])
1314     ymax<-max(z[,2])
1315

```

```

1316 # Create a grid of values
1317 a=seq(xmin,xmax,length=100)
1318 b=seq(ymax,ymin,length=100)
1319 grid <- expand.grid(x=a, y=b)
1320 colnames(grid) <- c('x1', 'x2')
1321 grid1 <-t(grid)
1322 # Predict the output for this grid
1323 q <-predict(retvals$parameters,grid1,keep_prob=1, hiddenActivationFunc)
1324 q1 <- t(data.frame(q))
1325 q2 <- as.numeric(q1)
1326 grid2 <- cbind(grid,q2)
1327 colnames(grid2) <- c('x1', 'x2','q2')
1328
1329 z1 <- data.frame(z)
1330 names(z1) <- c("x1","x2","y")
1331 atitle=paste("Decision boundary for learning rate:",lr)
1332 # Plot the contour of the boundary
1333 ggplot(z1) +
1334   geom_point(data = z1, aes(x = x1, y = x2, color = y)) +
1335   stat_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2), alpha =
1336 0.9)+ 
1337   ggtitle(atitle) + scale_colour_gradientn(colours = brewer.pal(10,
1338 "Spectral"))
1339 }
1340
1341 # Predict the probability scores for given data set
1342 # Input : parameters
1343 #           : X
1344 # Output: probability of output
1345 computeScores <- function(parameters, x,hiddenActivationFunc='relu'){
1346
1347 fwdProp <- forwardPropagationDeep(x, parameters,hiddenActivationFunc)
1348 scores <- fwdProp$AL
1349
1350 return (scores)
1351 }
1352
1353 # Create random mini batches
1354 # Input : X - Input features
1355 #           : Y- output
1356 #           : miniBatchsize
1357 #           : seed
1358 #output : mini_batches
1359 random_mini_batches <- function(X, Y, miniBatchsize = 64, seed = 0){
1360
1361
1362   set.seed(seed)
1363   # Get number of training samples
1364   m = dim(X)[2]
1365   # Initialize mini batches
1366   mini_batches = list()
1367
1368   # Create a list of random numbers < m
1369   permutation = c(sample(m))
1370   # Randomly shuffle the training data
1371   shuffled_X = X[, permutation]
1372   shuffled_Y = Y[1, permutation]
1373
1374   # Compute number of mini batches
1375   numCompleteMinibatches = floor(m/miniBatchsize)
1376   batch=0
1377   for(k in 0:(numCompleteMinibatches-1)){
1378     batch=batch+1
1379     # Set the lower and upper bound of the mini batches

```

```

1380     lower=(k*miniBatchSize)+1
1381     upper=((k+1) * miniBatchSize)
1382     mini_batch_X = shuffled_X[, lower:upper]
1383     mini_batch_Y = shuffled_Y[lower:upper]
1384     # Add it to the list of mini batches
1385     mini_batch =
1386     list("mini_batch_X"=mini_batch_X,"mini_batch_Y"=mini_batch_Y)
1387     mini_batches[[batch]] =mini_batch
1388
1389
1390 }
1391
1392 # If the batch size does not divide evenly with mini batch size
1393 if(m %% miniBatchSize != 0){
1394     p=floor(m/miniBatchSize)*miniBatchSize
1395     # Set the start and end of last batch
1396     q=p+m %% miniBatchSize
1397     mini_batch_X = shuffled_X[, (p+1):q]
1398     mini_batch_Y = shuffled_Y[(p+1):q]
1399 }
1400 # Return the list of mini batches
1401 mini_batch =
1402 list("mini_batch_X"=mini_batch_X,"mini_batch_Y"=mini_batch_Y)
1403 mini_batches[[batch]]=mini_batch
1404
1405 return(mini_batches)
1406 }
1407
1408 # Plot a decision boundary
1409 # This function uses ggplot2
1410 plotDecisionBoundary1 <- function(z,parameters,keep_prob=1){
1411     xmin<-min(z[,1])
1412     xmax<-max(z[,1])
1413     ymin<-min(z[,2])
1414     ymax<-max(z[,2])
1415
1416     # Create a grid of points
1417     a=seq(xmin,xmax,length=100)
1418     b=seq(ymin,ymax,length=100)
1419     grid <- expand.grid(x=a, y=b)
1420     colnames(grid) <- c('x1', 'x2')
1421     grid1 <-t(grid)
1422
1423     retvals = forwardPropagationDeep(grid1, parameters,keep_prob, "relu",
1424                                         outputActivationFunc="softmax")
1425
1426
1427     AL <- retvals$AL
1428     # From the softmax probabilities pick the one with the highest
1429     probability
1430     q= apply(AL,1,which.max)
1431
1432     q1 <- t(data.frame(q))
1433     q2 <- as.numeric(q1)
1434     grid2 <- cbind(grid,q2)
1435     colnames(grid2) <- c('x1', 'x2','q2')
1436
1437     z1 <- data.frame(z)
1438     names(z1) <- c("x1", "x2","y")
1439     atitle=paste("Decision boundary")
1440     ggplot(z1) +
1441         geom_point(data = z1, aes(x = x1, y = x2, color = y)) +
1442         stat_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2),
alpha = 0.9)+
```

```

1444     ggtile(atitle) + scale_colour_gradientn(colours = brewer.pal(10,
1445 "Spectral"))
1446 }

```

7.3 Octave

```

1 ##########
2 #####
3 #
4 # File   : DLfunctions7.R
5 # Author : Tinniam V Ganesh
6 # Date   : 16 Apr 2018
7 #
8 #####
9 #####
10 1;
11 # Define sigmoid function
12 function [A,cache] = sigmoid(z)
13     A = 1 ./ (1+ exp(-z));
14     cache=z;
15 end
16
17 # Define Relu function
18 function [A,cache] = relu(z)
19     A = max(0,Z);
20     cache=Z;
21 end
22
23 # Define Relu function
24 function [A,cache] = tanhAct(z)
25     A = tanh(Z);
26     cache=Z;
27 end
28
29 # Define Softmax function
30 function [A,cache] = softmax(z)
31     # get unnormalized probabilities
32     exp_scores = exp(z');
33     # normalize them for each example
34     A = exp_scores ./ sum(exp_scores,2);
35     cache=Z;
36 end
37
38 # Define Stable Softmax function
39 function [A,cache] = stableSoftmax(z)
40     # Normalize by max value in each row
41     shiftz = z' - max(z',[],2);
42     exp_scores = exp(shiftz);
43     # normalize them for each example
44     A = exp_scores ./ sum(exp_scores,2);
45     #disp("sm")
46     #disp(A);
47     cache=Z;
48 end
49
50 # Define Relu Derivative
51 function [dz] = reluDerivative(dA,cache)
52     Z = cache;
53     dz = dA;
54     # Get elements that are greater than 0

```

```

55     a = (z > 0);
56     # Select only those elements where z > 0
57     dz = dz .* a;
58 end
59
60 # Define Sigmoid Derivative
61 function [dZ] = sigmoidDerivative(dA,cache)
62     Z = cache;
63     s = 1 ./ (1+ exp(-z));
64     dz = dA .* s .* (1-s);
65 end
66
67 # Define Tanh Derivative
68 function [dZ] = tanhDerivative(dA,cache)
69     Z = cache;
70     a = tanh(z);
71     dz = dA .* (1 - a .^ 2);
72 end
73
74 # Populate a matrix with 1s in rows where Y=1
75 # This function may need to be modified if K is not 3, 10
76 # This function is used in computing the softmax derivative
77 function [Y1] = popMatrix(Y,numClasses)
78     Y1=zeros(length(Y),numClasses);
79     if(numClasses==3) # For 3 output classes
80         Y1(Y==0,1)=1;
81         Y1(Y==1,2)=1;
82         Y1(Y==2,3)=1;
83     elseif(numClasses==10) # For 10 output classes
84         Y1(Y==0,1)=1;
85         Y1(Y==1,2)=1;
86         Y1(Y==2,3)=1;
87         Y1(Y==3,4)=1;
88         Y1(Y==4,5)=1;
89         Y1(Y==5,6)=1;
90         Y1(Y==6,7)=1;
91         Y1(Y==7,8)=1;
92         Y1(Y==8,9)=1;
93         Y1(Y==9,10)=1;
94     endif
95 end
96
97 # Define Softmax Derivative
98 function [dZ] = softmaxDerivative(dA,cache,Y, numclasses)
99     Z = cache;
100    # get unnormalized probabilities
101    shiftZ = Z' - max(Z',[],2);
102    exp_scores = exp(shiftZ);
103
104    # normalize them for each example
105    probs = exp_scores ./ sum(exp_scores,2);
106    # dz = pi - yi
107    yi=popMatrix(Y,numClasses);
108    dz=probs-yi;
109
110 end
111
112 # Define Stable Softmax Derivative
113 function [dZ] = stableSoftmaxDerivative(dA,cache,Y, numclasses)
114     Z = cache;
115     # get unnormalized probabilities
116     exp_scores = exp(Z');
117     # normalize them for each example

```

```

119  probs = exp_scores ./ sum(exp_scores,2);
120  # dZ = pi - yi
121  yi=popMatrix(Y,numClasses);
122  dZ=probs-yi;
123
124 end
125
126 # Initialize model for L layers
127 # Input : Vector of units in each layer
128 # Returns: Initial weights and biases matrices for all layers
129 function [w b] = initializeDeepModel(layerDimensions)
130    rand ("seed", 3);
131    # note the weight matrix at layer '1' is a matrix of size (1,1-1)
132    # The Bias is a vectors of size (1,1)
133
134    # Loop through the layer dimension from 1.. L
135    # Create cell arrays for weights and biases
136
137    for l =2:size(layerDimensions)(2)
138        w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*0.01; #
139        Multiply by .01
140        b{l-1} = zeros(layerDimensions(l),1);
141
142    endfor
143 end
144
145 # He Initialization for L layers
146 # Input : Vector of units in each layer
147 # Returns: Initial weights and biases matrices for all layers
148 function [w b] = HeInitializeDeepModel(layerDimensions)
149    rand ("seed", 3);
150    # note the weight matrix at layer '1' is a matrix of size (1,1-1)
151    # The Bias is a vectors of size (1,1)
152
153    # Loop through the layer dimension from 1.. L
154    # Create cell arrays for weights and biases
155
156    for l =2:size(layerDimensions)(2)
157        w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*#
158        sqrt(2/layerDimensions(l-1)); # Multiply by .01
159        b{l-1} = zeros(layerDimensions(l),1);
160
161    endfor
162 end
163
164 # Xavier Initialization for L layers
165 # Input : Vector of units in each layer
166 # Returns: Initial weights and biases matrices for all layers
167 function [w b] = XavInitializeDeepModel(layerDimensions)
168    rand ("seed", 3);
169    # note the weight matrix at layer '1' is a matrix of size (1,1-1)
170    # The Bias is a vectors of size (1,1)
171
172    # Loop through the layer dimension from 1.. L
173    # Create cell arrays for weights and biases
174
175    for l =2:size(layerDimensions)(2)
176        w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*#
177        sqrt(1/layerDimensions(l-1)); # Multiply by .01
178        b{l-1} = zeros(layerDimensions(l),1);
179
180    endfor
181 end
182

```

```

183 # Initialize velocity
184 # Input : weights, biases
185 # Returns: vdw, vdB - Initial velocity
186 function[vdw vdB] = initializeVelocity(weights, biases)
187
188     L = size(weights)(2) # Create an integer
189     # Initialize a cell array
190     v = {}
191
192     # Initialize velocity with the same dimensions as w
193     for l=1:L
194         sz = size(weights{l});
195         vdw{l} = zeros(sz(1),sz(2));
196         sz = size(biases{l});
197         vdB{l} =zeros(sz(1),sz(2));
198     endfor;
199 end
200
201 # Initialize RMSProp
202 # Input : weights, biases
203 # Returns: sdw, sdb - Initial RMSProp
204 function[sdw sdb] = initializeRMSProp(weights, biases)
205
206     L = size(weights)(2) # Create an integer
207     # Initialize a cell array
208     s = {}
209
210     # Initialize velocity with the same dimensions as w
211     for l=1:L
212         sz = size(weights{l});
213         sdw{l} = zeros(sz(1),sz(2));
214         sz = size(biases{l});
215         sdb{l} =zeros(sz(1),sz(2));
216     endfor;
217 end
218
219 # Initialize Adam
220 # Input : parameters
221 # Returns: vdw, vdB, sdw, sdb -Initial Adam
222 function[vdw vdB sdw sdb] = initializeAdam(weights, biases)
223
224     L = size(weights)(2) # Create an integer
225     # Initialize a cell array
226     s = {}
227
228     # Initialize velocity with the same dimensions as w
229     for l=1:L
230         sz = size(weights{l});
231         vdw{l} = zeros(sz(1),sz(2));
232         sdw{l} = zeros(sz(1),sz(2));
233         sz = size(biases{l});
234         sdb{l} =zeros(sz(1),sz(2));
235         vdB{l} =zeros(sz(1),sz(2));
236     endfor;
237 end
238
239 # Compute the activation at a layer 'l' for forward prop in a Deep Network
240 # Input : A_prev - Activation of previous layer
241 #          W,b - Weight and bias matrices and vectors
242 #          activationFunc - Activation function - sigmoid, tanh, relu etc
243 # Returns : The Activation of this layer
244 #          :
245 #          Z = W * X + b
246 #          A = sigmoid(Z), A= Relu(Z), A= tanh(Z)

```

```

247 function [A forward_cache activation_cache] = layerActivationForward(A_prev,
248 w, b, activationFunc)
249
250     # Compute Z
251     Z = w * A_prev + b;
252     # Create a cell array
253     forward_cache = {A_prev w b};
254     # Compute the activation for sigmoid
255     if (strcmp(activationFunc, "sigmoid"))
256         [A activation_cache] = sigmoid(Z);
257     elseif (strcmp(activationFunc, "relu")) # Compute the activation for
258     Relu
259         [A activation_cache] = relu(Z);
260     elseif(strcmp(activationFunc, 'tanh')) # Compute the activation for
261     tanh
262         [A activation_cache] = tanhAct(Z);
263     elseif(strcmp(activationFunc, 'softmax')) # Compute the activation for
264     tanh
265         #[A activation_cache] = softmax(Z);
266         [A activation_cache] = stableSoftmax(Z);
267     endif
268
269 end
270
271 # Compute the forward propagation for layers 1..L
272 # Input : X - Input Features
273 #           parameters: Weights and biases
274 #           keep_prob
275 #           hiddenActivationFunc - Activation function at hidden layers
276 Relu/tanh/sigmoid
277 #           outputActivationFunc- sigmoid/softmax
278 # Returns : AL, forward_caches, activation_caches, dropoutMat
279 # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and
280 sigmoid actiovation at layer L
281 function [AL forward_caches activation_caches dropoutMat] =
282 forwardPropagationDeep(X, weights,biases, keep_prob=1,
283                                     hiddenActivationFunc='relu',
284 outputActivationFunc='sigmoid')
285     # Create an empty cell array
286     forward_caches = {};
287     activation_caches = {};
288     dropoutMat = {};
289     # Set A to X (A0)
290     A = X;
291     L = length(weights); # number of layers in the neural network
292     # Loop through from layer 1 to upto layer L
293     for l =1:L-1
294         A_prev = A;
295         # Zi = Wi x Ai-1 + bi and Ai = g(Zi)
296         w = weights{l};
297         b = biases{l};
298         [A forward_cache activation_cache] = layerActivationForward(A_prev,
299 w,b, activationFunc=hiddenActivationFunc);
300         D=rand(size(A)(1),size(A)(2));
301         D = (D < keep_prob) ;
302         # Multiply by DropoutMat
303         A=A.*D;
304         # Divide by keep_prob to keep expected value same
305         A = A ./ keep_prob;
306         # Store D
307         dropoutMat{l}=D;
308         forward_caches{l}=forward_cache;
309         activation_caches{l} = activation_cache;
310     endfor

```

```

311     # Since this is binary classification use the sigmoid activation function
312     in
313         # last layer
314         w = weights{L};
315         b = biases{L};
316         [AL, forward_cache activation_cache] = layerActivationForward(A, w, b,
317 activationFunc = outputActivationFunc);
318         forward_caches{L}=forward_cache;
319         activation_caches{L} = activation_cache;
320
321     end
322
323     # Pick columns where Y==1
324     # This function is used in computeCost
325     function [a] = pickColumns(AL,Y,numClasses)
326         if(numClasses==3)
327             a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)];
328         elseif (numClasses==10)
329             a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3);AL(Y==3,4);AL(Y==4,5);
330                 AL(Y==5,6); AL(Y==6,7);AL(Y==7,8);AL(Y==8,9);AL(Y==9,10)];
331         endif
332     end
333
334
335     # Compute the cost
336     # Input : AL-Activation of last layer
337     #          : Y-Output from data
338     #          : outputActivationFunc- sigmoid/softmax
339     #          : numClasses
340     # Output: cost
341     function [cost]= computeCost(AL, Y,
342 outputActivationFunc="sigmoid",numClasses)
343         if(strcmp(outputActivationFunc,"sigmoid"))
344             numTraining= size(Y)(2);
345             # Element wise multiply for logprobs
346             cost = -1/numTraining * sum((Y .* log(AL)) + (1-Y) .* log(1-AL));
347
348
349         elseif(strcmp(outputActivationFunc,'softmax'))
350             numTraining = size(Y)(2);
351             Y=Y';
352             # Select rows where Y=0,1, and 2 and concatenate to a long vector
353             #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)];
354             a =pickColumns(AL,Y,numClasses);
355
356             #Select the correct column for log prob
357             correct_probs = -log(a);
358             #Compute log loss
359             cost= sum(correct_probs)/numTraining;
360         endif
361     end
362
363     # Compute the cost with regularization
364     # Input : weights
365     #          : AL - Activation of last layer
366     #          : Output from data
367     #          : lambd
368     #          : outputActivationFunc- sigmoid/softmax
369     #          : numClasses
370     # Output: cost
371     function [cost]= computeCostwithReg(weights, AL, Y, lambd,
372 outputActivationFunc="sigmoid",numClasses)
373
374         if(strcmp(outputActivationFunc,"sigmoid"))

```

```

375     numTraining= size(Y)(2);
376     # Element wise multiply for logprobs
377     cost = -1/numTraining * sum((Y .* log(AL)) + (1-Y) .* log(1-AL));
378
379     # Regularization cost
380     L = size(weights)(2);
381     L2RegularizationCost=0;
382     for l=1:L
383         wtSqr = weights{l} .* weights{l};
384         #disp(sum(sum(wtSqr,1)));
385         L2RegularizationCost+=sum(sum(wtSqr,1));
386     endfor
387     L2RegularizationCost = (lambd/(2*numTraining))*L2RegularizationCost;
388     cost = cost + L2RegularizationCost ;
389
390
391 elseif(strcmp(outputActivationFunc, 'softmax'))
392     numTraining = size(Y)(2);
393     Y=Y';
394     # Select rows where Y=0,1, and 2 and concatenate to a long vector
395     #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)];
396     a =pickColumns(AL,Y,numClasses);
397
398     #Select the correct column for log prob
399     correct_probs = -log(a);
400     #Compute log loss
401     cost= sum(correct_probs)/numTraining;
402         # Regularization cost
403     L = size(weights)(2);
404     L2RegularizationCost=0;
405     for l=1:L
406         # Compute L2 Norm
407         wtSqr = weights{l} .* weights{l};
408         #disp(sum(sum(wtSqr,1)));
409         L2RegularizationCost+=sum(sum(wtSqr,1));
410     endfor
411     L2RegularizationCost = (lambd/(2*numTraining))*L2RegularizationCost;
412     cost = cost + L2RegularizationCost ;
413 endif
414 end
415
416
417
418 # Compute the backpropagation for 1 cycle
419 # Input : dA- Neural Network parameters
420 #          # cache - forward_cache & activation_cache
421 #          # Y-Output values
422 #          # outputActivationFunc- sigmoid/softmax
423 #          # numClasses
424 # Returns: Gradients
425 # dL/dwi= dL/dzi*A_l-1
426 # dL/dbl = dL/dz_l
427 # dL/dz_prev=dL/dz_l*w
428 function [dA_prev dw db] = layerActivationBackward(dA, forward_cache,
429 activation_cache, Y, activationFunc,numClasses)
430
431     A_prev = forward_cache{1};
432     w =forward_cache{2};
433     b = forward_cache{3};
434     numTraining = size(A_prev)(2);
435     if (strcmp(activationFunc,"relu"))
436         dz = reluDerivative(dA, activation_cache);
437     elseif (strcmp(activationFunc,"sigmoid"))
438         dz = sigmoidDerivative(dA, activation_cache);

```

```

439     elseif(strcmp(activationFunc, "tanh"))
440         dZ = tanhDerivative(dA, activation_cache);
441     elseif(strcmp(activationFunc, "softmax"))
442         #dZ = softmaxDerivative(dA, activation_cache, Y, numClasses);
443         dZ = stableSoftmaxDerivative(dA, activation_cache, Y, numClasses);
444     endif
445
446
447     if (strcmp(activationFunc, "softmax"))
448         w = forward_cache{2};
449         b = forward_cache{3};
450         # Add the regularization factor
451         dw = 1/numTraining * A_prev * dZ;
452         db = 1/numTraining * sum(dZ,1);
453         dA_prev = dZ*w;
454     else
455         w = forward_cache{2};
456         b = forward_cache{3};
457         # Add the regularization factor
458         dw = 1/numTraining * dZ * A_prev';
459         db = 1/numTraining * sum(dZ,2);
460         dA_prev = w'*dZ;
461     endif
462
463 end
464
465 # Compute the backpropagation with regularization for 1 cycle
466 # Input : dA-Neural Network parameters
467 #          # cache - forward_cache & activation_cache
468 #          # Y-Output values
469 #          # lambd
470 #          # outputActivationFunc- sigmoid/softmax
471 #          # numClasses
472 # Returns: Gradients
473 # dL/dwi= dL/dzi*A1-1
474 # dL/db1 = dL/dZ1
475 # dL/dz_prev=dL/dZ1*w
476 function [dA_prev dw db] = layerActivationBackwardWithReg(dA, forward_cache,
477 activation_cache, Y, lambd=0, activationFunc,numClasses)
478
479     A_prev = forward_cache{1};
480     w = forward_cache{2};
481     b = forward_cache{3};
482     numTraining = size(A_prev)(2);
483     if (strcmp(activationFunc, "relu"))
484         dZ = reluDerivative(dA, activation_cache);
485     elseif (strcmp(activationFunc, "sigmoid"))
486         dZ = sigmoidDerivative(dA, activation_cache);
487     elseif(strcmp(activationFunc, "tanh"))
488         dZ = tanhDerivative(dA, activation_cache);
489     elseif(strcmp(activationFunc, "softmax"))
490         #dZ = softmaxDerivative(dA, activation_cache, Y, numClasses);
491         dZ = stableSoftmaxDerivative(dA, activation_cache, Y, numClasses);
492     endif
493
494     if (strcmp(activationFunc, "softmax"))
495         w = forward_cache{2};
496         b = forward_cache{3};
497         # Add the regularization factor
498         dw = 1/numTraining * A_prev * dZ + (lambd/numTraining) * w';
499         db = 1/numTraining * sum(dZ,1);
500         dA_prev = dZ*w;
501     else
502         w = forward_cache{2};

```

```

503     b = forward_cache{3};
504     # Add the regularization factor
505     dw = 1/numTraining * dz * A_prev' + (lambd/numTraining) * w;
506     db = 1/numTraining * sum(dz,2);
507     dA_prev = w'*dz;
508   endif
509
510 end
511
512
513 # Compute the backpropagation for 1 cycle
514 # Input : AL: Output of L layer Network - weights
515 #           Y Real output
516 #           caches -- list of caches containing:
517 #           every cache of layerActivationForward() with "relu"/"tanh"
518 #           #(it's caches[1], for l in range(L-1) i.e l = 0...L-2)
519 #           #the cache of layerActivationForward() with "sigmoid" (it's caches[L-
520 1])
521 #           dropoutMat
522 #           lambd
523 #           keep_prob
524 #           hiddenActivationFunc - Activation function at hidden layers
525 sigmoid/tanh/relu
526 #           outputActivationFunc- sigmoid/softmax
527 #           numClasses
528 #
529 # Returns:
530 #   gradients -- A dictionary with the gradients
531 #                 gradients["dA" + str(l)] = ...
532 #                 gradients["dw" + str(l)] = ...
533 #                 gradients["db" + str(l)] = ...
534 function [gradsDA gradsDW gradsDB]= backwardPropagationDeep(AL, Y,
535 activation_caches,forward_caches,
536                         dropoutMat, lambd=0, keep_prob=1,
537 hiddenActivationFunc='relu',outputActivationFunc="sigmoid",numClasses)
538
539
540 # Set the number of layers
541 L = length(activation_caches);
542 m = size(AL)(2);
543
544 if (strcmp(outputActivationFunc,"sigmoid"))
545   # Initializing the backpropagation
546   # d1/dAL= -(y/a + (1-y)/(1-a)) - At the output layer
547   dAL = -(Y ./ AL) - (1 - Y) ./ (1 - AL);
548 elseif (strcmp(outputActivationFunc,"softmax"))
549   dAL=0;
550   Y=Y';
551 endif
552
553
554 # Since this is a binary classification the activation at output is
555 sigmoid
556   # Get the gradients at the last layer
557   # Inputs: "AL, Y, caches".
558   # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dB"]
559   activation_cache = activation_caches{L};
560   forward_cache = forward_caches(L);
561   # Note the cell array includes an array of forward caches. To get to this
562 we need to include the index {1}
563   if (lambd==0)
564     [dA dw db] = layerActivationBackward(dAL, forward_cache{1},
565 activation_cache, Y, activationFunc = outputActivationFunc,numClasses);
566 else

```

```

567     [dA dw db] = layerActivationBackwardWithReg(dAL, forward_cache{1},
568 activation_cache, Y, lambd, activationFunc =
569 outputActivationFunc,numClasses);
570     endif
571     if (strcmp(outputActivationFunc,"sigmoid"))
572         gradsDA{L}= dA;
573     elseif (strcmp(outputActivationFunc,"softmax"))
574         gradsDA{L}= dA';#Note the transpose
575     endif
576     gradsDW{L}= dw;
577     gradsDB{L}= db;
578
579     # Traverse in the reverse direction
580     for l =(L-1):-1:1
581         # Compute the gradients for L-1 to 1 for Relu/tanh
582         # Inputs: "gradients["dA" + str(l + 2)], caches".
583         # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l +
584 1)] , gradients["db" + str(l + 1)]
585         activation_cache = activation_caches{l};
586         forward_cache = forward_caches(l);
587
588         #dA_prev_temp, dw_temp, db_temp =
589         layerActivationBackward(gradients['dA'+str(l+1)], current_cache,
590 activationFunc = "relu")
591         # dAl the derivative of the activation of the lth layer,is the first
592 element
593         dAl= gradsDA{l+1};
594         if(lambd == 0)
595             # Get the dropout mat
596             D = dropoutMat{l};
597             #Multiply by the dropoutMat
598             dAl= dAl .* D;
599             # Divide by keep_prob to keep expected value same
600             dAl = dAl ./ keep_prob;
601             [dA_prev_temp, dw_temp, db_temp] = layerActivationBackward(dAl,
602 forward_cache{1}, activation_cache, Y, activationFunc =
603 hiddenActivationFunc,numClasses);
604         else
605             [dA_prev_temp, dw_temp, db_temp] =
606             layerActivationBackwardWithReg(dAl, forward_cache{1}, activation_cache, Y,
607 lambd, activationFunc = hiddenActivationFunc,numClasses);
608         endif
609         gradsDA{1}= dA_prev_temp;
610         gradsDW{1}= dw_temp;
611         gradsDB{1}= db_temp;
612
613     endfor
614
615 end
616
617
618 # Perform Gradient Descent
619 # Input : weights and biases
620 #           : gradients -gradsW,gradsB
621 #           : Learning rate
622 #           : outputActivationFunc
623 #output : Updated weights after 1 iteration
624 function [weights biases] = gradientDescent(weights, biases,gradsW,gradsB,
625 learningRate,outputActivationFunc="sigmoid")
626
627 L = size(weights)(2); # number of layers in the neural network
628 # Update rule for each parameter.
629 for l=1:(L-1)
630     weights{l} = weights{l} -learningRate* gradsW{l};

```

```

631     biases{1} = biases{1} - learningRate* gradsB{1};
632 endfor
633
634 if (strcmp(outputActivationFunc,"sigmoid"))
635     weights{L} = weights{L} - learningRate* gradsW{L};
636     biases{L} = biases{L} - learningRate* gradsB{L};
637 elseif (strcmp(outputActivationFunc,"softmax"))
638     weights{L} = weights{L} - learningRate* gradsW{L}';
639     biases{L} = biases{L} - learningRate* gradsB{L}';
640 endif
641
642
643
644 end
645
646
647 # Update parameters with momentum
648 # Input : parameters
649 #      : gradients -gradsDW,gradsDB
650 #      : v -vdW, vdB
651 #      : beta
652 #      : learningRate
653 #      : outputActivationFunc
654 #output : Updated weights, biases
655 function [weights biases] = gradientDescentWithMomentum(weights,
656 biases,gradsDW,gradsDB, vdW, vdB, beta,
657 learningRate,outputActivationFunc="sigmoid")
658 L = size(weights)(2); # number of layers in the neural network
659 # Update rule for each parameter.
660 for l=1:(L-1)
661     # Compute velocities
662     # v['dwk'] = beta *v['dwk'] + (1-beta)*dwk
663     vdW{l} = beta*vdW{l} + (1 -beta) * gradsDW{l};
664     vdB{l} = beta*vdB{l} + (1 -beta) * gradsDB{l};
665     weights{l} = weights{l} -learningRate* vdW{l};
666     biases{l} = biases{l} -learningRate* vdB{l};
667 endfor
668
669 if (strcmp(outputActivationFunc,"sigmoid"))
670     vdW{L} = beta*vdW{L} + (1 -beta) * gradsDW{L};
671     vdB{L} = beta*vdB{L} + (1 -beta) * gradsDB{L};
672     weights{L} = weights{L} -learningRate* vdW{L};
673     biases{L} = biases{L} -learningRate* vdB{L};
674 elseif (strcmp(outputActivationFunc,"softmax"))
675     vdW{L} = beta*vdW{L} + (1 -beta) * gradsDW{L}';
676     vdB{L} = beta*vdB{L} + (1 -beta) * gradsDB{L}';
677     weights{L} = weights{L} -learningRate* vdW{L};
678     biases{L} = biases{L} -learningRate* vdB{L};
679 endif
680
681
682 end
683
684
685 # Update parameters with RMSProp
686 # Input : parameters - weights, biases
687 #      : gradients - gradsDW,gradsDB
688 #      : s -sdw, sdB
689 #      : beta1
690 #      : epsilon
691 #      : learningRate
692 #      : outputActivationFunc
693 #output : Updated weights and biases RMSProp

```

```

694 function [weights biases] = gradientDescentWithRMSProp(weights,
695 biases,gradsDW,gradsDB, sdW, sdB, beta1, epsilon,
696 learningRate,outputActivationFunc="sigmoid")
697 L = size(weights)(2); # number of layers in the neural network
698 # Update rule for each parameter.
699 for l=1:(L-1)
700     sdw{l} = beta1*sdw{l} + (1 -beta1) .* gradsDW{l} .* gradsDW{l};
701     sdB{l} = beta1*sdB{l} + (1 -beta1) .* gradsDB{l} .* gradsDB{l};
702     weights{l} = weights{l} - learningRate* gradsDW{l} ./ sqrt(sdw{l} +
703 epsilon);
704     biases{l} = biases{l} - learningRate* gradsDB{l} ./ sqrt(sdB{l} +
705 epsilon);
706 endfor
707
708 if (strcmp(outputActivationFunc,"sigmoid"))
709     sdw{L} = beta1*sdw{L} + (1 -beta1) .* gradsDW{L} .* gradsDW{L};
710     sdB{L} = beta1*sdB{L} + (1 -beta1) .* gradsDB{L} .* gradsDB{L};
711     weights{L} = weights{L} -learningRate* gradsDW{L} ./ sqrt(sdw{L} +
712 +epsilon);
713     biases{L} = biases{L} -learningRate* gradsDB{L} ./ sqrt(sdB{L} +
714 epsilon);
715 elseif (strcmp(outputActivationFunc,"softmax"))
716     sdw{L} = beta1*sdw{L} + (1 -beta1) .* gradsDW{L}' .* gradsDW{L}';
717     sdB{L} = beta1*sdB{L} + (1 -beta1) .* gradsDB{L}' .* gradsDB{L}';
718     weights{L} = weights{L} -learningRate* gradsDW{L}' ./ sqrt(sdw{L} +
719 +epsilon);
720     biases{L} = biases{L} -learningRate* gradsDB{L}' ./ sqrt(sdB{L} +
721 epsilon);
722 endif
723
724 end
725
726
727 # Update parameters with Adam
728 # Input : parameters - weights, biases
729 #          : gradients -gradsDW,gradsDB
730 #          : v - vdw, vdB
731 #          : s - sdw, sdB
732 #          : t
733 #          : beta1
734 #          : beta2
735 #          : epsilon
736 #          : learningRate
737 #          : epsilon
738 #output : Updated weights and biases
739 function [weights biases] = gradientDescentWithAdam(weights,
740 biases,gradsDW,gradsDB,
741 vdw, vdB, sdW, sdB, t, beta1, beta2, epsilon,
742 learningRate,epsilon="sigmoid")
743 vdw_corrected = {};
744 vdB_corrected = {};
745 sdw_corrected = {};
746 sdB_corrected = {};
747 L = size(weights)(2); # number of layers in the neural network
748 # Update rule for each parameter.
749 for l=1:(L-1)
750     vdw{l} = beta1*vdw{l} + (1 -beta1) * gradsDW{l};
751     vdB{l} = beta1*vdB{l} + (1 -beta1) * gradsDB{l};
752
753     # Compute bias-corrected first moment estimate.
754     vdw_corrected{l} = vdw{l}/(1-beta1^t);
755     vdB_corrected{l} = vdB{l}/(1-beta1^t);
756
757

```

```

758     sdw{1} = beta2*sdw{1} + (1 -beta2) * gradsDW{1} .* gradsDW{1};
759     sdb{1} = beta2*sdb{1} + (1 -beta2) * gradsDB{1} .* gradsDB{1};
760
761     # Compute bias-corrected second moment estimate.
762     sdw_corrected{1} = sdw{1}/(1-beta2^t);
763     sdb_corrected{1} = sdb{1}/(1-beta2^t);
764
765     # Update parameters.
766     d1=sqrt(sdw_corrected{1}+epsilon);
767     d2=sqrt(sdb_corrected{1}+epsilon);
768
769     weights{1} = weights{1} - learningRate* vdw_corrected{1} ./ d1;
770     biases{1} = biases{1} -learningRate* vdb_corrected{1} ./ d2;
771 endfor
772
773 if (strcmp(outputActivationFunc,"sigmoid"))
774     vdw{L} = beta1*vdw{L} + (1 -beta1) * gradsDW{L};
775     vdb{L} = beta1*vdb{L} + (1 -beta1) * gradsDB{L};
776
777     # Compute bias-corrected first moment estimate.
778     vdw_corrected{L} = v{L}/(1-beta1^t);
779     vdb_corrected{L} = v{L}/(1-beta1^t);
780
781     sdw{L} = beta2*sdw{L} + (1 -beta2) * gradsDW{L} .* gradsDW{L};
782     sdb{L} = beta2*sdb{L} + (1 -beta2) * gradsDB{L} .* gradsDB{L};
783
784     # Compute bias-corrected second moment estimate.
785     sdw_corrected{L} = s{L}/(1-beta2^t);
786     sdb_corrected{L} = s{L}/(1-beta2^t);
787
788     # update parameters.
789     d1=sqrt(sdw_corrected{L}+epsilon);
790     d2=sqrt(sdb_corrected{L}+epsilon);
791
792     weights{L} = weights{L} - learningRate* vdw_corrected{L} ./ d1;
793     biases{L} = biases{L} -learningRate* vdb_corrected{L} ./ d2;
794 elseif (strcmp(outputActivationFunc,"softmax"))
795     vdw{L} = beta1*vdw{L} + (1 -beta1) * gradsDW{L}';
796     vdb{L} = beta1*vdb{L} + (1 -beta1) * gradsDB{L}';
797
798     # Compute bias-corrected first moment estimate.
799     vdw_corrected{L} = vdw{L}/(1-beta1^t);
800     vdb_corrected{L} = vdb{L}/(1-beta1^t);
801
802     sdw{L} = beta2*sdw{L} + (1 -beta2) * gradsDW{L}' .* gradsDW{L}';
803     sdb{L} = beta2*sdb{L} + (1 -beta2) * gradsDB{L}' .* gradsDB{L}';
804
805     # Compute bias-corrected second moment estimate.
806     sdw_corrected{L} = sdw{L}/(1-beta2^t);
807     sdb_corrected{L} = sdb{L}/(1-beta2^t);
808
809     # update parameters.
810     d1=sqrt(sdw_corrected{L}+epsilon);
811     d2=sqrt(sdb_corrected{L}+epsilon);
812
813     weights{L} = weights{L} - learningRate* vdw_corrected{L} ./ d1;
814     biases{L} = biases{L} -learningRate* vdb_corrected{L} ./ d2;
815 endif
816
817
818 end
819
820 # Execute a L layer Deep learning model
821 # Input : X - Input features

```

```

822 #      : Y output
823 #      : layersDimensions - Dimension of layers
824 #      : hiddenActivationFunc - Activation function at hidden layer relu
825 /tanh
826 #      : outputActivationFunc - Activation function at hidden layer
827 sigmoid/softmax
828 #      : learning rate
829 #      : lambd
830 #      : keep_prob
831 #      : num of iterations
832 #output : Updated weights and biases after each iteration
833 function [weights biases costs] = L_Layer_DeepModel(X, Y, layersDimensions,
834 hiddenActivationFunc='relu',
835             outputActivationFunc="sigmoid", learning_rate = .3, lambd=0,
836 keep_prob=1, num_iterations = 10000, initType="default")#lr was 0.009
837
838     rand ("seed", 1);
839     costs = [] ;
840     if (strcmp(initType,"He"))
841         # He Initialization
842         [weights biases] = HeInitializeDeepModel(layersDimensions);
843     elseif (strcmp(initType,"xav"))
844         # Xavier Initialization
845         [weights biases] = XavInitializeDeepModel(layersDimensions);
846     else
847         # Default initialization.
848         [weights biases] = initializeDeepModel(layersDimensions);
849     endif
850
851     # Loop (gradient descent)
852     for i = 0:num_iterations
853         # Forward propagation: [LINEAR -> RELU]*^(L-1) -> LINEAR -> SIGMOID.
854         [AL forward_caches activation_caches dropoutMat] =
855 forwardPropagationDeep(X, weights, biases,keep_prob, hiddenActivationFunc,
856 outputActivationFunc=outputActivationFunc);
857
858         # Regularization parameter is 0
859         if (lambd==0)
860             # Compute cost.
861             cost = computeCost(AL,
862 Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(
863 layersDimensions)(2)));
864         else
865             # Compute cost with regularization
866             cost = computeCostWithReg(weights, AL, Y, lambd,
867 outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(1a
868 yersDimensions)(2)));
869         endif
870         # Backward propagation.
871         [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y,
872 activation_caches,forward_caches, dropoutMat, lambd, keep_prob,
873 hiddenActivationFunc, outputActivationFunc=outputActivationFunc,
874 numClasses=layersDimensions(size(layersDimensions)(2)));
875         # Update parameters.
876         [weights biases] = gradientDescent(weights,biases,
877 gradsDW,gradsDB,learning_rate,outputActivationFunc=outputActivationFunc);
878
879
880         # Print the cost every 1000 iterations
881         if ( mod(i,1000) == 0)
882             costs =[costs cost];
883             #disp ("Cost after iteration"),
884             L2RegularizationCost(i),disp(cost);
885

```

```

886         printf("Cost after iteration i=%i cost=%d\n",i,cost);
887     endif
888 endfor
889
890 end
891
892 # Execute a L layer Deep learning model with Stochastic Gradient descent
893 # Input : X - Input features
894 #           Y output
895 #           layersDimensions - Dimension of layers
896 #           hiddenActivationFunc - Activation function at hidden layer relu
897 /tanh/sigmoid
898 #           outputActivationFunc - Activation function at hidden layer
899 sigmoid/softmax
900 #           learning rate
901 #           lrDecay
902 #           decayRate
903 #           lambd
904 #           keep_prob
905 #           optimizer
906 #           beta
907 #           beta1
908 #           beta2
909 #           epsilon
910 #           mini_batch_size
911 #           num of epochs
912 #output : Updated weights and biases after each iteration
913 function [weights biases costs] = L_Layer_DeepModel_SGD(X, Y,
914 layersDimensions, hiddenActivationFunc='relu',
915
916 outputActivationFunc="sigmoid",learningRate = .3,
917 optimizer
918 lrDecay=false,decayRate=1,
919
920
921 beta2=0.999,epsilon=10^-8,
922
923 mini_batch_size = 64, num_epochs =
924 2500)
925
926 disp("values");
927 printf("learningRate=%f ",learningRate);
928 printf("lrDecay=%d ",lrDecay);
929 printf("decayRate=%f ",decayRate);
930 printf("lambd=%d ",lambd);
931 printf("keep_prob=%f ",keep_prob);
932 printf("optimizer=%s ",optimizer);
933 printf("beta=%f ",beta);
934 printf("beta1=%f ",beta1);
935 printf("beta2=%f ",beta2);
936 printf("epsilon=%f ",epsilon);
937 printf("mini_batch_size=%d ",mini_batch_size);
938 printf("num_epochs=%d ",num_epochs);
939 t=0;
940 rand ("seed", 1);
941 costs = [] ;
942 # Parameters initialization.
943 [weights biases] = initializeDeepModel(layersDimensions);
944
945 if (strcmp(optimizer,"momentum"))
946     [vdW vdB] = initializeVelocity(weights, biases);
947
948 elseif(strcmp(optimizer,"rmsprop"))
949     [sdW sdB] = initializeRMSProp(weights, biases);

```

```

950     elseif(strcmp(optimizer,"adam"))
951         [vdW vdB sdW sdB] = initializeAdam(weights, biases);
952     endif
953     seed=10;
954     # Loop (gradient descent)
955     for i = 0:num_epochs
956         seed = seed + 1;
957         [mini_batches_X mini_batches_Y] = random_mini_batches(X, Y,
958         mini_batch_size, seed);
959
960         minibatches=length(mini_batches_X);
961         for batch=1:minibatches
962             X=mini_batches_X{batch};
963             Y=mini_batches_Y{batch};
964             # Forward propagation: [LINEAR -> RELU]^(L-1) -> LINEAR ->
965             SIGMOID/SOFTMAX.
966             [AL forward_caches activation_caches dropoutMat] =
967             forwardPropagationDeep(X, weights, biases, keep_prob,hiddenActivationFunc,
968             outputActivationFunc=outputActivationFunc);
969             #disp(batch);
970             #disp(size(X));
971             #disp(size(Y));
972             if (lambd==0)
973                 # Compute cost.
974                 cost = computeCost(AL,
975                 Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(
976                 layersDimensions)(2)));
977             else
978                 # Compute cost with regularization
979                 cost = computeCostWithReg(weights, AL, Y, lambd,
980                 outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(1a
981                 yersDimensions)(2)));
982             endif
983             #disp(cost);
984             # Backward propagation.
985             [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y,
986             activation_caches,forward_caches, dropoutMat, lambd, keep_prob,
987             hiddenActivationFunc, outputActivationFunc=outputActivationFunc,
988
989             numClasses=layersDimensions(size(layersDimensions)(2)));
990
991             if (strcmp(optimizer,"gd"))
992                 # Update parameters.
993                 [weights biases] = gradientDescent(weights,biases,
994                 gradsDW,gradsDB,learningRate,outputActivationFunc=outputActivationFunc);
995             elseif (strcmp(optimizer,"momentum"))
996                 [weights biases] = gradientDescentWithMomentum(weights,
997                 biases,gradsDW,gradsDB, vdw, vdB, beta, learningRate,outputActivationFunc);
998             elseif (strcmp(optimizer,"rmsprop"))
999                 [weights biases] = gradientDescentWithRMSPROP(weights,
1000                 biases,gradsDW,gradsDB, sdW, sdB, beta1, epsilon,
1001                 learningRate,outputActivationFunc);
1002
1003             elseif (strcmp(optimizer,"adam"))
1004                 t=t+1;
1005                 [weights biases] = gradientDescentWithAdam(weights,
1006                 biases,gradsDW,gradsDB,vdW, vdB, sdW, sdB, t, beta1, beta2, epsilon,
1007                 learningRate,outputActivationFunc);
1008             endif
1009             endfor
1010             # Print the cost every 1000 iterations
1011             if ( mod(i,1000) == 0)
1012                 costs =[costs cost];
1013                 #disp ("Cost after iteration"), disp(i),disp(cost);

```

```

1014         printf("Cost after iteration i=%i cost=%d\n",i,cost);
1015     endif
1016     if(lrDecay==true)
1017         learningRate=decayRate^(num_epochs/1000)*learningRate;
1018     endif
1019 endfor
1020
1021 end
1022
1023 # Plot cost vs iterations
1024 function plotCostVsIterations(maxIterations,costs,fig1)
1025     iterations=[0:1000:maxIterations];
1026     plot(iterations,costs);
1027     title ("Cost vs no of iterations ");
1028     xlabel("No of iterations");
1029     ylabel("Cost");
1030     print -dpng figReg2-o
1031 end;
1032
1033 # Plot cost vs epochs
1034 function plotCostVsEpochs(maxEpochs,costs,fig1)
1035     epochs=[0:1000:maxEpochs];
1036     plot(epochs,costs);
1037     title ("Cost vs no of epochs ");
1038     xlabel("No of epochs");
1039     ylabel("Cost");
1040     print -dpng fig5-o
1041 end;
1042
1043 # Compute the predicted value for a given input
1044 # Input : Neural Network parameters
1045 #           : Input data
1046 function [predictions]= predict(weights, biases,
1047 x,keep_prob=1,hiddenActivationFunc="relu")
1048     [AL forward_caches activation_caches] = forwardPropagationDeep(x,
1049 weights, biases,keep_prob,hiddenActivationFunc);
1050     predictions = (AL>0.5);
1051 end
1052
1053 # Plot the decision boundary
1054 function plotDecisionBoundary(data,weights,
1055 biases,keep_prob=1,hiddenActivationFunc="relu",fig2)
1056     %Plot a non-linear decision boundary learned by the SVM
1057     colormap ("summer");
1058
1059     % Make classification predictions over a grid of values
1060     x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)';
1061     x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)';
1062     [X1, X2] = meshgrid(x1plot, x2plot);
1063     vals = zeros(size(X1));
1064     # Plot the prediction for the grid
1065     for i = 1:size(X1, 2)
1066         gridPoints = [X1(:, i), X2(:, i)];
1067         vals(:, i)=predict(weights, biases,gridPoints',keep_prob,
1068 hiddenActivationFunc=hiddenActivationFunc);
1069     endfor
1070
1071     scatter(data(:,1),data(:,2),8,c=data(:,3),"filled");
1072     % Plot the boundary
1073     hold on
1074     #contour(X1, X2, vals, [0 0], 'Linewidth', 2);
1075     contour(X1, X2, vals,"linewidth",4);
1076     title ({"3 layer Neural Network decision boundary"});
1077     hold off;

```

```

1078     print -dpng figReg22-o
1079
1080 end
1081
1082 # Compute scores
1083 function [AL]= scores(weights, biases, x,hiddenActivationFunc="relu")
1084     [AL forward_caches activation_caches] = forwardPropagationDeep(X,
1085 weights, biases,hiddenActivationFunc);
1086 end
1087
1088 # Create Random mini batches. Return cell arrays with the mini batches
1089 # Input : X, Y
1090 #           : Size of minibatch
1091 #Output : mini batches X & Y
1092 function [mini_batches_X mini_batches_Y]= random_mini_batches(X, Y,
1093 miniBatchSize = 64, seed = 0)
1094
1095     rand ("seed", seed);
1096     # Get number of training samples
1097     m = size(X)(2);
1098
1099
1100    # Create a list of random numbers < m
1101    permutation = randperm(m);
1102    # Randomly shuffle the training data
1103    shuffled_X = X(:, permutation);
1104    shuffled_Y = Y(:, permutation);
1105
1106    # Compute number of mini batches
1107    numCompleteMinibatches = floor(m/miniBatchSize);
1108    batch=0;
1109    for k = 0:(numCompleteMinibatches-1)
1110        #Set the start and end of each mini batch
1111        batch=batch+1;
1112        lower=(k*miniBatchSize)+1;
1113        upper=(k+1) * miniBatchSize;
1114        mini_batch_X = shuffled_X(:, lower:upper);
1115        mini_batch_Y = shuffled_Y(:, lower:upper);
1116        # Create cell arrays
1117        mini_batches_X{batch} = mini_batch_X;
1118        mini_batches_Y{batch} = mini_batch_Y;
1119    endfor
1120
1121    # If the batc size does not cleanly divide with number of mini batches
1122    if mod(m ,miniBatchSize) != 0
1123        # Set the start and end of the last mini batch
1124        l=floor(m/miniBatchSize)*miniBatchSize;
1125        m=l+ mod(m,miniBatchSize);
1126        mini_batch_X = shuffled_X(:,(l+1):m);
1127        mini_batch_Y = shuffled_Y(:,(l+1):m);
1128
1129        batch=batch+1;
1130        mini_batches_X{batch} = mini_batch_X;
1131        mini_batches_Y{batch} = mini_batch_Y;
1132    endif
1133 end
1134
1135 # Plot decision boundary
1136 function plotDecisionBoundary1( data,weights, biases,keep_prob=1,
1137 hiddenActivationFunc="relu")
1138     % Make classification predictions over a grid of values
1139     x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)';
1140     x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)';
1141     [X1, X2] = meshgrid(x1plot, x2plot);

```

```

1142     vals = zeros(size(X1));
1143     for i = 1:size(X1, 2)
1144         gridPoints = [x1(:, i), x2(:, i)];
1145         [AL forward_caches activation_caches] =
1146         forwardPropagationDeep(gridPoints', weights,
1147         biases, keep_prob, hiddenActivationFunc, outputActivationFunc="softmax");
1148         [l m] = max(AL, [ ], 2);
1149         vals(:, i)= m;
1150     endfor
1151
1152     scatter(data(:,1),data(:,2),8,c=data(:,3),"filled");
1153     % Plot the boundary
1154     hold on
1155     contour(x1, x2, vals,"linewidth",4);
1156     print -dpng "fig-01.png"
1157 end

```

9.Appendix 8 – Gradient Check

8.1 Python

```

1 # -*- coding: utf-8 -*-
2 ######
3 #####
4 #
5 # File: DLfunctions8.py
6 # Developer: Tinniam V Ganesh
7 # Date : 6 May 2018
8 #
9 #####
10 #####
11 import numpy as np
12 import matplotlib.pyplot as plt
13 import matplotlib
14 import matplotlib.pyplot as plt
15 from matplotlib import cm
16 import math
17 import sklearn
18 import sklearn.datasets
19
20 # Compute the sigmoid of a vector
21 def sigmoid(z):
22     A=1/(1+np.exp(-z))
23     cache=z
24     return A,cache
25
26 # Compute the Relu of a vector
27 def relu(z):
28     A = np.maximum(0,z)
29     cache=z
30     return A,cache
31
32 # Compute the tanh of a vector
33 def tanh(z):
34     A = np.tanh(z)

```

```

35     cache=z
36     return A,cache
37
38 # Compute the softmax of a vector
39 def softmax(Z):
40     # get unnormalized probabilities
41     exp_scores = np.exp(Z.T)
42     # normalize them for each example
43     A = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
44     cache=z
45     return A,cache
46
47 # Compute the Stable softmax of a vector
48 def stableSoftmax(Z):
49     #Compute the softmax of vector x in a numerically stable way.
50     shiftZ = Z.T - np.max(Z.T, axis=1).reshape(-1,1)
51     exp_scores = np.exp(shiftZ)
52
53     # normalize them for each example
54     A = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
55     cache=z
56     return A,cache
57
58 # Compute the derivative of Relu
59 def reluDerivative(dA, cache):
60
61     Z = cache
62     dz = np.array(dA, copy=True) # just converting dz to a correct object.
63     # When z <= 0, you should set dz to 0 as well.
64     dz[Z <= 0] = 0
65     return dz
66
67 # Compute the derivative of sigmoid
68 def sigmoidDerivative(dA, cache):
69     Z = cache
70     s = 1/(1+np.exp(-Z))
71     dz = dA * s * (1-s)
72     return dz
73
74 # Compute the derivative of tanh
75 def tanhDerivative(dA, cache):
76     Z = cache
77     a = np.tanh(Z)
78     dz = dA * (1 - np.power(a, 2))
79     return dz
80
81 # Compute the derivative of softmax
82 def softmaxDerivative(dA, cache,y,numTraining):
83     # Note : dA not used. dL/dZ = dL/dA * dA/dZ = pi-yi
84     Z = cache
85     # Compute softmax
86     exp_scores = np.exp(Z.T)
87     # normalize them for each example
88     probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
89
90     # compute the gradient on scores
91     dz = probs
92
93     # dz = pi- yi
94     dz[range(int(numTraining)),y[:,0]] -= 1
95     return(dz)
96
97 # Compute the derivative of Stable softmax
98 def stableSoftmaxDerivative(dA, cache,y,numTraining):

```

```

99      # Note : dA not used. dL/dZ = dL/dA * dA/dZ = pi-yi
100     Z = cache
101     # Compute stable softmax
102     shiftz = Z.T - np.max(Z.T, axis=1).reshape(-1,1)
103     exp_scores = np.exp(shiftz)
104     # normalize them for each example
105     probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
106     #print(probs)
107     # compute the gradient on scores
108     dz = probs
109
110     # dz = pi- yi
111     dz[range(int(numTraining)),y[:,0]] -= 1
112     return(dz)
113
114
115     # Initialize the model
116     # Input : number of features
117     #           number of hidden units
118     #           number of units in output
119     # Returns: Weight and bias matrices and vectors
120     def initializeModel(numFeats,numHidden,numOutput):
121         np.random.seed(1)
122         w1=np.random.randn(numHidden,numFeats)*0.01 # Multiply by .01
123         b1=np.zeros((numHidden,1))
124         w2=np.random.randn(numOutput,numHidden)*0.01
125         b2=np.zeros((numOutput,1))
126
127         # Create a dictionary of the neural network parameters
128         nnParameters={'W1':w1,'b1':b1,'W2':w2,'b2':b2}
129         return(nnParameters)
130
131
132     # Initialize model for L layers
133     # Input : List of units in each layer
134     # Returns: Initial weights and biases matrices for all layers
135     def initializeDeepModel(layerDimensions):
136         np.random.seed(3)
137         # note the weight matrix at layer 'l' is a matrix of size (l,l-1)
138         # The Bias is a vectors of size (l,1)
139
140         # Loop through the layer dimension from 1.. L
141         layerParams = {}
142         for l in range(1,len(layerDimensions)):
143             layerParams['w' + str(l)] =
144                 np.random.randn(layerDimensions[l],layerDimensions[l-1])*0.01 # Multiply by
145                 .01
146             layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1))
147             np.savetxt('w' + str(l)+'.csv',layerParams['w' +
148                 str(l)],delimiter=',')
149             np.savetxt('b' + str(l)+'.csv',layerParams['b' +
150                 str(l)],delimiter=',')
151         return(layerParams)
152         return Z, cache
153
154     # He Initialization model for L layers
155     # Input : List of units in each layer
156     # Returns: Initial weights and biases matrices for all layers
157     # He initialization multiplies the random numbers with
158     sqrt(2/layerDimensions[l-1])
159     def HeInitializeDeepModel(layerDimensions):
160         np.random.seed(3)
161         # note the weight matrix at layer 'l' is a matrix of size (l,l-1)
162         # The Bias is a vectors of size (l,1)

```

```

163
164     # Loop through the layer dimension from 1.. L
165     layerParams = {}
166     for l in range(1,len(layerDimensions)):
167         layerParams['w' + str(l)] = np.random.randn(layerDimensions[l],
168                                         layerDimensions[l-1])*np.sqrt(2/layerDimensions[l-1])
169         layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1))
170
171     return(layerParams)
172     return Z, cache
173
174 # Xavier Initialization model for L layers
175 # Input : List of units in each layer
176 # Returns: Initial weights and biases matrices for all layers
177 # Xavier initialization multiplies the random numbers with
178 # sqrt(1/layerDimensions[l-1])
179 def XavInitializeDeepModel(layerDimensions):
180     np.random.seed(3)
181     # note the weight matrix at layer 'l' is a matrix of size (l,l-1)
182     # The Bias is a vectors of size (l,1)
183
184     # Loop through the layer dimension from 1.. L
185     layerParams = {}
186     for l in range(1,len(layerDimensions)):
187         layerParams['w' + str(l)] = np.random.randn(layerDimensions[l],
188                                         layerDimensions[l-1])*np.sqrt(1/layerDimensions[l-1])
189         layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1))
190
191     return(layerParams)
192     return Z, cache
193
194 # Initialize velocity of
195 # Input : parameters
196 # Returns: v - Initial velocity
197 def initializeVelocity(parameters):
198
199     L = len(parameters)//2 # Create an integer
200     v = {}
201
202     # Initialize velocity with the same dimensions as w
203     for l in range(L):
204         v["dw" + str(l+1)] = np.zeros((parameters['w' + str(l+1)].shape[0],
205                                         parameters['w' + str(l+1)].shape[1]))
206         v["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0],
207                                         parameters['b' + str(l+1)].shape[1]))
208
209     return v
210
211 # Initialize RMSProp param
212 # Input : List of units in each layer
213 # Returns: s - Initial RMSProp
214 def initializeRMSProp(parameters):
215
216     L = len(parameters)//2 # Create an integer
217     s = {}
218
219     # Initialize velocity with the same dimensions as w
220     for l in range(L):
221         s["dw" + str(l+1)] = np.zeros((parameters['w' + str(l+1)].shape[0],
222                                         parameters['w' + str(l+1)].shape[1]))
223         s["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0],
224                                         parameters['b' + str(l+1)].shape[1]))
225
226     return s

```

```

227
228 # Initialize Adam param
229 # Input : List of units in each layer
230 # Returns: v and s - Adam paramaters
231 def initializeAdam(parameters) :
232
233     L = len(parameters) // 2 # number of layers in the neural networks
234     v = {}
235     s = {}
236
237     # Initialize v, s.
238     for l in range(L):
239
240         v["dw" + str(l+1)] = np.zeros((parameters['w' + str(l+1)].shape[0],
241                                         parameters['w' + str(l+1)].shape[1]))
242         v["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0],
243                                         parameters['b' + str(l+1)].shape[1]))
244         s["dw" + str(l+1)] = np.zeros((parameters['w' + str(l+1)].shape[0],
245                                         parameters['w' + str(l+1)].shape[1]))
246         s["db" + str(l+1)] = np.zeros((parameters['b' + str(l+1)].shape[0],
247                                         parameters['b' + str(l+1)].shape[1]))
248
249     return v, s
250
251 # Compute the activation at a layer 'l' for forward prop in a Deep Network
252 # Input : A_prev - Activation of previous layer
253 #           w,b - Weight and bias matrices and vectors
254 #           keep_prob
255 #           activationFunc - Activation function - sigmoid, tanh, relu etc
256 # Returns : A, cache
257 #           :
258 # Z = W * X + b
259 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
260 def layerActivationForward(A_prev, w, b, keep_prob=0, activationFunc="relu"):
261
262     # Compute z
263     z = np.dot(w,A_prev) + b
264     forward_cache = (A_prev, w, b)
265     # Compute the activation for sigmoid
266     if activationFunc == "sigmoid":
267         A, activation_cache = sigmoid(z)
268     # Compute the activation for Relu
269     elif activationFunc == "relu":
270         A, activation_cache = relu(z)
271     # Compute the activation for tanh
272     elif activationFunc == 'tanh':
273         A, activation_cache = tanh(z)
274     elif activationFunc == 'softmax':
275         A, activation_cache = stableSoftmax(z)
276
277     cache = (forward_cache, activation_cache)
278     return A, cache
279
280 # Compute the forward propagation for layers 1..L
281 # Input : X - Input Features
282 #           parameters: Weights and biases
283 #           keep_prob
284 #           hiddenActivationFunc - Activation function at hidden layers
285 #           Relu/tanh
286 #           outputActivationFunc - Activation function at output -
287 #           sigmoid/softmax
288 # Returns : AL
289 #           caches
290 #           dropoutMat

```

```

290 # The forward propagation uses the Relu/tanh activation from layer 1..L-1 and
291 sigmoid activation at layer L
292 def forwardPropagationDeep(x, parameters, keep_prob=0,
293 hiddenActivationFunc='relu', outputActivationFunc='sigmoid'):
294     caches = []
295     # initialize the dropout matrix
296     dropoutMat = {}
297     # Set A to X (A0)
298     A = X
299     L = len(parameters)//2 # number of layers in the neural network
300     # Loop through from layer 1 to upto layer L
301     for l in range(1, L):
302         A_prev = A
303         #  $Z_i = W_i \times A_{i-1} + b_i$  and  $A_i = g(Z_i)$ 
304         A, cache = layerActivationForward(A_prev, parameters['W'+str(l)],
305 parameters['b'+str(l)], keep_prob, activationFunc = hiddenActivationFunc)
306
307         # Randomly drop some activation units
308         # Create a matrix as the same shape as A
309         D = np.random.rand(A.shape[0], A.shape[1])
310         D = (D < keep_prob)
311         # We need to use the same 'dropout' matrix in backward propagation
312         # Save the dropout matrix for use in backprop
313         dropoutMat["D" + str(l)] = D
314         A= np.multiply(A,D)
315         A = np.divide(A,keep_prob)
316
317         caches.append(cache)
318
319
320     # last layer
321     AL, cache = layerActivationForward(A, parameters['W'+str(L)],
322 parameters['b'+str(L)], activationFunc = outputActivationFunc)
323     caches.append(cache)
324
325     return AL, caches, dropoutMat
326
327
328 # Compute the cost
329 # Input : Activation of last layer
330 #          : Output from data
331 #          : Y
332 #          :outputActivationFunc - Activation function at output -
333 sigmoid/softmax
334 # Output: cost
335 def computeCost(AL,Y,outputActivationFunc="sigmoid"):
336     if outputActivationFunc=="sigmoid":
337         m= float(Y.shape[1])
338         # Element wise multiply for logprobs
339         cost=-1/m *np.sum(Y*np.log(AL) + (1-Y)*(np.log(1-AL)))
340         cost = np.squeeze(cost)
341     elif outputActivationFunc=="softmax":
342         # Take transpose of Y for softmax
343         Y=Y.T
344         m= float(len(Y))
345         # Compute log probs. Take the log prob of correct class based on
346 output y
347         correct_logprobs = -np.log(AL[range(int(m)),Y.T])
348         # Compute loss
349         cost = np.sum(correct_logprobs)/m
350
351
352     return cost
353
354
355 # Compute the cost with regularization

```

```

354 # Input : parameters
355 #      : AL
356 #      : Y
357 #      : lambd
358 #      :outputActivationFunc - Activation function at output -
359 sigmoid/softmax/tanh
360 # Output: cost
361 def computeCostWithReg(parameters,AL,Y,lambd,
362 outputActivationFunc="sigmoid"):
363
364
365     if outputActivationFunc=="sigmoid":
366         m= float(Y.shape[1])
367         # Element wise multiply for logprobs
368         cost=-1/m *np.sum(Y*np.log(AL) + (1-Y)*(np.log(1-AL)))
369         cost = np.squeeze(cost)
370
371         # Regularization cost
372         L= int(len(parameters)/2)
373         L2RegularizationCost=0
374         for l in range(L):
375             L2RegularizationCost+=np.sum(np.square(parameters['w'+str(l+1)]))
376
377         L2RegularizationCost = (lambd/(2*m))*L2RegularizationCost
378         cost = cost + L2RegularizationCost
379
380
381     elif outputActivationFunc=="softmax":
382         # Take transpose of Y for softmax
383         Y=Y.T
384         m= float(len(Y))
385         # Compute log probs. Take the log prob of correct class based on
386         output y
387         correct_logprobs = -np.log(AL[range(int(m)),Y.T])
388         # Compute loss
389         cost = np.sum(correct_logprobs)/m
390
391         # Regularization cost
392         L= int(len(parameters)/2)
393         L2RegularizationCost=0
394         for l in range(L):
395             L2RegularizationCost+=np.sum(np.square(parameters['w'+str(l+1)]))
396
397         L2RegularizationCost = (lambd/(2*m))*L2RegularizationCost
398         cost = cost + L2RegularizationCost
399
400     return cost
401
402 # Compute the backpropagation for 1 cycle
403 # Input : Neural Network parameters - dA
404 #      # cache - forward_cache & activation_cache
405 #      # Input features
406 #      # keep_prob
407 #      # Output values Y
408 # Returns: Gradients
409 # dL/dwi= dL/dzi*A_l-1
410 # dL/dbl = dL/dz_l
411 # dL/dZ_prev=dL/dz_l*w
412 def layerActivationBackward(dA, cache, Y, keep_prob=1,
413 activationFunc="relu"):
414     forward_cache, activation_cache = cache
415     A_prev, w, b = forward_cache
416     numtraining = float(A_prev.shape[1])
417     if activationFunc == "relu":

```

```

418     dz = reluDerivative(dA, activation_cache)
419 elif activationFunc == "sigmoid":
420     dz = sigmoidDerivative(dA, activation_cache)
421 elif activationFunc == "tanh":
422     dz = tanhDerivative(dA, activation_cache)
423 elif activationFunc == "softmax":
424     dz = stableSoftmaxDerivative(dA, activation_cache, Y, numtraining)
425
426 if activationFunc == 'softmax':
427     dw = 1/numtraining * np.dot(A_prev, dz)
428     db = 1/numtraining * np.sum(dZ, axis=0, keepdims=True)
429     dA_prev = np.dot(dz, w)
430 else:
431
432     dw = 1/numtraining *(np.dot(dZ, A_prev.T))
433     db = 1/numtraining * np.sum(dZ, axis=1, keepdims=True)
434     dA_prev = np.dot(w.T, dZ)
435
436 return dA_prev, dw, db
437
438
439 # Compute the backpropagation with regularization for 1 cycle
440 # Input : dA- Neural Network parameters
441 #          # cache - forward_cache & activation_cache
442 #          # Output values Y
443 #          # lambd
444 #          # activationFunc
445 # Returns dA_prev, dw, db
446 # Returns: Gradients
447 # dL/dwi= dL/dzi*A1-1
448 # dL/dbl = dL/dzl
449 # dL/dz_prev=dL/dzl*w
450 def layerActivationBackwardWithReg(dA, cache, Y, lambd, activationFunc):
451     forward_cache, activation_cache = cache
452     A_prev, w, b = forward_cache
453     numtraining = float(A_prev.shape[1])
454
455     #print("n=",numtraining)
456     #print("no=",numtraining)
457     if activationFunc == "relu":
458         dz = reluDerivative(dA, activation_cache)
459     elif activationFunc == "sigmoid":
460         dz = sigmoidDerivative(dA, activation_cache)
461     elif activationFunc == "tanh":
462         dz = tanhDerivative(dA, activation_cache)
463     elif activationFunc == "softmax":
464         dz = stableSoftmaxDerivative(dA, activation_cache, Y, numtraining)
465
466     if activationFunc == 'softmax':
467         # Add the regularization factor
468         dw = 1/numtraining * np.dot(A_prev, dz) + (lambd/numtraining) * w.T
469         db = 1/numtraining * np.sum(dZ, axis=0, keepdims=True)
470         dA_prev = np.dot(dz, w)
471     else:
472
473         # Add the regularization factor
474         dw = 1/numtraining *(np.dot(dZ, A_prev.T)) + (lambd/numtraining) * w
475         #print("dw=",dw)
476         db = 1/numtraining * np.sum(dZ, axis=1, keepdims=True)
477         #print("db=",db)
478         dA_prev = np.dot(w.T, dZ)
479
480
481 return dA_prev, dw, db

```

```

482
483 # Compute the backpropagation for 1 cycle
484 # Input : AL: Output of L layer Network - weights
485 #           # Y Real output
486 #           # caches -- list of caches containing:
487 #           # dropoutMat
488 #           # lambd
489 #           # keep_prob
490 #           every cache of layerActivationForward() with "relu"/"tanh"
491 #           #(it's caches[1], for l in range(L-1) i.e l = 0...L-2)
492 #           #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1])
493 #           # hiddenActivationFunc - Activation function at hidden layers -
494 #           relu/sigmoid/tanh
495 #           #outputActivationFunc - Activation function at output -
496 #           sigmoid/softmax
497 #
498 #     Returns:
499 #     gradients -- A dictionary with the gradients
500 #                 gradients["dA" + str(l)] = ...
501 #                 gradients["dw" + str(l)] = ...
502 #                 gradients["db" + str(l)] = ...
503 #
504 def backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd=0, keep_prob=1,
505 hiddenActivationFunc='relu',outputActivationFunc="sigmoid"):
506     #initialize the gradients
507     gradients = {}
508     # Set the number of layers
509     L = len(caches)
510     m = float(AL.shape[1])
511
512     if outputActivationFunc == "sigmoid":
513         Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL
514         # Initializing the backpropagation
515         #  $dL/dAL = -(y/a + (1-y)/(1-a))$  - At the output layer
516         dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
517     else:
518         dAL = 0
519         Y=Y.T
520
521     # Since this is a binary classification the activation at output is
522     sigmoid
523     # Get the gradients at the last layer
524     # Inputs: "AL, Y, caches".
525     # Outputs: "gradients["dAL"], gradients["dwL"], gradients["dbL"]"
526     current_cache = caches[L-1]
527     if lambd==0:
528         gradients["dA" + str(L)], gradients["dw" + str(L)], gradients["db" +
529         str(L)] = layerActivationBackward(dAL, current_cache,
530                                         Y, activationFunc =
531                                         outputActivationFunc)
532     else: #Regularization
533         gradients["dA" + str(L)], gradients["dw" + str(L)], gradients["db" +
534         str(L)] = layerActivationBackwardWithReg(dAL, current_cache,
535                                         Y, lambd, activationFunc =
536                                         outputActivationFunc)
537
538     # Note dA for softmax is the transpose
539     if outputActivationFunc == "softmax":
540         gradients["dA" + str(L)] = gradients["dA" + str(L)].T
541     # Traverse in the reverse direction
542     for l in reversed(range(L-1)):
543
544         # Compute the gradients for L-1 to 1 for Relu/tanh
545         # Inputs: "gradients["dA" + str(l + 2)], caches".

```

```

546         # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l +
547 1)] , gradients["db" + str(l + 1)]
548         current_cache = caches[l]
549
550         #dA_prev_temp, dw_temp, db_temp =
551 layerActivationBackward(gradients['dA'+str(l+2)], current_cache,
552 activationFunc = "relu")
553         if lambd==0:
554             # In the reverse direction use the same dropout matrix
555             # Random dropout
556             # Multiply dA'l' with the dropoutMat and divide to keep the
557 expected value same
558             D = dropoutMat["D" + str(l+1)]
559             # Drop some dA'l's
560             gradients['dA'+str(l+2)]= np.multiply(gradients['dA'+str(l+2)],D)
561             # Divide by keep_prob to keep expected value same
562             gradients['dA'+str(l+2)] =
563 np.divide(gradients['dA'+str(l+2)],keep_prob)
564
565         dA_prev_temp, dw_temp, db_temp =
566 layerActivationBackward(gradients['dA'+str(l+2)], current_cache, Y,
567 keep_prob=1, activationFunc = hiddenActivationFunc)
568
569         else:
570             dA_prev_temp, dw_temp, db_temp =
571 layerActivationBackwardWithReg(gradients['dA'+str(l+2)], current_cache, Y,
572 lambd, activationFunc = hiddenActivationFunc)
573             gradients["dA" + str(l + 1)] = dA_prev_temp
574             gradients["dw" + str(l + 1)] = dw_temp
575             gradients["db" + str(l + 1)] = db_temp
576
577     return gradients
578
579 # Perform Gradient Descent
580 # Input : Weights and biases
581 #           : gradients
582 #           : learning rate
583 #           : outputActivationFunc - Activation function at output -
584 sigmoid/softmax
585 #output : Updated weights after 1 iteration
586 def gradientDescent(parameters, gradients,
587 learningRate,outputActivationFunc="sigmoid"):
588
589     L = int(len(parameters) / 2)
590     # Update rule for each parameter.
591     for l in range(L-1):
592         parameters["w" + str(l+1)] = parameters['w'+str(l+1)] -learningRate*
593 gradients['dw' + str(l+1)]
594         parameters["b" + str(l+1)] = parameters['b'+str(l+1)] -learningRate*
595 gradients['db' + str(l+1)]
596
597         if outputActivationFunc=="sigmoid":
598             parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
599 gradients['dw' + str(L)]
600             parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
601 gradients['db' + str(L)]
602         elif outputActivationFunc=="softmax":
603             parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
604 gradients['dw' + str(L)].T
605             parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
606 gradients['db' + str(L)].T
607
608     return parameters
609

```

```

610 # Update parameters with momentum
611 # Input : parameters
612 #       : gradients
613 #       : v
614 #       : beta
615 #       : learningRate
616 #       : outputActivationFunc - softmax/sigmoid
617 #output : Updated parameters and velocity
618 def gradientDescentWithMomentum(parameters, gradients, v, beta, learningRate,
619 outputActivationFunc="sigmoid"):
620
621     L = len(parameters) // 2 # number of layers in the neural networks
622     # Momentum update for each parameter
623     for l in range(L-1):
624
625         # Compute velocities
626         # v['dwk'] = beta *v['dwk'] + (1-beta)*dW
627         v["dw" + str(l+1)] = beta*v["dw" + str(l+1)] + (1-beta) *
628 gradients['dw' + str(l+1)]
629         v["db" + str(l+1)] = beta*v["db" + str(l+1)] + (1-beta) *
630 gradients['db' + str(l+1)]
631         # Update parameters with velocities
632         parameters["w" + str(l+1)] = parameters['w' + str(l+1)] -
633 learningRate* v["dw" + str(l+1)]
634         parameters["b" + str(l+1)] = parameters['b' + str(l+1)] -
635 learningRate* v["db" + str(l+1)]
636
637         if outputActivationFunc=="sigmoid":
638             v["dw" + str(L)] = beta*v["dw" + str(L)] + (1-beta) * gradients['dw' +
639 str(L)]
640             v["db" + str(L)] = beta*v["db" + str(L)] + (1-beta) * gradients['db' +
641 str(L)]
642             parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
643 gradients['dw' + str(L)]
644             parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
645 gradients['db' + str(L)]
646         elif outputActivationFunc=="softmax":
647             v["dw" + str(L)] = beta*v["dw" + str(L)] + (1-beta) * gradients['dw' +
648 str(L)].T
649             v["db" + str(L)] = beta*v["db" + str(L)] + (1-beta) * gradients['db' +
650 str(L)].T
651             parameters["w" + str(L)] = parameters['w'+str(L)] -learningRate*
652 gradients['dw' + str(L)].T
653             parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate*
654 gradients['db' + str(L)].T
655
656         return parameters, v
657
658
659 # Update parameters with RMSProp
660 # Input : parameters
661 #       : gradients
662 #       : s
663 #       : beta1
664 #       : learningRate
665 #       : outputActivationFunc - sigmoid/softmax
666 # output : Updated parameters and RMSProp
667 def gradientDescentWithRMSProp(parameters, gradients, s, beta1, epsilon,
668 learningRate, outputActivationFunc="sigmoid"):
669
670     L = len(parameters) // 2 # number of layers in the neural networks
671     # Momentum update for each parameter
672     for l in range(L-1):

```

```

674     # Compute RMSProp
675     # s['dwk'] = beta1 *s['dwk'] + (1-beta1)*dwk**2/sqrt(s['dwk'])
676     s["dw" + str(l+1)] = beta1*s["dw" + str(l+1)] + (1-beta1) * \
677         np.multiply(gradients['dw' + str(l+1)],gradients['dw' +
678             str(l+1)])
679     s["db" + str(l+1)] = beta1*s["db" + str(l+1)] + (1-beta1) * \
680         np.multiply(gradients['db' + str(l+1)],gradients['db' +
681             str(l+1)])
682     # Update parameters with RMSProp
683     parameters["w" + str(l+1)] = parameters['w' + str(l+1)] - \
684         learningRate* gradients['dw' + str(l+1)]/np.sqrt(s["dw" +
685             str(l+1)] + epsilon)
686     parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - \
687         learningRate* gradients['db' + str(l+1)]/np.sqrt(s["db" +
688             str(l+1)] + epsilon)
689
690     if outputActivationFunc=="sigmoid":
691         s["dw" + str(L)] = beta1*s["dw" + str(L)] + (1-beta1) * \
692             np.multiply(gradients['dw' + str(L)],gradients['dw' + str(L)])
693         s["db" + str(L)] = beta1*s["db" + str(L)] + (1-beta1) * \
694             np.multiply(gradients['db' + str(L)],gradients['db' + str(L)])
695         parameters["w" + str(L)] = parameters['w'+str(L)] - \
696             learningRate* gradients['dw' + str(L)]/np.sqrt(s["dw" + str(L)]
697             + epsilon)
698         parameters["b" + str(L)] = parameters['b'+str(L)] - \
699             learningRate* gradients['db' + str(L)]/np.sqrt(s["db" + str(L)]
700             + epsilon)
701     elif outputActivationFunc=="softmax":
702         s["dw" + str(L)] = beta1*s["dw" + str(L)] + (1-beta1) * \
703             np.multiply(gradients['dw' + str(L)].T,gradients['dw' +
704                 str(L)].T)
705         s["db" + str(L)] = beta1*s["db" + str(L)] + (1-beta1) * \
706             np.multiply(gradients['db' + str(L)].T,gradients['db' +
707                 str(L)].T)
708         parameters["w" + str(L)] = parameters['w'+str(L)] - \
709             learningRate* gradients['dw' + str(L)].T/np.sqrt(s["dw" +
710                 str(L)] + epsilon)
711         parameters["b" + str(L)] = parameters['b'+str(L)] - \
712             learningRate* gradients['db' + str(L)].T/np.sqrt(s["db" +
713                 str(L)] + epsilon)
714
715     return parameters, s
716
717 # Update parameters with Adam
718 # Input : parameters
719 #       : gradients
720 #       : v
721 #       : s
722 #       : t
723 #       : beta1
724 #       : beta2
725 #       : epsilon
726 #       : learningRate
727 #       : outputActivationFunc - sigmoid/softmax
728 # output : Updated parameters and RMSProp
729 def gradientDescentWithAdam(parameters, gradients, v, s, t,
730                             beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8,
731                             learningRate=0.1,
732                             outputActivationFunc="sigmoid"):
733
734     L = len(parameters) // 2
735     # Initializing first moment estimate, python dictionary
736     v_corrected = {}

```

```

738     # Initializing second moment estimate, python dictionary
739     s_corrected = {}
740
741     # Perform Adam upto L-1
742     for l in range(L-1):
743
744         # Compute momentum
745         v["dw" + str(l+1)] = beta1*v["dw" + str(l+1)] + \
746             (1-beta1) * gradients['dw' + str(l+1)]
747         v["db" + str(l+1)] = beta1*v["db" + str(l+1)] + \
748             (1-beta1) * gradients['db' + str(l+1)]
749
750
751         # Compute bias-corrected first moment estimate.
752         v_corrected["dw" + str(l+1)] = v["dw" + str(l+1)]/(1-
753             np.power(beta1,t))
754         v_corrected["db" + str(l+1)] = v["db" + str(l+1)]/(1-
755             np.power(beta1,t))
756
757
758         # Moving average of the squared gradients like RMSProp
759         s["dw" + str(l+1)] = beta2*s["dw" + str(l+1)] + \
760             (1-beta2) * np.multiply(gradients['dw' +
761                 str(l+1)],gradients['dw' + str(l+1)])
762         s["db" + str(l+1)] = beta2*s["db" + str(l+1)] + \
763             (1-beta2) * np.multiply(gradients['db' +
764                 str(l+1)],gradients['db' + str(l+1)])
765
766
767         # Compute bias-corrected second raw moment estimate.
768         s_corrected["dw" + str(l+1)] = s["dw" + str(l+1)]/(1-
769             np.power(beta2,t))
770         s_corrected["db" + str(l+1)] = s["db" + str(l+1)]/(1-
771             np.power(beta2,t))
772
773         # Update parameters.
774         d1=np.sqrt(s_corrected["dw" + str(l+1)]+epsilon)
775         d2=np.sqrt(s_corrected["db" + str(l+1)]+epsilon)
776         parameters["w" + str(l+1)] = parameters['w' + str(l+1)] - \
777             (learningRate* v_corrected["dw" + str(l+1)]/d1)
778         parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - \
779             (learningRate* v_corrected["db" + str(l+1)]/d2)
780
781         if outputActivationFunc=="sigmoid":
782             #Compute 1st moment for L
783             v["dw" + str(L)] = beta1*v["dw" + str(L)] + (1-beta1) *
784             gradients['dw' + str(L)]
785             v["db" + str(L)] = beta1*v["db" + str(L)] + (1-beta1) *
786             gradients['db' + str(L)]
787             # Compute bias-corrected first moment estimate.
788             v_corrected["dw" + str(L)] = v["dw" + str(L)]/(1-
789                 np.power(beta1,t))
790             v_corrected["db" + str(L)] = v["db" + str(L)]/(1-
791                 np.power(beta1,t))
792
793             # Compute 2nd moment for L
794             s["dw" + str(L)] = beta2*s["dw" + str(L)] + (1-beta2) * \
795                 np.multiply(gradients['dw' + str(L)],gradients['dw' +
796                     str(L)])
797             s["db" + str(L)] = beta2*s["db" + str(L)] + (1-beta2) * \
798                 np.multiply(gradients['db' + str(L)],gradients['db' +
799                     str(L)])
800
801             # Compute bias-corrected second raw moment estimate.

```

```

802         s_corrected["dw" + str(L)] = s["dw" + str(L)]/(1-
803 np.power(beta2,t))
804         s_corrected["db" + str(L)] = s["db" + str(L)]/(1-
805 np.power(beta2,t))
806
807         # Update parameters.
808         d1=np.sqrt(s_corrected["dw" + str(L)]+epsilon)
809         d2=np.sqrt(s_corrected["db" + str(L)]+epsilon)
810         parameters["w" + str(L)] = parameters['w' + str(L)]- \
811             (learningRate* v_corrected["dw" + str(L)]/d1)
812         parameters["b" + str(L)] = parameters['b' + str(L)] - \
813             (learningRate* v_corrected["db" + str(L)]/d2)
814
815     elif outputActivationFunc=="softmax":
816         # Compute 1st moment
817         v["dw" + str(L)] = beta1*v["dw" + str(L)] + (1-beta1) *
818 gradients['dw' + str(L)].T
819         v["db" + str(L)] = beta1*v["db" + str(L)] + (1-beta1) *
820 gradients['db' + str(L)].T
821         # Compute bias-corrected first moment estimate.
822         v_corrected["dw" + str(L)] = v["dw" + str(L)]/(1-
823 np.power(beta1,t))
824         v_corrected["db" + str(L)] = v["db" + str(L)]/(1-
825 np.power(beta1,t))
826
827         #Compute 2nd moment
828         s["dw" + str(L)] = beta2*s["dw" + str(L)] + (1-beta2) *
829 np.multiply(gradients['dw' + str(L)].T,gradients['dw' + str(L)].T)
830         s["db" + str(L)] = beta2*s["db" + str(L)] + (1-beta2) *
831 np.multiply(gradients['db' + str(L)].T,gradients['db' + str(L)].T)
832         # Compute bias-corrected second raw moment estimate.
833         s_corrected["dw" + str(L)] = s["dw" + str(L)]/(1-
834 np.power(beta2,t))
835         s_corrected["db" + str(L)] = s["db" + str(L)]/(1-
836 np.power(beta2,t))
837
838         # Update parameters.
839         d1=np.sqrt(s_corrected["dw" + str(L)]+epsilon)
840         d2=np.sqrt(s_corrected["db" + str(L)]+epsilon)
841         parameters["w" + str(L)] = parameters['w' + str(L)]- \
842             (learningRate* v_corrected["dw" + str(L)]/d1)
843         parameters["b" + str(L)] = parameters['b' + str(L)] - \
844             (learningRate* v_corrected["db" + str(L)]/d2)
845
846
847     return parameters, v, s
848
849
850 # Execute a L layer Deep learning model
851 # Input : X - Input features
852 #           : Y output
853 #           : layersDimensions - Dimension of layers
854 #           : hiddenActivationFunc - Activation function at hidden layer relu
855 /tanh/sigmoid
856 #           : outputActivationFunc - Activation function at output layer
857 sigmoid/softmax
858 #           : learning rate
859 #           : lambd
860 #           : keep_prob
861 #           : num of iteration
862 #           : initType
863 #output : parameters
864

```

```

865 def L_Layer_DeepModel(X1, Y1, layersDimensions, hiddenActivationFunc='relu',
866     outputActivationFunc="sigmoid",
867             learningRate = .3, lambd=0, keep_prob=1,
868     num_iterations = 10000,initType="default",
869     print_cost=False,figure="figa.png"):
870
871     np.random.seed(1)
872     costs = []
873
874     # Parameters initialization.
875     if initType == "He":
876         parameters = HeInitializeDeepModel(layersDimensions)
877     elif initType == "Xavier":
878         parameters = XavInitializeDeepModel(layersDimensions)
879     else: #Default
880         parameters = initializeDeepModel(layersDimensions)
881
882     # Loop (gradient descent)
883     for i in range(0, num_iterations):
884
885         AL, caches, dropoutMat = forwardPropagationDeep(X1, parameters,
886         keep_prob,
887         hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
888
889         # Regularization parameter is 0
890         if lambd==0:
891             # Compute cost
892             cost = computeCost(parameters,AL, Y1,
893             outputActivationFunc=outputActivationFunc)
894             # Include L2 regularization
895         else:
896             # Compute cost
897             cost = computeCostWithReg(parameters,AL, Y1, lambd,
898             outputActivationFunc=outputActivationFunc)
899
900         # Backward propagation.
901         gradients = backwardPropagationDeep(AL, Y1, caches, dropoutMat,
902         lambd, keep_prob,
903         hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
904
905         # Update parameters.
906         parameters = gradientDescent(parameters, gradients,
907         learningRate=learningRate,outputActivationFunc=outputActivationFunc)
908
909         # Print the cost every 100 training example
910         if print_cost and i % 1000 == 0:
911             print ("Cost after iteration %i: %f" %(i, cost))
912         if print_cost and i % 1000 == 0:
913             costs.append(cost)
914
915     # plot the cost
916     plt.plot(np.squeeze(costs))
917     plt.ylabel('Cost')
918     plt.xlabel('No of iterations (x1000)')
919     plt.title("Learning rate =" + str(learningRate))
920     plt.savefig(figure,bbox_inches='tight')
921     #plt.show()
922     plt.clf()
923     plt.close()
924
925     return parameters
926
927 # Execute a L layer Deep learning model Stoachastic Gradient Descent
928 # Input : X - Input features

```

```

929 #      : Y output
930 #      : layersDimensions - Dimension of layers
931 #      : hiddenActivationFunc - Activation function at hidden layer relu
932 #      : /tanh/sigmoid
933 #      : outputActivationFunc - Activation function at output -
934 #      : sigmoid/softmax
935 #      : learning rate
936 #      : lrDecay
937 #      : lambd
938 #      : keep_prob
939 #      : optimizer
940 #      : beta
941 #      : beta1
942 #      : beta2
943 #      : epsilon
944 #      : mini_batch_size
945 #      : num_epochs
946 #      :
947 #output : Updated weights and biases
948
949 def L_Layer_DeepModel_SGD(X1, Y1, layersDimensions,
950     hiddenActivationFunc='relu', outputActivationFunc="sigmoid",
951             learningRate = .3, lrDecay=False, decayRate=1,
952             lambd=0, keep_prob=1,
953     optimizer="gd", beta=0.9, beta1=0.9, beta2=0.999,
954             epsilon = 1e-8,mini_batch_size = 64, num_epochs =
955     2500, print_cost=False, figure="figa.png"):
956
957     print("lr=",learningRate)
958     print("lrDecay=",lrDecay)
959     print("decayRate=",decayRate)
960     print("lambd=",lambd)
961     print("keep_prob=",keep_prob)
962     print("optimizer=",optimizer)
963     print("beta=",beta)
964
965     print("beta1=",beta1)
966     print("beta2=",beta2)
967     print("epsilon=",epsilon)
968
969     print("mini_batch_size=",mini_batch_size)
970     print("num_epochs=",num_epochs)
971     print("epsilon=",epsilon)
972
973
974 t =0 # Adam counter
975 np.random.seed(1)
976 costs = []
977
978 # Parameters initialization.
979 parameters = initializeDeepModel(layersDimensions)
980
981 #Initialize the optimizer
982 if optimizer == "gd":
983     pass # no initialization required for gradient descent
984 elif optimizer == "momentum":
985     v = initializeVelocity(parameters)
986 elif optimizer == "rmsprop":
987     s = initializeRMSProp(parameters)
988 elif optimizer == "adam":
989     v,s = initializeAdam(parameters)
990
991 seed=10
992 # Loop for number of epochs

```

```

993     for i in range(num_epochs):
994         # Define the random minibatches. We increment the seed to reshuffle
995         # differently the dataset after each epoch
996         seed = seed + 1
997         minibatches = random_mini_batches(x1, Y1, mini_batch_size, seed)
998
999         batch=0
1000        # Loop through each mini batch
1001        for minibatch in minibatches:
1002            #print("batch=",batch)
1003            batch=batch+1
1004            # Select a minibatch
1005            (minibatch_X, minibatch_Y) = minibatch
1006
1007            # Perform forward propagation
1008            AL, caches, dropoutMat = forwardPropagationDeep(minibatch_X,
1009 parameters, keep_prob,
1010 hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
1011
1012            # Compute cost
1013            # Regularization parameter is 0
1014            if lambd==0:
1015                # Compute cost
1016                cost = computeCost(parameters, AL, minibatch_Y,
1017 outputActivationFunc=outputActivationFunc)
1018            else: # Include L2 regularization
1019                # Compute cost
1020                cost = computeCostWithReg(parameters, AL, minibatch_Y, lambd,
1021 outputActivationFunc=outputActivationFunc)
1022
1023            # Backward propagation.
1024            gradients = backwardPropagationDeep(AL, minibatch_Y,
1025 caches,dropoutMat, lambd,
1026 keep_prob,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFu
1027 nc)
1028
1029            if optimizer == "gd":
1030                # Update parameters normal gradient descent
1031                parameters = gradientDescent(parameters, gradients,
1032 learningRate=learningRate,outputActivationFunc=outputActivationFunc)
1033            elif optimizer == "momentum":
1034                # Update parameters for gradient descent with momentum
1035                parameters, v = gradientDescentWithMomentum(parameters,
1036 gradients, v, beta, \
1037
1038 learningRate=learningRate,outputActivationFunc=outputActivationFunc)
1039            elif optimizer == "rmsprop":
1040                # Update parameters for gradient descent with RMSProp
1041                parameters, s = gradientDescentWithRMSProp(parameters,
1042 gradients, s, beta1, epsilon, \
1043
1044 learningRate=learningRate,outputActivationFunc=outputActivationFunc)
1045            elif optimizer == "adam":
1046                t = t + 1 # Adam counter
1047                parameters, v, s = gradientDescentWithAdam(parameters,
1048 gradients, v, s,
1049                                         t, beta1,
1050 beta2, epsilon,
1051
1052 learningRate=learningRate,outputActivationFunc=outputActivationFunc)
1053
1054            # Print the cost every 1000 epoch
1055            if print_cost and i % 100 == 0:
1056                print ("Cost after epoch %i: %f" %(i, cost))

```

```

1057         if print_cost and i % 100 == 0:
1058             costs.append(cost)
1059         if lrDecay == True:
1060             learningRate = np.power(decayRate,(num_epochs/1000)) *
1061             learningRate
1062
1063
1064         # plot the cost
1065         plt.plot(np.squeeze(costs))
1066         plt.ylabel('Cost')
1067         plt.xlabel('No of epochs(x100)')
1068         plt.title("Learning rate =" + str(learningRate))
1069         plt.savefig(figure,bbox_inches='tight')
1070         #plt.show()
1071         plt.clf()
1072         plt.close()
1073
1074
1075 # Create random mini batches
1076 # Input : X - Input features
1077 #           : Y- output
1078 #           : miniBatchsizes
1079 #           : seed
1080 #output : mini_batches
1081 def random_mini_batches(X, Y, miniBatchsize = 64, seed = 0):
1082
1083     np.random.seed(seed)
1084     # Get number of training samples
1085     m = X.shape[1]
1086     # Initialize mini batches
1087     mini_batches = []
1088
1089     # Create a list of random numbers < m
1090     permutation = list(np.random.permutation(m))
1091     # Randomly shuffle the training data
1092     shuffled_X = X[:, permutation]
1093     shuffled_Y = Y[:, permutation].reshape((1,m))
1094
1095     # Compute number of mini batches
1096     numCompleteMinibatches = math.floor(m/miniBatchsize)
1097
1098     # For the number of mini batches
1099     for k in range(0, numCompleteMinibatches):
1100
1101         # Set the start and end of each mini batch
1102         mini_batch_X = shuffled_X[:, k*miniBatchSize : (k+1) * miniBatchsize]
1103         mini_batch_Y = shuffled_Y[:, k*miniBatchSize : (k+1) * miniBatchsize]
1104
1105         mini_batch = (mini_batch_X, mini_batch_Y)
1106         mini_batches.append(mini_batch)
1107
1108
1109     #if m % miniBatchSize != 0:. The batch does not evenly divide by the mini
1110     batch
1111     if m % miniBatchSize != 0:
1112         l=math.floor(m/miniBatchSize)*miniBatchSize
1113         # Set the start and end of last mini batch
1114         m=l+m % miniBatchSize
1115         mini_batch_X = shuffled_X[:,l:m]
1116         mini_batch_Y = shuffled_Y[:,l:m]
1117
1118         mini_batch = (mini_batch_X, mini_batch_Y)
1119         mini_batches.append(mini_batch)
1120

```

```

1121     return mini_batches
1122
1123
1124 # Plot a decision boundary
1125 # Input : Input Model,
1126 #           X
1127 #           Y
1128 #           sz - Num of hidden units
1129 #           lr - Learning rate
1130 #           Fig to be saved as
1131 # Returns Null
1132 def plot_decision_boundary(model, x, y, lr, figure1="figb.png"):
1133     print("plot")
1134     # Set min and max values and give it some padding
1135     x_min, x_max = x[0, :].min() - 1, x[0, :].max() + 1
1136     y_min, y_max = x[1, :].min() - 1, x[1, :].max() + 1
1137     colors=['black', 'gold']
1138     cmap = matplotlib.colors.ListedColormap(colors)
1139     h = 0.01
1140     # Generate a grid of points with distance h between them
1141     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,
1142                           h))
1143     # Predict the function value for the whole grid
1144     z = model(np.c_[xx.ravel(), yy.ravel()])
1145     z = z.reshape(xx.shape)
1146     # Plot the contour and training examples
1147     plt.contourf(xx, yy, z, cmap="coolwarm")
1148     plt.ylabel('x2')
1149     plt.xlabel('x1')
1150     x=X.T
1151     y=y.T.reshape(300, )
1152     plt.scatter(x[:, 0], x[:, 1], c=y, s=20);
1153     print(x.shape)
1154     plt.title("Decision Boundary for learning rate:"+str(lr))
1155     plt.savefig(figure1, bbox_inches='tight')
1156     #plt.show()
1157
1158 # Predict output
1159 def predict(parameters,
1160             x, keep_prob=1, hiddenActivationFunc="relu", outputActivationFunc="sigmoid"):
1161     A2, cache, dropoutMat = forwardPropagationDeep(x, parameters, keep_prob=1,
1162             hiddenActivationFunc=hiddenActivationFunc, outputActivationFunc=outputActivationFunc)
1163     predictions = (A2>0.5)
1164     return predictions
1165
1166 # Predict probability
1167 def predict_proba(parameters, x, outputActivationFunc="sigmoid"):
1168     A2, cache = forwardPropagationDeep(x, parameters)
1169     if outputActivationFunc=="sigmoid":
1170         proba=A2
1171     elif outputActivationFunc=="softmax":
1172         proba=np.argmax(A2, axis=0).reshape(-1,1)
1173         print("A2=", A2.shape)
1174     return proba
1175
1176 # Plot a decision boundary
1177 # Input : Input Model,
1178 #           X
1179 #           Y
1180 #           sz - Num of hidden units
1181 #           lr - Learning rate
1182 #           Fig to be saved as
1183 # Returns Null
1184 def plot_decision_boundary1(x, y, w1, b1, w2, b2, figure2="figc.png"):
```

```

1185     #plot_decision_boundary(lambda x: predict(parameters, x.T),
1186     x1,y1.T,str(0.3),"fig2.png")
1187     h = 0.02
1188     x_min, x_max = x[:, 0].min() - 1, x[:, 0].max() + 1
1189     y_min, y_max = x[:, 1].min() - 1, x[:, 1].max() + 1
1190     xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
1191                           np.arange(y_min, y_max, h))
1192     Z = np.dot(np.maximum(0, np.dot(np.c_[xx.ravel(), yy.ravel()], w1.T) +
1193     b1.T), w2.T) + b2.T
1194     Z = np.argmax(Z, axis=1)
1195     Z = Z.reshape(xx.shape)
1196
1197     fig = plt.figure()
1198     plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8)
1199     print(X.shape)
1200     y1=y.reshape(300,)
1201     plt.scatter(X[:, 0], X[:, 1], c=y1, s=40, cmap=plt.cm.Spectral)
1202     plt.xlim(xx.min(), xx.max())
1203     plt.ylim(yy.min(), yy.max())
1204     plt.savefig('figure2', bbox_inches='tight')
1205
1206 # Load the circles dataset
1207 def load_dataset():
1208     np.random.seed(1)
1209     train_X, train_Y = sklearn.datasets.make_circles(n_samples=300,
1210 noise=.05)
1211     np.random.seed(2)
1212     test_X, test_Y = sklearn.datasets.make_circles(n_samples=100, noise=.05)
1213     # Visualize the data
1214     print(train_X.shape)
1215     print(train_Y.shape)
1216     #plt.scatter(train_X[:, 0], train_X[:, 1], c=train_Y, s=40,
1217     cmap=plt.cm.Spectral);
1218     train_X = train_X.T
1219     train_Y = train_Y.reshape((1, train_Y.shape[0]))
1220     test_X = test_X.T
1221     test_Y = test_Y.reshape((1, test_Y.shape[0]))
1222     return train_X, train_Y, test_X, test_Y
1223
1224 #####
1225 # Note: Using dictionary_to_vector followed by vector_to_dictionary =>
1226 original dictionary
1227 #####
1228 # Convert a weight,biases dictionary to a vector
1229 # Input : parameter dictionary
1230 # Returns : vector
1231 def dictionary_to_vector(parameters):
1232     """
1233         Roll all our parameters dictionary into a single vector satisfying our
1234 specific required shape.
1235     """
1236     keys = []
1237     count = 0
1238     for key in parameters:
1239         # flatten parameter
1240         new_vector = np.reshape(parameters[key], (-1,1))
1241         keys = keys + [key]*new_vector.shape[0]
1242
1243         if count == 0:
1244             theta = new_vector
1245         else:
1246             theta = np.concatenate((theta, new_vector), axis=0)
1247         count = count + 1

```

```

1249     return theta, keys
1250
1251
1252 # Convert a gradient dictionary to a vector
1253 # Input : parameter
1254 #       : gradient dictionary
1255 # Returns : gradient vector
1256 def gradients_to_vector(parameters, gradients):
1257
1258     #Roll all our gradients dictionary into a single vector satisfying our
1259     specific required shape.
1260
1261     keyvals=[]
1262     L=len(parameters)//2
1263     count = 0
1264     for l in range(L):
1265         # flatten parameter
1266         keyvals.append('dw'+str(l+1))
1267         keyvals.append('db'+str(l+1))
1268
1269     for key in keyvals:
1270         new_vector = np.reshape(gradients[key], (-1,1))
1271
1272         if count == 0:
1273             theta = new_vector
1274         else:
1275             theta = np.concatenate((theta, new_vector), axis=0)
1276         count = count + 1
1277
1278     return theta
1279
1280 # Convert a vector to a dictionary
1281 # Input : parameter
1282 #       : theta
1283 # Returns : parameters1 (dictionary)
1284 def vector_to_dictionary(parameters, theta):
1285     #Unroll all our parameters dictionary from a single vector satisfying our
1286     specific required shape.
1287
1288     start=0
1289     parameters1 = {}
1290     #For key
1291     for key in parameters:
1292         (a,b) = parameters[key].shape
1293         # Create a dictionary
1294         parameters1[key]= theta[start:start+a*b].reshape((a,b))
1295         start=start+a*b
1296
1297     return parameters1
1298
1299 # Convert a vector to a dictionary
1300 # Input : parameter
1301 #       : theta
1302 # Returns : parameters1 (dictionary)
1303 def vector_to_dictionary2(parameters, theta):
1304     #Unroll all our parameters dictionary from a single vector satisfying our
1305     specific required shape.
1306
1307     start=0
1308     parameters2= {}
1309     # For key
1310     for key in parameters:
1311         (a,b) = parameters[key].shape
1312         # Create a key value pair

```

```

1313     parameters2['d'+key]= theta[start:start+a*b].reshape((a,b))
1314     start=start+a*b
1315
1316     return parameters2
1317
1318 # Perform a gradient check
1319 # Input : parameters
1320 #           : gradients
1321 #           : train_X
1322 #           : train_Y
1323 #           : epsilon
1324 #           : outputActivationFunc
1325 # Returns :
1326 def gradient_check_n(parameters, gradients, train_X, train_Y, epsilon = 1e-
1327 7,outputActivationFunc="sigmoid"):
1328     # Set-up variables
1329     parameters_values, _ = dictionary_to_vector(parameters)
1330     grad = gradients_to_vector(parameters,gradients)
1331     num_parameters = parameters_values.shape[0]
1332     J_plus = np.zeros((num_parameters, 1))
1333     J_minus = np.zeros((num_parameters, 1))
1334     gradapprox = np.zeros((num_parameters, 1))
1335
1336     # Compute gradapprox using 2 sided derivative
1337     for i in range(num_parameters):
1338         # Compute J_plus[i]. Inputs: "parameters_values, epsilon". Output =
1339         "J_plus[i]".
1340         thetaplus = np.copy(parameters_values)
1341         thetaplus[i][0] = thetaplus[i][0] + epsilon
1342         AL, caches, dropoutMat = forwardPropagationDeep(train_X,
1343         vector_to_dictionary(parameters,thetaplus), keep_prob=1,
1344
1345         hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
1346         J_plus[i] = computeCost(AL, train_Y,
1347         outputActivationFunc=outputActivationFunc)
1348
1349
1350         # Compute J_minus[i]. Inputs: "parameters_values, epsilon". Output =
1351         "J_minus[i]".
1352         thetaminus = np.copy(parameters_values)
1353         thetaminus[i][0] = thetaminus[i][0] - epsilon
1354         AL, caches, dropoutMat = forwardPropagationDeep(train_X,
1355         vector_to_dictionary(parameters,thetaminus), keep_prob=1,
1356
1357         hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
1358         J_minus[i] = computeCost(AL, train_Y,
1359         outputActivationFunc=outputActivationFunc)
1360
1361
1362         # Compute gradapprox[i]
1363         gradapprox[i] = (J_plus[i] - J_minus[i])/(2*epsilon)
1364
1365     # Compare gradapprox to backward propagation gradients by computing
1366     difference.
1367     numerator = np.linalg.norm(grad-gradapprox)
1368     denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)
1369     difference = numerator/denominator
1370
1371
1372     if difference > 1e-5:
1373         print ("\033[93m" + "There is a mistake in the backward propagation!
1374 difference = " + str(difference) + "\033[0m")
1375     else:

```

```

1376     print("\033[92m" + "Your backward propagation works perfectly fine!
1377 difference = " + str(difference) + "\033[0m")
1378     print(difference)
1379     print("\n")
1380     # Convert grad to dictionary
1381     m=vector_to_dictionary2(parameters,grad)
1382     print("Gradients from backprop")
1383     print(m)
1384     print("\n")
1385     # Convert gradapprox to dictionary
1386     n=vector_to_dictionary2(parameters,gradapprox)
1387     print("Gradapprox from gradient check")
1388     print(n)
1389

```

8.2 R

```

1 ##########
2 #####
3 #
4 # File   : DLfunctions8.R
5 # Author : Tinniam V Ganesh
6 # Date   : 6 May 2018
7 #
8 #####
9 #####
10 library(ggplot2)
11 library(PRROC)
12 library(dplyr)
13
14 # Compute the sigmoid of a vector
15 sigmoid <- function(z){
16   A <- 1/(1+ exp(-z))
17   cache<-z
18   retvals <- list("A"=A, "Z"=z)
19   return(retvals)
20 }
21
22 # This is the older version. Very performance intensive
23 reluOld <-function(z){
24   A <- apply(z, 1:2, function(x) max(0,x))
25   cache<-z
26   retvals <- list("A"=A, "Z"=z)
27   return(retvals)
28 }
29
30 # Compute the Relu of a vector (current version)
31 relu  <-function(z{
32   # Perform relu. Set values less than equal to 0 as 0
33   z[z<0]=0
34   A=z
35   cache<-z
36   retvals <- list("A"=A, "Z"=z)
37   return(retvals)
38 }
39
40 # Compute the tanh activation of a vector
41 tanhActivation <- function(z{

```

```

43   A <- tanh(Z)
44   cache<-Z
45   retvals <- list("A"=A, "Z"=Z)
46   return(retvals)
47 }
48
49 # Compute the softmax of a vector
50 softmax <- function(z){
51   # get unnormalized probabilities
52   exp_scores = exp(t(z))
53   # normalize them for each example
54   A = exp_scores / rowSums(exp_scores)
55   retvals <- list("A"=A, "Z"=Z)
56   return(retvals)
57 }
58
59 # Compute the derivative of Relu
60 # g'(z) = 1 if z >0 and 0 otherwise
61 reluDerivative <-function(dA, cache){
62   Z <- cache
63   dZ <- dA
64   # Create a logical matrix of values > 0
65   a <- Z > 0
66   # When z <= 0, you should set dz to 0 as well. Perform an element wise
67   multiply
68   dZ <- dZ * a
69   return(dZ)
70 }
71
72 # Compute the derivative of sigmoid
73 # Derivative g'(z) = a* (1-a)
74 sigmoidDerivative <- function(dA, cache){
75   Z <- cache
76   s <- 1/(1+exp(-Z))
77   dZ <- dA * s * (1-s)
78   return(dZ)
79 }
80
81 # Compute the derivative of tanh
82 # Derivative g'(z) = 1- a^2
83 tanhDerivative <- function(dA, cache){
84   Z = cache
85   a = tanh(Z)
86   dZ = dA * (1 - a^2)
87   return(dZ)
88 }
89
90 # Populate a matrix of 1s in rows where Y==1
91 # This may need to be extended for K classes. Currently
92 # supports K=3 & K=10
93 popMatrix <- function(Y,numClasses){
94   a=rep(0,times=length(Y))
95   Y1=matrix(a,nrow=length(Y),ncol=numClasses)
96   #Set the rows and columns as 1's where Y is the class value
97   if(numClasses==3){
98     Y1[Y==0,1]=1
99     Y1[Y==1,2]=1
100    Y1[Y==2,3]=1
101  } else if (numClasses==10){
102    Y1[Y==0,1]=1
103    Y1[Y==1,2]=1
104    Y1[Y==2,3]=1
105    Y1[Y==3,4]=1
106    Y1[Y==4,5]=1

```

```

107         Y1[Y==5,6]=1
108         Y1[Y==6,7]=1
109         Y1[Y==7,8]=1
110         Y1[Y==8,9]=1
111         Y1[Y==9,0]=1
112     }
113     return(Y1)
114 }
115
116 # Compute the softmax derivative
117 softmaxDerivative <- function(dA, cache ,y,numTraining,numClasses){
118   # Note : dA not used. dL/dZ = dL/dA * dA/dZ = pi-yi
119   Z <- cache
120   # Compute softmax
121   exp_scores = exp(t(Z))
122   # normalize them for each example
123   probs = exp_scores / rowSums(exp_scores)
124   # Create a matrix of zeros
125   Y1=popMatrix(y,numClasses)
126   #a=rep(0,times=length(Y))
127   #Y1=matrix(a,nrow=length(Y),ncol=numClasses)
128   #Set the rows and columns as 1's where Y is the class value
129   dz = probs-Y1
130   return(dz)
131 }
132
133
134 # Initialize model for L layers
135 # Input : List of units in each layer
136 # Returns: Initial weights and biases matrices for all layers
137 initializeDeepModel <- function(layerDimensions){
138   set.seed(2)
139
140   # Initialize empty list
141   layerParams <- list()
142
143   # Note the weight matrix at layer '1' is a matrix of size (1,1-1)
144   # The Bias is a vectors of size (1,1)
145
146   # Loop through the layer dimension from 1.. L
147   # Indices in R start from 1
148   for(l in 2:length(layersDimensions)){
149     # Initialize a matrix of small random numbers of size 1 x 1-1
150     # Create random numbers of size 1 x 1-1
151     w=rnorm(layersDimensions[1]*layersDimensions[1-1])*0.01
152     # Create a weight matrix of size 1 x 1-1 with this initial weights and
153     # Add to list W1,W2... WL
154     layerParams[[paste('w',l-1,sep="")]] = matrix(w,nrow=layersDimensions[1],
155                                                 ncol=layersDimensions[1-1])
156     layerParams[[paste('b',l-1,sep="")]] = matrix(rep(0,layersDimensions[1]),
157
158     nrow=layersDimensions[1],ncol=1)
159   }
160   return(layerParams)
161 }
162
163
164
165 # He Initialization model for L layers
166 # Input : List of units in each layer
167 # Returns: Initial weights and biases matrices for all layers
168 # He initialization multiplies the random numbers with
169 sqrt(2/layerDimensions[previouslayer])
170 HeInitializeDeepModel <- function(layerDimensions){
```

```

171 set.seed(2)
172
173 # Initialize empty list
174 layerParams <- list()
175
176 # Note the weight matrix at layer '1' is a matrix of size (1,1-1)
177 # The Bias is a vectors of size (1,1)
178
179 # Loop through the layer dimension from 1.. L
180 # Indices in R start from 1
181 for(l in 2:length(layersDimensions)){
182     # Initialize a matrix of small random numbers of size 1 x 1-1
183     # Create random numbers of size 1 x 1-1
184     w=rnorm(layersDimensions[l]*layersDimensions[l-1])
185
186     # Create a weight matrix of size 1 x 1-1 with this initial weights
187     and
188     # Add to list w1,w2... WL
189     # He initialization - Divide by sqrt(2/layersDimensions[previous
190     layer])
191     layerParams[[paste('w',l-1,sep="")]] =
192     matrix(w,nrow=layersDimensions[l],
193
194     ncol=layersDimensions[l-1])*sqrt(2/layersDimensions[l-1])
195     layerParams[[paste('b',l-1,sep="")]] =
196     matrix(rep(0,layersDimensions[l]),
197
198     nrow=layersDimensions[l],ncol=1)
199 }
200 return(layerParams)
201 }
202
203 # XavInitializeDeepModel Initialization model for L layers
204 # Input : List of units in each layer
205 # Returns: Initial weights and biases matrices for all layers
206 # He initialization multiplies the random numbers with
207 # sqrt(1/layersDimensions[previouslayer])
208 XavInitializeDeepModel <- function(layersDimensions){
209     set.seed(2)
210
211     # Initialize empty list
212     layerParams <- list()
213
214     # Note the weight matrix at layer '1' is a matrix of size (1,1-1)
215     # The Bias is a vectors of size (1,1)
216
217     # Loop through the layer dimension from 1.. L
218     # Indices in R start from 1
219     for(l in 2:length(layersDimensions)){
220         # Initialize a matrix of small random numbers of size 1 x 1-1
221         # Create random numbers of size 1 x 1-1
222         w=rnorm(layersDimensions[l]*layersDimensions[l-1])
223
224         # Create a weight matrix of size 1 x 1-1 with this initial weights
225         and
226         # Add to list w1,w2... WL
227         # He initialization - Divide by sqrt(2/layersDimensions[previous
228         layer])
229         layerParams[[paste('w',l-1,sep="")]] =
230         matrix(w,nrow=layersDimensions[l],
231
232         ncol=layersDimensions[l-1])*sqrt(1/layersDimensions[l-1])
233         layerParams[[paste('b',l-1,sep="")]] =
234         matrix(rep(0,layersDimensions[l]),
```

```

235 nrow=layerDimensions[1],ncol=1)
236     }
237     return(layerParams)
238 }
239
240
241 # Initialize velocity
242 # Input : parameters
243 # Returns: v -Initial velocity
244 initializevelocity <- function(parameters){
245
246     L <- length(parameters)/2
247     v <- list()
248
249     # Initialize velocity with the same dimensions as w
250     for(l in 1:L){
251         # Get the size of weight matrix
252         sz <- dim(parameters[[paste('w',l,sep="")]])
253         v[[paste('dw',l,sep="")]] = matrix(rep(0,sz[1]*sz[2]),
254                                         nrow=sz[1],ncol=sz[2])
255         #Get the size of bias matrix
256         sz <- dim(parameters[[paste('b',l,sep="")]])
257         v[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]*sz[2]),
258                                         nrow=sz[1],ncol=sz[2])
259     }
260
261     return(v)
262 }
263
264
265 # Initialize RMSProp
266 # Input : parameters
267 # Returns: s - Initial RMSProp
268 initializeRMSProp <- function(parameters){
269
270     L <- length(parameters)/2
271     s <- list()
272
273     # Initialize velocity with the same dimensions as w
274     for(l in 1:L){
275         # Get the size of weight matrix
276         sz <- dim(parameters[[paste('w',l,sep="")]])
277         s[[paste('dw',l,sep="")]] = matrix(rep(0,sz[1]*sz[2]),
278                                         nrow=sz[1],ncol=sz[2])
279         #Get the size of bias matrix
280         sz <- dim(parameters[[paste('b',l,sep="")]])
281         s[[paste('db',l,sep="")]] = matrix(rep(0,sz[1]*sz[2]),
282                                         nrow=sz[1],ncol=sz[2])
283     }
284
285     return(s)
286 }
287
288 # Initialize Adam
289 # Input : parameters
290 # Returns: (v,s) - Initial Adam parameters
291 initializeAdam <- function(parameters){
292
293     L <- length(parameters)/2
294     v <- list()
295     s <- list()
296
297     # Initialize velocity with the same dimensions as w
298     for(l in 1:L){

```

```

299      # Get the size of weight matrix
300      sz <- dim(parameters[[paste('w',l,sep="")]])
301      v[[paste('dw',l,sep="")]] = matrix(rep(0,sz[1]*sz[2]),
302                                              nrow=sz[1],ncol=sz[2])
303      s[[paste('dw',l,sep="")]] = matrix(rep(0,sz[1]*sz[2]),
304                                              nrow=sz[1],ncol=sz[2])
305      #Get the size of bias matrix
306      sz <- dim(parameters[[paste('b',l,sep="")]])
307      v[[paste('db',l,sep="")]] =  matrix(rep(0,sz[1]*sz[2]),
308                                              nrow=sz[1],ncol=sz[2])
309      s[[paste('db',l,sep="")]] =  matrix(rep(0,sz[1]*sz[2]),
310                                              nrow=sz[1],ncol=sz[2])
311  }
312  retvals <- list("v"=v,"s"=s)
313  return(retvals)
314 }
315
316 # Compute the activation at a layer 'l' for forward prop in a Deep Network
317 # Input : A_prev - Activation of previous layer
318 #          w,b - Weight and bias matrices and vectors
319 #          activationFunc - Activation function - sigmoid, tanh, relu etc
320 # Returns : The Activation of this layer
321 #
322 # Z = W * X + b
323 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
324 layerActivationForward <- function(A_prev, w, b, activationFunc){
325
326     # Compute Z
327     z = w %*% A_prev
328     # Broadcast the bias 'b' by column
329     Z <- sweep(z,1,b,'+')
330
331     forward_cache <- list("A_prev"=A_prev, "w"=w, "b"=b)
332     # Compute the activation for sigmoid
333     if(activationFunc == "sigmoid"){
334         vals = sigmoid(Z)
335     } else if (activationFunc == "relu"){ # Compute the activation for relu
336         vals = relu(Z)
337     } else if(activationFunc == 'tanh'){ # Compute the activation for tanh
338         vals = tanhActivation(Z)
339     } else if(activationFunc == 'softmax'){
340         vals = softmax(Z)
341     }
342     # Create a list of forward and activation cache
343     cache <- list("forward_cache"=forward_cache,
344 "activation_cache"=vals[['Z']])
345     retvals <- list("A"=vals[['A']], "cache"=cache)
346     return(retvals)
347 }
348
349 # Compute the forward propagation for layers 1..L
350 # Input : X - Input Features
351 #          parameters: weights and biases
352 #          keep_prob
353 #          hiddenActivationFunc - relu/sigmoid/tanh
354 #          outputActivationFunc - Activation function at hidden layer
355 #          sigmoid/softmax
356 # Returns : AL
357 #          caches
358 #          dropoutMat
359 # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and
360 # sigmoid actiovation at layer L
361 forwardPropagationDeep <- function(x, parameters,keep_prob=1,
362 hiddenActivationFunc='relu',

```

```

363                                         outputActivationFunc='sigmoid'){

364     caches <- list()
365     dropoutMat <- list()
366     # Set A to X (A0)
367     A <- X
368     L <- length(parameters)/2 # number of layers in the neural network
369     # Loop through from layer 1 to upto layer L
370     for(l in 1:(L-1)){
371         A_prev <- A
372         # Zi = Wi x Ai-1 + bi and Ai = g(Zi)
373         # Set W and b for layer 'l'
374         # Loop throug from W1,W2... WL-1
375         W <- parameters[[paste("w",l,sep="")]]
376         b <- parameters[[paste("b",l,sep="")]]
377         # Compute the forward propagation through layer 'l' using the activation
378         function
379         actForward <- layerActivationForward(A_prev,
380                                              W,
381                                              b,
382                                              activationFunc =
383         hiddenActivationFunc)
384         A <- actForward[['A']]
385         # Append the cache A_prev,W,b, Z
386         caches[[l]] <- actForward
387
388         # Randomly drop some activation units
389         # Create a matrix as the same shape as A
390         set.seed(1)
391         i=dim(A)[1]
392         j=dim(A)[2]
393         a<-rnorm(i*j)
394         # Normalize a between 0 and 1
395         a = (a - min(a))/(max(a) - min(a))
396         # Create a matrix of D
397         D <- matrix(a,nrow=i, ncol=j)
398         # Find D which is less than equal to keep_prob
399         D <- D < keep_prob
400         # Remove some A's
401         A <- A * D
402         # Divide by keep_prob to keep expected value same
403         A <- A/keep_prob
404         dropoutMat[[paste("D",l,sep="")]] <- D
405     }
406
407     # Since this is binary classification use the sigmoid activation function
408     in
409     # last layer
410     # Set the weights and biases for the last layer
411     W <- parameters[[paste("W",L,sep="")]]
412     b <- parameters[[paste("b",L,sep="")]]
413     # Last layer
414     actForward = layerActivationForward(A, W, b, activationFunc =
415     outputActivationFunc)
416     AL <- actForward[['A']]
417     # Append the output of this forward propagation through the last layer
418     caches[[L]] <- actForward
419     # Create a list of the final output and the caches
420     fwdPropDeep <- list("AL"=AL,"caches"=caches,"dropoutMat"=dropoutMat)
421     return(fwdPropDeep)
422
423 }
424
425 # Function pickColumns(). This function is in computeCost()
426 # Pick columns

```

```

427 # Input : AL
428 #      : Y
429 #      : numClasses
430 # Output: a
431 pickColumns <- function(AL,Y,numClasses){
432   if(numClasses==3){
433     a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3])
434   }
435   else if (numClasses==10){
436     a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3],AL[Y==3,4],AL[Y==4,5],
437          AL[Y==5,6],AL[Y==6,7],AL[Y==7,8],AL[Y==8,9],AL[Y==9,10])
438   }
439   return(a)
440 }
441
442
443 # Compute the cost
444 # Input : Activation of last layer
445 #      : Output from data
446 #      :outputActivationFunc - Activation function at hidden layer
447 sigmoid/softmax
448 #      : numClasses
449 # Output: cost
450 computeCost <- function(AL,Y,outputActivationFunc="sigmoid",numClasses=3){
451   if(outputActivationFunc=="sigmoid"){
452     m= length(Y)
453     cost=-1/m*sum(Y*log(AL) + (1-Y)*log(1-AL))
454
455   }else if (outputActivationFunc=="softmax"){
456     # Select the elements where the y values are 0, 1 or 2 and make a vector
457     # Pick columns
458     #a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3])
459     m= length(Y)
460     a =pickColumns(AL,Y,numClasses)
461     #a = c(A2[y=k,k+1])
462     # Take log
463     correct_probs = -log(a)
464
465     # Compute loss
466     cost= sum(correct_probs)/m
467   }
468   return(cost)
469 }
470
471
472
473
474 # Compute the cost with Regularization
475 # Input : parameters
476 #      : AL-Activation of last layer
477 #      : Y-Output from data
478 #      : lambd
479 #      : outputActivationFunc - Activation function at hidden layer
480 sigmoid/softmax
481 #      : numClasses
482 # Output: cost
483 computeCostWithReg <- function(parameters, AL,Y,lambd,
484 outputActivationFunc="sigmoid",numClasses=3){
485
486   if(outputActivationFunc=="sigmoid"){
487     m= length(Y)
488     cost=-1/m*sum(Y*log(AL) + (1-Y)*log(1-AL))
489
490     # Regularization cost

```

```

491     L <- length(parameters)/2
492     L2RegularizationCost=0
493     for(l in 1:L){
494         L2RegularizationCost = L2RegularizationCost +
495             sum(parameters[[paste("w",l,sep="")]]^2)
496     }
497     L2RegularizationCost = (lambd/(2*m))*L2RegularizationCost
498     cost = cost + L2RegularizationCost
499
500 }else if (outputActivationFunc=="softmax"){
501     # Select the elements where the y values are 0, 1 or 2 and make a
502     vector
503     # Pick columns
504     #a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3])
505     m= length(Y)
506     a =pickColumns(AL,Y,numClasses)
507     #a = c(A2[y=k,k+1])
508     # Take log
509     correct_probs = -log(a)
510     # Compute loss
511     cost= sum(correct_probs)/m
512
513     # Regularization cost
514     L <- length(parameters)/2
515     L2RegularizationCost=0
516     # Add L2 norm
517     for(l in 1:L){
518         L2RegularizationCost = L2RegularizationCost +
519             sum(parameters[[paste("W",l,sep="")]]^2)
520     }
521     L2RegularizationCost = (lambd/(2*m))*L2RegularizationCost
522     cost = cost + L2RegularizationCost
523 }
524 return(cost)
525 }
526
527 # Compute the backpropagation through a layer
528 # Input : Neural Network parameters - dA
529 #         # cache - forward_cache & activation_cache
530 #         # Input features
531 #         # Output values Y
532 #         # activationFunc
533 #         # numClasses
534 # Returns: Gradients
535 # dL/dwi= dL/dzi*A_l-1
536 # dL/dbl = dL/dz_l
537 # dL/dz_prev=dL/dz_l*w
538
539 layerActivationBackward <- function(dA, cache, Y,
540 activationFunc,numClasses){
541     # Get A_prev,w,b
542     forward_cache <-cache[['forward_cache']]
543     activation_cache <- cache[['activation_cache']]
544     A_prev <- forward_cache[['A_prev']]
545     numtraining = dim(A_prev)[2]
546     # Get Z
547     activation_cache <- cache[['activation_cache']]
548     if(activationFunc == "relu"){
549         dz <- reluDerivative(dA, activation_cache)
550     } else if(activationFunc == "sigmoid"){
551         dz <- sigmoidDerivative(dA, activation_cache)
552     } else if(activationFunc == "tanh"){
553         dz <- tanhDerivative(dA, activation_cache)
554     } else if(activationFunc == "softmax"){

```

```

555   dz <- softmaxDerivative(dA, activation_cache, Y, numtraining, numClasses)
556 }
557
558 if (activationFunc == 'softmax'){
559   w <- forward_cache[['w']]
560   b <- forward_cache[['b']]
561   dw = 1/numtraining * A_prev%*%dz
562   db = 1/numtraining* matrix(colSums(dz), nrow=1, ncol=numClasses)
563   dA_prev = dz %*% w
564 } else {
565   w <- forward_cache[['w']]
566   b <- forward_cache[['b']]
567   numtraining = dim(A_prev)[2]
568   dw = 1/numtraining * dz %*% t(A_prev)
569   db = 1/numtraining * rowSums(dz)
570   dA_prev = t(w) %*% dz
571 }
572 retvals <- list("dA_prev"=dA_prev, "dw"=dw, "db"=db)
573 return(retvals)
574 }
575
576 # Compute the backpropagation through a layer with Regularization
577 # Input : dA-Neural Network parameters
578 #          # cache - forward_cache & activation_cache
579 #          # Output values Y
580 #          # lambd
581 #          # activationFunc
582 #          # numClasses
583 # Returns: Gradients
584 # dL/dwi= dL/dzi*A1-1
585 # dL/dbl = dL/dz1
586 # dL/dz_prev=dL/dz1*w
587
588 layerActivationBackwardwithReg <- function(dA, cache, Y, lambd,
589 activationFunc, numClasses){
590   # Get A_prev,w,b
591   forward_cache <- cache[['forward_cache']]
592   activation_cache <- cache[['activation_cache']]
593   A_prev <- forward_cache[['A_prev']]
594   numtraining = dim(A_prev)[2]
595   # Get Z
596   activation_cache <- cache[['activation_cache']]
597   if(activationFunc == "relu"){
598     dz <- reluDerivative(dA, activation_cache)
599   } else if(activationFunc == "sigmoid"){
600     dz <- sigmoidDerivative(dA, activation_cache)
601   } else if(activationFunc == "tanh"){
602     dz <- tanhDerivative(dA, activation_cache)
603   } else if(activationFunc == "softmax"){
604     dz <- softmaxDerivative(dA,
605 activation_cache, Y, numtraining, numClasses)
606   }
607
608   if (activationFunc == 'softmax'){
609     w <- forward_cache[['w']]
610     b <- forward_cache[['b']]
611     # Add the regularization factor
612     dw = 1/numtraining * A_prev%*%dz + (lambd/numtraining) * t(w)
613     db = 1/numtraining* matrix(colSums(dz), nrow=1, ncol=numClasses)
614     dA_prev = dz %*% w
615   } else {
616     w <- forward_cache[['w']]
617     b <- forward_cache[['b']]
618     numtraining = dim(A_prev)[2]

```

```

619      # Add the regularization factor
620      dw = 1/numtraining * dz %*% t(A_prev) + (lambd/numtraining) * w
621      db = 1/numtraining * rowSums(dz)
622      dA_prev = t(w) %*% dz
623    }
624    retvals <- list("dA_prev"=dA_prev, "dw"=dw, "db"=db)
625    return(retvals)
626  }
627
628 # Compute the backpropagation for 1 cycle through all layers
629 # Input : AL: Output of L layer Network - weights
630 #           Y Real output
631 #           caches -- list of caches containing:
632 #             every cache of layerActivationForward() with "relu"/"tanh"
633 #             #(it's caches[], for l in range(L-1) i.e l = 0...L-2)
634 #             #the cache of layerActivationForward() with "sigmoid" (it's caches[L-
635 1])
636 #           dropoutMat
637 #           lambd
638 #           keep_prob
639 #           hiddenActivationFunc - Activation function at hidden layers -
640 #           relu/tanh/sigmoid
641 #           outputActivationFunc - Activation function at hidden layer
642 #           sigmoid/softmax
643 #           numClasses
644 #   Returns:
645 #     gradients -- A dictionary with the gradients
646 #                   gradients["dA" + str(l)]
647 #                   gradients["dw" + str(l)]
648 #                   gradients["db" + str(l)]
649 backwardPropagationDeep <- function(AL, Y, caches, dropoutMat, lambd=0,
650 keep_prob=0, hiddenActivationFunc='relu',
651                                         outputActivationFunc="sigmoid", numClasses){
652   #initialize the gradients
653   gradients = list()
654   # Set the number of layers
655   L = length(caches)
656   numTraining = dim(AL)[2]
657
658   if(outputActivationFunc == "sigmoid")
659     # Initializing the backpropagation
660     # d1/dAL= -(y/a) - ((1-y)/(1-a)) - At the output layer
661     dAL = -( (Y/AL) -(1 - Y)/(1 - AL))
662   else if(outputActivationFunc == "softmax"){
663     dAL=0
664     Y=t(Y)
665   }
666
667   # Get the gradients at the last layer
668   # Inputs: "AL, Y, caches".
669   # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dB_L"]
670   # Start with Layer L
671   # Get the current cache
672   current_cache = caches[[L]]$cache
673   if (lambd==0){
674     retvals <- layerActivationBackward(dAL, current_cache, Y,
675                                         activationFunc =
676                                         outputActivationFunc, numClasses)
677   } else {
678     retvals = layerActivationBackwardWithReg(dAL, current_cache, Y, lambd,
679                                         activationFunc =
680                                         outputActivationFunc, numClasses)
681   }

```

```

683
684
685 #Note: Take the transpose of dA
686 if(outputActivationFunc == "sigmoid")
687   gradients[[paste("dA",L,sep="")]] <- retvals[['dA_prev']]
688 else if(outputActivationFunc == "softmax")
689   gradients[[paste("dA",L,sep="")]] <- t(retvals[['dA_prev']])
690
691 gradients[[paste("dw",L,sep="")]] <- retvals[['dw']]
692 gradients[[paste("db",L,sep="")]] <- retvals[['db']]
693
694
695
696 # Traverse in the reverse direction
697 for(l in (L-1):1){
698   # Compute the gradients for L-1 to 1 for Relu/tanh
699   # Inputs: "gradients["dA" + str(l + 2)]", caches".
700   # Outputs: "gradients["dA" + str(l + 1)]", gradients["dw" + str(l + 1)]",
701   gradients["db" + str(l + 1)]
702   current_cache = caches[[1]]$cache
703   if (lambd==0){
704     cat("l=",l)
705     # Get the dropout matrix
706     D <- dropoutMat[[paste("D",l,sep="")]]
707     # Multiply gradient with dropout matrix
708     gradients[[paste('dA',l+1,sep="")]] =
709     gradients[[paste('dA',l+1,sep="")]] * D
710     # Divide by keep_prob to keep expected value same
711     gradients[[paste('dA',l+1,sep="")]] =
712     gradients[[paste('dA',l+1,sep="")]]/keep_prob
713     retvals =
714     layerActivationBackward(gradients[[paste('dA',l+1,sep="")]],
715                             current_cache, Y,
716                             activationFunc = hiddenActivationFunc)
717   } else {
718     retvals =
719     layerActivationBackwardWithReg(gradients[[paste('dA',l+1,sep="")]],
720                                   current_cache, Y, lambd,
721                                   activationFunc =
722     hiddenActivationFunc)
723   }
724
725   gradients[[paste("dA",l,sep="")]] <-retvals[['dA_prev']]
726   gradients[[paste("dw",l,sep="")]] <- retvals[['dw']]
727   gradients[[paste("db",l,sep="")]] <- retvals[['db']]
728 }
729
730
731
732   return(gradients)
733 }
734
735
736 # Perform Gradient Descent
737 # Input : Weights and biases
738 #       : gradients
739 #       : learning rate
740 #       : outputActivationFunc - Activation function at hidden layer
741 # sigmoid/softmax
742 #output : Updated weights after 1 iteration
743 gradientDescent <- function(parameters, gradients,
744   learningRate,outputActivationFunc="sigmoid"){
745
746   L = length(parameters)/2 # number of layers in the neural network

```

```

747 # Update rule for each parameter. Use a for loop.
748 for(l in 1:(L-1)){
749   parameters[[paste("w",l,sep="")]] = parameters[[paste("w",l,sep="")]] -
750     learningRate* gradients[[paste("dw",l,sep="")]]
751   parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] -
752     learningRate* gradients[[paste("db",l,sep="")]]
753 }
754 if(outputActivationFunc=="sigmoid"){
755   parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]] -
756     learningRate* gradients[[paste("dw",L,sep="")]]
757   parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] -
758     learningRate* gradients[[paste("db",L,sep="")]]
759 }
760 }else if (outputActivationFunc=="softmax"){
761   parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]] -
762     learningRate* t(gradients[[paste("dw",L,sep="")]])
763   parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] -
764     learningRate* t(gradients[[paste("db",L,sep="")]])
765 }
766 }
767 return(parameters)
768 }
769
770 # Perform Gradient Descent with momentum
771 # Input : weights and biases
772 #          : beta
773 #          : gradients
774 #          : learning rate
775 #          : outputActivationFunc - Activation function at hidden layer
776 sigmoid/softmax
777 #output : Updated weights after 1 iteration
778 gradientDescentwithMomentum <- function(parameters, gradients,v, beta,
779 learningRate,outputActivationFunc="sigmoid"){
780
781 L = length(parameters)/2 # number of layers in the neural network
782
783 # Update rule for each parameter. Use a for loop.
784 for(l in 1:(L-1)){
785   # Compute velocities
786   # v['dwk'] = beta *v['dwk'] + (1-beta)*dwk
787   v[[paste("dw",l, sep="")]] = beta*v[[paste("dw",l, sep="")]] +
788     (1-beta) * gradients[[paste('dw',l,sep="")]]
789   v[[paste("db",l, sep="")]] = beta*v[[paste("db",l, sep="")]] +
790     (1-beta) * gradients[[paste('db',l,sep="")]]
791
792   parameters[[paste("w",l,sep="")]] = parameters[[paste("w",l,sep="")]] -
793     learningRate* v[[paste("dw",l, sep="")]]
794   parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] -
795     learningRate* v[[paste("db",l, sep="")]]
796 }
797
798 # Compute for the Lth layer
799 if(outputActivationFunc=="sigmoid"){
800   v[[paste("dw",L, sep="")]] = beta*v[[paste("dw",L, sep="")]] +
801     (1-beta) * gradients[[paste('dw',L,sep="")]]
802   v[[paste("db",L, sep="")]] = beta*v[[paste("db",L, sep="")]] +
803     (1-beta) * gradients[[paste('db',L,sep="")]]
804
805   parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]] -
806     learningRate* v[[paste("dw",l, sep="")]]
807
808
809

```

```

810     parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
811     -
812     learningRate* v[[paste("db",l, sep="")]]
813
814 }else if (outputActivationFunc=="softmax"){
815   v[[paste("dw",L, sep="")]] = beta*v[[paste("dw",L, sep="")]] +
816   (1-beta) * t(gradients[[paste('dw',L,sep="")]])]
817   v[[paste("db",L, sep="")]] = beta*v[[paste("db",L, sep="")]] +
818   (1-beta) * t(gradients[[paste('db',L,sep="")]])]
819   parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
820   -
821   learningRate* t(gradients[[paste("dw",L,sep="")]])
822   parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
823   -
824   learningRate* t(gradients[[paste("db",L,sep="")]])
825 }
826 return(parameters)
827 }
828
829
830 # Perform Gradient Descent with RMSProp
831 # Input : parameters
832 #       : gradients
833 #       : s
834 #       : beta1
835 #       : epsilon
836 #       : learning rate
837 #       : outputActivationFunc - Activation function at hidden layer
838 # sigmoid/softmax
839 #output : Updated weights after 1 iteration
840 gradientDescentWithRMSProp <- function(parameters, gradients,s, beta1,
841 epsilon, learningRate,outputActivationFunc="sigmoid"){
842   L = length(parameters)/2 # number of layers in the neural network
843   # Update rule for each parameter. Use a for loop.
844   for(l in 1:(L-1)){
845     # Compute RMSProp
846     # s['dwk'] = beta1 *s['dwk'] + (1-beta1)*dwk**2/sqrt(s['dwk'])
847     # Element wise multiply of gradients
848     s[[paste("dw",l, sep="")]] = beta1*s[[paste("dw",l, sep="")]] +
849     (1-beta1) * gradients[[paste('dw',l,sep="")]] *
850     gradients[[paste('dw',l,sep="")]]
851     s[[paste("db",l, sep="")]] = beta1*s[[paste("db",l, sep="")]] +
852     (1-beta1) * gradients[[paste('db',l,sep="")]] *
853     gradients[[paste('db',l,sep="")]]
854
855     parameters[[paste("w",l,sep="")]] = parameters[[paste("w",l,sep="")]]
856     -
857     learningRate *
858     gradients[[paste('dw',l,sep="")]]/sqrt(s[[paste("dw",l, sep="")]]+epsilon)
859     parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]]
860     -
861     learningRate*gradients[[paste('db',l,sep="")]]/sqrt(s[[paste("db",l,
862     sep="")]]+epsilon)
863   }
864
865   # Compute for the Lth layer
866   if(outputActivationFunc=="sigmoid"){
867     s[[paste("dw",L, sep="")]] = beta1*s[[paste("dw",L, sep="")]] +
868     (1-beta1) * gradients[[paste('dw',L,sep="")]] *
869     gradients[[paste('dw',L,sep="")]]
870     s[[paste("db",L, sep="")]] = beta1*s[[paste("db",L, sep="")]] +
871     (1-beta1) * gradients[[paste('db',L,sep="")]] *
872     gradients[[paste('db',L,sep="")]]

```

```

874     parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
875     -
876         learningRate*
877     gradients[[paste('dw',l,sep="")]]/sqrt(s[[paste("dw",L, sep="")]]+epsilon)
878         parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
879     -
880         learningRate* gradients[[paste('db',l,sep="")]]/sqrt(
881     s[[paste("db",L, sep="")]]+epsilon)
882
883     }else if (outputActivationFunc=="softmax"){
884         s[[paste("dw",L, sep="")]] = beta1*s[[paste("dw",L, sep="")]] +
885             (1-beta1) * t(gradients[[paste('dw',L,sep="")]]) *
886             t(gradients[[paste('dw',L,sep="")]])
887         s[[paste("db",L, sep="")]] = beta1*s[[paste("db",L, sep="")]] +
888             (1-beta1) * t(gradients[[paste('db',L,sep="")]]) *
889             t(gradients[[paste('db',L,sep="")]])
890
891         parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
892     -
893         learningRate*
894     t(gradients[[paste("dw",L,sep="")]])/sqrt(s[[paste("dw",L, sep="")]]+epsilon)
895         parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
896     -
897         learningRate* t(gradients[[paste("db",L,sep="")]])/sqrt(
898     s[[paste("db",L, sep="")]]+epsilon)
899     }
900     return(parameters)
901 }
902
903
904 # Perform Gradient Descent with Adam
905 # Input : parameters
906 #       : gradients
907 #       : v
908 #       : s
909 #       : t
910 #       : beta1
911 #       : beta2
912 #       : epsilon
913 #       : learning rate
914 #       : outputActivationFunc - Activation function at hidden layer
915 #       : sigmoid/softmax
916 #output : Updated weights after 1 iteration
917 gradientDescentWithAdam <- function(parameters, gradients,v, s, t,
918                                         beta1=0.9, beta2=0.999, epsilon=10^-8,
919                                         learningRate=0.1, outputActivationFunc="sigmoid"){
920
921 L = length(parameters)/2 # number of layers in the neural network
922 v_corrected <- list()
923 s_corrected <- list()
924 # Update rule for each parameter. Use a for loop.
925 for(l in 1:(L-1)){
926     # v['dwk'] = beta *v['dwk'] + (1-beta)*dwk
927     v[[paste("dw",l, sep="")]] = beta1*v[[paste("dw",l, sep="")]] +
928         (1-beta1) * gradients[[paste('dw',l,sep="")]]
929     v[[paste("db",l, sep="")]] = beta1*v[[paste("db",l, sep="")]] +
930         (1-beta1) * gradients[[paste('db',l,sep="")]]
931
932     # Compute bias-corrected first moment estimate.
933     v_corrected[[paste("dw",l, sep="")]] = v[[paste("dw",l, sep="")]]/(1-
934     beta1^t)

```

```

937     v_corrected[[paste("db",1, sep="")]] = v[[paste("db",1, sep="")]]/(1-
938 beta1^t)
939
940
941         # Element wise multiply of gradients
942         s[[paste("dw",1, sep="")]] = beta2*s[[paste("dw",1, sep="")]] +
943             (1-beta2) * gradients[[paste('dw',1,sep="")]] *
944 gradients[[paste('dw',1,sep="")]]
945         s[[paste("db",1, sep="")]] = beta2*s[[paste("db",1, sep="")]] +
946             (1-beta2) * gradients[[paste('db',1,sep="")]] *
947 gradients[[paste('db',1,sep="")]]
948
949         # Compute bias-corrected second moment estimate.
950         s_corrected[[paste("dw",1, sep="")]] = s[[paste("dw",1, sep="")]]/(1-
951 beta2^t)
952         s_corrected[[paste("db",1, sep="")]] = s[[paste("db",1, sep="")]]/(1-
953 beta2^t)
954
955         # Update parameters.
956         d1=sqrt(s_corrected[[paste("dw",1, sep="")]]+epsilon)
957         d2=sqrt(s_corrected[[paste("db",1, sep="")]]+epsilon)
958
959         parameters[[paste("w",1,sep="")]] = parameters[[paste("w",1,sep="")]]-
960
961             learningRate * v_corrected[[paste("dw",1, sep="")]]/d1
962         parameters[[paste("b",1,sep="")]] = parameters[[paste("b",1,sep="")]]-
963
964             learningRate*v_corrected[[paste("db",1, sep="")]]/d2
965     }
966
967     # Compute for the Lth layer
968     if(outputActivationFunc=="sigmoid"){
969         v[[paste("dw",L, sep="")]] = beta1*v[[paste("dw",L, sep="")]] +
970             (1-beta1) * gradients[[paste('dw',L,sep="")]]
971         v[[paste("db",L, sep="")]] = beta1*v[[paste("db",L, sep="")]] +
972             (1-beta1) * gradients[[paste('db',L,sep="")]]
973
974
975         # Compute bias-corrected first moment estimate.
976         v_corrected[[paste("dw",L, sep="")]] = v[[paste("dw",L, sep="")]]/(1-
977 beta1^t)
978         v_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-
979 beta1^t)
980
981
982         # Element wise multiply of gradients
983         s[[paste("dw",L, sep="")]] = beta2*s[[paste("dw",L, sep="")]] +
984             (1-beta2) * gradients[[paste('dw',L,sep="")]] *
985 gradients[[paste('dw',L,sep="")]]
986         s[[paste("db",L, sep="")]] = beta2*s[[paste("db",L, sep="")]] +
987             (1-beta2) * gradients[[paste('db',L,sep="")]] *
988 gradients[[paste('db',L,sep="")]]
989
990         # Compute bias-corrected second moment estimate.
991         s_corrected[[paste("dw",L, sep="")]] = s[[paste("dw",L, sep="")]]/(1-
992 beta2^t)
993         s_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-
994 beta2^t)
995
996         # update parameters.
997         d1=sqrt(s_corrected[[paste("dw",L, sep="")]]+epsilon)
998         d2=sqrt(s_corrected[[paste("db",L, sep="")]]+epsilon)
999

```

```

1000     parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
1001     -
1002         learningRate * v_corrected[[paste("dw",L, sep="")]]/d1
1003     parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
1004     -
1005         learningRate*v_corrected[[paste("db",L, sep="")]]/d2
1006
1007 }else if (outputActivationFunc=="softmax"){
1008     v[[paste("dw",L, sep="")]] = beta1*v[[paste("dw",L, sep="")]] +
1009         (1-beta1) * t(gradients[[paste('dw',L,sep="")]])
1010     v[[paste("db",L, sep="")]] = beta1*v[[paste("db",L, sep="")]] +
1011         (1-beta1) * t(gradients[[paste('db',L,sep="")]])
1012
1013
1014 # Compute bias-corrected first moment estimate.
1015 v_corrected[[paste("dw",L, sep="")]] = v[[paste("dw",L, sep="")]]/(1-
1016 beta1^t)
1017 v_corrected[[paste("db",L, sep="")]] = v[[paste("db",L, sep="")]]/(1-
1018 beta1^t)
1019
1020
1021 # Element wise multiply of gradients
1022 s[[paste("dw",L, sep="")]] = beta2*s[[paste("dw",L, sep="")]] +
1023     (1-beta2) * t(gradients[[paste('dw',L,sep="")]]) *
1024 t(gradients[[paste('dw',L,sep="")]])
1025 s[[paste("db",L, sep="")]] = beta2*s[[paste("db",L, sep="")]] +
1026     (1-beta2) * t(gradients[[paste('db',L,sep="")]]) *
1027 t(gradients[[paste('db',L,sep="")]])
1028
1029 # Compute bias-corrected second moment estimate.
1030 s_corrected[[paste("dw",L, sep="")]] = s[[paste("dw",L, sep="")]]/(1-
1031 beta2^t)
1032 s_corrected[[paste("db",L, sep="")]] = s[[paste("db",L, sep="")]]/(1-
1033 beta2^t)
1034
1035 # Update parameters.
1036 d1=sqrt(s_corrected[[paste("dw",L, sep="")]]+epsilon)
1037 d2=sqrt(s_corrected[[paste("db",L, sep="")]]+epsilon)
1038
1039 parameters[[paste("w",L,sep="")]] = parameters[[paste("w",L,sep="")]]
1040
1041     learningRate * v_corrected[[paste("dw",L, sep="")]]/d1
1042     parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]]
1043
1044         learningRate*v_corrected[[paste("db",L, sep="")]]/d2
1045     }
1046     return(parameters)
1047 }
1048
1049 # Execute a L layer Deep learning model
1050 # Input : X - Input features
1051 #           : Y output
1052 #           : layersDimensions - Dimension of layers
1053 #           : hiddenActivationFunc - Activation function at hidden layer relu
1054 /tanh
1055 #           : outputActivationFunc - Activation function at hidden layer
1056 sigmoid/softmax
1057 #           : learning rate
1058 #           : lambd
1059 #           : keep_prob
1060 #           : learning rate
1061 #           : num of iterations
1062 #           : initType
1063 #output : Updated weights

```

```

1064
1065 L_Layer_DeepModel <- function(X, Y, layersDimensions,
1066                               hiddenActivationFunc='relu',
1067                               outputActivationFunc= 'sigmoid',
1068                               learningRate = 0.5,
1069                               lambd=0,
1070                               keep_prob=1,
1071                               numIterations = 10000,
1072                               initType="default",
1073                               print_cost=False){
1074     #Initialize costs vector as NULL
1075     costs <- NULL
1076
1077     # Parameters initialization.
1078     if (initType=="He"){
1079         parameters =HeInitializeDeepModel(layersDimensions)
1080     } else if (initType=="Xav"){
1081         parameters =XavInitializeDeepModel(layersDimensions)
1082     }
1083     else{
1084         parameters = initializeDeepModel(layersDimensions)
1085     }
1086
1087
1088     # Loop (gradient descent)
1089     for( i in 0:numIterations){
1090         # Forward propagation: [LINEAR -> RELU]^(L-1) -> LINEAR ->
1091 SIGMOID/SOFTMAX.
1092         retvals = forwardPropagationDeep(X, parameters,keep_prob,
1093 hiddenActivationFunc,
1094
1095 outputActivationFunc=outputActivationFunc)
1096         AL <- retvals[['AL']]
1097         caches <- retvals[['caches']]
1098         dropoutMat <- retvals[['dropoutMat']]
1099
1100         # Compute cost.
1101         if(lambd==0){
1102             cost <- computeCost(AL,
1103 Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
1104 h(layersDimensions))])
1105         } else {
1106             cost <- computeCostWithReg(parameters, AL, Y,lambd,
1107 outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
1108 layersDimensions))])
1109         }
1110         # Backward propagation.
1111         gradients = backwardPropagationDeep(AL, Y, caches, dropoutMat, lambd,
1112 keep_prob, hiddenActivationFunc,
1113
1114 outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
1115 layersDimensions))])
1116
1117         # Update parameters.
1118         parameters = gradientDescent(parameters, gradients, learningRate,
1119                                         outputActivationFunc=outputActivationFunc)
1120
1121
1122         if(i%%1000 == 0){
1123             costs=c(costs,cost)
1124             print(cost)
1125         }
1126     }
1127

```

```

1128     retvals <- list("parameters"=parameters,"costs"=costs)
1129
1130     return(retvals)
1131 }
1132
1133 # Execute a L layer Deep learning model with stochastic Gradient descent
1134 # Input : X - Input features
1135 #
1136 #           : Y output
1137 #           : layersDimensions - Dimension of layers
1138 #           : hiddenActivationFunc - Activation function at hidden layer relu
1139 #           : outputActivationFunc - Activation function at hidden layer
1140 #           : sigmoid/softmax
1141 #           : learning rate
1142 #           : lrDecay
1143 #           : decayRate
1144 #           : lambd
1145 #           : keep_prob
1146 #           : optimizer
1147 #           : beta
1148 #           : beta1
1149 #           : beta2
1150 #           : epsilon
1151 #           : mini_batch_size
1152 #           : num of epochs
1153 #output : Updated weights after each iteration
1154 L_Layer_DeepModel_SGD <- function(X, Y, layersDimensions,
1155                                     hiddenActivationFunc='relu',
1156                                     outputActivationFunc= 'sigmoid',
1157                                     learningRate = .3,
1158                                     lrDecay=FALSE,
1159                                     decayRate=1,
1160                                     lambd=0,
1161                                     keep_prob=1,
1162                                     optimizer="gd",
1163                                     beta=0.9,
1164                                     beta1=0.9,
1165                                     beta2=0.999,
1166                                     epsilon=10^-8,
1167                                     mini_batch_size = 64,
1168                                     num_epochs = 2500,
1169                                     print_cost=False){
1170
1171
1172
1173     print("Values")
1174     cat("learningRate= ",learningRate)
1175     cat("\n")
1176     cat("lambd=",lambd)
1177     cat("\n")
1178     cat("keep_prob=",keep_prob)
1179     cat("\n")
1180     cat("optimizer=",optimizer)
1181     cat("\n")
1182     cat("lrDecay=",lrDecay)
1183     cat("\n")
1184     cat("decayRate=",decayRate)
1185     cat("\n")
1186     cat("beta=",beta)
1187     cat("\n")
1188     cat("beta1=",beta1)
1189     cat("\n")
1190     cat("beta2=",beta2)
1191     cat("\n")

```

```

1192 cat("epsilon=", epsilon)
1193 cat("\n")
1194 cat("mini_batch_size=", mini_batch_size)
1195 cat("\n")
1196 cat("num_epochs=", num_epochs)
1197 cat("\n")
1198 set.seed(1)
1199 #Initialize costs vector as NULL
1200 costs <- NULL
1201 t <- 0
1202 # Parameters initialization.
1203 parameters = initializeDeepModel(layersDimensions)
1204
1205
1206 #Initialize the optimizer
1207
1208 if(optimizer == "momentum"){
1209   v <- initializeVelocity(parameters)
1210 } else if(optimizer == "rmsprop"){
1211   s <- initializeRMSProp(parameters)
1212 } else if (optimizer == "adam"){
1213   adamVals <- initializeAdam(parameters)
1214 }
1215
1216 seed=10
1217
1218 # Loop for number of epochs
1219 for( i in 0:num_epochs){
1220   seed=seed+1
1221   minibatches = random_mini_batches(X, Y, mini_batch_size, seed)
1222
1223   for(batch in 1:length(minibatches)){
1224
1225     mini_batch_X=minibatches[[batch]][['mini_batch_X']]
1226     mini_batch_Y=minibatches[[batch]][['mini_batch_Y']]
1227     # Forward propagation:
1228     retvals = forwardPropagationDeep(mini_batch_X,
parameters,keep_prob, hiddenActivationFunc,
1229
1230 outputActivationFunc=outputActivationFunc)
1231
1232     AL <- retvals[['AL']]
1233     caches <- retvals[['caches']]
1234     dropoutMat <- retvals[['dropoutMat']]
1235
1236     # Compute cost.
1237     # Compute cost.
1238     if(lambd==0){
1239       cost <- computeCost(AL,
1240     mini_batch_Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)])
1241     } else {
1242       cost <- computeCostWithReg(parameters, AL, Y,lambd,
1243     outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
1244     layersDimensions)])
1245     }
1246
1247     # Backward propagation.
1248     gradients = backwardPropagationDeep(AL, mini_batch_Y, caches,
1249     dropoutMat, lambd, keep_prob, hiddenActivationFunc,
1250
1251     outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(
1252     layersDimensions)])
1253
1254     if(optimizer == "gd"){
1255       # Update parameters.

```

```

1256         parameters = gradientDescent(parameters, gradients,
1257     learningRate,
1258
1259     outputActivationFunc=outputActivationFunc)
1260     }else if(optimizer == "momentum"){
1261         # Update parameters with Momentum
1262         parameters = gradientDescentWithMomentum(parameters,
1263     gradients,v,beta, learningRate,
1264
1265     outputActivationFunc=outputActivationFunc)
1266
1267     } else if(optimizer == "rmsprop"){
1268         # Update parameters with RMSProp
1269         parameters = gradientDescentWithRMSProp(parameters,
1270     gradients,s,beta1, epsilon,learningRate,
1271
1272     outputActivationFunc=outputActivationFunc)
1273
1274     } else if(optimizer == "adam"){
1275         # Update parameters with Adam
1276         #Get v and s
1277         t <- t+1
1278         v <- adamVals[['v']]
1279         s <- adamVals[['s']]
1280         parameters = gradientDescentWithAdam(parameters, gradients,v,
1281     s,t, beta1,beta2, epsilon,learningRate,
1282
1283     outputActivationFunc=outputActivationFunc)
1284
1285     }
1286
1287     if(i%1000 == 0){
1288         costs=c(costs,cost)
1289         print(cost)
1290
1291     if(lrDecay==TRUE){
1292         learningRate = decayRate^(num_epochs/1000) * learningRate
1293     }
1294
1295     retvals <- list("parameters"=parameters,"costs"=costs)
1296
1297     return(retvals)
1298 }
1299
1300 # Predict the output for given input
1301 # Input : parameters
1302 #           : X
1303 # Output: predictions
1304 predict <- function(parameters, x,keep_prob=1, hiddenActivationFunc='relu'){
1305
1306     fwdProp <- forwardPropagationDeep(x, parameters,keep_prob,
1307     hiddenActivationFunc)
1308     predictions <- fwdProp$AL>0.5
1309
1310     return (predictions)
1311 }
1312
1313
1314 # Plot a decision boundary
1315 # This function uses ggplot2
1316 plotDecisionBoundary <-
1317 function(z,retvals,keep_prob=1,hiddenActivationFunc="sigmoid",lr=0.5){
1318     # Find the minimum and maximum for the data
1319     xmin<-min(z[,1])

```

```

1320 xmax<-max(z[,1])
1321 ymin<-min(z[,2])
1322 ymax<-max(z[,2])
1323
1324 # Create a grid of values
1325 a=seq(xmin,xmax,length=100)
1326 b=seq(ymin,ymax,length=100)
1327 grid <- expand.grid(x=a, y=b)
1328 colnames(grid) <- c('x1', 'x2')
1329 grid1 <-t(grid)
1330 # Predict the output for this grid
1331 q <-predict(retvals$parameters,grid1,keep_prob=1, hiddenActivationFunc)
1332 q1 <- t(data.frame(q))
1333 q2 <- as.numeric(q1)
1334 grid2 <- cbind(grid,q2)
1335 colnames(grid2) <- c('x1', 'x2','q2')
1336
1337 z1 <- data.frame(z)
1338 names(z1) <- c("x1","x2","y")
1339 atitle=paste("Decision boundary for learning rate:",lr)
1340 # Plot the contour of the boundary
1341 ggplot(z1) +
1342   geom_point(data = z1, aes(x = x1, y = x2, color = y)) +
1343   stat_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2), alpha =
1344 0.9)+ 
1345   ggtitle(atitle) + scale_colour_gradientn(colours = brewer.pal(10,
1346 "Spectral"))
1347 }
1348
1349 # Predict the probability scores for given data set
1350 # Input : parameters
1351 # : X
1352 # Output: probability of output
1353 computeScores <- function(parameters, x,hiddenActivationFunc='relu'){
1354
1355 fwdProp <- forwardPropagationDeep(X, parameters,hiddenActivationFunc)
1356 scores <- fwdProp$AL
1357
1358 return (scores)
1359 }
1360
1361 # Create random mini batches
1362 # Input : X - Input features
1363 # : Y- output
1364 # : miniBatchSize
1365 # : seed
1366 #output : mini_batches
1367 random_mini_batches <- function(X, Y, miniBatchSize = 64, seed = 0){
1368
1369   set.seed(seed)
1370   # Get number of training samples
1371   m = dim(X)[2]
1372   # Initialize mini batches
1373   mini_batches = list()
1374
1375   # Create a list of random numbers < m
1376   permutation = c(sample(m))
1377   # Randomly shuffle the training data
1378   shuffled_X = X[, permutation]
1379   shuffled_Y = Y[1, permutation]
1380
1381   # Compute number of mini batches
1382   numCompleteMinibatches = floor(m/miniBatchSize)
1383   batch=0

```

```

1384     for(k in 0:(numCompleteMinibatches-1)){
1385         batch=batch+1
1386         # Set the lower and upper bound of the mini batches
1387         lower=(k*miniBatchSize)+1
1388         upper=((k+1) * miniBatchSize)
1389         mini_batch_X = shuffled_X[, lower:upper]
1390         mini_batch_Y = shuffled_Y[lower:upper]
1391         # Add it to the list of mini batches
1392         mini_batch =
1393         list("mini_batch_X"=mini_batch_X,"mini_batch_Y"=mini_batch_Y)
1394         mini_batches[[batch]] =mini_batch
1395
1396     }
1397
1398     # If the batch size does not divide evenly with mini batch size
1399     if(m %% miniBatchSize != 0){
1400         p=floor(m/miniBatchSize)*miniBatchSize
1401         # Set the start and end of last batch
1402         q=p+m %% miniBatchSize
1403         mini_batch_X = shuffled_X[, (p+1):q]
1404         mini_batch_Y = shuffled_Y[(p+1):q]
1405     }
1406     # Return the list of mini batches
1407     mini_batch =
1408     list("mini_batch_X"=mini_batch_X,"mini_batch_Y"=mini_batch_Y)
1409     mini_batches[[batch]] =mini_batch
1410
1411     return(mini_batches)
1412 }
1413
1414 # Plot a decision boundary
1415 # This function uses ggplot2
1416 plotDecisionBoundary1 <- function(z,parameters,keep_prob=1){
1417     xmin<-min(z[,1])
1418     xmax<-max(z[,1])
1419     ymin<-min(z[,2])
1420     ymax<-max(z[,2])
1421
1422     # Create a grid of points
1423     a=seq(xmin,xmax,length=100)
1424     b=seq(ymin,ymax,length=100)
1425     grid <- expand.grid(x=a, y=b)
1426     colnames(grid) <- c('x1', 'x2')
1427     grid1 <-t(grid)
1428
1429     retvals = forwardPropagationDeep(grid1, parameters,keep_prob, "relu",
1430                                         outputActivationFunc="softmax")
1431
1432
1433     AL <- retvals$AL
1434     # From the softmax probabilities pick the one with the highest
1435     probability
1436     q= apply(AL,1,which.max)
1437
1438     q1 <- t(data.frame(q))
1439     q2 <- as.numeric(q1)
1440     grid2 <- cbind(grid,q2)
1441     colnames(grid2) <- c('x1', 'x2','q2')
1442
1443     z1 <- data.frame(z)
1444     names(z1) <- c("x1", "x2","y")
1445     atitle=paste("Decision boundary")
1446     ggplot(z1) +
1447         geom_point(data = z1, aes(x = x1, y = x2, color = y)) +

```

```

1448     stat_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2),
1449 alpha = 0.9)+  

1450         ggtitle(atitle) + scale_colour_gradientn(colours = brewer.pal(10,
1451 "Spectral"))
1452 }
1453
1454 #####
1455 # Note: Using list_to_vector followed by vector_to_list => original list
1456 #####
1457 # Convert a weight,biases as a list to a vector
1458 # Input : parameter dictionary
1459 # Returns : vector
1460 list_to_vector <- function(parameters){
1461   vec <- NULL
1462   L=length(parameters)/2
1463   for(l in 1:L){
1464     vec1= as.vector(t(parameters[[paste('w',l,sep="")]])) #Take transpose
1465     vec1=as.matrix(vec1,nrow=length(vec1),ncol=1)
1466     vec2= as.vector(t(parameters[[paste('b',l,sep="")]])) #Take transpose
1467     vec2=as.matrix(vec2,nrow=length(vec2),ncol=1)
1468     vec <- rbind(vec,vec1)
1469     vec <- rbind(vec,vec2)
1470   }
1471   return(vec)
1472 }
1473
1474 # Convert a list of gradients to a vector
1475 # Input : parameter
1476 #       : gradient list
1477 # Returns : gradient vector
1478 gradients_to_vector <- function(parameters,gradients){
1479   vec <- NULL
1480   L=length(parameters)/2
1481   for(l in 1:L){
1482     vec1= as.vector(t(gradients[[paste('dw',l,sep="")]])) #Take transpose
1483     vec1=as.matrix(vec1,nrow=length(vec1),ncol=1)
1484     vec2= as.vector(t(gradients[[paste('db',l,sep="")]])) #Take transpose
1485     vec2=as.matrix(vec2,nrow=length(vec2),ncol=1)
1486     vec <- rbind(vec,vec1)
1487     vec <- rbind(vec,vec2)
1488   }
1489   return(vec)
1490 }
1491
1492 # Convert vector to a list
1493 # This should be a mirror copy of list_to_vector
1494 # Input : parameters
1495 #       : theta
1496 # Returns : parameters1 (list)
1497 vector_to_list <- function(parameters,theta){
1498   L<-length(parameters)/2
1499   start<-1
1500   parameters1 <- list()
1501   for(l in 1:L){
1502     m = dim(parameters[[paste('w',l,sep="")]])
1503     a = theta[start:(start+m[1]*m[2]-1),1]
1504     parameters1[[paste('w',l,sep="")]] = t(matrix(a,nrow=m[2],ncol=m[1]))
1505     start=start+m[1]*m[2]
1506     n = dim(parameters[[paste('b',l,sep="")]])
1507     b= theta[start:(start+n[1]*n[2]-1),1]
1508     parameters1[[paste('b',l,sep="")]]=t(matrix(b,nrow=n[2],ncol=n[1]))
1509     start=start+n[1]*n[2]
1510   }
1511 }
```

```

1512     return(parameters1)
1513 }
1514
1515 # Convert vector of gradients to list of gradients
1516 # Input : parameters
1517 #       : grads
1518 # Returns : gradients1 (list)
1519 vector_to_list2 <- function(parameters,grads){
1520   L<-length(parameters)/2
1521   start<-1
1522   gradients1 <- list()
1523   for(l in 1:L){
1524     m = dim(parameters[[paste('w',l,sep="")]])[1]
1525     a = grads[start:(start+m[1]*m[2]-1),1]
1526     gradients1[[paste('dw',l,sep="")]] = matrix(a,nrow=m[1],ncol=m[2])
1527     start=start+m[1]*m[2]
1528     n = dim(parameters[[paste('b',l,sep="")]])[1]
1529     b= grads[start:(start+n[1]*n[2]-1),1]
1530     gradients1[[paste('db',l,sep="")]]=matrix(b,nrow=n[1],ncol=n[2])
1531     start=start+n[1]*n[2]
1532   }
1533
1534   return(gradients1)
1535 }
1536
1537 # Compute L2Norm
1538 L2NormVec <- function(x) {
1539   sqrt(sum(x^2))
1540 }
1541
1542 # Perform Gradient check
1543 # Input : parameters
1544 #       : gradients
1545 #       : X
1546 #       : Y
1547 #       : epsilon
1548 #       : outputActivationFunc
1549 # Returns :
1550 gradient_check_n <- function(parameters, gradients, X, Y,
1551                               epsilon = 1e-7, outputActivationFunc="sigmoid"){
1552   # Convert parameters to a vector
1553   parameters_values = list_to_vector(parameters)
1554   # Convert gradients to a vector
1555   grad = gradients_to_vector(parameters,gradients)
1556   num_parameters = dim(parameters_values)[1]
1557   #Initialize
1558   J_plus = matrix(rep(0,num_parameters),
1559                   nrow=num_parameters,ncol=1)
1560   J_minus = matrix(rep(0,num_parameters),
1561                   nrow=num_parameters,ncol=1)
1562   gradapprox = matrix(rep(0,num_parameters),
1563                      nrow=num_parameters,ncol=1)
1564
1565   # Compute gradapprox
1566   for(i in 1:num_parameters){
1567     # Compute J_plus[i].
1568     thetaplus = parameters_values
1569     thetaplus[i][1] = thetaplus[i][1] + epsilon
1570     retvals = forwardPropagationDeep(X,
1571                                     vector_to_list(parameters,thetaplus), keep_prob=1,
1572                                     hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
1573
1574     AL <- retvals[['AL']]

```

```

1576         J_plus[i] = computeCost(AL, Y,
1577 outputActivationFunc=outputActivationFunc)
1578
1579
1580     # Compute J_minus[i].
1581     thetaminus = parameters_values
1582     thetaminus[i][1] = thetaminus[i][1] - epsilon
1583     retvals = forwardPropagationDeep(x,
1584 vector_to_list(parameters,thetaminus), keep_prob=1,
1585
1586 hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc)
1587     AL <- retvals[['AL']]
1588     J_minus[i] = computeCost(AL, Y,
1589 outputActivationFunc=outputActivationFunc)
1590
1591
1592     # Compute gradapprox[i]
1593     gradapprox[i] = (J_plus[i] - J_minus[i])/(2*epsilon)
1594 }
1595
1596 # Compare gradapprox to backward propagation gradients by computing
1597 difference.
1598 numerator = L2NormVec(grad-gradapprox)
1599 denominator = L2NormVec(grad) + L2NormVec(gradapprox)
1600 difference = numerator/denominator
1601 if(difference > 1e-5){
1602     cat("There is a mistake, the difference is too high",difference)
1603 } else{
1604     cat("The implementations works perfectly", difference)
1605 }
1606
1607 # This can be used to check the structure of gradients and gradapprox
1608 print("Gradients from backprop")
1609 m=vector_to_list2(parameters,grad)
1610 print(m)
1611 print("Grad approx from gradient check")
1612 n=vector_to_list2(parameters,gradapprox)
1613 print(n)
1614 }
```

8.3 Octave

```

1 #####
2 #####
3 #
4 # File   : DLfunctions8.R
5 # Author : Tinniam V Ganesh
6 # Date   : 6 May 2018
7 #
8 #####
9 #####
10 1;
11 # Define sigmoid function
12 function [A,cache] = sigmoid(z)
13     A = 1 ./ (1+ exp(-z));
14     cache=z;
15 end
16
17 # Define Relu function
18 function [A,cache] = relu(z)
19     A = max(0,z);
```

```

20     cache=z;
21 end
22
23 # Define Relu function
24 function [A,cache] = tanhAct(z)
25     A = tanh(z);
26     cache=z;
27 end
28
29 # Define Softmax function
30 function [A,cache] = softmax(z)
31     # get unnormalized probabilities
32     exp_scores = exp(z');
33     # normalize them for each example
34     A = exp_scores ./ sum(exp_scores,2);
35     cache=z;
36 end
37
38 # Define Stable Softmax function
39 function [A,cache] = stableSoftmax(z)
40     # Normalize by max value in each row
41     shiftz = z' - max(z',[],2);
42     exp_scores = exp(shiftz);
43     # normalize them for each example
44     A = exp_scores ./ sum(exp_scores,2);
45     #disp("sm")
46     #disp(A);
47     cache=z;
48 end
49
50 # Define Relu Derivative
51 function [dz] = reluDerivative(dA,cache)
52     z = cache;
53     dz = dA;
54     # Get elements that are greater than 0
55     a = (z > 0);
56     # Select only those elements where z > 0
57     dz = dz .* a;
58 end
59
60 # Define Sigmoid Derivative
61 function [dZ] = sigmoidDerivative(dA,cache)
62     z = cache;
63     s = 1 ./ (1+ exp(-z));
64     dZ = dA .* s .* (1-s);
65 end
66
67 # Define Tanh Derivative
68 function [dZ] = tanhDerivative(dA,cache)
69     z = cache;
70     a = tanh(z);
71     dZ = dA .* (1 - a .^ 2);
72 end
73
74 # Populate a matrix with 1s in rows where Y=1
75 # This function may need to be modified if K is not 3, 10
76 # This function is used in computing the softmax derivative
77 function [Y1] = popMatrix(Y,numClasses)
78     Y1=zeros(length(Y),numClasses);
79     if(numClasses==3) # For 3 output classes
80         Y1(Y==0,1)=1;
81         Y1(Y==1,2)=1;
82         Y1(Y==2,3)=1;
83     elseif(numClasses==10) # For 10 output classes

```

```

84         Y1(Y==0,1)=1;
85         Y1(Y==1,2)=1;
86         Y1(Y==2,3)=1;
87         Y1(Y==3,4)=1;
88         Y1(Y==4,5)=1;
89         Y1(Y==5,6)=1;
90         Y1(Y==6,7)=1;
91         Y1(Y==7,8)=1;
92         Y1(Y==8,9)=1;
93         Y1(Y==9,10)=1;
94
95     endif
96 end
97
98 # Define Softmax Derivative
99 function [dz] = softmaxDerivative(dA,cache,Y, numClasses)
100    Z = cache;
101    # get unnormalized probabilities
102    shiftZ = Z' - max(z',[],2);
103    exp_scores = exp(shiftZ);
104
105    # normalize them for each example
106    probs = exp_scores ./ sum(exp_scores,2);
107    # dz = pi- yi
108    yi=popMatrix(Y,numClasses);
109    dz=probs-yi;
110
111 end
112
113 # Define Stable Softmax Derivative
114 function [dZ] = stableSoftmaxDerivative(dA,cache,Y, numClasses)
115    Z = cache;
116    # get unnormalized probabilities
117    exp_scores = exp(Z');
118    # normalize them for each example
119    probs = exp_scores ./ sum(exp_scores,2);
120    # dz = pi- yi
121    yi=popMatrix(Y,numClasses);
122    dZ=probs-yi;
123
124 end
125
126 # Initialize model for L layers
127 # Input : List of units in each layer
128 # Returns: Initial weights and biases matrices for all layers
129 function [w b] = initializeDeepModel(layerDimensions)
130    rand ("seed", 3);
131    # note the weight matrix at layer 'l' is a matrix of size (l,l-1)
132    # The Bias is a vectors of size (l,1)
133
134    # Loop through the layer dimension from 1.. L
135    # Create cell arrays for weights and biases
136
137    for l =2:size(layerDimensions)(2)
138        w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*0.01; #
139        Multiply by .01
140        b{l-1} = zeros(layerDimensions(l),1);
141
142    endfor
143 end
144
145 # He Initialization for L layers
146 # Input : List of units in each layer
147 # Returns: Initial weights and biases matrices for all layers

```

```

148 function [w b] = HeInitializeDeepModel(layerDimensions)
149     rand ("seed", 3);
150     # note the weight matrix at layer '1' is a matrix of size (1,1-1)
151     # The Bias is a vectors of size (1,1)
152
153     # Loop through the layer dimension from 1.. L
154     # Create cell arrays for weights and biases
155
156     for l =2:size(layerDimensions)(2)
157         w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*%
158         sqrt(2/layerDimensions(l-1)); # Multiply by .01
159         b{l-1} = zeros(layerDimensions(l),1);
160
161     endfor
162 end
163
164 # Xavier Initialization for L layers
165 # Input : List of units in each layer
166 # Returns: Initial weights and biases matrices for all layers
167 function [w b] = XavInitializeDeepModel(layerDimensions)
168     rand ("seed", 3);
169     # note the weight matrix at layer '1' is a matrix of size (1,1-1)
170     # The Bias is a vectors of size (1,1)
171
172     # Loop through the layer dimension from 1.. L
173     # Create cell arrays for weights and biases
174
175     for l =2:size(layerDimensions)(2)
176         w{l-1} = rand(layerDimensions(l),layerDimensions(l-1))*%
177         sqrt(1/layerDimensions(l-1)); # Multiply by .01
178         b{l-1} = zeros(layerDimensions(l),1);
179
180     endfor
181 end
182
183 # Initialize velocity
184 # Input : weights, biases
185 # Returns: vdw, vdB - Initial velocity
186 function[vdw vdB] = initializeVelocity(weights, biases)
187
188     L = size(weights)(2) # Create an integer
189     # Initialize a cell array
190     v = {}
191
192     # Initialize velocity with the same dimensions as w
193     for l=1:L
194         sz = size(weights{l});
195         vdw{l} = zeros(sz(1),sz(2));
196         sz = size(biases{l});
197         vdB{l} =zeros(sz(1),sz(2));
198     endfor;
199 end
200
201 # Initialize RMSProp
202 # Input : weights, biases
203 # Returns: sdw, sdb - Initial RMSProp
204 function[sdw sdb] = initializeRMSProp(weights, biases)
205
206     L = size(weights)(2) # Create an integer
207     # Initialize a cell array
208     s = {}
209
210     # Initialize velocity with the same dimensions as w
211     for l=1:L

```

```

212     sz = size(weights{1});
213     sdw{1} = zeros(sz(1),sz(2));
214     sz = size(biases{1});
215     sdb{1} =zeros(sz(1),sz(2));
216   endfor;
217 end
218
219 # Initialize Adam
220 # Input : parameters
221 # Returns: vdw, vdB, sdw, sdb -Initial Adam
222 function[vdw vdB sdw sdb] = initializeAdam(weights, biases)
223
224 L = size(weights)(2) # Create an integer
225 # Initialize a cell array
226 s = {}
227
228 # Initialize velocity with the same dimensions as w
229 for l=1:L
230   sz = size(weights{l});
231   vdw{l} = zeros(sz(1),sz(2));
232   sdw{l} = zeros(sz(1),sz(2));
233   sz = size(biases{l});
234   sdb{l} =zeros(sz(1),sz(2));
235   vdB{l} =zeros(sz(1),sz(2));
236 endfor;
237
238 end
239
240 # Compute the activation at a layer 'l' for forward prop in a Deep Network
241 # Input : A_prev - Activation of previous layer
242 #           W,b - Weight and bias matrices and vectors
243 #           activationFunc - Activation function - sigmoid, tanh, relu etc
244 # Returns : The Activation of this layer
245 #           :
246 # Z = W * X + b
247 # A = sigmoid(Z), A= Relu(Z), A= tanh(Z)
248 function [A forward_cache activation_cache] = layerActivationForward(A_prev,
249 W, b, activationFunc)
250
251   # Compute Z
252   Z = W * A_prev +b;
253   # Create a cell array
254   forward_cache = {A_prev W b};
255   # Compute the activation for sigmoid
256   if (strcmp(activationFunc,"sigmoid"))
257     [A activation_cache] = sigmoid(Z);
258   elseif (strcmp(activationFunc, "relu")) # Compute the activation for
259   Relu
260     [A activation_cache] = relu(Z);
261   elseif(strcmp(activationFunc,'tanh')) # Compute the activation for
262   tanh
263     [A activation_cache] = tanhAct(Z);
264   elseif(strcmp(activationFunc,'softmax')) # Compute the activation for
265   tanh
266     #[A activation_cache] = softmax(Z);
267     [A activation_cache] = stableSoftmax(Z);
268   endif
269
270 end
271
272 # Compute the forward propagation for layers 1..L
273 # Input : X - Input Features
274 #           parameters: weights and biases
275 #           keep_prob

```

```

275      # hiddenActivationFunc - Activation function at hidden layers
276      Relu/tanh/sigmoid
277      # outputActivationFunc- sigmoid/softmax
278      # Returns : AL, forward_caches, activation_caches, dropoutMat
279      # The forward propagation uses the Relu/tanh activation from layer 1..L-1 and
280      # sigmoid activation at layer L
281      function [AL forward_caches activation_caches dropoutMat] =
282          forwardPropagationDeep(X, weights,biases, keep_prob=1,
283                                  hiddenActivationFunc='relu',
284                                  outputActivationFunc='sigmoid')
285          # Create an empty cell array
286          forward_caches = {};
287          activation_caches = {};
288          dropoutMat = {};
289          # Set A to X (A0)
290          A = X;
291          L = length(weights); # number of layers in the neural network
292          # Loop through from layer 1 to upto layer L
293          for l =1:L-1
294              A_prev = A;
295              # Zi = Wi x Ai-1 + bi and Ai = g(Zi)
296              W = weights{l};
297              b = biases{l};
298              [A forward_cache activation_cache] = layerActivationForward(A_prev,
299              w,b, activationFunc=hiddenActivationFunc);
300              D=rand(size(A)(1),size(A)(2));
301              D = (D < keep_prob) ;
302              # Multiply by DropoutMat
303              A=A.*D;
304              # Divide by keep_prob to keep expected value same
305              A = A ./ keep_prob;
306              # Store D
307              dropoutMat{l}=D;
308              forward_caches{l}=forward_cache;
309              activation_caches{l} = activation_cache;
310          endfor
311          # Since this is binary classification use the sigmoid activation function
312      in
313          # last layer
314          W = weights{L};
315          b = biases{L};
316          [AL, forward_cache activation_cache] = layerActivationForward(A, w,b,
317          activationFunc = outputActivationFunc);
318          forward_caches{L}=forward_cache;
319          activation_caches{L} = activation_cache;
320      end
321
322      # Pick columns where Y==1
323      # This function is used in computeCost
324      function [a] = pickColumns(AL,Y,numClasses)
325          if(numClasses==3)
326              a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)];
327          elseif (numClasses==10)
328              a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3);AL(Y==3,4);AL(Y==4,5);
329                  AL(Y==5,6); AL(Y==6,7);AL(Y==7,8);AL(Y==8,9);AL(Y==9,10)];
330          endif
331      end
332
333
334
335      # Compute the cost
336      # Input : AL-Activation of last layer
337      #           : Y-Output from data
338      #           : outputActivationFunc- sigmoid/softmax

```

```

339 #      : numClasses
340 # Output: cost
341 function [cost]= computeCost(AL, Y,
342 outputActivationFunc="sigmoid",numClasses)
343     if(strcmp(outputActivationFunc,"sigmoid"))
344         numTraining= size(Y)(2);
345         # Element wise multiply for logprobs
346         cost = -1/numTraining * sum((Y .* log(AL)) + (1-Y) .* log(1-AL));
347         #disp(cost);
348
349 elseif(strcmp(outputActivationFunc,'softmax'))
350     numTraining = size(Y)(2);
351     Y=Y';
352     # Select rows where Y=0,1, and 2 and concatenate to a long vector
353     #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)];
354     a =pickColumns(AL,Y,numClasses);
355
356     #Select the correct column for log prob
357     correct_probs = -log(a);
358     #Compute log loss
359     cost= sum(correct_probs)/numTraining;
360 endif
361 end
362
363 # Compute the cost with regularization
364 # Input : weights
365 #      : AL - Activation of last layer
366 #      : Output from data
367 #      : lambd
368 #      : outputActivationFunc- sigmoid/softmax
369 #      : numClasses
370 # Output: cost
371 function [cost]= computeCostWithReg(weights, AL, Y, lambd,
372 outputActivationFunc="sigmoid",numClasses)
373
374     if(strcmp(outputActivationFunc,"sigmoid"))
375         numTraining= size(Y)(2);
376         # Element wise multiply for logprobs
377         cost = -1/numTraining * sum((Y .* log(AL)) + (1-Y) .* log(1-AL));
378
379         # Regularization cost
380         L = size(weights)(2);
381         L2RegularizationCost=0;
382         for l=1:L
383             wtSqr = weights{l} .* weights{l};
384             #disp(sum(sum(wtSqr,1)));
385             L2RegularizationCost+=sum(sum(wtSqr,1));
386         endfor
387         L2RegularizationCost = (lambd/(2*numTraining))*L2RegularizationCost;
388         cost = cost + L2RegularizationCost ;
389
390
391 elseif(strcmp(outputActivationFunc,'softmax'))
392     numTraining = size(Y)(2);
393     Y=Y';
394     # Select rows where Y=0,1, and 2 and concatenate to a long vector
395     #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)];
396     a =pickColumns(AL,Y,numClasses);
397
398     #Select the correct column for log prob
399     correct_probs = -log(a);
400     #Compute log loss
401     cost= sum(correct_probs)/numTraining;
402         # Regularization cost

```

```

403     L = size(weights)(2);
404     L2RegularizationCost=0;
405     for l=1:L
406         # Compute L2 Norm
407         wtSqr = weights{l} .* weights{l};
408         #disp(sum(sum(wtSqr,1)));
409         L2RegularizationCost+=sum(sum(wtSqr,1));
410     endfor
411     L2RegularizationCost = (lambd/(2*numTraining))*L2RegularizationCost;
412     cost = cost + L2RegularizationCost ;
413   endif
414 end
415
416
417
418 # Compute the backpropagation for 1 cycle
419 # Input : Neural Network parameters - dA
420 #          # cache - forward_cache & activation_cache
421 #          # Input features
422 #          # Output values Y
423 #          # outputActivationFunc- sigmoid/softmax
424 #          # numClasses
425 # Returns: Gradients
426 # dL/dwi= dL/dzi*A{l-1}
427 # dL/dbl = dL/dz{l}
428 # dL/dz_{prev}=dL/dz{l}*w
429 function [dA_prev dw db] = layerActivationBackward(dA, forward_cache,
430 activation_cache, Y, activationFunc,numClasses)
431
432 A_prev = forward_cache{1};
433 W =forward_cache{2};
434 b = forward_cache{3};
435 numTraining = size(A_prev)(2);
436 if (strcmp(activationFunc,"relu"))
437     dz = reluDerivative(dA, activation_cache);
438 elseif (strcmp(activationFunc,"sigmoid"))
439     dz = sigmoidDerivative(dA, activation_cache);
440 elseif(strcmp(activationFunc, "tanh"))
441     dz = tanhDerivative(dA, activation_cache);
442 elseif(strcmp(activationFunc, "softmax"))
443     #dz = softmaxDerivative(dA, activation_cache,Y,numClasses);
444     dz = stableSoftmaxDerivative(dA, activation_cache,Y,numClasses);
445 endif
446
447
448 if (strcmp(activationFunc,"softmax"))
449     W =forward_cache{2};
450     b = forward_cache{3};
451     # Add the regularization factor
452     dw = 1/numTraining * A_prev * dz;
453     db = 1/numTraining * sum(dz,1);
454     dA_prev = dz*w;
455 else
456     W =forward_cache{2};
457     b = forward_cache{3};
458     # Add the regularization factor
459     dw = 1/numTraining * dz * A_prev';
460     db = 1/numTraining * sum(dz,2);
461     dA_prev = w'*dz;
462 endif
463
464 end
465
466 # Compute the backpropagation for 1 cycle

```

```

467 # Input : dA- Neural Network parameters
468 #      # cache - forward_cache & activation_cache
469 #      # Y-Output values
470 #      # outputActivationFunc- sigmoid/softmax
471 #      # numClasses
472 # Returns: Gradients
473 # dL/dwi= dL/dzi*A1-1
474 # dL/dbl = dL/dz1
475 # dL/dz_prev=dL/dz1*w
476 function [dA_prev dw db] = layerActivationBackwardWithReg(dA, forward_cache,
477 activation_cache, Y, lambd=0, activationFunc,numClasses)
478
479 A_prev = forward_cache{1};
480 w =forward_cache{2};
481 b = forward_cache{3};
482 numTraining = size(A_prev)(2);
483 if (strcmp(activationFunc,"relu"))
484     dz = reluDerivative(dA, activation_cache);
485 elseif (strcmp(activationFunc,"sigmoid"))
486     dz = sigmoidDerivative(dA, activation_cache);
487 elseif(strcmp(activationFunc, "tanh"))
488     dz = tanhDerivative(dA, activation_cache);
489 elseif(strcmp(activationFunc, "softmax"))
490     #dz = softmaxDerivative(dA, activation_cache,Y,numClasses);
491     dz = stableSoftmaxDerivative(dA, activation_cache,Y,numClasses);
492 endif
493
494 if (strcmp(activationFunc,"softmax"))
495     w =forward_cache{2};
496     b = forward_cache{3};
497     # Add the regularization factor
498     dw = 1/numTraining * A_prev * dz + (lambd/numTraining) * w';
499     db = 1/numTraining * sum(dz,1);
500     dA_prev = dz*w;
501 else
502     w =forward_cache{2};
503     b = forward_cache{3};
504     # Add the regularization factor
505     dw = 1/numTraining * dz * A_prev' + (lambd/numTraining) * w;
506     db = 1/numTraining * sum(dz,2);
507     dA_prev = w'*dz;
508 endif
509
510 end
511
512 # Compute the backpropagation with regularization for 1 cycle
513 # Input : dA-Neural Network parameters
514 #      # cache - forward_cache & activation_cache
515 #      # Y-Output values
516 #      # lambd
517 #      # outputActivationFunc- sigmoid/softmax
518 #      # numClasses
519 # Returns: Gradients
520 # dL/dwi= dL/dzi*A1-1
521 # dL/dbl = dL/dz1
522 # dL/dz_prev=dL/dz1*w
523 function [dA_prev dw db] = layerActivationBackwardWithReg(dA, forward_cache,
524 activation_cache, Y, lambd=0, activationFunc,numClasses)
525
526 A_prev = forward_cache{1};
527 w =forward_cache{2};
528 b = forward_cache{3};
529 numTraining = size(A_prev)(2);
530 if (strcmp(activationFunc,"relu"))

```

```

531     dz = reluDerivative(dA, activation_cache);
532 elseif (strcmp(activationFunc,"sigmoid"))
533     dz = sigmoidDerivative(dA, activation_cache);
534 elseif(strcmp(activationFunc, "tanh"))
535     dz = tanhDerivative(dA, activation_cache);
536 elseif(strcmp(activationFunc, "softmax"))
537     #dz = softmaxDerivative(dA, activation_cache,Y,numClasses);
538     dz = stableSoftmaxDerivative(dA, activation_cache,Y,numClasses);
539 endif
540
541 if (strcmp(activationFunc,"softmax"))
542     w =forward_cache{2};
543     b = forward_cache{3};
544     # Add the regularization factor
545     dw = 1/numTraining * A_prev * dz + (lambd/numTraining) * w';
546     db = 1/numTraining * sum(dz,1);
547     dA_prev = dz*w;
548 else
549     w =forward_cache{2};
550     b = forward_cache{3};
551     # Add the regularization factor
552     dw = 1/numTraining * dz * A_prev' + (lambd/numTraining) * w;
553     db = 1/numTraining * sum(dz,2);
554     dA_prev = w'*dz;
555 endif
556
557 end
558
559 # Compute the backpropagation for 1 cycle
560 # Input : AL: Output of L layer Network - weights
561 #          Y Real output
562 #          caches -- list of caches containing:
563 #          every cache of layerActivationForward() with "relu"/"tanh"
564 #          #(it's caches[1], for l in range(L-1) i.e l = 0...L-2)
565 #          #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1])
566 #
567 #          dropoutMat
568 #          lambd
569 #          keep_prob
570 #          hiddenActivationFunc - Activation function at hidden layers
571 sigmoid/tanh/relu
572 #          outputActivationFunc- sigmoid/softmax
573 #          numClasses
574 #
575 # Returns:
576 #     gradients -- A dictionary with the gradients
577 #                 gradients["dA" + str(l)] = ...
578 #                 gradients["dw" + str(l)] = ...
579 #                 gradients["db" + str(l)] = ...
580
581 function [gradsDA gradsDW gradsDB]= backwardPropagationDeep(AL, Y,
582 activation_caches,forward_caches,
583                         dropoutMat, lambd=0, keep_prob=1,
584 hiddenActivationFunc='relu',outputActivationFunc="sigmoid",numClasses)
585
586
587 # Set the number of layers
588 L = length(activation_caches);
589 m = size(AL)(2);
590
591 if (strcmp(outputActivationFunc,"sigmoid"))
592     # Initializing the backpropagation
593     # d1/dAL= -(y/a + (1-y)/(1-a)) - At the output layer
594     dAL = -((Y ./ AL) - (1 - Y) ./ (1 - AL));

```

```

595     elseif (strcmp(outputActivationFunc,"softmax"))
596         dAL=0;
597         Y=Y';
598     endif
599
600
601     # Since this is a binary classification the activation at output is
602     sigmoid
603     # Get the gradients at the last layer
604     # Inputs: "AL, Y, caches".
605     # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"]
606     activation_cache = activation_caches{L};
607     forward_cache = forward_caches(L);
608     # Note the cell array includes an array of forward caches. To get to this
609     we need to include the index {1}
610     if (lambd==0)
611         [dA dw db] = layerActivationBackward(dAL, forward_cache{1},
612 activation_cache, Y, activationFunc = outputActivationFunc,numClasses);
613     else
614         [dA dw db] = layerActivationBackwardWithReg(dAL, forward_cache{1},
615 activation_cache, Y, lambd, activationFunc =
616 outputActivationFunc,numClasses);
617     endif
618     if (strcmp(outputActivationFunc,"sigmoid"))
619         gradsDA{L}= dA;
620     elseif (strcmp(outputActivationFunc,"softmax"))
621         gradsDA{L}= dA';#Note the transpose
622     endif
623     gradsDW{L}= dw;
624     gradsDB{L}= db;
625
626     # Traverse in the reverse direction
627     for l =(L-1):-1:1
628         # Compute the gradients for L-1 to 1 for Relu/tanh
629         # Inputs: "gradients["dA" + str(l + 2)], caches".
630         # Outputs: "gradients["dA" + str(l + 1)] , gradients["dw" + str(l +
631 1)], gradients["db" + str(l + 1)]
632         activation_cache = activation_caches{l};
633         forward_cache = forward_caches(l);
634
635         #dA_prev_temp, dw_temp, db_temp =
636         layerActivationBackward(gradients['dA'+str(l+1)], current_cache,
637 activationFunc = "relu")
638         # dAl the derivative of the activation of the lth layer, is the first
639 element
640         dAl= gradsDA{l+1};
641         if(lambd == 0)
642             # Get the dropout mat
643             D = dropoutMat{l};
644             #Multiply by the dropoutMat
645             dAl= dAl .* D;
646             # Divide by keep_prob to keep expected value same
647             dAl = dAl ./ keep_prob;
648             [dA_prev_temp, dw_temp, db_temp] = layerActivationBackward(dAl,
649 forward_cache{1}, activation_cache, Y, activationFunc =
650 hiddenActivationFunc,numClasses);
651         else
652             [dA_prev_temp, dw_temp, db_temp] =
653             layerActivationBackwardWithReg(dAl, forward_cache{1}, activation_cache, Y,
654 lambd, activationFunc = hiddenActivationFunc,numClasses);
655         endif
656         gradsDA{l}= dA_prev_temp;
657         gradsDW{l}= dw_temp;
658         gradsDB{l}= db_temp;

```

```

659     endfor
660
661 end
662
663
664 # Perform Gradient Descent
665 # Input : Weights and biases
666 #      : gradients -gradsW,gradsB
667 #      : learning rate
668 #      : outputActivationFunc
669 #output : Updated weights after 1 iteration
670 function [weights biases] = gradientDescent(weights, biases,gradsW,gradsB,
671 learningRate,outputActivationFunc="sigmoid")
672
673 L = size(weights)(2); # number of layers in the neural network
674 # Update rule for each parameter.
675 for l=1:(L-1)
676     weights{l} = weights{l} -learningRate* gradsW{l};
677     biases{l} = biases{l} -learningRate* gradsB{l};
678 endfor
679
680
681 if (strcmp(outputActivationFunc,"sigmoid"))
682     weights{L} = weights{L} -learningRate* gradsW{L};
683     biases{L} = biases{L} -learningRate* gradsB{L};
684 elseif (strcmp(outputActivationFunc,"softmax"))
685     weights{L} = weights{L} -learningRate* gradsW{L}';
686     biases{L} = biases{L} -learningRate* gradsB{L}';
687 endif
688
689
690 end
691
692
693 # Update parameters with momentum
694 # Input : parameters
695 #      : gradients -gradsDW,gradsDB
696 #      : v -vdW, vdB
697 #      : beta
698 #      : learningRate
699 #      : outputActivationFunc
700 #output : Updated weights, biases
701 function [weights biases] = gradientDescentWithMomentum(weights,
702 biases,gradsDW,gradsDB, vDW, vdB, beta,
703 learningRate,outputActivationFunc="sigmoid")
704 L = size(weights)(2); # number of layers in the neural network
705 # Update rule for each parameter.
706 for l=1:(L-1)
707     # Compute velocities
708     # v['dwk'] = beta *v['dwk'] + (1-beta)*dwk
709     vDW{l} = beta*vdW{l} + (1 -beta) * gradsDW{l};
710     vdB{l} = beta*vdB{l} + (1 -beta) * gradsDB{l};
711     weights{l} = weights{l} -learningRate* vDW{l};
712     biases{l} = biases{l} -learningRate* vdB{l};
713 endfor
714
715 if (strcmp(outputActivationFunc,"sigmoid"))
716     vDW{L} = beta*vdW{L} + (1 -beta) * gradsDW{L};
717     vdB{L} = beta*vdB{L} + (1 -beta) * gradsDB{L};
718     weights{L} = weights{L} -learningRate* vDW{L};
719     biases{L} = biases{L} -learningRate* vdB{L};
720 elseif (strcmp(outputActivationFunc,"softmax"))
721     vDW{L} = beta*vdW{L} + (1 -beta) * gradsDW{L}';
722

```

```

723         vdB{L} = beta*vdB{L} + (1 -beta) * gradsDB{L}' ;
724         weights{L} = weights{L} -learningRate* vdw{L};
725         biases{L} = biases{L} -learningRate* vdB{L};
726     endif
727
728
729 end
730
731
732 # Update parameters with RMSProp
733 # Input : parameters - weights, biases
734 #           : gradients - gradsDW,gradsDB
735 #           : s -sdw, sdB
736 #           : beta1
737 #           : epsilon
738 #           : learningRate
739 #           : outputActivationFunc
740 #output : Updated weights and biases RMSProp
741 function [weights biases] = gradientDescentWithRMSProp(weights,
742 biases,gradsDW,gradsDB, sdW, sdB, beta1, epsilon,
743 learningRate,outputActivationFunc="sigmoid")
744 L = size(weights)(2); # number of layers in the neural network
745 # Update rule for each parameter.
746 for l=1:(L-1)
747     sdW{l} = beta1*sdW{l} + (1 -beta1) * gradsDW{l} .* gradsDW{l};
748     sdB{l} = beta1*sdB{l} + (1 -beta1) * gradsDB{l} .* gradsDB{l};
749     weights{l} = weights{l} - learningRate* gradsDW{l} ./ sqrt(sdW{l}) +
750 epsilon;
751     biases{l} = biases{l} - learningRate* gradsDB{l} ./ sqrt(sdB{l}) +
752 epsilon;
753 endfor
754
755 if (strcmp(outputActivationFunc,"sigmoid"))
756     sdW{L} = beta1*sdW{L} + (1 -beta1) * gradsDW{L} .* gradsDW{L};
757     sdB{L} = beta1*sdB{L} + (1 -beta1) * gradsDB{L} .* gradsDB{L};
758     weights{L} = weights{L} -learningRate* gradsDW{L} ./ sqrt(sdW{L})
759 +epsilon;
760     biases{L} = biases{L} -learningRate* gradsDB{L} ./ sqrt(sdB{L}) +
761 epsilon;
762 elseif (strcmp(outputActivationFunc,"softmax"))
763     sdW{L} = beta1*sdW{L} + (1 -beta1) * gradsDW{L}' .* gradsDW{L}';
764     sdB{L} = beta1*sdB{L} + (1 -beta1) * gradsDB{L}' .* gradsDB{L}';
765     weights{L} = weights{L} -learningRate* gradsDW{L}' ./ sqrt(sdW{L})
766 +epsilon;
767     biases{L} = biases{L} -learningRate* gradsDB{L}' ./ sqrt(sdB{L}) +
768 epsilon;
769 endif
770
771
772 end
773
774
775 # Update parameters with Adam
776 # Input : parameters - weights, biases
777 #           : gradients -gradsDW,gradsDB
778 #           : v - vdw, vdB
779 #           : s - sdw, sdB
780 #           : t
781 #           : beta1
782 #           : beta2
783 #           : epsilon
784 #           : learningRate
785 #           : epsilon
786 #output : Updated weights and biases

```

```

787 function [weights biases] = gradientDescentWithAdam(weights,
788 biases,gradsDW,gradsDB,
789 vdw, vdB, sdw, sdB, t, beta1, beta2, epsilon,
790 learningRate,outputActivationFunc="sigmoid")
791     vdw_corrected = {};
792     vdB_corrected = {};
793     sdw_corrected = {};
794     sdB_corrected = {};
795     L = size(weights)(2); # number of layers in the neural network
796     # Update rule for each parameter.
797     for l=1:(L-1)
798         vdw{l} = beta1*vdw{l} + (1 -beta1) * gradsDW{l};
799         vdB{l} = beta1*vdB{l} + (1 -beta1) * gradsDB{l};
800
801         # Compute bias-corrected first moment estimate.
802         vdw_corrected{l} = vdw{l}/(1-beta1^t);
803         vdB_corrected{l} = vdB{l}/(1-beta1^t);
804
805         sdw{l} = beta2*sdw{l} + (1 -beta2) * gradsDW{l} .* gradsDW{l};
806         sdB{l} = beta2*sdB{l} + (1 -beta2) * gradsDB{l} .* gradsDB{l};
807
808         # Compute bias-corrected second moment estimate.
809         sdw_corrected{l} = sdw{l}/(1-beta2^t);
810         sdB_corrected{l} = sdB{l}/(1-beta2^t);
811
812         # Update parameters.
813         d1=sqrt(sdw_corrected{l}+epsilon);
814         d2=sqrt(sdB_corrected{l}+epsilon);
815
816         weights{l} = weights{l} - learningRate* vdw_corrected{l} ./ d1;
817         biases{l} = biases{l} -learningRate* vdB_corrected{l} ./ d2;
818     endfor
819
820     if (strcmp(outputActivationFunc,"sigmoid"))
821         vdw{L} = beta1*vdw{L} + (1 -beta1) * gradsDW{L};
822         vdB{L} = beta1*vdB{L} + (1 -beta1) * gradsDB{L};
823
824         # Compute bias-corrected first moment estimate.
825         vdw_corrected{L} = v{L}/(1-beta1^t);
826         vdB_corrected{L} = v{L}/(1-beta1^t);
827
828         sdw{L} = beta2*sdw{L} + (1 -beta2) * gradsDW{L} .* gradsDW{L};
829         sdB{L} = beta2*sdB{L} + (1 -beta2) * gradsDB{L} .* gradsDB{L};
830
831         # Compute bias-corrected second moment estimate.
832         sdw_corrected{L} = s{L}/(1-beta2^t);
833         sdB_corrected{L} = s{L}/(1-beta2^t);
834
835         # Update parameters.
836         d1=sqrt(sdw_corrected{L}+epsilon);
837         d2=sqrt(sdB_corrected{L}+epsilon);
838
839         weights{L} = weights{L} - learningRate* vdw_corrected{L} ./ d1;
840         biases{L} = biases{L} -learningRate* vdB_corrected{L} ./ d2;
841     elseif (strcmp(outputActivationFunc,"softmax"))
842         vdw{L} = beta1*vdW{L} + (1 -beta1) * gradsDW{L}';
843         vdB{L} = beta1*vdB{L} + (1 -beta1) * gradsDB{L}';
844
845         # Compute bias-corrected first moment estimate.
846         vdw_corrected{L} = vdw{L}/(1-beta1^t);
847         vdB_corrected{L} = vdB{L}/(1-beta1^t);
848
849         sdw{L} = beta2*sdw{L} + (1 -beta2) * gradsDW{L}' .* gradsDW{L}';
850         sdB{L} = beta2*sdB{L} + (1 -beta2) * gradsDB{L}' .* gradsDB{L}';

```

```

851
852     # Compute bias-corrected second moment estimate.
853     sdW_corrected{L} = sdW{L}/(1-beta2^t);
854     sdB_corrected{L} = sdB{L}/(1-beta2^t);
855
856     # update parameters.
857     d1=sqrt(sdW_corrected{L}+epsilon);
858     d2=sqrt(sdB_corrected{L}+epsilon);
859
860     weights{L} = weights{L} - learningRate* vdw_corrected{L} ./ d1;
861     biases{L} = biases{L} -learningRate* vdB_corrected{L} ./ d2;
862   endif
863
864 end
865
866
867 # Execute a L layer Deep learning model
868 # Input : X - Input features
869 #           : Y output
870 #           : layersDimensions - Dimension of layers
871 #           : hiddenActivationFunc - Activation function at hidden layer relu
872 /tanh
873 #           : outputActivationFunc - Activation function at hidden layer
874 sigmoid/softmax
875 #           : learning rate
876 #           : lambd
877 #           : keep_prob
878 #           : num of iterations
879 #output : Updated weights and biases after each iteration
880 function [weights biases costs] = L_Layer_DeepModel(X, Y, layersDimensions,
881 hiddenActivationFunc='relu',
882           outputActivationFunc="sigmoid", learning_rate = .3, lambd=0,
883 keep_prob=1, num_iterations = 10000, initType="default")#lr was 0.009
884
885 rand ("seed", 1);
886 costs = [] ;
887 if (strcmp(initType,"He"))
888     # He Initialization
889     [weights biases] = HeInitializeDeepModel(layersDimensions);
890 elseif (strcmp(initType,"Xav"))
891     # Xavier Initialization
892     [weights biases] = XavInitializeDeepModel(layersDimensions);
893 else
894     # Default initialization.
895     [weights biases] = initializeDeepModel(layersDimensions);
896 endif
897
898 # Loop (gradient descent)
899 for i = 0:num_iterations
900     # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
901     [AL forward_caches activation_caches dropoutMat] =
902 forwardPropagationDeep(X, weights, biases,keep_prob, hiddenActivationFunc,
903 outputActivationFunc=outputActivationFunc);
904
905         # Regularization parameter is 0
906         if (lambd==0)
907             # Compute cost.
908             cost = computeCost(AL,
909 Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(
910 layersDimensions)(2)));
911         else
912             # Compute cost with regularization

```

```

913         cost = computeCostWithReg(weights, AL, Y, lambd,
914 outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2)));
915         endif
916     # Backward propagation.
917     [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y,
918 activation_caches,forward_caches, dropoutMat, lambd, keep_prob,
919 hiddenActivationFunc, outputActivationFunc=outputActivationFunc,
920 numClasses=layersDimensions(size(layersDimensions)(2)));
921     # Update parameters.
922     [weights biases] = gradientDescent(weights,biases,
923 gradsDW,gradsDB,learning_rate,outputActivationFunc=outputActivationFunc);
924
925
926
927     # Print the cost every 1000 iterations
928     if ( mod(i,1000) == 0)
929         costs =[costs cost];
930         #disp ("Cost after iteration"),
931 L2RegularizationCost(i),disp(cost);
932         printf("Cost after iteration i=%i cost=%d\n",i,cost);
933     endif
934 endfor
935
936
937 end
938
939 # Execute a L layer Deep learning model with stochastic Gradient descent
940 # Input : X - Input features
941 #
942 #       : Y output
943 #       : layersDimensions - Dimension of layers
944 #       : hiddenActivationFunc - Activation function at hidden layer relu
945 /tanh/sigmoid
946 #       : outputActivationFunc - Activation function at hidden layer
947 sigmoid/softmax
948 #       : learning rate
949 #       : lrDecay
950 #       : decayRate
951 #       : lambd
952 #       : keep_prob
953 #       : optimizer
954 #       : beta
955 #       : beta1
956 #       : beta2
957 #       : epsilon
958 #       : mini_batch_size
959 #output : Updated weights and biases after each iteration
960 function [weights biases costs] = L_Layer_DeepModel_SGD(X, Y,
961 layersDimensions, hiddenActivationFunc='relu',
962
963 outputActivationFunc="sigmoid",learningRate = .3,
964                                         lrDecay=false,decayRate=1,
965                                         lambd=0, keep_prob=1,
966                                         optimizer="gd", beta=0.9, beta1=0.9,
967 beta2=0.999,epsilon=10^-8,
968                                         mini_batch_size = 64, num_epochs =
969 2500)
970
971     disp("here");
972     printf("learningRate=%f ",learningRate);
973     printf("lrDecay=%d ",lrDecay);
974     printf("decayRate=%f ",decayRate);
975     printf("lambd=%d ",lambd);
976     printf("keep_prob=%f ",keep_prob);

```

```

977     printf("optimizer=%s ",optimizer);
978     printf("beta=%f ",beta);
979     printf("beta1=%f ",beta1);
980     printf("beta2=%f ",beta2);
981     printf("epsilon=%f ",epsilon);
982     printf("mini_batch_size=%d ",mini_batch_size);
983     printf("num_epochs=%d ",num_epochs);
984     t=0;
985     rand ("seed", 1);
986     costs = [] ;
987     # Parameters initialization.
988     [weights biases] = initializeDeepModel(layersDimensions);
989
990     if (strcmp(optimizer,"momentum"))
991         [vdW vdB] = initializeVelocity(weights, biases);
992
993     elseif(strcmp(optimizer,"rmsprop"))
994         [sdW sdB] = initializeRMSProp(weights, biases);
995
996     elseif(strcmp(optimizer,"adam"))
997         [vdW vdB sdW sdB] = initializeAdam(weights, biases);
998     endif
999     seed=10;
1000    # Loop (gradient descent)
1001    for i = 0:num_epochs
1002        seed = seed + 1;
1003        [mini_batches_X mini_batches_Y] = random_mini_batches(X, Y,
1004        mini_batch_size, seed);
1005
1006        minibatches=length(mini_batches_X);
1007        for batch=1:minibatches
1008            X=mini_batches_X{batch};
1009            Y=mini_batches_Y{batch};
1010            # Forward propagation: [LINEAR -> RELU]^(L-1) -> LINEAR ->
1011            SIGMOID/SOFTMAX.
1012            [AL forward_caches activation_caches dropoutMat] =
1013            forwardPropagationDeep(X, weights, biases, keep_prob,hiddenActivationFunc,
1014            outputActivationFunc=outputActivationFunc);
1015            #disp(batch);
1016            #disp(size(X));
1017            #disp(size(Y));
1018            if (lambd==0)
1019                # Compute cost.
1020                cost = computeCost(AL,
1021                Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(
1022                layersDimensions)(2)));
1023            else
1024                # Compute cost with regularization
1025                cost = computeCostWithReg(weights, AL, Y, lambd,
1026                outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(1a
1027                yersDimensions)(2)));
1028            endif
1029            #disp(cost);
1030            # Backward propagation.
1031            [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y,
1032            activation_caches,forward_caches, dropoutMat, lambd, keep_prob,
1033            hiddenActivationFunc, outputActivationFunc=outputActivationFunc,
1034            numClasses=layersDimensions(size(layersDimensions)(2)));
1035
1036            if (strcmp(optimizer,"gd"))
1037                # Update parameters.
1038                [weights biases] = gradientDescent(weights,biases,
1039                gradsDW,gradsDB,learningRate,outputActivationFunc);

```

```

1041         elseif (strcmp(optimizer,"momentum"))
1042             [weights biases] = gradientDescentWithMomentum(weights,
1043             biases,gradsDW,gradsDB, vdw, vdB, beta, learningRate,outputActivationFunc);
1044             elseif (strcmp(optimizer,"rmsprop"))
1045                 [weights biases] = gradientDescentWithRMSProp(weights,
1046                 biases,gradsDW,gradsDB, sdW, sdB, beta1, epsilon,
1047                 learningRate,outputActivationFunc);
1048
1049             elseif (strcmp(optimizer,"adam"))
1050                 t=t+1;
1051                 [weights biases] = gradientDescentWithAdam(weights,
1052                 biases,gradsDW,gradsDB, vdw, vdB, sdW, sdB, t, beta1, beta2, epsilon,
1053                 learningRate,outputActivationFunc);
1054             endif
1055         endfor
1056         # Print the cost every 1000 iterations
1057         if ( mod(i,1000) == 0)
1058             costs =[costs cost];
1059             #disp ("Cost after iteration"), disp(i),disp(cost);
1060             printf("Cost after iteration i=%i cost=%d\n",i,cost);
1061         endif
1062         if(lrDecay==true)
1063             learningRate=decayRate^(num_epochs/1000)*learningRate;
1064         endif
1065     endfor
1066
1067 end
1068
1069 # Plot cost vs iterations
1070 function plotCostVsIterations(maxIterations,costs,fig1)
1071     iterations=[0:1000:maxIterations];
1072     plot(iterations,costs);
1073     title ("Cost vs no of iterations ");
1074     xlabel("No of iterations");
1075     ylabel("Cost");
1076     print -dpng figReg2-o
1077 end;
1078
1079 # Plot cost vs number of epochs
1080 function plotCostVsEpochs(maxEpochs,costs,fig1)
1081     epochs=[0:1000:maxEpochs];
1082     plot(epochs,costs);
1083     title ("Cost vs no of epochs ");
1084     xlabel("No of epochs");
1085     ylabel("Cost");
1086     print -dpng fig5-o
1087 end;
1088
1089 # Compute the predicted value for a given input
1090 # Input : Neural Network parameters
1091 # : Input data
1092 function [predictions]= predict(weights, biases,
1093 x,keep_prob=1,hiddenActivationFunc="relu")
1094     [AL forward_caches activation_caches] = forwardPropagationDeep(x,
1095     weights, biases,keep_prob,hiddenActivationFunc);
1096     predictions = (AL>0.5);
1097 end
1098
1099 # Plot the decision boundary
1100 function plotDecisionBoundary(data,weights,
1101 biases,keep_prob=1,hiddenActivationFunc="relu",fig2)
1102     %Plot a non-linear decision boundary learned by the SVM
1103     colormap ("summer");
1104

```

```

1105 % Make classification predictions over a grid of values
1106 x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)';
1107 x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)';
1108 [X1, X2] = meshgrid(x1plot, x2plot);
1109 vals = zeros(size(X1));
1110 # Plot the prediction for the grid
1111 for i = 1:size(X1, 2)
1112     gridPoints = [X1(:, i), X2(:, i)];
1113     vals(:, i)=predict(weights, biases,gridPoints',keep_prob,
1114 hiddenActivationFunc=hiddenActivationFunc);
1115 endfor
1116
1117 scatter(data(:,1),data(:,2),8,c=data(:,3),"filled");
1118 % Plot the boundary
1119 hold on
1120 #contour(X1, X2, vals, [0 0], 'LineWidth', 2);
1121 contour(X1, X2, vals,"LineWidth",4);
1122 title ({"3 layer Neural Network decision boundary"});
1123 hold off;
1124 print -dpng figReg22-o
1125
1126 end
1127
1128 # Compute scores
1129 function [AL]= scores(weights, biases, x,hiddenActivationFunc="relu")
1130     [AL forward_caches activation_caches] = forwardPropagationDeep(X,
1131 weights, biases,hiddenActivationFunc);
1132 end
1133
1134 # Create Random mini batches. Return cell arrays with the mini batches
1135 # Input : X, Y
1136 #           : Size of minibatch
1137 #Output : mini batches X & Y
1138 function [mini_batches_X mini_batches_Y]= random_mini_batches(X, Y,
1139 miniBatchSize = 64, seed = 0)
1140
1141 rand ("seed", seed);
1142 # Get number of training samples
1143 m = size(X)(2);
1144
1145
1146 # Create a list of random numbers < m
1147 permutation = randperm(m);
1148 # Randomly shuffle the training data
1149 shuffled_X = X(:, permutation);
1150 shuffled_Y = Y(:, permutation);
1151
1152 # Compute number of mini batches
1153 numCompleteMinibatches = floor(m/miniBatchSize);
1154 batch=0;
1155 for k = 0:(numCompleteMinibatches-1)
1156     #Set the start and end of each mini batch
1157     batch=batch+1;
1158     lower=(k*miniBatchSize)+1;
1159     upper=(k+1) * miniBatchSize;
1160     mini_batch_X = shuffled_X(:, lower:upper);
1161     mini_batch_Y = shuffled_Y(:, lower:upper);
1162
1163     # Create cell arrays
1164     mini_batches_X{batch} = mini_batch_X;
1165     mini_batches_Y{batch} = mini_batch_Y;
1166 endfor
1167
1168 # If the batc size does not cleanly divide with number of mini batches

```

```

1169     if mod(m ,miniBatchsize) != 0
1170         # Set the start and end of the last mini batch
1171         l=floor(m/miniBatchsize)*miniBatchSize;
1172         m=l+ mod(m,miniBatchSize);
1173         mini_batch_X = shuffled_X(:,(l+1):m);
1174         mini_batch_Y = shuffled_Y(:,(l+1):m);
1175
1176         batch=batch+1;
1177         mini_batches_X{batch} = mini_batch_X;
1178         mini_batches_Y{batch} = mini_batch_Y;
1179     endif
1180 end
1181
1182 # Plot decision boundary
1183 function plotDecisionBoundary1( data,weights, biases,keep_prob=1,
1184 hiddenActivationFunc="relu")
1185     % Make classification predictions over a grid of values
1186     x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)';
1187     x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)';
1188     [X1, X2] = meshgrid(x1plot, x2plot);
1189     vals = zeros(size(X1));
1190     for i = 1:size(X1, 2)
1191         gridPoints = [X1(:, i), X2(:, i)];
1192         [AL forward_caches activation_caches] =
1193         forwardPropagationDeep(gridPoints', weights,
1194         biases,keep_prob,hiddenActivationFunc, outputActivationFunc="softmax");
1195         [l m] = max(AL, [ ], 2);
1196         vals(:, i)= m;
1197     endfor
1198
1199     scatter(data(:,1),data(:,2),8,c=data(:,3),"filled");
1200     % Plot the boundary
1201     hold on
1202     contour(X1, X2, vals,"linewidth",4);
1203     print -dpng "fig-o1.png"
1204 end
1205
1206 #####
1207 # Note: Using cellArray_to_vector followed by vector_to_cellArray => original
1208 cellArray
1209 #####
1210 # Convert a weight,biases as a cell array to a vector
1211 # Input : weight and biases cell array
1212 # Returns : vector
1213 function [vec] = cellArray_to_vector(weights,biases)
1214     vec=[];
1215     for i = 1: size(weights)(2)
1216         w= weights{i};
1217         sz=size(w);
1218         # Take transpose before reshaping
1219         l=reshape(w',sz(1)*sz(2),1);
1220
1221         b=biases{i};
1222         sz1=size(b);
1223         m=reshape(b',sz1(1)*sz1(2),1);
1224         #Concatenate
1225         vec=[vec;l;m];
1226     endfor
1227 end
1228
1229 # Convert gradients cell array to a vector
1230 # Input : gradients cell array
1231 # Returns : vector
1232 function [vec] = gradients_to_vector(gradsDW,gradsDB)

```

```

1233     vec=[];
1234     for i = 1: size(gradsDW)(2)
1235         gw= gradsDW{i};
1236         sz=size(gw);
1237         # Take transpose before reshaping
1238         l=reshape(gw',sz(1)*sz(2),1);
1239
1240         gB=gradsDB{i};
1241         sz1=size(gB);
1242         m=reshape(gB',sz1(1)*sz1(2),1);
1243         #Concatenate
1244         vec=[vec;l;m];
1245     endfor
1246 end
1247
1248 # Convert a vector to a cell array
1249 # Input : vector
1250 # Returns : cell array
1251 function [weights1 biases1] = vector_to_cellArray(weights, biases,params)
1252     vec=[];
1253     weights1 = {};
1254     biases1 = {};
1255     start=1;
1256     for i = 1: size(weights)(2)
1257         w= weights{i};
1258         sz=size(w);
1259         # Take transpose before reshaping
1260         a = params(start:start+sz(1)*sz(2)-1,1);
1261         b = reshape(a,sz(2),sz(1));
1262         weights1{i}= b';
1263         start=start+sz(1)*sz(2);
1264         b=biases{i};
1265         sz=size(b);
1266         c = params(start:start+sz(1)*sz(2)-1,1);
1267         d = reshape(c,sz(2),sz(1));
1268         biases1{i}= d';
1269         start=start+sz(1)*sz(2);
1270     endfor
1271 end
1272
1273
1274 # Convert a vector to a cell array
1275 # Input : vector
1276 # Returns : cell array
1277 function [weights1 biases1] = vector_to_cellArray1(weights, biases,gradients)
1278     vec=[];
1279     weights1 = {};
1280     biases1 = {};
1281     start=1;
1282     for i = 1: size(weights)(2)
1283         w= weights{i};
1284         sz=size(w);
1285         # Take transpose before reshaping
1286         a = grads(start:start+sz(1)*sz(2)-1,1);
1287         b = reshape(a,sz(2),sz(1));
1288         weights1{i}= b';
1289         start=start+sz(1)*sz(2);
1290         b=biases{i};
1291         sz=size(b);
1292         c = grads(start:start+sz(1)*sz(2)-1,1);
1293         d = reshape(c,sz(2),sz(1));
1294         biases1{i}= d';
1295         start=start+sz(1)*sz(2);
1296

```

```

1297     endfor
1298 end
1299
1300 # Perform Gradient check
1301 # Input : weights,biases
1302 #      : gradsDW,gradsDB
1303 #      : X
1304 #      : Y
1305 #      : epsilon
1306 #      : outputActivationFunc
1307 #      : numClasses
1308 # Returns :
1309 function [difference]= gradient_check_n(weights,biases,gradsDW,gradsDB , X,
1310 Y, epsilon = 1e-7,outputActivationFunc="sigmoid",numClasses)
1311 # Convert cell array to vector
1312 parameters_values = cellArray_to_vector(weights, biases);
1313 # Convert gradient cell array to vector
1314 grad = gradients_to_vector(gradsDW,gradsDB);
1315 num_parameters = size(parameters_values)(1);
1316 #Initialize
1317 J_plus = zeros(num_parameters, 1);
1318 J_minus = zeros(num_parameters, 1);
1319 gradapprox = zeros(num_parameters, 1);
1320
1321
1322 # Compute gradapprox
1323 for i = 1:num_parameters
1324     # Compute J_plus[i]. Inputs: "parameters_values, epsilon". Output =
1325 "J_plus[i]".
1326     thetaplus = parameters_values;
1327     thetaplus(i,1) = thetaplus(i,1) + epsilon;
1328     [weights1 biases1] =vector_to_cellArray(weights, biases,thetaplus);
1329     [AL forward_caches activation_caches dropoutMat] =
1330 forwardPropagationDeep(X', weights1, biases1, keep_prob=1,
1331 hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc);
1332     J_plus(i) = computeCost(AL, Y',
1333 outputActivationFunc=outputActivationFunc,numClasses);
1334
1335
1336     # Compute J_minus[i]. Inputs: "parameters_values, epsilon". Output =
1337 "J_minus[i]".
1338     thetaminus = parameters_values;
1339     thetaminus(i,1) = thetaminus(i,1) - epsilon ;
1340     [weights1 biases1] = vector_to_cellArray(weights, biases,thetaminus);
1341     [AL forward_caches activation_caches dropoutMat] =
1342 forwardPropagationDeep(X',weights1, biases1, keep_prob=1,
1343 hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc);
1344     J_minus(i) = computeCost(AL, Y',
1345 outputActivationFunc=outputActivationFunc,numClasses);
1346
1347     # Compute gradapprox[i]
1348     gradapprox(i) = (J_plus(i) - J_minus(i))/(2*epsilon);
1349
1350 endfor
1351
1352 # Compute L2Norm
1353 numerator = L2NormVec(grad-gradapprox);
1354 denominator = L2NormVec(grad) + L2NormVec(gradapprox);
1355 difference = numerator/denominator;
1356 ;
1357 if difference > 1e-04
1358     printf("There is a mistake in the implementation ");
1359
1360

```

```

1361     disp(difference);
1362 else
1363     printf("The implementation works perfectly");
1364     disp(difference);
1365 endif
1366 # This can be used to compare the gradients from backprop and gradapprox
1367 [weights1 biases1] = vector_to_cellArray(weights, biases,grad);
1368 printf("Gradients from back propagation");
1369 disp(weights1);
1370 disp(biases1);
1371 [weights2 biases2] = vector_to_cellArray(weights, biases,gradapprox);
1372 printf("Gradients from gradient check");
1373 disp(weights2);
1374 disp(biases2);
1375
1376 end
1377
1378 # Compute L2Norm
1379 function [l2norm] = L2NormVec(x)
1380     l2norm=sqrt(sum(x .^ 2));
1381 end

```

References

1. Deep Learning Specialization - <https://www.coursera.org/specializations/deep-learning>
2. Neural Networks for Machine Learning - <https://www.coursera.org/learn/neural-networks>
3. Deep Learning by Ian Goodfellow, Yoshua Bengio and Aaron Courville - <http://www.deeplearningbook.org/>
4. Neural Networks: The mechanics of backpropagation - <https://gigadom.wordpress.com/2017/01/21/neural-networks-the-mechanics-of-backpropagation/>
5. Machine Learning - <https://www.coursera.org/learn/machine-learning>
6. CS231n Convolutional Neural Networks for Visual Recognition -<http://cs231n.github.io/neural-networks-case-study/>
7. The Softmax function and its derivative - <https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/>
8. Cross Validated - <https://stats.stackexchange.com/questions/235528/backpropagation-with-softmax-cross-entropy>
9. CS231n: How to calculate gradient for Softmax loss function? - <https://stackoverflow.com/questions/41663874/cs231n-how-to-calculate-gradient-for-softmax-loss-function>
10. Derivative of Softmax loss function - <https://math.stackexchange.com/questions/945871/derivative-of-softmax-loss-function>
11. The Matrix Calculus You Need For Deep Learning - <https://arxiv.org/abs/1802.01528>
12. A Step by Step Backpropagation Example - <https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>
13. The Backpropagation Algorithm - <https://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf>
14. Backpropagation Learning- <https://www.cs.cmu.edu/afs/cs/academic/class/15883-f15/slides/backprop.pdf>
15. Practical Machine Learning with R and Python – Machine Learning in stereo - <https://www.amazon.com/dp/1973443503>