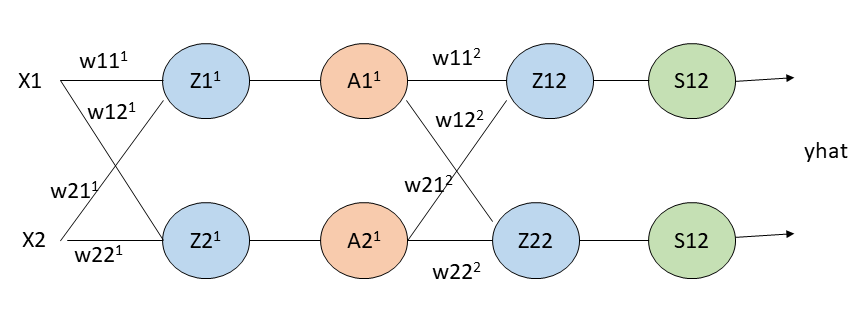
The Softmax function takes an N dimensional vector as input and generates a N dimensional vector as output.

The Softmax function is given by  
S_{j}= \frac{e_{j}}{\sum_{i}^{N}e_{k}}  
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,

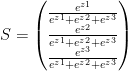
Let us take a simple 2 layered neural network with just 2 activation units in the hidden layer is shown below  
  
Z_{1}^{1} =W_{11}^{1}x_{1} + W_{21}^{1}x_{2} + b_{1}^{1}  
Z_{2}^{1} =W_{12}^{1}x_{1} + W_{22}^{1}x_{2} + b_{2}^{1}  
and  
A_{1}^{1} = g'(Z_{1}^{1})  
A_{2}^{1} = g'(Z_{2}^{1})  
where 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 above denotes the equation 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(Z_{1}^{2})  
A_{2}^{2} = S(Z_{2}^{2})  
where S() is the Softmax activation function  
S=\begin{pmatrix} S(Z_{1}^{2})\\ S(Z_{2}^{2}) \end{pmatrix}  
S=\begin{pmatrix} \frac{e^{Z1}}{e^{Z1}+e^{Z2}}\\ \frac{e^{Z2}}{e^{Z1}+e^{Z2}} \end{pmatrix}

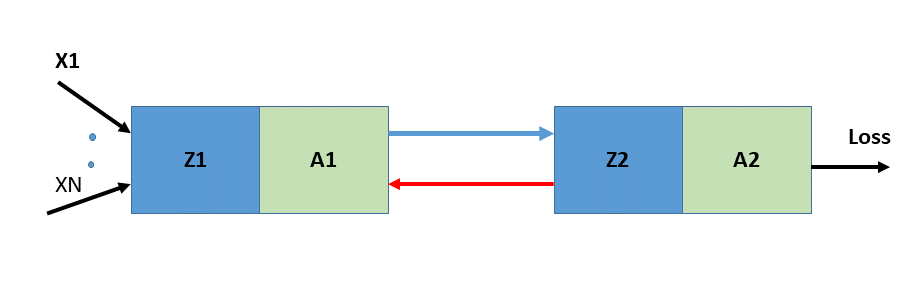
The Jacobian of the softmax ‘S’ is given by  
\begin{pmatrix} \frac {\partial S_{1}}{\partial Z_{1}} & \frac {\partial S_{1}}{\partial Z_{2}}\\ \frac {\partial S_{2}}{\partial Z_{1}} & \frac {\partial S_{2}}{\partial Z_{2}} \end{pmatrix}  
\begin{pmatrix} \frac{\partial}{\partial Z_{1}} \frac {e^{Z1}}{e^{Z1}+ e^{Z2}} & \frac{\partial}{\partial Z_{2}} \frac {e^{Z1}}{e^{Z1}+ e^{Z2}}\\ \frac{\partial}{\partial Z_{1}} \frac {e^{Z2}}{e^{Z1}+ e^{Z2}} & \frac{\partial}{\partial Z_{2}} \frac {e^{Z2}}{e^{Z1}+ e^{Z2}} \end{pmatrix}     – (A)

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 {vdu -udv}{v^{2}}  
Using this to compute each element of the above Jacobian matrix, we see that  
when i=j we have  
\frac {\partial}{\partial Z1}\frac{e^{Z1}}{e^{Z1}+e^{Z2}} = \frac {\sum e^{Z1} - e^{Z1^{2}}}{\sum ^{2}}  
and when i \neq j  
\frac {\partial}{\partial Z1}\frac{e^{Z2}}{e^{Z1}+e^{Z2}} = \frac {0 - e^{z1}e^{Z2}}{\sum ^{2}}  
This is of the general form  
\frac {\partial S_{j}}{\partial z_{i}} = S_{i}( 1-S_{j})  when i=j  
and  
\frac {\partial S_{j}}{\partial z_{i}} = -S_{i}S_{j}  when i \neq j  
**Note**: Since the Softmax essentially gives the probability the following  
notation is also used  
\frac {\partial p_{j}}{\partial z_{i}} = p_{i}( 1-p_{j}) when i=j  
and  
\frac {\partial p_{j}}{\partial z_{i}} = -p_{i}p_{j} 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_{j}}{\partial z_{i}} = p_{i} (\delta_{ij} - p_{j})  
where \delta_{ij} = 1when 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}}(1)  
and  
\frac {\partial p_{i}}{\partial w_{pq}} = \sum _{k}\frac {\partial p_{i}}{\partial z_{k}} \frac {\partial z_{k}}{\partial w_{pq}}(2)  
In expanded form this is  
\frac {\partial L}{\partial w_{pq}} = \sum _{i}\frac {\partial L}{\partial p_{i}} \sum _{k}\frac {\partial p_{i}}{\partial z_{k}} \frac {\partial z_{k}}{\partial w_{pq}}  
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 {\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 z_{1}}=-\frac {y1}{p1} p1(1-p1) - \frac {y2}{p2}*(-p_{2}p_{1})  
Since  
\frac {\partial p_{j}}{\partial z_{i}} = p_{i}( 1-p_{j})when i=j  
and  
\frac {\partial p_{j}}{\partial z_{i}} = -p_{i}p_{j}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 {\partial L}{\partial p_{1}} \frac {\partial p_{1}}{\partial Z_{2}} +\frac {\partial L}{\partial p_{2}} \frac {\partial p_{2}}{\partial Z_{2}}   
\frac {\partial L}{\partial z_{2}}=-\frac {y1}{p1}*(p_{1}p_{2}) - \frac {y2}{p2}*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}  
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  yi  would be yi=[0,1,0] therefore the gradient pi-yi = [0.1,-0.3,0.2]

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


We could derive  
\frac {\partial L}{\partial z1}= \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}}which similarly reduces to  
\frac {\partial L}{\partial z_{1}}=-\frac {y1}{p1} p1(1-p1) - \frac {y2}{p2}*(-p_{2}p_{1}) - \frac {y3}{p3}*(-p_{3}p_{1})  
-y_{1}+ y_{1}p_{1} + y_{2}p_{1} + y_{3}p1 = 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 the key equations for forward and backward propagation are:  
  
**Forward propagation equations layer 1**  
Z_{1} = W_{1}X +b_{1}     and  A_{1} = g(Z_{1})  
**Forward propagation equations layer 1**  
Z_{2} = W_{2}A_{1} +b_{2}  and  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})

**2.0 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.

import numpy as np

import matplotlib.pyplot as plt

import os

os.chdir("C:/junk/dl-4/dl-4")

exec(open("././DLfunctions41.py").read())

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

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

N = 100 # number of points per class

D = 2 # dimensionality

K = 3 # number of classes

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

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

for j in range(K):

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

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

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

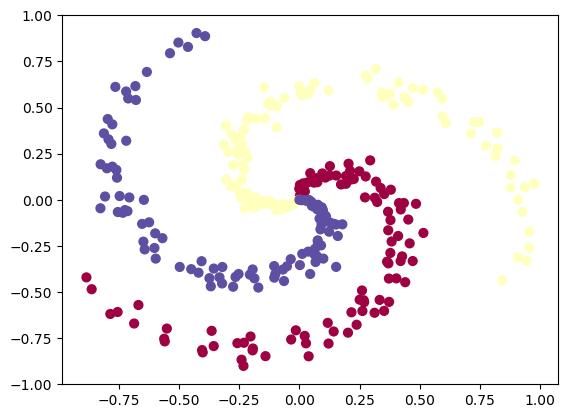
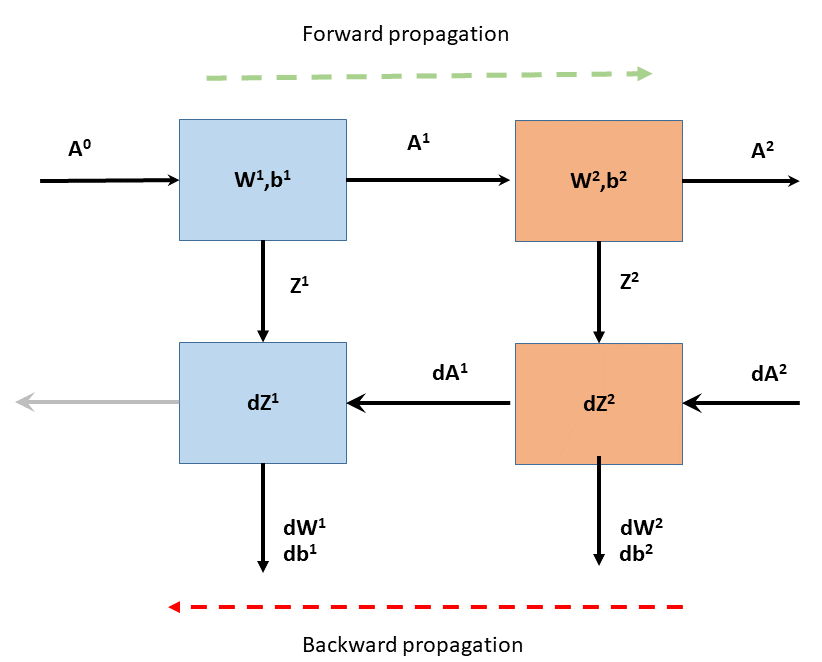
X[ix] = np.c\_[r\*np.sin(t), r\*np.cos(t)]

y[ix] = j

# Plot the data

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

plt.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

import numpy as np

import matplotlib.pyplot as plt

import os

os.chdir("C:/junk/dl-4/dl-4")

exec(open("././DLfunctions41.py").read())

# Read the input data

N = 100 # number of points per class

D = 2 # dimensionality

K = 3 # number of classes

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

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

for j in range(K):

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

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

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

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

y[ix] = j

# Set the number of features, hidden units in hidden layer and number of classess

numHidden=100 # No of hidden units in hidden layer

numFeats= 2 # dimensionality

numOutput = 3 # number of classes

# Initialize the model

parameters=initializeModel(numFeats,numHidden,numOutput)

W1= parameters['W1']

b1= parameters['b1']

W2= parameters['W2']

b2= parameters['b2']

# Set the learning rate

learningRate=0.6

# Initialize losses

losses=[]

# Perform Gradient descent

for i in range(10000):

# Forward propagation through hidden layer with Relu units

A1,cache1= layerActivationForward(X.T,W1,b1,'relu')

# Forward propagation through output layer with Softmax

A2,cache2 = layerActivationForward(A1,W2,b2,'softmax')

# No of training examples

numTraining = X.shape[0]

# Compute log probs. Take the log prob of correct class based on output y

correct\_logprobs = -np.log(A2[range(numTraining),y])

# Conpute loss

loss = np.sum(correct\_logprobs)/numTraining

# Print the loss

if i % 1000 == 0:

print("iteration %d: loss %f" % (i, loss))

losses.append(loss)

dA=0

# Backward propagation through output layer with Softmax

dA1,dW2,db2 = layerActivationBackward(dA, cache2, y, activationFunc='softmax')

# Backward propagation through hidden layer with Relu unit

dA0,dW1,db1 = layerActivationBackward(dA1.T, cache1, y, activationFunc='relu')

#Update paramaters with the learning rate

W1 += -learningRate \* dW1

b1 += -learningRate \* db1

W2 += -learningRate \* dW2.T

b2 += -learningRate \* db2.T

#Plot losses vs iterations

i=np.arange(0,10000,1000)

plt.plot(i,losses)

plt.xlabel('Iterations')

plt.ylabel('Loss')

plt.title('Losses vs Iterations')

plt.savefig("fig2.png", bbox="tight")

#Compute the multi-class Confusion Matrix

from sklearn.metrics import confusion\_matrix

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

# We need to determine the predicted values from the learnt data

# Forward propagation through hidden layer with Relu units

A1,cache1= layerActivationForward(X.T,W1,b1,'relu')

# Forward propagation through output layer with Softmax

A2,cache2 = layerActivationForward(A1,W2,b2,'softmax')

#Compute predicted values from weights and biases

yhat=np.argmax(A2, axis=1)

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

print("Multi-class Confusion Matrix")

print(a)

## iteration 0: loss 1.098507

## iteration 1000: loss 0.214611

## iteration 2000: loss 0.043622

## iteration 3000: loss 0.032525

## iteration 4000: loss 0.025108

## iteration 5000: loss 0.021365

## iteration 6000: loss 0.019046

## iteration 7000: loss 0.017475

## iteration 8000: loss 0.016359

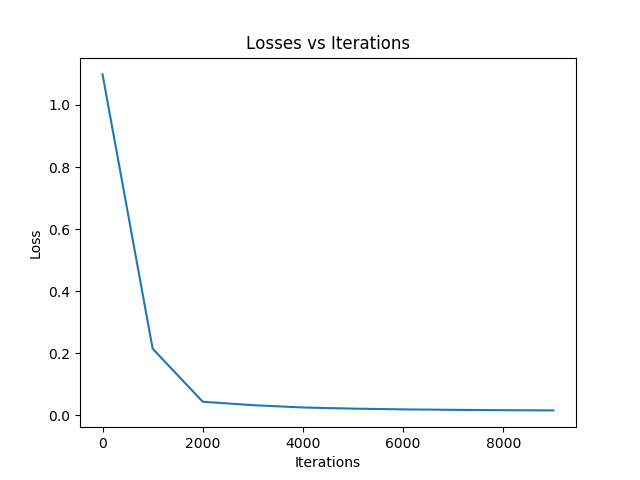
## iteration 9000: loss 0.015703

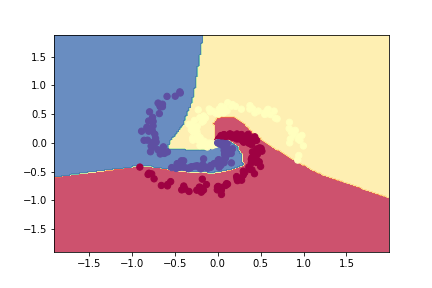
## Multi-class Confusion Matrix

## [[ 99 1 0]

## [ 0 100 0]

## [ 0 1 99]]





**2.2 Multi-class classification with Softmax – R code**

The spiral data set created with Python was saved, and is used as the input with R code. The R Neural Network seems to perform much,much slower than both Python and Octave. Not sure why! Incidentally the computation of loss and the softmax derivative are identical for

both R and Octave. yet R is much slower. To compute the softmax derivative I create matrices for the One Hot Encoded yi and then stack them before subtracting pi-yi. I am sure there is a more elegant and more efficient way to do this, much like Python. Any suggestions?

library(ggplot2)

library(dplyr)

library(RColorBrewer)

source("DLfunctions41.R")

# Read the spiral dataset

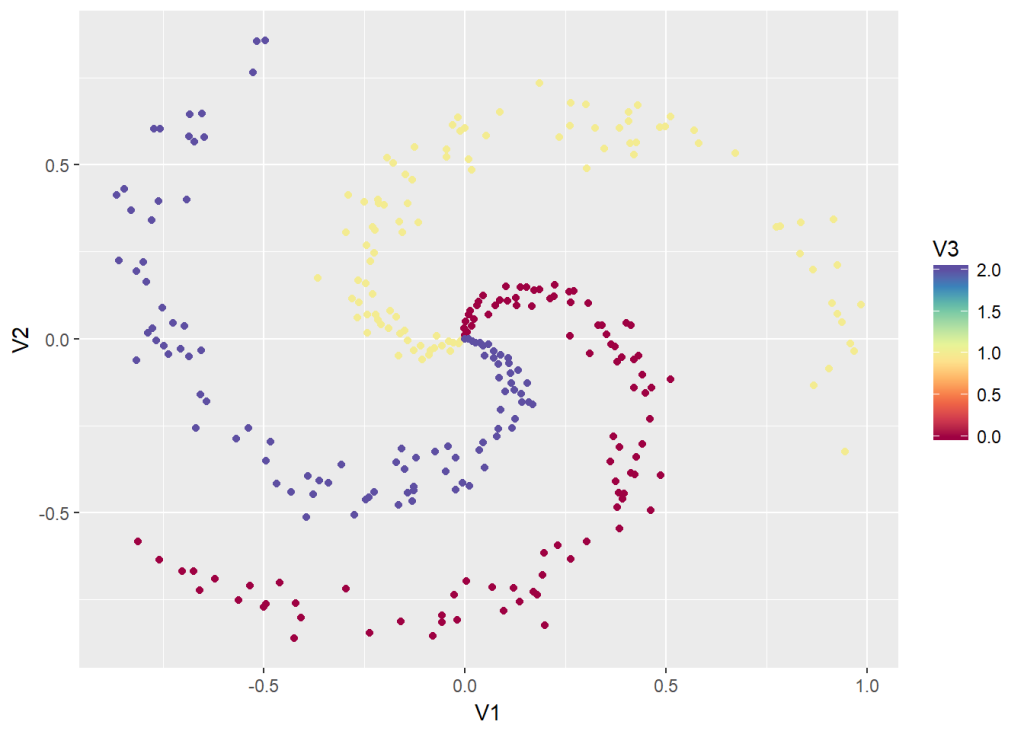
Z <- as.matrix(read.csv("spiral.csv",header=FALSE))

Z1=data.frame(Z)

#Plot the dataset

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

scale\_colour\_gradientn(colours = brewer.pal(10, "Spectral"))



# Setup the data

X <- Z[,1:2]

y <- Z[,3]

X1 <- t(X)

Y1 <- t(y)

# Initialize number of features, number of hidden units in hidden layer and

# number of classes

numFeats<-2 # No features

numHidden<-100 # No of hidden units

numOutput<-3 # No of classes

# Initialize model

parameters <-initializeModel(numFeats, numHidden,numOutput)

W1 <-parameters[['W1']]

b1 <-parameters[['b1']]

W2 <-parameters[['W2']]

b2 <-parameters[['b2']]

# Set the learning rate

learningRate <- 0.5

# Initialize losses

losses <- NULL

# Perform gradient descent

for(i in 0:9000){

# Forward propagation through hidden layer with Relu units

retvals <- layerActivationForward(X1,W1,b1,'relu')

A1 <- retvals[['A']]

cache1 <- retvals[['cache']]

forward\_cache1 <- cache1[['forward\_cache1']]

activation\_cache <- cache1[['activation\_cache']]

# Forward propagation through output layer with Softmax units

retvals = layerActivationForward(A1,W2,b2,'softmax')

A2 <- retvals[['A']]

cache2 <- retvals[['cache']]

forward\_cache2 <- cache2[['forward\_cache1']]

activation\_cache2 <- cache2[['activation\_cache']]

# No oftraining examples

numTraining <- dim(X)[1]

dA <-0

# Select the elements where the y values are 0, 1 or 2 and make a vector

a=c(A2[y==0,1],A2[y==1,2],A2[y==2,3])

# Take log

correct\_probs = -log(a)

# Compute loss

loss= sum(correct\_probs)/numTraining

if(i %% 1000 == 0){

sprintf("iteration %d: loss %f",i, loss)

print(loss)

}

# Backward propagation through output layer with Softmax units

retvals = layerActivationBackward(dA, cache2, y, activationFunc='softmax')

dA1 = retvals[['dA\_prev']]

dW2= retvals[['dW']]

db2= retvals[['db']]

# Backward propagation through hidden layer with Relu units

retvals = layerActivationBackward(t(dA1), cache1, y, activationFunc='relu')

dA0 = retvals[['dA\_prev']]

dW1= retvals[['dW']]

db1= retvals[['db']]

# Update parameters

W1 <- W1 - learningRate \* dW1

b1 <- b1 - learningRate \* db1

W2 <- W2 - learningRate \* t(dW2)

b2 <- b2 - learningRate \* t(db2)

}

## [1] 1.212487

## [1] 0.5740867

## [1] 0.4048824

## [1] 0.3561941

## [1] 0.2509576

## [1] 0.7351063

## [1] 0.2066114

## [1] 0.2065875

## [1] 0.2151943

## [1] 0.1318807

#Create iterations

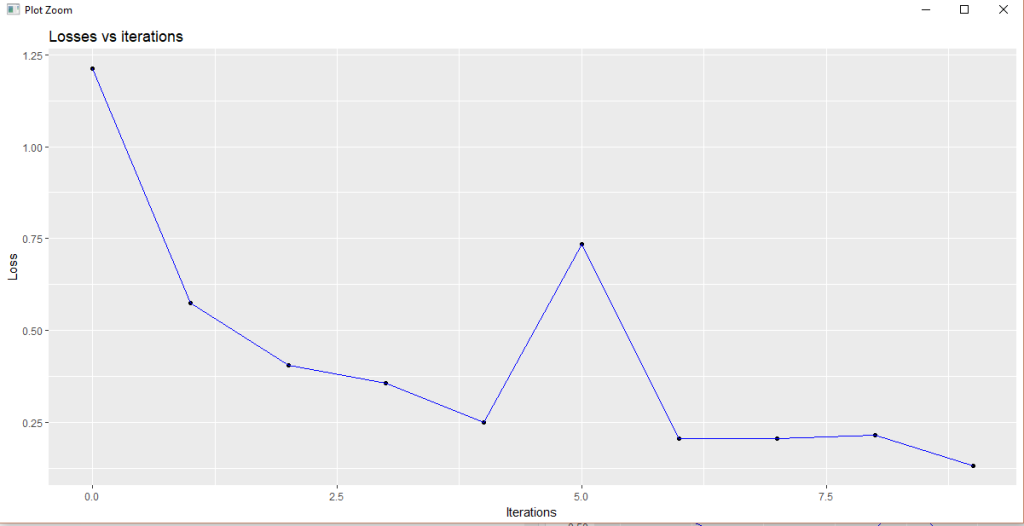
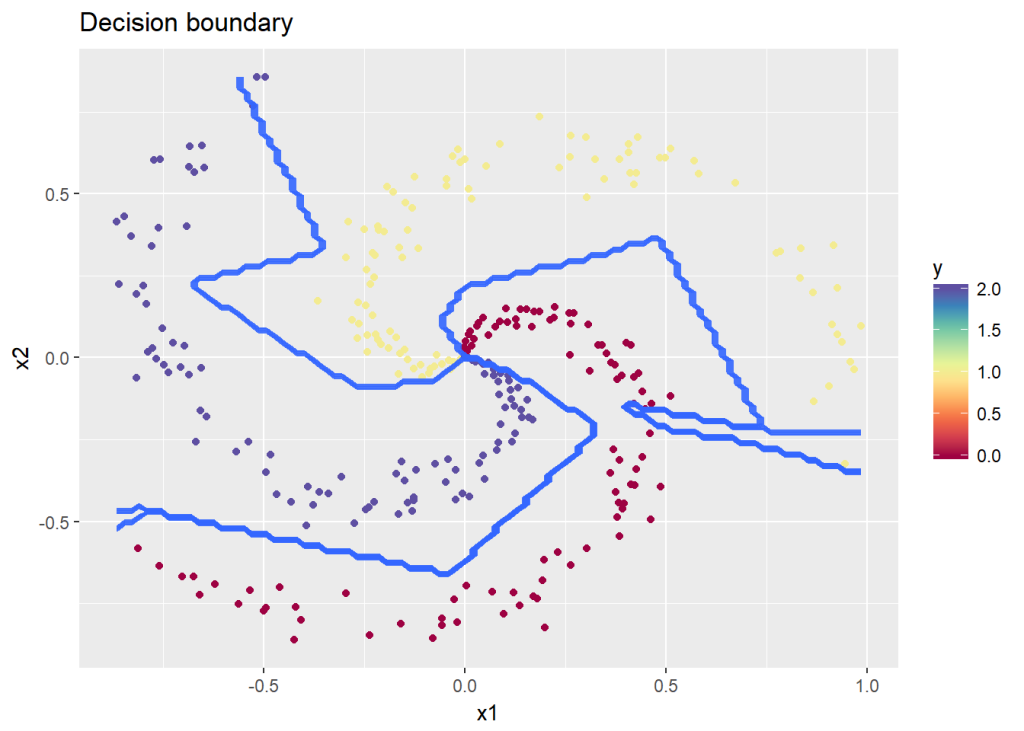
iterations <- seq(0,10)

#df=data.frame(iterations,losses)

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

ggtitle("Losses vs iterations") + xlab("Iterations") + ylab("Loss")

plotDecisionBoundary(Z,W1,b1,W2,b2)

  
  
**Multi-class Confusion Matrix**

library(caret)

library(e1071)

# Forward propagation through hidden layer with Relu units

retvals <- layerActivationForward(X1,W1,b1,'relu')

A1 <- retvals[['A']]

# Forward propagation through output layer with Softmax units

retvals = layerActivationForward(A1,W2,b2,'softmax')

A2 <- retvals[['A']]

yhat <- apply(A2, 1,which.max) -1

Confusion Matrix and Statistics

Reference

Prediction 0 1 2

0 97 0 1

1 2 96 4

2 1 4 95

Overall Statistics

Accuracy : 0.96

95% CI : (0.9312, 0.9792)

No Information Rate : 0.3333

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

Kappa : 0.94

Mcnemar's Test P-Value : 0.5724

Statistics by Class:

Class: 0 Class: 1 Class: 2

Sensitivity 0.9700 0.9600 0.9500

Specificity 0.9950 0.9700 0.9750

Pos Pred Value 0.9898 0.9412 0.9500

Neg Pred Value 0.9851 0.9798 0.9750

Prevalence 0.3333 0.3333 0.3333

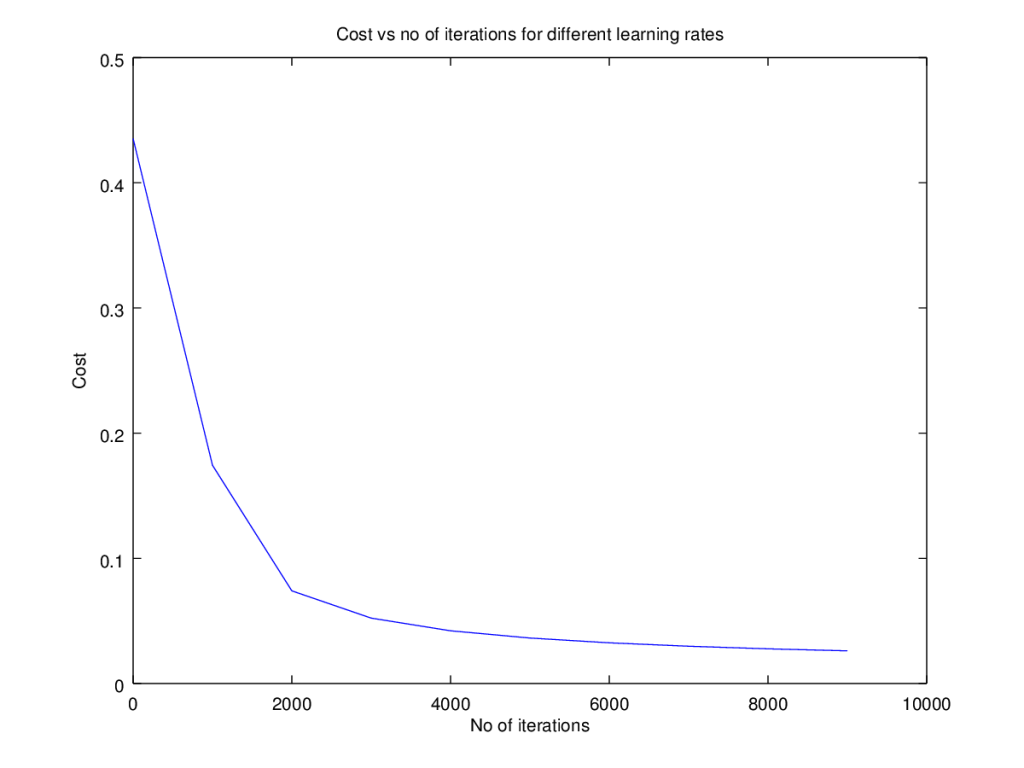
Detection Rate 0.3233 0.3200 0.3167

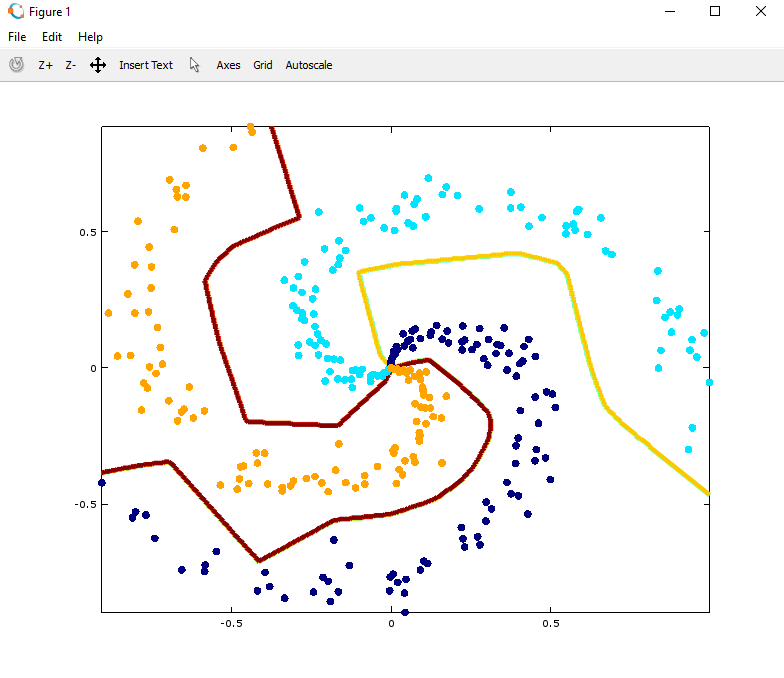
Detection Prevalence 0.3267 0.3400 0.3333

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 set is used for Octave also  
  
source("DL41functions.m")  
# Read the spiral data  
data=csvread("spiral.csv");  
# Setup the data  
X=data(:,1:2);  
Y=data(:,3);  
# Set the number of features, number of hidden units in hidden layer and number of classes  
numFeats=2; #No features  
numHidden=100; # No of hidden units  
numOutput=3; # No of classes  
# Initialize model  
[W1 b1 W2 b2] = initializeModel(numFeats,numHidden,numOutput);  
# Initialize losses  
losses=[]  
#Initialize learningRate  
learningRate=0.5;  
for k =1:10000  
# Forward propagation through hidden layer with Relu units  
[A1,cache1 activation\_cache1]= layerActivationForward(X',W1,b1,activationFunc ='relu');  
# Forward propagation through output layer with Softmax units  
[A2,cache2 activation\_cache2] =  
layerActivationForward(A1,W2,b2,activationFunc='softmax');  
# No of training examples  
numTraining = size(X)(1);  
# Select rows where Y=0,1,and 2 and concatenate to a long vector  
a=[A2(Y==0,1) ;A2(Y==1,2) ;A2(Y==2,3)];  
#Select the correct column for log prob  
correct\_probs = -log(a);  
#Compute log loss  
loss= sum(correct\_probs)/numTraining;  
if(mod(k,1000) == 0)  
disp(loss);  
losses=[losses loss];  
endif  
dA=0;  
# Backward propagation through output layer with Softmax units  
[dA1 dW2 db2] = layerActivationBackward(dA, cache2, activation\_cache2,Y,activationFunc='softmax');  
# Backward propagation through hidden layer with Relu units  
[dA0,dW1,db1] = layerActivationBackward(dA1', cache1, activation\_cache1, Y, activationFunc='relu');  
#Update parameters  
W1 += -learningRate \* dW1;  
b1 += -learningRate \* db1;  
W2 += -learningRate \* dW2';  
b2 += -learningRate \* db2';  
endfor  
# Plot Losses vs Iterations  
iterations=0:1000:9000  
plotCostVsIterations(iterations,losses)  
# Plot the decision boundary  
plotDecisionBoundary( X,Y,W1,b1,W2,b2)



****

DL41Functions.m

|  |
| --- |
|  |
| 1; |
|  |  |
|  | # Define Relu function |
|  | function [A,cache] = relu(Z) |
|  | A = max(0,Z); |
|  | cache=Z; |
|  | end |
|  |  |
|  |  |
|  | # Define Softmax function |
|  | function [A,cache] = softmax(Z) |
|  | # get unnormalized probabilities |
|  | exp\_scores = exp(Z'); |
|  | # normalize them for each example |
|  | A = exp\_scores ./ sum(exp\_scores,2); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Relu Derivative |
|  | function [dZ] = reluDerivative(dA,cache) |
|  | Z = cache; |
|  | dZ = dA; |
|  | # Get elements that are greater than 0 |
|  | a = (Z > 0); |
|  | # Select only those elements where Z > 0 |
|  | dZ = dZ .\* a; |
|  | end |
|  |  |
|  |  |
|  | # Define Softmax Derivative |
|  | function [dZ] = softmaxDerivative(dA,cache,Y) |
|  | Z = cache; |
|  | # get unnormalized probabilities |
|  | exp\_scores = exp(Z'); |
|  | # normalize them for each example |
|  | probs = exp\_scores ./ sum(exp\_scores,2); |
|  |  |
|  | # dZ = pi- yi |
|  | a=sum(Y==0); |
|  | b=sum(Y==1); |
|  | c=sum(Y==2); |
|  | m= repmat([1 0 0],a,1); |
|  | n= repmat([0 1 0],b,1); |
|  | o= repmat([0 0 1],c,1); |
|  | yi=[m;n;o]; |
|  | dZ=probs-yi; |
|  |  |
|  | end |
|  |  |
|  | # Initialize the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  | function [W1 b1 W2 b2] = initializeModel(numFeats,numHidden,numOutput) |
|  | rand ("seed", 3); |
|  | W1=rand(numHidden,numFeats)\*0.01; # Multiply by .01 |
|  | b1=zeros(numHidden,1); |
|  | W2=rand(numOutput,numHidden)\*0.01; |
|  | b2=zeros(numOutput,1); |
|  | end |
|  |  |
|  |  |
|  |  |
|  | # Compute the activation at a layer 'l' for forward prop in a Deep Network |
|  | # Input : A\_prec - Activation of previous layer |
|  | # W,b - Weight and bias matrices and vectors |
|  | # activationFunc - Activation function - sigmoid, tanh, relu etc |
|  | # Returns : The Activation of this layer |
|  | # : |
|  | # Z = W \* X + b |
|  | # A = sigmoid(Z), A= Relu(Z), A= tanh(Z) |
|  | function [A forward\_cache activation\_cache] = layerActivationForward(A\_prev, W, b, activationFunc) |
|  |  |
|  | # Compute Z |
|  | Z = W \* A\_prev +b; |
|  | # Create a cell array |
|  | forward\_cache = {A\_prev W b}; |
|  | # Compute the activation for sigmoid |
|  | if (strcmp(activationFunc,"sigmoid")) |
|  | [A activation\_cache] = sigmoid(Z); |
|  | elseif (strcmp(activationFunc, "relu")) # Compute the activation for Relu |
|  | [A activation\_cache] = relu(Z); |
|  | elseif(strcmp(activationFunc,'tanh')) # Compute the activation for tanh |
|  | [A activation\_cache] = tanhAct(Z); |
|  | elseif(strcmp(activationFunc,'softmax')) # Compute the activation for softmax |
|  | [A activation\_cache] = softmax(Z); |
|  |  |
|  | endif |
|  |  |
|  | end |
|  |  |
|  |  |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  | function [dA\_prev dW db] = layerActivationBackward(dA, forward\_cache, activation\_cache, Y, activationFunc) |
|  |  |
|  | if (strcmp(activationFunc,"relu")) |
|  | dZ = reluDerivative(dA, activation\_cache); |
|  | elseif (strcmp(activationFunc,"sigmoid")) |
|  | dZ = sigmoidDerivative(dA, activation\_cache); |
|  | elseif(strcmp(activationFunc, "tanh")) |
|  | dZ = tanhDerivative(dA, activation\_cache); |
|  | elseif(strcmp(activationFunc, "softmax")) |
|  | dZ = softmaxDerivative(dA, activation\_cache,Y); |
|  | endif |
|  | A\_prev = forward\_cache{1}; |
|  | numTraining = size(A\_prev)(2); |
|  |  |
|  | if(strcmp(activationFunc, "softmax")) |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | dW = 1/numTraining \* A\_prev \* dZ; |
|  | db = 1/numTraining \* sum(dZ,1); |
|  | dA\_prev = dZ\*W; |
|  | else |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | dW = 1/numTraining \* dZ \* A\_prev'; |
|  | db = 1/numTraining \* sum(dZ,2); |
|  | dA\_prev = W'\*dZ; |
|  | endif |
|  | end |
|  |  |
|  |  |
|  | function plotCostVsIterations(iterations,costs) |
|  |  |
|  | plot(iterations,costs); |
|  | title ("Cost vs no of iterations for different learning rates"); |
|  | xlabel("No of iterations"); |
|  | ylabel("Cost"); |
|  | print -dpng "figo2.png" |
|  | end; |
|  |  |
|  | function plotDecisionBoundary( X,Y,W1,b1,W2,b2) |
|  | % Make classification predictions over a grid of values |
|  | x1plot = linspace(min(X(:,1)), max(X(:,1)), 400)'; |
|  | x2plot = linspace(min(X(:,2)), max(X(:,2)), 400)'; |
|  | [X1, X2] = meshgrid(x1plot, x2plot); |
|  | vals = zeros(size(X1)); |
|  |  |
|  | for i = 1:size(X1, 2) |
|  | gridPoints = [X1(:, i), X2(:, i)]; |
|  | [A1,cache1 activation\_cache1]= layerActivationForward(gridPoints',W1,b1,activationFunc ='relu'); |
|  | [A2,cache2 activation\_cache2] = layerActivationForward(A1,W2,b2,activationFunc='softmax'); |
|  | [l m] = max(A2, [ ], 2); |
|  | vals(:, i)= m; |
|  | endfor |
|  |  |
|  | scatter(X(:,1),X(:,2),8,c=Y,"filled"); |
|  | % Plot the boundary |
|  | hold on |
|  | contour(X1, X2, vals,"linewidth",4); |
|  | print -dpng "fig-o1.png" |
|  | end |
|  |  |

DLFunctions41.R

|  |
| --- |
| # Conmpute the Relu of a vector |
|  | relu <-function(Z){ |
|  | A <- apply(Z, 1:2, function(x) max(0,x)) |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  |  |
|  |  |
|  | # Conmpute the softmax of a vector |
|  | softmax <- function(Z){ |
|  | # get unnormalized probabilities |
|  | exp\_scores = exp(t(Z)) |
|  | # normalize them for each example |
|  | A = exp\_scores / rowSums(exp\_scores) |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the detivative of Relu |
|  | reluDerivative <-function(dA, cache){ |
|  | Z <- cache |
|  | dZ <- dA |
|  | # Create a logical matrix of values > 0 |
|  | a <- Z > 0 |
|  | # When z <= 0, you should set dz to 0 as well. Perform an element wise multiple |
|  | dZ <- dZ \* a |
|  | return(dZ) |
|  | } |
|  |  |
|  |  |
|  | softmaxDerivative <- function(dA, cache ,y,numTraining){ |
|  | # Note : dA not used. dL/dZ = dL/dA \* dA/dZ = pi-yi |
|  | Z <- cache |
|  | # Compute softmax |
|  | exp\_scores = exp(t(Z)) |
|  | # normalize them for each example |
|  | probs = exp\_scores / rowSums(exp\_scores) |
|  | # Get the number of 0, 1 and 2 classes and store in a,b,c |
|  | a=sum(y==0) |
|  | b=sum(y==1) |
|  | c=sum(y==2) |
|  | # Create a yi matrix based on yi for each class |
|  | m= matrix(rep(c(1,0,0),a),nrow=a,ncol=3,byrow=T) |
|  | n= matrix(rep(c(0,1,0),b),nrow=b,ncol=3,byrow=T) |
|  | o= matrix(rep(c(0,0,1),c),nrow=c,ncol=3,byrow=T) |
|  | # Stack them vertically |
|  | yi=rbind(m,n,o) |
|  |  |
|  | dZ = probs-yi |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Initialize the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  | initializeModel <- function(numFeats,numHidden,numOutput){ |
|  | set.seed(2) |
|  | a<-rnorm(numHidden\*numFeats)\*0.01 # Multiply by .01 |
|  | W1 <- matrix(a,nrow=numHidden,ncol=numFeats) |
|  | a<-rnorm(numHidden\*1) |
|  | b1 <- matrix(a,nrow=numHidden,ncol=1) |
|  | a<-rnorm(numOutput\*numHidden)\*0.01 |
|  | W2 <- matrix(a,nrow=numOutput,ncol=numHidden) |
|  | a<-rnorm(numOutput\*1) |
|  | b2 <- matrix(a,nrow=numOutput,ncol=1) |
|  | parameters <- list("W1"=W1,"b1"=b1,"W2"=W2,"b2"=b2) |
|  | return(parameters) |
|  |  |
|  | } |
|  |  |
|  |  |
|  |  |
|  |  |
|  | # Compute the activation at a layer 'l' for forward prop in a Deep Network |
|  | # Input : A\_prec - Activation of previous layer |
|  | # W,b - Weight and bias matrices and vectors |
|  | # activationFunc - Activation function - sigmoid, tanh, relu etc |
|  | # Returns : The Activation of this layer |
|  | # : |
|  | # Z = W \* X + b |
|  | # A = sigmoid(Z), A= Relu(Z), A= tanh(Z) |
|  | layerActivationForward <- function(A\_prev, W, b, activationFunc){ |
|  |  |
|  | # Compute Z |
|  | z = W %\*% A\_prev |
|  | Z <-sweep(z,1,b,'+') |
|  |  |
|  | forward\_cache <- list("A\_prev"=A\_prev, "W"=W, "b"=b) |
|  | # Compute the activation for sigmoid |
|  | if(activationFunc == "sigmoid"){ |
|  | vals = sigmoid(Z) |
|  | } else if (activationFunc == "relu"){ # Compute the activation for relu |
|  | vals = relu(Z) |
|  | } else if(activationFunc == 'tanh'){ # Compute the activation for tanh |
|  | vals = tanhActivation(Z) |
|  | } else if(activationFunc == 'softmax'){ |
|  | vals = softmax(Z) |
|  | } |
|  |  |
|  | cache <- list("forward\_cache"=forward\_cache, "activation\_cache"=vals[['Z']]) |
|  | retvals <- list("A"=vals[['A']],"cache"=cache) |
|  | return(retvals) |
|  | } |
|  |  |
|  |  |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  | layerActivationBackward <- function(dA, cache, y, activationFunc){ |
|  | # Get A\_prev,W,b |
|  | forward\_cache <-cache[['forward\_cache']] |
|  | activation\_cache <- cache[['activation\_cache']] |
|  | A\_prev <- forward\_cache[['A\_prev']] |
|  | numtraining = dim(A\_prev)[2] |
|  | # Get Z |
|  |  |
|  | if(activationFunc == "relu"){ |
|  | dZ <- reluDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "sigmoid"){ |
|  | dZ <- sigmoidDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "tanh"){ |
|  | dZ <- tanhDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "softmax"){ |
|  | dZ <- softmaxDerivative(dA, activation\_cache,y,numtraining) |
|  | } |
|  | #print(dZ) |
|  | if (activationFunc == 'softmax'){ |
|  | W <- forward\_cache[['W']] |
|  | b <- forward\_cache[['b']] |
|  | dW = 1/numtraining \* A\_prev%\*%dZ |
|  | db = 1/numtraining\* matrix(colSums(dZ),nrow=1,ncol=3) |
|  | dA\_prev = dZ %\*%W |
|  | } else { |
|  | W <- forward\_cache[['W']] |
|  | b <- forward\_cache[['b']] |
|  | numtraining = dim(A\_prev)[2] |
|  | dW = 1/numtraining \* dZ %\*% t(A\_prev) |
|  | db = 1/numtraining \* rowSums(dZ) |
|  | dA\_prev = t(W) %\*% dZ |
|  | } |
|  | retvals <- list("dA\_prev"=dA\_prev,"dW"=dW,"db"=db) |
|  | return(retvals) |
|  | } |
|  |  |
|  |  |
|  |  |
|  |  |
|  | # Plot a decision boundary |
|  | # This function uses ggplot2 |
|  | plotDecisionBoundary <- function(Z,W1,b1,W2,b2){ |
|  | xmin<-min(Z[,1]) |
|  | xmax<-max(Z[,1]) |
|  | ymin<-min(Z[,2]) |
|  | ymax<-max(Z[,2]) |
|  |  |
|  | # Create a grid of points |
|  | a=seq(xmin,xmax,length=100) |
|  | b=seq(ymin,ymax,length=100) |
|  | grid <- expand.grid(x=a, y=b) |
|  | colnames(grid) <- c('x1', 'x2') |
|  | grid1 <-t(grid) |
|  |  |
|  |  |
|  | # Predict the output based on the grid of points |
|  | retvals <- layerActivationForward(grid1,W1,b1,'relu') |
|  | A1 <- retvals[['A']] |
|  | cache1 <- retvals[['cache']] |
|  | forward\_cache1 <- cache1[['forward\_cache1']] |
|  | activation\_cache <- cache1[['activation\_cache']] |
|  |  |
|  | retvals = layerActivationForward(A1,W2,b2,'softmax') |
|  | A2 <- retvals[['A']] |
|  | cache2 <- retvals[['cache']] |
|  | forward\_cache2 <- cache2[['forward\_cache1']] |
|  | activation\_cache2 <- cache2[['activation\_cache']] |
|  |  |
|  | # From the softmax probabilities pick the one with the highest probability |
|  | q= apply(A2,1,which.max) |
|  |  |
|  |  |
|  | ### |
|  |  |
|  |  |
|  | q1 <- t(data.frame(q)) |
|  | q2 <- as.numeric(q1) |
|  | grid2 <- cbind(grid,q2) |
|  | colnames(grid2) <- c('x1', 'x2','q2') |
|  |  |
|  | z1 <- data.frame(Z) |
|  | names(z1) <- c("x1","x2","y") |
|  | atitle=paste("Decision boundary") |
|  | ggplot(z1) + |
|  | geom\_point(data = z1, aes(x = x1, y = x2, color = y)) + |
|  | stat\_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2), alpha = 0.9)+ |
|  | ggtitle(atitle) + scale\_colour\_gradientn(colours = brewer.pal(10, "Spectral")) |
|  | } |
|  |  |
|  | # Predict the output |
|  | computeScores <- function(parameters, X,hiddenActivationFunc='relu'){ |
|  |  |
|  | fwdProp <- forwardPropagationDeep(X, parameters,hiddenActivationFunc) |
|  | scores <- fwdProp$AL |
|  |  |
|  | return (scores) |
|  | } |

DLFunctions41.py

|  |
| --- |
| import numpy as np |
|  | import matplotlib.pyplot as plt |
|  | import matplotlib |
|  | import matplotlib.pyplot as plt |
|  | from matplotlib import cm |
|  |  |
|  |  |
|  |  |
|  | # Conmpute the Relu of a vector |
|  | def relu(Z): |
|  | A = np.maximum(0,Z) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Conmpute the softmax of a vector |
|  | def softmax(Z): |
|  | # get unnormalized probabilities |
|  | exp\_scores = np.exp(Z.T) |
|  | # normalize them for each example |
|  | A = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Compute the detivative of Relu |
|  | def reluDerivative(dA, cache): |
|  |  |
|  | Z = cache |
|  | dZ = np.array(dA, copy=True) # just converting dz to a correct object. |
|  | # When z <= 0, you should set dz to 0 as well. |
|  | dZ[Z <= 0] = 0 |
|  | return dZ |
|  |  |
|  |  |
|  | # Compute the derivative of softmax |
|  | def softmaxDerivative(dA, cache,y,numTraining): |
|  | # Note : dA not used. dL/dZ = dL/dA \* dA/dZ = pi-yi |
|  | Z = cache |
|  | # Compute softmax |
|  | exp\_scores = np.exp(Z.T) |
|  | # normalize them for each example |
|  | probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) |
|  |  |
|  | # compute the gradient on scores |
|  | dZ = probs |
|  | # dZ = pi- yi |
|  | dZ[range(int(numTraining)),y] -= 1 |
|  | return(dZ) |
|  |  |
|  |  |
|  | # Initialize the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  | def initializeModel(numFeats,numHidden,numOutput): |
|  | np.random.seed(1) |
|  | W1=np.random.randn(numHidden,numFeats)\*0.01 # Multiply by .01 |
|  | b1=np.zeros((numHidden,1)) |
|  | W2=np.random.randn(numOutput,numHidden)\*0.01 |
|  | b2=np.zeros((numOutput,1)) |
|  |  |
|  | # Create a dictionary of the neural network parameters |
|  | nnParameters={'W1':W1,'b1':b1,'W2':W2,'b2':b2} |
|  | return(nnParameters) |
|  |  |
|  |  |
|  |  |
|  | # Compute the activation at a layer 'l' for forward prop in a Deep Network |
|  | # Input : A\_prec - Activation of previous layer |
|  | # W,b - Weight and bias matrices and vectors |
|  | # activationFunc - Activation function - sigmoid, tanh, relu etc |
|  | # Returns : The Activation of this layer |
|  | # : |
|  | # Z = W \* X + b |
|  | # A = sigmoid(Z), A= Relu(Z), A= tanh(Z) |
|  | def layerActivationForward(A\_prev, W, b, activationFunc): |
|  |  |
|  | # Compute Z |
|  | Z = np.dot(W,A\_prev) + b |
|  | forward\_cache = (A\_prev, W, b) |
|  | # Compute the activation for sigmoid |
|  | if activationFunc == "sigmoid": |
|  | A, activation\_cache = sigmoid(Z) |
|  | # Compute the activation for Relu |
|  | elif activationFunc == "relu": |
|  | A, activation\_cache = relu(Z) |
|  | # Compute the activation for tanh |
|  | elif activationFunc == 'tanh': |
|  | A, activation\_cache = tanh(Z) |
|  | elif activationFunc == 'softmax': |
|  | A, activation\_cache = softmax(Z) |
|  | cache = (forward\_cache, activation\_cache) |
|  | return A, cache |
|  |  |
|  |  |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  | def layerActivationBackward(dA, cache, y, activationFunc): |
|  | forward\_cache, activation\_cache = cache |
|  | A\_prev, W, b = forward\_cache |
|  | numtraining = float(A\_prev.shape[1]) |
|  | if activationFunc == "relu": |
|  | dZ = reluDerivative(dA, activation\_cache) |
|  | elif activationFunc == "sigmoid": |
|  | dZ = sigmoidDerivative(dA, activation\_cache) |
|  | elif activationFunc == "tanh": |
|  | dZ = tanhDerivative(dA, activation\_cache) |
|  | elif activationFunc == "softmax": |
|  | dZ = softmaxDerivative(dA, activation\_cache,y,numtraining) |
|  |  |
|  | if activationFunc == 'softmax': |
|  | dW = 1/numtraining \* np.dot(A\_prev,dZ) |
|  | db = np.sum(dZ, axis=0, keepdims=True) |
|  | dA\_prev = np.dot(dZ,W) |
|  | else: |
|  | #print(numtraining) |
|  | dW = 1/numtraining \*(np.dot(dZ,A\_prev.T)) |
|  | #print("dW=",dW) |
|  | db = 1/numtraining \* np.sum(dZ, axis=1, keepdims=True) |
|  | #print("db=",db) |
|  | dA\_prev = np.dot(W.T,dZ) |
|  |  |
|  | return dA\_prev, dW, db |
|  |  |
|  |  |
|  | # Plot a decision boundary |
|  | # Input : Input Model, |
|  | # X |
|  | # Y |
|  | # sz - Num of hiden units |
|  | # lr - Learning rate |
|  | # Fig to be saved as |
|  | # Returns Null |
|  | def plot\_decision\_boundary(X, y,W1,b1,W2,b2,fig1): |
|  | #plot\_decision\_boundary(lambda x: predict(parameters, x.T), x1,y1.T,str(0.3),"fig2.png") |
|  | h = 0.02 |
|  | x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1 |
|  | y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1 |
|  | xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), |
|  | np.arange(y\_min, y\_max, h)) |
|  | Z = np.dot(np.maximum(0, np.dot(np.c\_[xx.ravel(), yy.ravel()], W1.T) + b1.T), W2.T) + b2.T |
|  | Z = np.argmax(Z, axis=1) |
|  | Z = Z.reshape(xx.shape) |
|  |  |
|  | fig = plt.figure() |
|  | plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8) |
|  | plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral) |
|  | plt.xlim(xx.min(), xx.max()) |
|  | plt.ylim(yy.min(), yy.max()) |