**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}}}  
S_{j} =\frac{e^{Z_{j}}}{\sum_{i}^{k}e^{Z_{i}}} = \frac{Ce^{Z_{j}}}{C\sum_{i}^{k}e^{Z_{i}}} = \frac{e^{Z_{j}+log(C)}}{\sum_{i}^{k}e^{Z_{i}+log(C)}}  
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},... Z_{k})

Here is the stable Softmax implementation in Python

# A numerically stable Softmax implementation

def stableSoftmax(Z):

#Compute the softmax of vector x in a numerically stable way.

shiftZ = Z.T - np.max(Z.T,axis=1).reshape(-1,1)

exp\_scores = np.exp(shiftZ)

# normalize them for each example

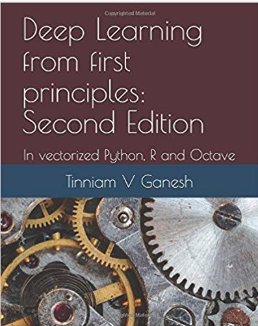
A = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True)

cache=Z

return A,cache

While trying to create a 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(Part 3 & Part4) , 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.

Checkout my book ‘Deep Learning from first principles: Second Edition – In vectorized Python, R and Octave’. My book starts with the implementation of a simple 2-layer Neural Network and works its way to a generic L-Layer Deep Learning Network, with all the bells and whistles. The derivations have been discussed in detail. The code has been extensively commented and included in its entirety in the Appendix sections.

**1. Random dataset with Sigmoid activation – Python**

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import pandas as pd

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

exec(open("DLfunctions51.py").read()) # Cannot import in Rmd.

# Create a random data set with 9 centeres

X1, Y1 = make\_blobs(n\_samples = 400, n\_features = 2, centers = 9,cluster\_std = 1.3, random\_state =4)

#Create 2 classes

Y1=Y1.reshape(400,1)

Y1 = Y1 % 2

X2=X1.T

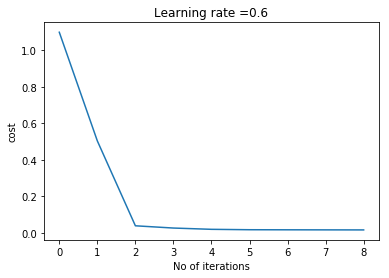
Y2=Y1.T

# Set the dimensions of L -layer DL network

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

# Execute DL network with hidden activation=relu and sigmoid output function

parameters = L\_Layer\_DeepModel(X2, Y2, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid",learningRate = 0.3,num\_iterations = 2500, print\_cost = True)

****

**2. Spiral dataset with Softmax activation – Python**

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import pandas as pd

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

exec(open("DLfunctions51.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

X1=X.T

Y1=y.reshape(-1,1).T

numHidden=100 # No of hidden units in hidden layer

numFeats= 2 # dimensionality

numOutput = 3 # number of classes

# Set the dimensions of the layers

layersDimensions=[numFeats,numHidden,numOutput]

parameters = L\_Layer\_DeepModel(X1, Y1, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="softmax",learningRate = 0.6,num\_iterations = 9000, print\_cost = True)

## Cost after iteration 0: 1.098759

## Cost after iteration 1000: 0.112666

## Cost after iteration 2000: 0.044351

## Cost after iteration 3000: 0.027491

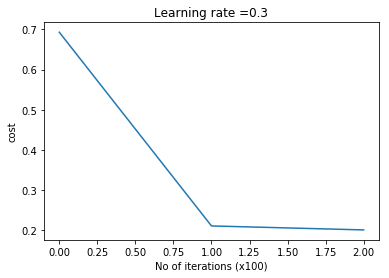
## Cost after iteration 4000: 0.021898

## Cost after iteration 5000: 0.019181

## Cost after iteration 6000: 0.017832

## Cost after iteration 7000: 0.017452

## Cost after iteration 8000: 0.017161

****

**3. 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 also tweaking the learning rate and the number of epochs.

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import pandas as pd

import math

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

from sklearn.metrics import confusion\_matrix

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

exec(open("DLfunctions51.py").read())

exec(open("load\_mnist.py").read())

# Read the MNIST training and test sets

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

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

# Create labels and pixel arrays

lbls=[]

pxls=[]

print(len(training))

#for i in range(len(training)):

for i in range(60000):

l,p=training[i]

lbls.append(l)

pxls.append(p)

labels= np.array(lbls)

pixels=np.array(pxls)

y=labels.reshape(-1,1)

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

X1=X.T

Y1=y.T

# Set the dimensions of the layers. The MNIST data is 28x28 pixels= 784

# Hence input layer is 784. For the 10 digits the Softmax classifier

# has to handle 10 outputs

layersDimensions=[784, 15,9,10] # Works very well,lr=0.01,mini\_batch =1000, total=20000

np.random.seed(1)

costs = []

# Run Stochastic Gradient Descent with Learning Rate=0.01, mini batch size=1000

# number of epochs=3000

parameters = L\_Layer\_DeepModel\_SGD(X1, Y1, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="softmax",learningRate = 0.01 ,mini\_batch\_size =1000, num\_epochs = 3000, print\_cost = True)

# Compute the Confusion Matrix on Training set

# Compute the training accuracy, precision and recall

proba=predict\_proba(parameters, X1,outputActivationFunc="softmax")

#A2, cache = forwardPropagationDeep(X1, parameters)

#proba=np.argmax(A2, axis=0).reshape(-1,1)

a=confusion\_matrix(Y1.T,proba)

print(a)

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

print('Accuracy: {:.2f}'.format(accuracy\_score(Y1.T, proba)))

print('Precision: {:.2f}'.format(precision\_score(Y1.T, proba,average="micro")))

print('Recall: {:.2f}'.format(recall\_score(Y1.T, proba,average="micro")))

# Read the test data

lbls=[]

pxls=[]

print(len(test))

for i in range(10000):

l,p=test[i]

lbls.append(l)

pxls.append(p)

testLabels= np.array(lbls)

testPixels=np.array(pxls)

ytest=testLabels.reshape(-1,1)

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

X1test=Xtest.T

Y1test=ytest.T

# Compute the Confusion Matrix on Test set

# Compute the test accuracy, precision and recall

probaTest=predict\_proba(parameters, X1test,outputActivationFunc="softmax")

#A2, cache = forwardPropagationDeep(X1, parameters)

#proba=np.argmax(A2, axis=0).reshape(-1,1)

a=confusion\_matrix(Y1test.T,probaTest)

print(a)

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

print('Accuracy: {:.2f}'.format(accuracy\_score(Y1test.T, probaTest)))

print('Precision: {:.2f}'.format(precision\_score(Y1test.T, probaTest,average="micro")))

print('Recall: {:.2f}'.format(recall\_score(Y1test.T, probaTest,average="micro")))

##1. Confusion Matrix of Training set

0 1 2 3 4 5 6 7 8 9

## [[5854 0 19 2 10 7 0 1 24 6]

## [ 1 6659 30 10 5 3 0 14 20 0]

## [ 20 24 5805 18 6 11 2 32 37 3]

## [ 5 4 175 5783 1 27 1 58 60 17]

## [ 1 21 9 0 5780 0 5 2 12 12]

## [ 29 9 21 224 6 4824 18 17 245 28]

## [ 5 4 22 1 32 12 5799 0 43 0]

## [ 3 13 148 154 18 3 0 5883 4 39]

## [ 11 34 30 21 13 16 4 7 5703 12]

## [ 10 4 1 32 135 14 1 92 134 5526]]

##2. Accuracy, Precision, Recall of Training set

## Accuracy: 0.96

## Precision: 0.96

## Recall: 0.96

##3. Confusion Matrix of Test set

0 1 2 3 4 5 6 7 8 9

## [[ 954 1 8 0 3 3 2 4 4 1]

## [ 0 1107 6 5 0 0 1 2 14 0]

## [ 11 7 957 10 5 0 5 20 16 1]

## [ 2 3 37 925 3 13 0 8 18 1]

## [ 2 6 1 1 944 0 7 3 4 14]

## [ 12 5 4 45 2 740 24 8 42 10]

## [ 8 4 4 2 16 9 903 0 12 0]

## [ 4 10 27 18 5 1 0 940 1 22]

## [ 11 13 6 13 9 10 7 2 900 3]

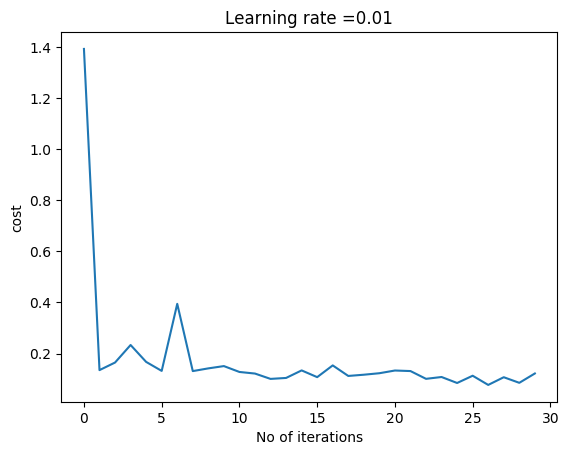
## [ 8 5 1 7 50 7 0 20 29 882]]

##4. Accuracy, Precision, Recall of Training set

## Accuracy: 0.93

## Precision: 0.93

## Recall: 0.93

****

**4. Random dataset with Sigmoid activation – R code**

This is the random data set used in the Python code above which was saved as a CSV. The code is used to test a L -Layer DL network with Sigmoid Activation in R.

source("DLfunctions5.R")

# Read the random data set

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

x <- z[,1:2]

y <- z[,3]

X <- t(x)

Y <- t(y)

# Set the dimensions of the layer

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

# Run Gradient Descent on the data set with relu hidden unit activation

# sigmoid activation unit in the output layer

retvals = L\_Layer\_DeepModel(X, Y, layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="sigmoid",

learningRate = 0.3,

numIterations = 5000,

print\_cost = True)

#Plot the cost vs iterations

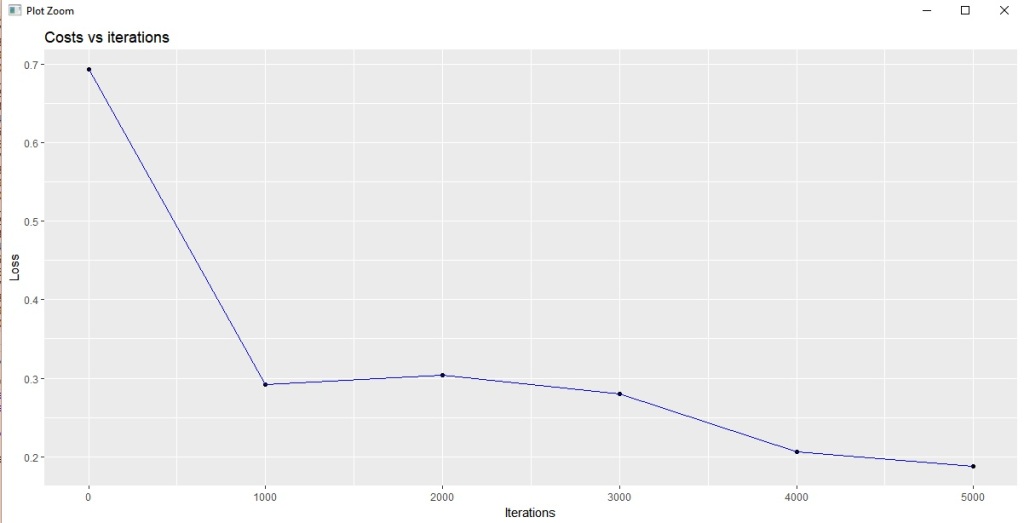
iterations <- seq(0,5000,1000)

costs=retvals$costs

df=data.frame(iterations,costs)

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

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

****

**5. Spiral dataset with Softmax activation – R**

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

source("DLfunctions5.R")

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

# Setup the data

X <- Z[,1:2]

y <- Z[,3]

X <- t(X)

Y <- 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

# Set the layer dimensions

layersDimensions = c(numFeats,numHidden,numOutput)

# Perform gradient descent with relu activation unit for hidden layer

# and softmax activation in the output

retvals = L\_Layer\_DeepModel(X, Y, layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.5,

numIterations = 9000,

print\_cost = True)

#Plot cost vs iterations

iterations <- seq(0,9000,1000)

costs=retvals$costs

df=data.frame(iterations,costs)

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

ggtitle("Costs vs iterations") + xlab("Iterations") + ylab("Costs")

**6. 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 entire 60000 data set was used to train the data. R takes almost 8 hours to process this data set with a mini-batch size of 1000.  The use of ‘for’ loops is limited to iterating through epochs, mini batches and for creating the mini batches itself. All other code is vectorized. Yet, it seems to crawl. Most likely the use of ‘lists’ in R, to return multiple values is performance intensive. Some day, I will try to profile the code, and see where the issue is. However the code works!

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!

source("DLfunctions5.R")

source("mnist.R")

#Load the mnist data

load\_mnist()

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

#Set the layer dimensions

layersDimensions=c(784, 15,9, 10) # Works at 1500

x <- t(train$x)

X <- x[,1:60000]

y <-train$y

y1 <- y[1:60000]

y2 <- as.matrix(y1)

Y=t(y2)

# Subset 32768 random samples from MNIST

permutation = c(sample(2^15))

# Randomly shuffle the training data

X1 = X[, permutation]

y1 = Y[1, permutation]

y2 <- as.matrix(y1)

Y1=t(y2)

# Execute Stochastic Gradient Descent on the entire training set

# with Softmax activation

retvalsSGD= L\_Layer\_DeepModel\_SGD(X1, Y1, layersDimensions,

hiddenActivationFunc='relu',

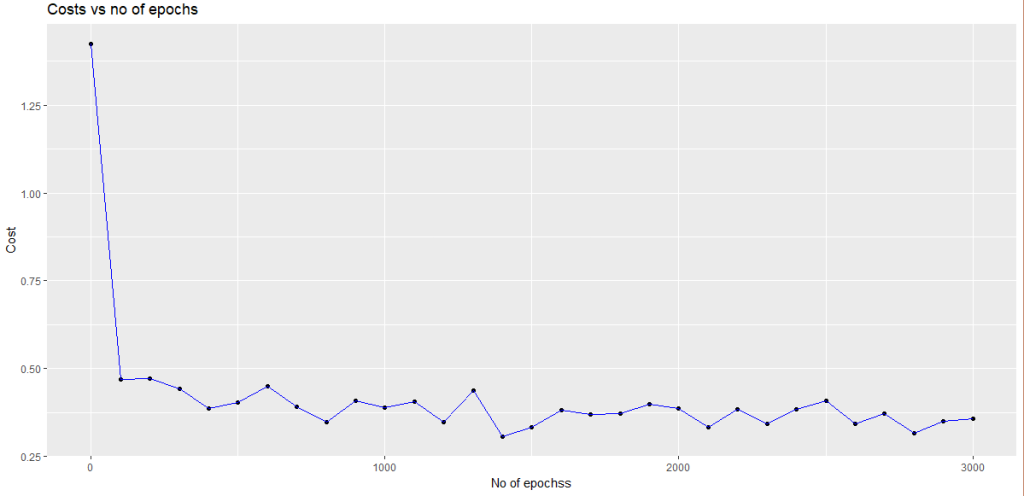
outputActivationFunc="softmax",

learningRate = 0.05,

mini\_batch\_size = 512,

num\_epochs = 1,

print\_cost = True)



# Compute the Confusion Matrix

library(caret)

library(e1071)

predictions=predictProba(retvalsSGD[['parameters']], X,hiddenActivationFunc='relu',

outputActivationFunc="softmax")

confusionMatrix(predictions,Y)

# Confusion Matrix on the Training set

> confusionMatrix(predictions,Y)

Confusion Matrix and Statistics

Reference

Prediction 0 1 2 3 4 5 6 7 8 9

0 5738 1 21 5 16 17 7 15 9 43

1 5 6632 21 24 25 3 2 33 13 392

2 12 32 5747 106 25 28 3 27 44 4779

3 0 27 12 5715 1 21 1 20 1 13

4 10 5 21 18 5677 9 17 30 15 166

5 142 21 96 136 93 5306 5884 43 60 413

6 0 0 0 0 0 0 0 0 0 0

7 6 9 13 13 3 4 0 6085 0 55

8 8 12 7 43 1 32 2 7 5703 69

9 2 3 20 71 1 1 2 5 6 19

Overall Statistics

Accuracy : 0.777

95% CI : (0.7737, 0.7804)

No Information Rate : 0.1124

P-Value [Acc > NIR] : < 2.2e-16

Kappa : 0.7524

Mcnemar's Test P-Value : NA

Statistics by Class:

Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5 Class: 6

Sensitivity 0.96877 0.9837 0.96459 0.93215 0.97176 0.97879 0.00000

Specificity 0.99752 0.9903 0.90644 0.99822 0.99463 0.87380 1.00000

Pos Pred Value 0.97718 0.9276 0.53198 0.98348 0.95124 0.43513 NaN

Neg Pred Value 0.99658 0.9979 0.99571 0.99232 0.99695 0.99759 0.90137

Prevalence 0.09872 0.1124 0.09930 0.10218 0.09737 0.09035 0.09863

Detection Rate 0.09563 0.1105 0.09578 0.09525 0.09462 0.08843 0.00000

Detection Prevalence 0.09787 0.1192 0.18005 0.09685 0.09947 0.20323 0.00000

Balanced Accuracy 0.98314 0.9870 0.93551 0.96518 0.98319 0.92629 0.50000

Class: 7 Class: 8 Class: 9

Sensitivity 0.9713 0.97471 0.0031938

Specificity 0.9981 0.99666 0.9979464

Pos Pred Value 0.9834 0.96924 0.1461538

Neg Pred Value 0.9967 0.99727 0.9009521

Prevalence 0.1044 0.09752 0.0991500

Detection Rate 0.1014 0.09505 0.0003167

Detection Prevalence 0.1031 0.09807 0.0021667

Balanced Accuracy 0.9847 0.98568 0.5005701

# Confusion Matrix on the Training set xtest <- t(test$x) Xtest <- xtest[,1:10000] ytest <-test$y ytest1 <- ytest[1:10000] ytest2 <- as.matrix(ytest1) Ytest=t(ytest2)

Confusion Matrix and Statistics

Reference

Prediction 0 1 2 3 4 5 6 7 8 9

0 950 2 2 3 0 6 9 4 7 6

1 3 1110 4 2 9 0 3 12 5 74

2 2 6 965 21 9 14 5 16 12 789

3 1 2 9 908 2 16 0 21 2 6

4 0 1 9 5 938 1 8 6 8 39

5 19 5 25 35 20 835 929 8 54 67

6 0 0 0 0 0 0 0 0 0 0

7 4 4 7 10 2 4 0 952 5 6

8 1 5 8 14 2 16 2 3 876 21

9 0 0 3 12 0 0 2 6 5 1

Overall Statistics

Accuracy : 0.7535

95% CI : (0.7449, 0.7619)

No Information Rate : 0.1135

P-Value [Acc > NIR] : < 2.2e-16

Kappa : 0.7262

Mcnemar's Test P-Value : NA

Statistics by Class:

Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5 Class: 6

Sensitivity 0.9694 0.9780 0.9351 0.8990 0.9552 0.9361 0.0000

Specificity 0.9957 0.9874 0.9025 0.9934 0.9915 0.8724 1.0000

Pos Pred Value 0.9606 0.9083 0.5247 0.9390 0.9241 0.4181 NaN

Neg Pred Value 0.9967 0.9972 0.9918 0.9887 0.9951 0.9929 0.9042

Prevalence 0.0980 0.1135 0.1032 0.1010 0.0982 0.0892 0.0958

Detection Rate 0.0950 0.1110 0.0965 0.0908 0.0938 0.0835 0.0000

Detection Prevalence 0.0989 0.1222 0.1839 0.0967 0.1015 0.1997 0.0000

Balanced Accuracy 0.9825 0.9827 0.9188 0.9462 0.9733 0.9043 0.5000

Class: 7 Class: 8 Class: 9

Sensitivity 0.9261 0.8994 0.0009911

Specificity 0.9953 0.9920 0.9968858

Pos Pred Value 0.9577 0.9241 0.0344828

Neg Pred Value 0.9916 0.9892 0.8989068

Prevalence 0.1028 0.0974 0.1009000

Detection Rate 0.0952 0.0876 0.0001000

Detection Prevalence 0.0994 0.0948 0.0029000

Balanced Accuracy 0.9607 0.9457 0.4989384

**7. Random dataset with Sigmoid activation – Octave**

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

source("DL5functions.m")

# Read the data

data=csvread("data.csv");

X=data(:,1:2);

Y=data(:,3);

#Set the layer dimensions

layersDimensions = [2 9 7 1]; #tanh=-0.5(ok), #relu=0.1 best!

# Perform gradient descent

[weights biases costs]=L\_Layer\_DeepModel(X', Y', layersDimensions,

hiddenActivationFunc='relu',

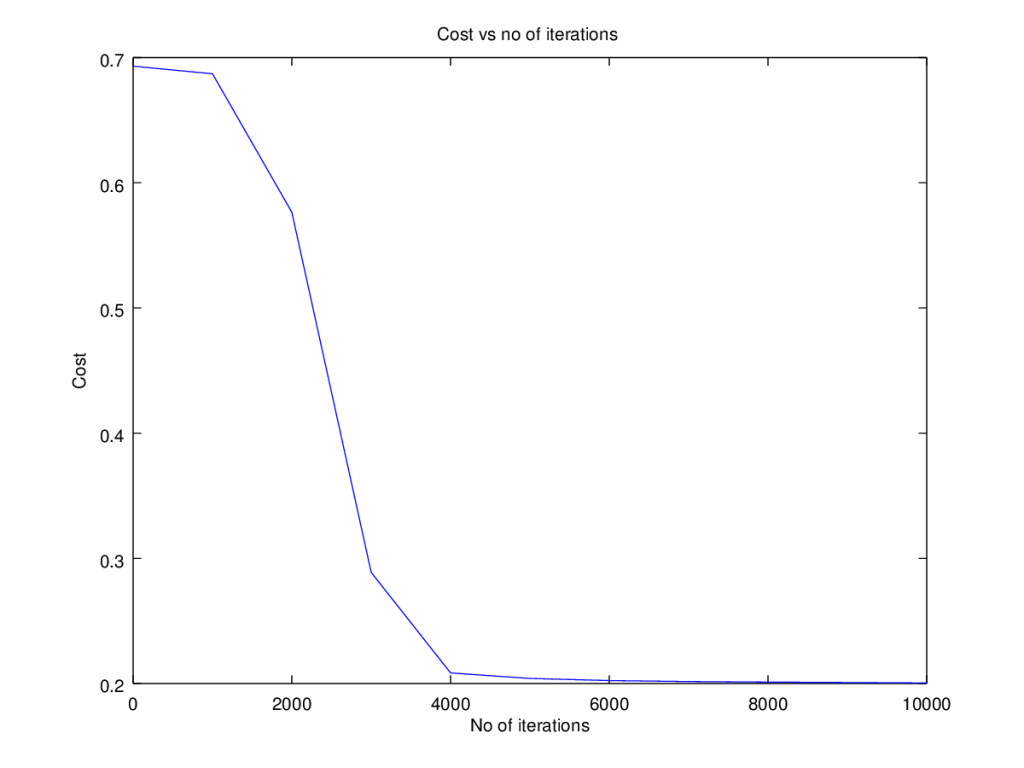
outputActivationFunc="sigmoid",

learningRate = 0.1,

numIterations = 10000);

# Plot cost vs iterations

plotCostVsIterations(10000,costs);



**8. Spiral dataset with Softmax activation – Octave**

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

# Read the 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 classess

numFeats=2; #No features

numHidden=100; # No of hidden units

numOutput=3; # No of classes

# Set the layer dimensions

layersDimensions = [numFeats numHidden numOutput];

#Perform gradient descent with softmax activation unit

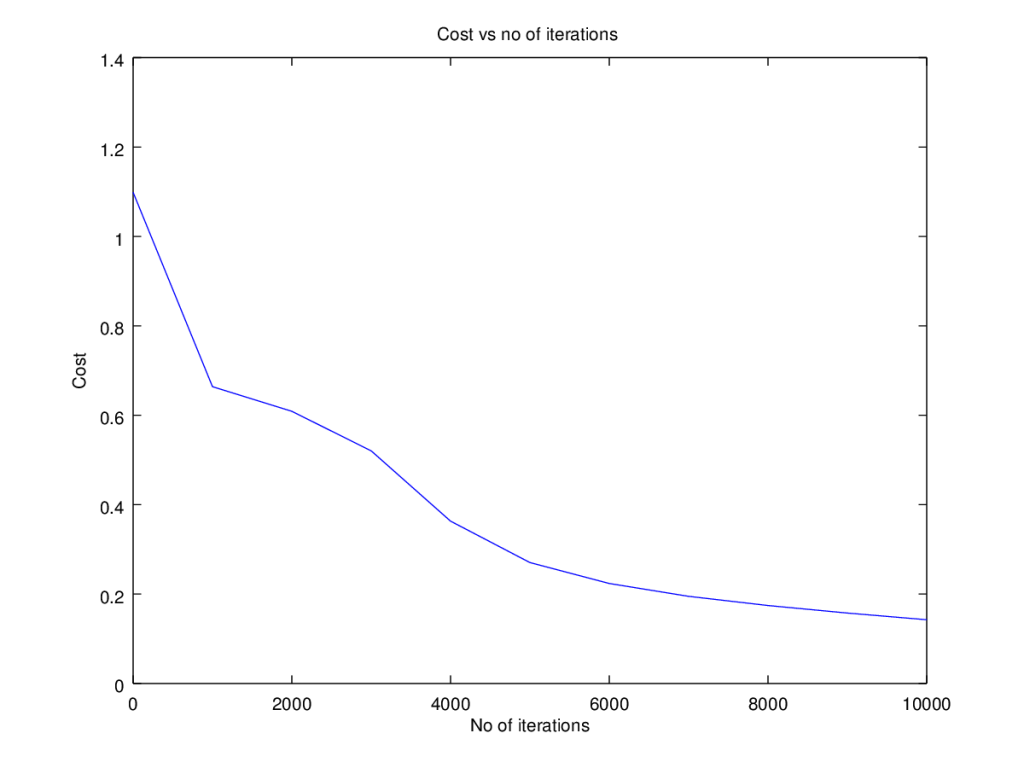
[weights biases costs]=L\_Layer\_DeepModel(X', Y', layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.1,

numIterations = 10000);



**9. 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.

I thought if the performance was not good, I could iterate through these random batches and refining the gradients as follows

# Pseudo code that could be used since Octave only allows 10K batches

# at a time

# Randomly create weights

[weights biases] = initialize\_weights()

for i=1:k

# Create a random permutation and create a random batch

permutation = randperm(10000);

X=trainX(permutation,:);

Y=trainY(permutation,:);

# Compute weights from SGD and update weights in the next batch update

[weights biases costs]=L\_Layer\_DeepModel\_SGD(X,Y,mini\_bactch=1000,weights, biases,...);

   ...

endfor

# Load the MNIST data

load('./mnist/mnist.txt.gz');

#Create a random permutatation from 60K

permutation = randperm(10000);

disp(length(permutation));

# Use this 10K as the batch

X=trainX(permutation,:);

Y=trainY(permutation,:);

# Set layer dimensions

layersDimensions=[784, 15, 9, 10];

# Run Stochastic Gradient descent with batch size=10K and mini\_batch\_size=1000

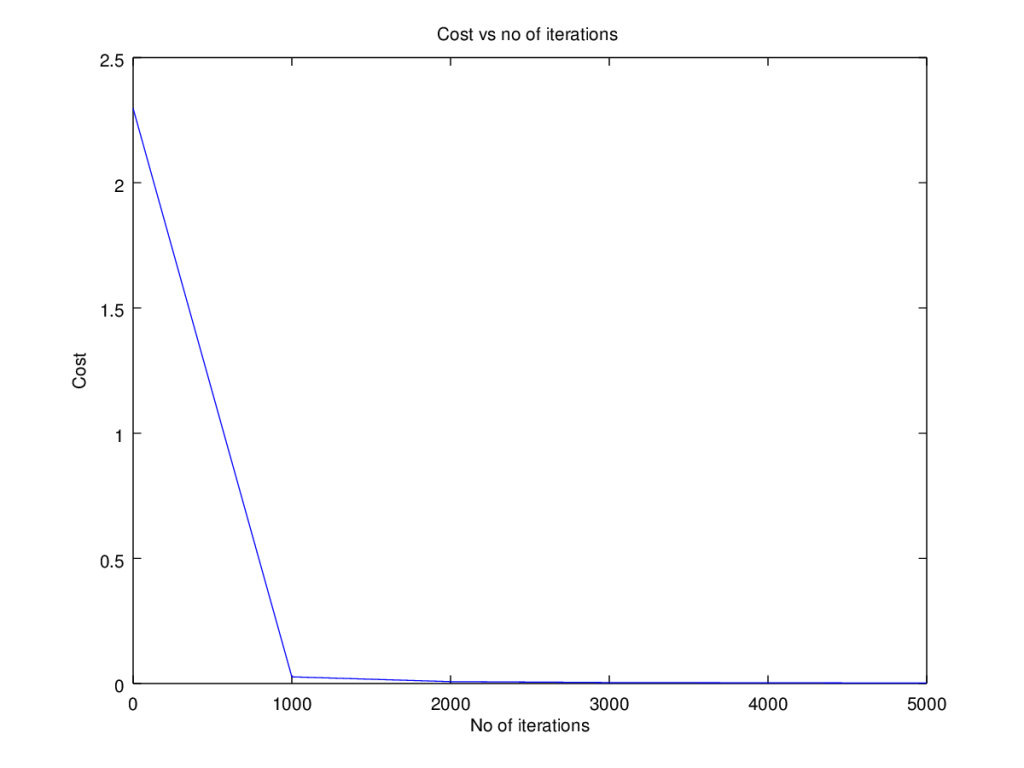
[weights biases costs]=L\_Layer\_DeepModel\_SGD(X', Y', layersDimensions,

hiddenActivationFunc='relu',

outputActivationFunc="softmax",

learningRate = 0.01,

mini\_batch\_size = 2000, num\_epochs = 5000);



**9. 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 libaries of R and Python. They possibly exist in its sibling, Matlab

DL5Functions.m

|  |
| --- |
| 1; |
|  | # Define sigmoid function |
|  | function [A,cache] = sigmoid(Z) |
|  | A = 1 ./ (1+ exp(-Z)); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Relu function |
|  | function [A,cache] = relu(Z) |
|  | A = max(0,Z); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Relu function |
|  | function [A,cache] = tanhAct(Z) |
|  | A = tanh(Z); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define 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 Softmax function |
|  | function [A,cache] = stableSoftmax(Z) |
|  | # Normalize by max value in each row |
|  | shiftZ = Z' - max(Z',[],2); |
|  | exp\_scores = exp(shiftZ); |
|  | # normalize them for each example |
|  | A = exp\_scores ./ sum(exp\_scores,2); |
|  | #disp("sm") |
|  | #disp(A); |
|  | cache=Z; |
|  | end |
|  |  |
|  | # Define Relu Derivative |
|  | function [dZ] = reluDerivative(dA,cache) |
|  | Z = cache; |
|  | dZ = dA; |
|  | # Get elements that are greater than 0 |
|  | a = (Z > 0); |
|  | # Select only those elements where Z > 0 |
|  | dZ = dZ .\* a; |
|  | end |
|  |  |
|  | # Define Sigmoid Derivative |
|  | function [dZ] = sigmoidDerivative(dA,cache) |
|  | Z = cache; |
|  | s = 1 ./ (1+ exp(-Z)); |
|  | dZ = dA .\* s .\* (1-s); |
|  | end |
|  |  |
|  | # Define Tanh Derivative |
|  | function [dZ] = tanhDerivative(dA,cache) |
|  | Z = cache; |
|  | a = tanh(Z); |
|  | dZ = dA .\* (1 - a .^ 2); |
|  | end |
|  |  |
|  | # Populate a matrix with 1s in rows where Y=1 |
|  | # This function may need to be modified if K is not 3, 10 |
|  | function [Y1] = popMatrix(Y,numClasses) |
|  | Y1=zeros(length(Y),numClasses); |
|  | if(numClasses==3) # For 3 output classes |
|  | Y1(Y==0,1)=1; |
|  | Y1(Y==1,2)=1; |
|  | Y1(Y==2,3)=1; |
|  | elseif(numClasses==10) # For 10 output classes |
|  | Y1(Y==0,1)=1; |
|  | Y1(Y==1,2)=1; |
|  | Y1(Y==2,3)=1; |
|  | Y1(Y==3,4)=1; |
|  | Y1(Y==4,5)=1; |
|  | Y1(Y==5,6)=1; |
|  | Y1(Y==6,7)=1; |
|  | Y1(Y==7,8)=1; |
|  | Y1(Y==8,9)=1; |
|  | Y1(Y==9,10)=1; |
|  |  |
|  | endif |
|  | end |
|  |  |
|  | # Define Softmax Derivative |
|  | function [dZ] = softmaxDerivative(dA,cache,Y, numClasses) |
|  | Z = cache; |
|  | # get unnormalized probabilities |
|  | shiftZ = Z' - max(Z',[],2); |
|  | exp\_scores = exp(shiftZ); |
|  |  |
|  | # normalize them for each example |
|  | probs = exp\_scores ./ sum(exp\_scores,2); |
|  | # dZ = pi- yi |
|  | yi=popMatrix(Y,numClasses); |
|  | dZ=probs-yi; |
|  |  |
|  | end |
|  |  |
|  | # Define Softmax Derivative |
|  | function [dZ] = stableSoftmaxDerivative(dA,cache,Y, numClasses) |
|  | Z = cache; |
|  | # get unnormalized probabilities |
|  | exp\_scores = exp(Z'); |
|  | # normalize them for each example |
|  | probs = exp\_scores ./ sum(exp\_scores,2); |
|  | # dZ = pi- yi |
|  | yi=popMatrix(Y,numClasses); |
|  | 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 |
|  |  |
|  |  |
|  | # Initialize model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | function [W b] = initializeDeepModel(layerDimensions) |
|  | rand ("seed", 3); |
|  | # note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Create cell arrays for Weights and biases |
|  |  |
|  | for l =2:size(layerDimensions)(2) |
|  | W{l-1} = rand(layerDimensions(l),layerDimensions(l-1))\*0.01; # Multiply by .01 |
|  | b{l-1} = zeros(layerDimensions(l),1); |
|  |  |
|  | endfor |
|  | end |
|  |  |
|  | # Compute the activation at a layer 'l' for forward prop in a Deep Network |
|  | # Input : A\_prec - Activation of previous layer |
|  | # W,b - Weight and bias matrices and vectors |
|  | # activationFunc - Activation function - sigmoid, tanh, relu etc |
|  | # Returns : The Activation of this layer |
|  | # : |
|  | # Z = W \* X + b |
|  | # A = sigmoid(Z), A= Relu(Z), A= tanh(Z) |
|  | function [A forward\_cache activation\_cache] = layerActivationForward(A\_prev, W, b, activationFunc) |
|  |  |
|  | # Compute Z |
|  | Z = W \* A\_prev +b; |
|  | # Create a cell array |
|  | forward\_cache = {A\_prev W b}; |
|  | # Compute the activation for sigmoid |
|  | if (strcmp(activationFunc,"sigmoid")) |
|  | [A activation\_cache] = sigmoid(Z); |
|  | elseif (strcmp(activationFunc, "relu")) # Compute the activation for Relu |
|  | [A activation\_cache] = relu(Z); |
|  | elseif(strcmp(activationFunc,'tanh')) # Compute the activation for tanh |
|  | [A activation\_cache] = tanhAct(Z); |
|  | elseif(strcmp(activationFunc,'softmax')) # Compute the activation for tanh |
|  | #[A activation\_cache] = softmax(Z); |
|  | [A activation\_cache] = stableSoftmax(Z); |
|  | endif |
|  |  |
|  | end |
|  |  |
|  | # Compute the forward propagation for layers 1..L |
|  | # Input : X - Input Features |
|  | # paramaters: Weights and biases |
|  | # hiddenActivationFunc - Activation function at hidden layers Relu/tanh |
|  | # outputActivationFunc- sigmoid/softmax |
|  | # Returns : AL |
|  | # caches |
|  | # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and sigmoid actiovation at layer L |
|  | function [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights,biases, |
|  | hiddenActivationFunc='relu', outputActivationFunc='sigmoid') |
|  | # Create an empty cell array |
|  | forward\_caches = {}; |
|  | activation\_caches = {}; |
|  | # Set A to X (A0) |
|  | A = X; |
|  | L = length(weights); # number of layers in the neural network |
|  | # Loop through from layer 1 to upto layer L |
|  | for l =1:L-1 |
|  | A\_prev = A; |
|  | # Zi = Wi x Ai-1 + bi and Ai = g(Zi) |
|  | W = weights{l}; |
|  | b = biases{l}; |
|  | [A forward\_cache activation\_cache] = layerActivationForward(A\_prev, W,b, activationFunc=hiddenActivationFunc); |
|  | forward\_caches{l}=forward\_cache; |
|  | activation\_caches{l} = activation\_cache; |
|  | endfor |
|  | # Since this is binary classification use the sigmoid activation function in |
|  | # last layer |
|  | W = weights{L}; |
|  | b = biases{L}; |
|  | [AL, forward\_cache activation\_cache] = layerActivationForward(A, W,b, activationFunc = outputActivationFunc); |
|  | forward\_caches{L}=forward\_cache; |
|  | activation\_caches{L} = activation\_cache; |
|  |  |
|  | end |
|  |  |
|  | # Pick columns where Y==1 |
|  | function [a] = pickColumns(AL,Y,numClasses) |
|  | if(numClasses==3) |
|  | a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)]; |
|  | elseif (numClasses==10) |
|  | a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3);AL(Y==3,4);AL(Y==4,5); |
|  | AL(Y==5,6); AL(Y==6,7);AL(Y==7,8);AL(Y==8,9);AL(Y==9,10)]; |
|  | endif |
|  | end |
|  |  |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # : outputActivationFunc- sigmoid/softmax |
|  | # : numClasses |
|  | # Output: cost |
|  | function [cost]= computeCost(AL, Y, outputActivationFunc="sigmoid",numClasses) |
|  | if(strcmp(outputActivationFunc,"sigmoid")) |
|  | numTraining= size(Y)(2); |
|  | # Element wise multiply for logprobs |
|  | cost = -1/numTraining \* sum((Y .\* log(AL)) + (1-Y) .\* log(1-AL)); |
|  | elseif(strcmp(outputActivationFunc,'softmax')) |
|  | numTraining = size(Y)(2); |
|  | Y=Y'; |
|  | # Select rows where Y=0,1,and 2 and concatenate to a long vector |
|  | #a=[AL(Y==0,1) ;AL(Y==1,2) ;AL(Y==2,3)]; |
|  | a =pickColumns(AL,Y,numClasses); |
|  |  |
|  | #Select the correct column for log prob |
|  | correct\_probs = -log(a); |
|  | #Compute log loss |
|  | cost= sum(correct\_probs)/numTraining; |
|  | endif |
|  | end |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # # outputActivationFunc- sigmoid/softmax |
|  | # # numClasses |
|  | # 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,numClasses) |
|  |  |
|  | A\_prev = forward\_cache{1}; |
|  | W =forward\_cache{2}; |
|  | b = forward\_cache{3}; |
|  | numTraining = size(A\_prev)(2); |
|  | 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,numClasses); |
|  | dZ = stableSoftmaxDerivative(dA, activation\_cache,Y,numClasses); |
|  | endif |
|  |  |
|  |  |
|  | 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 |
|  |  |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : AL: Output of L layer Network - weights |
|  | # # Y Real output |
|  | # # caches -- list of caches containing: |
|  | # every cache of layerActivationForward() with "relu"/"tanh" |
|  | # #(it's caches[l], for l in range(L-1) i.e l = 0...L-2) |
|  | # #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1]) |
|  | # hiddenActivationFunc - Activation function at hidden layers |
|  | # # outputActivationFunc- sigmoid/softmax |
|  | # # numClasses |
|  | # |
|  | # Returns: |
|  | # gradients -- A dictionary with the gradients |
|  | # gradients["dA" + str(l)] = ... |
|  | # gradients["dW" + str(l)] = ... |
|  |  |
|  | function [gradsDA gradsDW gradsDB]= backwardPropagationDeep(AL, Y, activation\_caches,forward\_caches, |
|  | hiddenActivationFunc='relu',outputActivationFunc="sigmoid",numClasses) |
|  |  |
|  |  |
|  | # Set the number of layers |
|  | L = length(activation\_caches); |
|  | m = size(AL)(2); |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | # Initializing the backpropagation |
|  | # dl/dAL= -(y/a + (1-y)/(1-a)) - At the output layer |
|  | dAL = -((Y ./ AL) - (1 - Y) ./ ( 1 - AL)); |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | dAL=0; |
|  | Y=Y'; |
|  | endif |
|  |  |
|  |  |
|  | # Since this is a binary classification the activation at output is sigmoid |
|  | # Get the gradients at the last layer |
|  | # Inputs: "AL, Y, caches". |
|  | # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"] |
|  | activation\_cache = activation\_caches{L}; |
|  | forward\_cache = forward\_caches(L); |
|  | # Note the cell array includes an array of forward caches. To get to this we need to include the index {1} |
|  | [dA dW db] = layerActivationBackward(dAL, forward\_cache{1}, activation\_cache, Y, activationFunc = outputActivationFunc,numClasses); |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | gradsDA{L}= dA; |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | gradsDA{L}= dA';#Note the transpose |
|  | endif |
|  | gradsDW{L}= dW; |
|  | gradsDB{L}= db; |
|  |  |
|  | # Traverse in the reverse direction |
|  | for l =(L-1):-1:1 |
|  | # Compute the gradients for L-1 to 1 for Relu/tanh |
|  | # Inputs: "gradients["dA" + str(l + 2)], caches". |
|  | # Outputs: "gradients["dA" + str(l + 1)] , gradients["dW" + str(l + 1)] , gradients["db" + str(l + 1)] |
|  | activation\_cache = activation\_caches{l}; |
|  | forward\_cache = forward\_caches(l); |
|  |  |
|  | #dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackward(gradients['dA'+str(l+1)], current\_cache, activationFunc = "relu") |
|  | # dAl the dervative of the activation of the lth layer,is the first element |
|  | dAl= gradsDA{l+1}; |
|  | [dA\_prev\_temp, dW\_temp, db\_temp] = layerActivationBackward(dAl, forward\_cache{1}, activation\_cache, Y, activationFunc = hiddenActivationFunc,numClasses); |
|  | gradsDA{l}= dA\_prev\_temp; |
|  | gradsDW{l}= dW\_temp; |
|  | gradsDB{l}= db\_temp; |
|  |  |
|  | endfor |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc |
|  | #output : Updated weights after 1 iteration |
|  | function [weights biases] = gradientDescent(weights, biases,gradsW,gradsB, learningRate,outputActivationFunc="sigmoid") |
|  |  |
|  | L = size(weights)(2); # number of layers in the neural network |
|  |  |
|  | # Update rule for each parameter. |
|  | for l=1:(L-1) |
|  | weights{l} = weights{l} -learningRate\* gradsW{l}; |
|  | biases{l} = biases{l} -learningRate\* gradsB{l}; |
|  | endfor |
|  |  |
|  |  |
|  | if (strcmp(outputActivationFunc,"sigmoid")) |
|  | weights{L} = weights{L} -learningRate\* gradsW{L}; |
|  | biases{L} = biases{L} -learningRate\* gradsB{L}; |
|  | elseif (strcmp(outputActivationFunc,"softmax")) |
|  | weights{L} = weights{L} -learningRate\* gradsW{L}'; |
|  | biases{L} = biases{L} -learningRate\* gradsB{L}'; |
|  | endif |
|  |  |
|  |  |
|  | end |
|  |  |
|  |  |
|  | # Execute a L layer Deep learning model |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : num of iterations |
|  | #output : Updated weights and biases after each iteration |
|  | function [weights biases costs] = L\_Layer\_DeepModel(X, Y, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid",learning\_rate = .3, num\_iterations = 10000)#lr was 0.009 |
|  |  |
|  | rand ("seed", 1); |
|  | costs = [] ; |
|  |  |
|  | # Parameters initialization. |
|  | [weights biases] = initializeDeepModel(layersDimensions); |
|  |  |
|  | # Loop (gradient descent) |
|  | for i = 0:num\_iterations |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID. |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights, biases,hiddenActivationFunc, outputActivationFunc=outputActivationFunc); |
|  |  |
|  | # Compute cost. |
|  | cost = computeCost(AL, Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2))); |
|  |  |
|  | # Backward propagation. |
|  | [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y, activation\_caches,forward\_caches,hiddenActivationFunc, outputActivationFunc=outputActivationFunc, |
|  | numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | # Update parameters. |
|  | [weights biases] = gradientDescent(weights,biases, gradsDW,gradsDB,learning\_rate,outputActivationFunc=outputActivationFunc); |
|  |  |
|  |  |
|  | # Print the cost every 1000 iterations |
|  | if ( mod(i,1000) == 0) |
|  | costs =[costs cost]; |
|  | #disp ("Cost after iteration"), disp(i),disp(cost); |
|  | printf("Cost after iteration i=%i cost=%d\n",i,cost); |
|  | endif |
|  | endfor |
|  |  |
|  | end |
|  |  |
|  | # Execute a L layer Deep learning model with Stochastic Gradient descent |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : mini\_batch\_size |
|  | # : num of epochs |
|  | #output : Updated weights and biases after each iteration |
|  | function [weights biases costs] = L\_Layer\_DeepModel\_SGD(X, Y, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid",learning\_rate = .3, |
|  | mini\_batch\_size = 64, num\_epochs = 2500)#lr was 0.009 |
|  |  |
|  | rand ("seed", 1); |
|  | costs = [] ; |
|  |  |
|  | # Parameters initialization. |
|  | [weights biases] = initializeDeepModel(layersDimensions); |
|  | seed=10; |
|  | # Loop (gradient descent) |
|  | for i = 0:num\_epochs |
|  | seed = seed + 1; |
|  | [mini\_batches\_X mini\_batches\_Y] = random\_mini\_batches(X, Y, mini\_batch\_size, seed); |
|  |  |
|  | minibatches=length(mini\_batches\_X); |
|  | for batch=1:minibatches |
|  | X=mini\_batches\_X{batch}; |
|  | Y=mini\_batches\_Y{batch}; |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID/SOFTMAX. |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights, biases,hiddenActivationFunc, outputActivationFunc=outputActivationFunc); |
|  | #disp(batch); |
|  | #disp(size(X)); |
|  | #disp(size(Y)); |
|  |  |
|  | # Compute cost. |
|  | cost = computeCost(AL, Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions(size(layersDimensions)(2))); |
|  |  |
|  | #disp(cost); |
|  | # Backward propagation. |
|  | [gradsDA gradsDW gradsDB] = backwardPropagationDeep(AL, Y, activation\_caches,forward\_caches,hiddenActivationFunc, outputActivationFunc=outputActivationFunc, |
|  | numClasses=layersDimensions(size(layersDimensions)(2))); |
|  | # Update parameters. |
|  | [weights biases] = gradientDescent(weights,biases, gradsDW,gradsDB,learning\_rate,outputActivationFunc=outputActivationFunc); |
|  |  |
|  | endfor |
|  | # Print the cost every 1000 iterations |
|  | if ( mod(i,1000) == 0) |
|  | costs =[costs cost]; |
|  | #disp ("Cost after iteration"), disp(i),disp(cost); |
|  | printf("Cost after iteration i=%i cost=%d\n",i,cost); |
|  | endif |
|  | endfor |
|  |  |
|  | end |
|  |  |
|  |  |
|  | function plotCostVsIterations(maxIterations,costs) |
|  | iterations=[0:1000:maxIterations]; |
|  | plot(iterations,costs); |
|  | title ("Cost vs no of iterations "); |
|  | xlabel("No of iterations"); |
|  | ylabel("Cost"); |
|  | print -dpng figure23.jpg |
|  | end; |
|  |  |
|  | # Compute the predicted value for a given input |
|  | # Input : Neural Network parameters |
|  | # : Input data |
|  | function [predictions]= predict(weights, biases, X,hiddenActivationFunc="relu") |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights, biases,hiddenActivationFunc); |
|  | predictions = (AL>0.5); |
|  | end |
|  |  |
|  | # Plot the decision boundary |
|  | function plotDecisionBoundary(data,weights, biases,hiddenActivationFunc="relu") |
|  | %Plot a non-linear decision boundary learned by the SVM |
|  | colormap ("summer"); |
|  |  |
|  | % Make classification predictions over a grid of values |
|  | x1plot = linspace(min(data(:,1)), max(data(:,1)), 400)'; |
|  | x2plot = linspace(min(data(:,2)), max(data(:,2)), 400)'; |
|  | [X1, X2] = meshgrid(x1plot, x2plot); |
|  | vals = zeros(size(X1)); |
|  | # Plot the prediction for the grid |
|  | for i = 1:size(X1, 2) |
|  | gridPoints = [X1(:, i), X2(:, i)]; |
|  | vals(:, i)=predict(weights, biases,gridPoints',hiddenActivationFunc=hiddenActivationFunc); |
|  | endfor |
|  |  |
|  | scatter(data(:,1),data(:,2),8,c=data(:,3),"filled"); |
|  | % Plot the boundary |
|  | hold on |
|  | #contour(X1, X2, vals, [0 0], 'LineWidth', 2); |
|  | contour(X1, X2, vals,"linewidth",4); |
|  | title ({"3 layer Neural Network decision boundary"}); |
|  | hold off; |
|  | print -dpng figure32.jpg |
|  |  |
|  | end |
|  |  |
|  | function [AL]= scores(weights, biases, X,hiddenActivationFunc="relu") |
|  | [AL forward\_caches activation\_caches] = forwardPropagationDeep(X, weights, biases,hiddenActivationFunc); |
|  | end |
|  |  |
|  | # Create Random mini batches. Return cell arrays with the mini batches |
|  | # Input : X, Y |
|  | # : Size of minibatch |
|  | #Output : mini batches X & Y |
|  | function [mini\_batches\_X mini\_batches\_Y]= random\_mini\_batches(X, Y, miniBatchSize = 64, seed = 0) |
|  |  |
|  | rand ("seed", seed); |
|  | # Get number of training samples |
|  | m = size(X)(2); |
|  |  |
|  |  |
|  | # Create a list of random numbers < m |
|  | permutation = randperm(m); |
|  | # Randomly shuffle the training data |
|  | shuffled\_X = X(:, permutation); |
|  | shuffled\_Y = Y(:, permutation); |
|  |  |
|  | # Compute number of mini batches |
|  | numCompleteMinibatches = floor(m/miniBatchSize); |
|  | batch=0; |
|  | for k = 0:(numCompleteMinibatches-1) |
|  | #Set the start and end of each mini batch |
|  | batch=batch+1; |
|  | lower=(k\*miniBatchSize)+1; |
|  | upper=(k+1) \* miniBatchSize; |
|  | mini\_batch\_X = shuffled\_X(:, lower:upper); |
|  | mini\_batch\_Y = shuffled\_Y(:, lower:upper); |
|  |  |
|  | # Create cell arrays |
|  | mini\_batches\_X{batch} = mini\_batch\_X; |
|  | mini\_batches\_Y{batch} = mini\_batch\_Y; |
|  | endfor |
|  |  |
|  | # If the batc size does not cleanly divide with number of mini batches |
|  | if mod(m ,miniBatchSize) != 0 |
|  | # Set the start and end of the last mini batch |
|  | l=floor(m/miniBatchSize)\*miniBatchSize; |
|  | m=l+ mod(m,miniBatchSize); |
|  | mini\_batch\_X = shuffled\_X(:,(l+1):m); |
|  | mini\_batch\_Y = shuffled\_Y(:,(l+1):m); |
|  |  |
|  | batch=batch+1; |
|  | mini\_batches\_X{batch} = mini\_batch\_X; |
|  | mini\_batches\_Y{batch} = mini\_batch\_Y; |
|  | endif |
|  | end |

DLFunctions5.R

|  |
| --- |
| library(ggplot2) |
|  | library(PRROC) |
|  | library(dplyr) |
|  |  |
|  | # Compute the sigmoid of a vector |
|  | sigmoid <- function(Z){ |
|  | A <- 1/(1+ exp(-Z)) |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  |  |
|  | } |
|  |  |
|  | # Compute the Relu(old) of a vector (performance hog!) |
|  | reluOld <-function(Z){ |
|  | A <- apply(Z, 1:2, function(x) max(0,x)) |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the Relu of a vector (much better performance!) |
|  | relu <-function(Z){ |
|  | # Perform relu. Set values less that equal to 0 as 0 |
|  | Z[Z<0]=0 |
|  | A=Z |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the tanh activation of a vector |
|  | tanhActivation <- function(Z){ |
|  | A <- tanh(Z) |
|  | cache<-Z |
|  | retvals <- list("A"=A,"Z"=Z) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # 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 |
|  | # g'(z) = 1 if z >0 and 0 otherwise |
|  | reluDerivative <-function(dA, cache){ |
|  | Z <- cache |
|  | dZ <- dA |
|  | # Create a logical matrix of values > 0 |
|  | a <- Z > 0 |
|  | # When z <= 0, you should set dz to 0 as well. Perform an element wise multiple |
|  | dZ <- dZ \* a |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Compute the derivative of sigmoid |
|  | # Derivative g'(z) = a\* (1-a) |
|  | sigmoidDerivative <- function(dA, cache){ |
|  | Z <- cache |
|  | s <- 1/(1+exp(-Z)) |
|  | dZ <- dA \* s \* (1-s) |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Compute the derivative of tanh |
|  | # Derivative g'(z) = 1- a^2 |
|  | tanhDerivative <- function(dA, cache){ |
|  | Z = cache |
|  | a = tanh(Z) |
|  | dZ = dA \* (1 - a^2) |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Populate a matrix of 1s in rows where Y==1 |
|  | # This may need to be extended for K classes. Currently |
|  | # supports K=3 & K=10 |
|  | popMatrix <- function(Y,numClasses){ |
|  | a=rep(0,times=length(Y)) |
|  | Y1=matrix(a,nrow=length(Y),ncol=numClasses) |
|  | #Set the rows and columns as 1's where Y is the class value |
|  | if(numClasses==3){ |
|  | Y1[Y==0,1]=1 |
|  | Y1[Y==1,2]=1 |
|  | Y1[Y==2,3]=1 |
|  | } else if (numClasses==10){ |
|  | Y1[Y==0,1]=1 |
|  | Y1[Y==1,2]=1 |
|  | Y1[Y==2,3]=1 |
|  | Y1[Y==3,4]=1 |
|  | Y1[Y==4,5]=1 |
|  | Y1[Y==5,6]=1 |
|  | Y1[Y==6,7]=1 |
|  | Y1[Y==7,8]=1 |
|  | Y1[Y==8,9]=1 |
|  | Y1[Y==9,0]=1 |
|  | } |
|  | return(Y1) |
|  | } |
|  |  |
|  | softmaxDerivative <- function(dA, cache ,y,numTraining,numClasses){ |
|  | # 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) |
|  | # Create a matrix of zeros |
|  | Y1=popMatrix(y,numClasses) |
|  | #a=rep(0,times=length(Y)) |
|  | #Y1=matrix(a,nrow=length(Y),ncol=numClasses) |
|  | #Set the rows and columns as 1's where Y is the class value |
|  |  |
|  | dZ = probs-Y1 |
|  | return(dZ) |
|  | } |
|  |  |
|  | # Initialize the model |
|  | # Input : number of features |
|  | # number of hidden units |
|  | # number of units in output |
|  | # Returns: Weight and bias matrices and vectors |
|  |  |
|  |  |
|  | # Initialize model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | initializeDeepModel <- function(layerDimensions){ |
|  | set.seed(2) |
|  |  |
|  | # Initialize empty list |
|  | layerParams <- list() |
|  |  |
|  | # Note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | # Indices in R start from 1 |
|  | for(l in 2:length(layersDimensions)){ |
|  | # Initialize a matrix of small random numbers of size l x l-1 |
|  | # Create random numbers of size l x l-1 |
|  | w=rnorm(layersDimensions[l]\*layersDimensions[l-1])\*0.01 |
|  |  |
|  | # Create a weight matrix of size l x l-1 with this initial weights and |
|  | # Add to list W1,W2... WL |
|  | layerParams[[paste('W',l-1,sep="")]] = matrix(w,nrow=layersDimensions[l], |
|  | ncol=layersDimensions[l-1]) |
|  | layerParams[[paste('b',l-1,sep="")]] = matrix(rep(0,layersDimensions[l]), |
|  | nrow=layersDimensions[l],ncol=1) |
|  | } |
|  | return(layerParams) |
|  | } |
|  |  |
|  |  |
|  | # Compute the activation at a layer 'l' for forward prop in a Deep Network |
|  | # Input : A\_prec - Activation of previous layer |
|  | # W,b - Weight and bias matrices and vectors |
|  | # activationFunc - Activation function - sigmoid, tanh, relu etc |
|  | # Returns : The Activation of this layer |
|  | # : |
|  | # Z = W \* X + b |
|  | # A = sigmoid(Z), A= Relu(Z), A= tanh(Z) |
|  | layerActivationForward <- function(A\_prev, W, b, activationFunc){ |
|  |  |
|  | # Compute Z |
|  | z = W %\*% A\_prev |
|  | # Broadcast the bias 'b' by column |
|  | Z <-sweep(z,1,b,'+') |
|  |  |
|  | forward\_cache <- list("A\_prev"=A\_prev, "W"=W, "b"=b) |
|  | # Compute the activation for sigmoid |
|  | if(activationFunc == "sigmoid"){ |
|  | vals = sigmoid(Z) |
|  | } else if (activationFunc == "relu"){ # Compute the activation for relu |
|  | vals = relu(Z) |
|  | } else if(activationFunc == 'tanh'){ # Compute the activation for tanh |
|  | vals = tanhActivation(Z) |
|  | } else if(activationFunc == 'softmax'){ |
|  | vals = softmax(Z) |
|  | } |
|  | # Create a list of forward and activation cache |
|  | cache <- list("forward\_cache"=forward\_cache, "activation\_cache"=vals[['Z']]) |
|  | retvals <- list("A"=vals[['A']],"cache"=cache) |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Compute the forward propagation for layers 1..L |
|  | # Input : X - Input Features |
|  | # paramaters: Weights and biases |
|  | # hiddenActivationFunc - elu/sigmoid/tanh |
|  | # outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # Returns : AL |
|  | # caches |
|  | # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and sigmoid actiovation at layer L |
|  | forwardPropagationDeep <- function(X, parameters,hiddenActivationFunc='relu', |
|  | outputActivationFunc='sigmoid'){ |
|  | caches <- list() |
|  | # Set A to X (A0) |
|  | A <- X |
|  | L <- length(parameters)/2 # number of layers in the neural network |
|  | # Loop through from layer 1 to upto layer L |
|  | for(l in 1:(L-1)){ |
|  | A\_prev <- A |
|  | # Zi = Wi x Ai-1 + bi and Ai = g(Zi) |
|  | # Set W and b for layer 'l' |
|  | # Loop throug from W1,W2... WL-1 |
|  | W <- parameters[[paste("W",l,sep="")]] |
|  | b <- parameters[[paste("b",l,sep="")]] |
|  | # Compute the forward propagation through layer 'l' using the activation function |
|  | actForward <- layerActivationForward(A\_prev, |
|  | W, |
|  | b, |
|  | activationFunc = hiddenActivationFunc) |
|  | A <- actForward[['A']] |
|  | # Append the cache A\_prev,W,b, Z |
|  | caches[[l]] <-actForward |
|  | } |
|  |  |
|  | # Since this is binary classification use the sigmoid activation function in |
|  | # last layer |
|  | # Set the weights and biases for the last layer |
|  | W <- parameters[[paste("W",L,sep="")]] |
|  | b <- parameters[[paste("b",L,sep="")]] |
|  | # Compute the sigmoid activation |
|  | actForward = layerActivationForward(A, W, b, activationFunc = outputActivationFunc) |
|  | AL <- actForward[['A']] |
|  | # Append the output of this forward propagation through the last layer |
|  | caches[[L]] <- actForward |
|  | # Create a list of the final output and the caches |
|  | fwdPropDeep <- list("AL"=AL,"caches"=caches) |
|  | return(fwdPropDeep) |
|  |  |
|  | } |
|  |  |
|  | pickColumns <- function(AL,Y,numClasses){ |
|  | if(numClasses==3){ |
|  | a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3]) |
|  | } |
|  | else if (numClasses==10){ |
|  | a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3],AL[Y==3,4],AL[Y==4,5], |
|  | AL[Y==5,6],AL[Y==6,7],AL[Y==7,8],AL[Y==8,9],AL[Y==9,10]) |
|  | } |
|  | return(a) |
|  | } |
|  |  |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # :outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : numClasses |
|  | # Output: cost |
|  | computeCost <- function(AL,Y,outputActivationFunc="sigmoid",numClasses=3){ |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | m= length(Y) |
|  | cost=-1/m\*sum(Y\*log(AL) + (1-Y)\*log(1-AL)) |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | # Select the elements where the y values are 0, 1 or 2 and make a vector |
|  | # Pick columns |
|  | #a=c(AL[Y==0,1],AL[Y==1,2],AL[Y==2,3]) |
|  | m= length(Y) |
|  | a =pickColumns(AL,Y,numClasses) |
|  | #a = c(A2[y=k,k+1]) |
|  | # Take log |
|  | correct\_probs = -log(a) |
|  | # Compute loss |
|  | cost= sum(correct\_probs)/m |
|  | } |
|  | #cost=-1/m\*sum(a+b) |
|  | return(cost) |
|  | } |
|  |  |
|  |  |
|  | # Compute the backpropagation through a layer |
|  | # Input : Neural Network parameters - dA |
|  | # # cache - forward\_cache & activation\_cache |
|  | # # Input features |
|  | # # Output values Y |
|  | # # activationFunc |
|  | # # numClasses |
|  | # Returns: Gradients |
|  | # dL/dWi= dL/dZi\*Al-1 |
|  | # dl/dbl = dL/dZl |
|  | # dL/dZ\_prev=dL/dZl\*W |
|  |  |
|  | layerActivationBackward <- function(dA, cache, Y, activationFunc,numClasses){ |
|  | # 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 |
|  | activation\_cache <- cache[['activation\_cache']] |
|  | if(activationFunc == "relu"){ |
|  | dZ <- reluDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "sigmoid"){ |
|  | dZ <- sigmoidDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "tanh"){ |
|  | dZ <- tanhDerivative(dA, activation\_cache) |
|  | } else if(activationFunc == "softmax"){ |
|  | dZ <- softmaxDerivative(dA, activation\_cache,Y,numtraining,numClasses) |
|  | } |
|  |  |
|  | 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=numClasses) |
|  | 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) |
|  | } |
|  |  |
|  | # Compute the backpropagation for 1 cycle through all layers |
|  | # Input : AL: Output of L layer Network - weights |
|  | # # Y Real output |
|  | # # caches -- list of caches containing: |
|  | # every cache of layerActivationForward() with "relu"/"tanh" |
|  | # #(it's caches[l], for l in range(L-1) i.e l = 0...L-2) |
|  | # #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1]) |
|  | # hiddenActivationFunc - Activation function at hidden layers - relu/tanh/sigmoid |
|  | # outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # |
|  | # Returns: |
|  | # gradients -- A dictionary with the gradients |
|  | # gradients["dA" + str(l)] = ... |
|  | # |
|  | backwardPropagationDeep <- function(AL, Y, caches,hiddenActivationFunc='relu', |
|  | outputActivationFunc="sigmoid",numClasses){ |
|  | #initialize the gradients |
|  | gradients = list() |
|  | # Set the number of layers |
|  | L = length(caches) |
|  | numTraining = dim(AL)[2] |
|  |  |
|  | if(outputActivationFunc == "sigmoid") |
|  | # Initializing the backpropagation |
|  | # dl/dAL= -(y/a) - ((1-y)/(1-a)) - At the output layer |
|  | dAL = -( (Y/AL) -(1 - Y)/(1 - AL)) |
|  | else if(outputActivationFunc == "softmax"){ |
|  | dAL=0 |
|  | Y=t(Y) |
|  | } |
|  |  |
|  | # Get the gradients at the last layer |
|  | # Inputs: "AL, Y, caches". |
|  | # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"] |
|  | # Start with Layer L |
|  | # Get the current cache |
|  | current\_cache = caches[[L]]$cache |
|  | #gradients["dA" + str(L)], gradients["dW" + str(L)], gradients["db" + str(L)] = layerActivationBackward(dAL, current\_cache, activationFunc = "sigmoid") |
|  | retvals <- layerActivationBackward(dAL, current\_cache, Y, activationFunc = outputActivationFunc,numClasses) |
|  | # Create gradients as lists |
|  | #Note: Take the transpose of dA |
|  | if(outputActivationFunc =="sigmoid") |
|  | gradients[[paste("dA",L,sep="")]] <- retvals[['dA\_prev']] |
|  | else if(outputActivationFunc =="softmax") |
|  | gradients[[paste("dA",L,sep="")]] <- t(retvals[['dA\_prev']]) |
|  | gradients[[paste("dW",L,sep="")]] <- retvals[['dW']] |
|  | gradients[[paste("db",L,sep="")]] <- retvals[['db']] |
|  |  |
|  | # Traverse in the reverse direction |
|  | for(l in (L-1):1){ |
|  | # Compute the gradients for L-1 to 1 for Relu/tanh |
|  | # Inputs: "gradients["dA" + str(l + 2)], caches". |
|  | # Outputs: "gradients["dA" + str(l + 1)] , gradients["dW" + str(l + 1)] , gradients["db" + str(l + 1)] |
|  | current\_cache = caches[[l]]$cache |
|  |  |
|  | retvals = layerActivationBackward(gradients[[paste('dA',l+1,sep="")]], |
|  | current\_cache, |
|  | activationFunc = hiddenActivationFunc) |
|  |  |
|  | gradients[[paste("dA",l,sep="")]] <-retvals[['dA\_prev']] |
|  | gradients[[paste("dW",l,sep="")]] <- retvals[['dW']] |
|  | gradients[[paste("db",l,sep="")]] <- retvals[['db']] |
|  | } |
|  |  |
|  |  |
|  |  |
|  | return(gradients) |
|  | } |
|  |  |
|  |  |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | gradientDescent <- function(parameters, gradients, learningRate,outputActivationFunc="sigmoid"){ |
|  |  |
|  | L = length(parameters)/2 # number of layers in the neural network |
|  |  |
|  | # Update rule for each parameter. Use a for loop. |
|  | for(l in 1:(L-1)){ |
|  | parameters[[paste("W",l,sep="")]] = parameters[[paste("W",l,sep="")]] - |
|  | learningRate\* gradients[[paste("dW",l,sep="")]] |
|  | parameters[[paste("b",l,sep="")]] = parameters[[paste("b",l,sep="")]] - |
|  | learningRate\* gradients[[paste("db",l,sep="")]] |
|  | } |
|  | if(outputActivationFunc=="sigmoid"){ |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* gradients[[paste("dW",L,sep="")]] |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* gradients[[paste("db",L,sep="")]] |
|  |  |
|  | }else if (outputActivationFunc=="softmax"){ |
|  | parameters[[paste("W",L,sep="")]] = parameters[[paste("W",L,sep="")]] - |
|  | learningRate\* gradients[[paste("dW",L,sep="")]] |
|  | parameters[[paste("b",L,sep="")]] = parameters[[paste("b",L,sep="")]] - |
|  | learningRate\* gradients[[paste("db",L,sep="")]] |
|  | } |
|  | return(parameters) |
|  | } |
|  |  |
|  |  |
|  | # Execute a L layer Deep learning model |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : num of iterations |
|  | #output : Updated weights after each iteration |
|  |  |
|  | L\_Layer\_DeepModel <- function(X, Y, layersDimensions, |
|  | hiddenActivationFunc='relu', |
|  | outputActivationFunc= 'sigmoid', |
|  | learningRate = 0.5, |
|  | numIterations = 10000, |
|  | print\_cost=False){ |
|  | #Initialize costs vector as NULL |
|  | costs <- NULL |
|  |  |
|  | # Parameters initialization. |
|  | parameters = initializeDeepModel(layersDimensions) |
|  |  |
|  |  |
|  | # Loop (gradient descent) |
|  | for( i in 0:numIterations){ |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID/SOFTMAX. |
|  | retvals = forwardPropagationDeep(X, parameters,hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc) |
|  | AL <- retvals[['AL']] |
|  | caches <- retvals[['caches']] |
|  |  |
|  | # Compute cost. |
|  | cost <- computeCost(AL, Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[3]) |
|  |  |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, Y, caches,hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[3]) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  |  |
|  |  |
|  | if(i%%1000 == 0){ |
|  | costs=c(costs,cost) |
|  | print(cost) |
|  | } |
|  | } |
|  |  |
|  | retvals <- list("parameters"=parameters,"costs"=costs) |
|  |  |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Execute a L layer Deep learning model with Stochastic Gradient descent |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh |
|  | # : outputActivationFunc - Activation function at hidden layer sigmoid/softmax |
|  | # : learning rate |
|  | # : mini\_batch\_size |
|  | # : num of epochs |
|  | #output : Updated weights after each iteration |
|  | L\_Layer\_DeepModel\_SGD <- function(X, Y, layersDimensions, |
|  | hiddenActivationFunc='relu', |
|  | outputActivationFunc= 'sigmoid', |
|  | learningRate = .3, |
|  | mini\_batch\_size = 64, |
|  | num\_epochs = 2500, |
|  | print\_cost=False){ |
|  |  |
|  | set.seed(1) |
|  | #Initialize costs vector as NULL |
|  | costs <- NULL |
|  |  |
|  | # Parameters initialization. |
|  | parameters = initializeDeepModel(layersDimensions) |
|  | seed=10 |
|  |  |
|  | # Loop for number of epochs |
|  | for( i in 0:num\_epochs){ |
|  | seed=seed+1 |
|  | minibatches = random\_mini\_batches(X, Y, mini\_batch\_size, seed) |
|  |  |
|  | for(batch in 1:length(minibatches)){ |
|  |  |
|  | mini\_batch\_X=minibatches[[batch]][['mini\_batch\_X']] |
|  | mini\_batch\_Y=minibatches[[batch]][['mini\_batch\_Y']] |
|  | # Forward propagation: |
|  | retvals = forwardPropagationDeep(mini\_batch\_X, parameters,hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc) |
|  | AL <- retvals[['AL']] |
|  | caches <- retvals[['caches']] |
|  |  |
|  | # Compute cost. |
|  | cost <- computeCost(AL, mini\_batch\_Y,outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  |  |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, mini\_batch\_Y, caches,hiddenActivationFunc, |
|  | outputActivationFunc=outputActivationFunc,numClasses=layersDimensions[length(layersDimensions)]) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate, |
|  | outputActivationFunc=outputActivationFunc) |
|  | } |
|  |  |
|  | if(i%%100 == 0){ |
|  | costs=c(costs,cost) |
|  | print(cost) |
|  | } |
|  | } |
|  |  |
|  | retvals <- list("parameters"=parameters,"costs"=costs) |
|  |  |
|  | return(retvals) |
|  | } |
|  |  |
|  | # Predict the output for given input |
|  | # Input : parameters |
|  | # : X |
|  | # Output: predictions |
|  | predict <- function(parameters, X,hiddenActivationFunc='relu'){ |
|  |  |
|  | fwdProp <- forwardPropagationDeep(X, parameters,hiddenActivationFunc) |
|  | predictions <- fwdProp$AL>0.5 |
|  |  |
|  | return (predictions) |
|  | } |
|  |  |
|  | # Predict the output |
|  | predictProba <- function(parameters, X,hiddenActivationFunc, |
|  | outputActivationFunc){ |
|  | retvals = forwardPropagationDeep(X, parameters,hiddenActivationFunc, |
|  | outputActivationFunc="softmax") |
|  | if(outputActivationFunc =="sigmoid") |
|  | predictions <- retvals$AL>0.5 |
|  | else if (outputActivationFunc =="softmax") |
|  | predictions <- apply(retvals$AL, 1,which.max) -1 |
|  |  |
|  | return (predictions) |
|  | } |
|  |  |
|  | # Plot a decision boundary |
|  | # This function uses ggplot2 |
|  | plotDecisionBoundary <- function(z,retvals,hiddenActivationFunc,lr){ |
|  | # Find the minimum and maximum for the data |
|  | xmin<-min(z[,1]) |
|  | xmax<-max(z[,1]) |
|  | ymin<-min(z[,2]) |
|  | ymax<-max(z[,2]) |
|  |  |
|  | # Create a grid of values |
|  | a=seq(xmin,xmax,length=100) |
|  | b=seq(ymin,ymax,length=100) |
|  | grid <- expand.grid(x=a, y=b) |
|  | colnames(grid) <- c('x1', 'x2') |
|  | grid1 <-t(grid) |
|  | # Predict the output for this grid |
|  | q <-predict(retvals$parameters,grid1,hiddenActivationFunc) |
|  | q1 <- t(data.frame(q)) |
|  | q2 <- as.numeric(q1) |
|  | grid2 <- cbind(grid,q2) |
|  | colnames(grid2) <- c('x1', 'x2','q2') |
|  |  |
|  | z1 <- data.frame(z) |
|  | names(z1) <- c("x1","x2","y") |
|  | atitle=paste("Decision boundary for learning rate:",lr) |
|  | # Plot the contour of the boundary |
|  | ggplot(z1) + |
|  | geom\_point(data = z1, aes(x = x1, y = x2, color = y)) + |
|  | stat\_contour(data = grid2, aes(x = x1, y = x2, z = q2,color=q2), alpha = 0.9)+ |
|  | ggtitle(atitle) |
|  | } |
|  |  |
|  | # Predict the probability scores for given data set |
|  | # Input : parameters |
|  | # : X |
|  | # Output: probability of output |
|  | computeScores <- function(parameters, X,hiddenActivationFunc='relu'){ |
|  |  |
|  | fwdProp <- forwardPropagationDeep(X, parameters,hiddenActivationFunc) |
|  | scores <- fwdProp$AL |
|  |  |
|  | return (scores) |
|  | } |
|  |  |
|  |  |
|  | random\_mini\_batches <- function(X, Y, miniBatchSize = 64, seed = 0){ |
|  |  |
|  |  |
|  | set.seed(seed) |
|  | # Get number of training samples |
|  | m = dim(X)[2] |
|  | # Initialize mini batches |
|  | mini\_batches = list() |
|  |  |
|  | # Create a list of random numbers < m |
|  | permutation = c(sample(m)) |
|  | # Randomly shuffle the training data |
|  | shuffled\_X = X[, permutation] |
|  | shuffled\_Y = Y[1, permutation] |
|  |  |
|  | # Compute number of mini batches |
|  | numCompleteMinibatches = floor(m/miniBatchSize) |
|  | batch=0 |
|  | for(k in 0:(numCompleteMinibatches-1)){ |
|  | batch=batch+1 |
|  | # Set the lower and upper bound of the mini batches |
|  | lower=(k\*miniBatchSize)+1 |
|  | upper=((k+1) \* miniBatchSize) |
|  | mini\_batch\_X = shuffled\_X[, lower:upper] |
|  | mini\_batch\_Y = shuffled\_Y[lower:upper] |
|  | # Add it to the list of mini batches |
|  | mini\_batch = list("mini\_batch\_X"=mini\_batch\_X,"mini\_batch\_Y"=mini\_batch\_Y) |
|  | mini\_batches[[batch]] =mini\_batch |
|  |  |
|  |  |
|  | } |
|  |  |
|  | # If the batch size does not divide evenly with mini batc size |
|  | if(m %% miniBatchSize != 0){ |
|  | p=floor(m/miniBatchSize)\*miniBatchSize |
|  | # Set the start and end of last batch |
|  | q=p+m %% miniBatchSize |
|  | mini\_batch\_X = shuffled\_X[,(p+1):q] |
|  | mini\_batch\_Y = shuffled\_Y[(p+1):q] |
|  | } |
|  | # Return the list of mini batches |
|  | mini\_batch = list("mini\_batch\_X"=mini\_batch\_X,"mini\_batch\_Y"=mini\_batch\_Y) |
|  | mini\_batches[[batch]]=mini\_batch |
|  |  |
|  | return(mini\_batches) |
|  | } |

DLFunctions51.py

|  |
| --- |
| import numpy as np |
|  | import matplotlib.pyplot as plt |
|  | import matplotlib |
|  | import matplotlib.pyplot as plt |
|  | from matplotlib import cm |
|  | import math |
|  |  |
|  | # Conmpute the sigmoid of a vector |
|  | def sigmoid(Z): |
|  | A=1/(1+np.exp(-Z)) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Conmpute the Relu of a vector |
|  | def relu(Z): |
|  | A = np.maximum(0,Z) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # Conmpute the tanh of a vector |
|  | def tanh(Z): |
|  | A = np.tanh(Z) |
|  | cache=Z |
|  | return A,cache |
|  |  |
|  | # 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 |
|  |  |
|  | # Conmpute the softmax of a vector |
|  | def stableSoftmax(Z): |
|  | #Compute the softmax of vector x in a numerically stable way. |
|  | shiftZ = Z.T - np.max(Z.T,axis=1).reshape(-1,1) |
|  | exp\_scores = np.exp(shiftZ) |
|  |  |
|  | # 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 sigmoid |
|  | def sigmoidDerivative(dA, cache): |
|  | Z = cache |
|  | s = 1/(1+np.exp(-Z)) |
|  | dZ = dA \* s \* (1-s) |
|  | return dZ |
|  |  |
|  | # Compute the derivative of tanh |
|  | def tanhDerivative(dA, cache): |
|  | Z = cache |
|  | a = np.tanh(Z) |
|  | dZ = dA \* (1 - np.power(a, 2)) |
|  | 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[:,0]] -= 1 |
|  | return(dZ) |
|  |  |
|  | # Compute the derivative of softmax |
|  | def stableSoftmaxDerivative(dA, cache,y,numTraining): |
|  | # Note : dA not used. dL/dZ = dL/dA \* dA/dZ = pi-yi |
|  | Z = cache |
|  | # Compute stable softmax |
|  | shiftZ = Z.T - np.max(Z.T,axis=1).reshape(-1,1) |
|  | exp\_scores = np.exp(shiftZ) |
|  | # normalize them for each example |
|  | probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True) |
|  | #print(probs) |
|  | # compute the gradient on scores |
|  | dZ = probs |
|  |  |
|  | # dZ = pi- yi |
|  | dZ[range(int(numTraining)),y[:,0]] -= 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) |
|  |  |
|  |  |
|  | # Initialize model for L layers |
|  | # Input : List of units in each layer |
|  | # Returns: Initial weights and biases matrices for all layers |
|  | def initializeDeepModel(layerDimensions): |
|  | np.random.seed(3) |
|  | # note the Weight matrix at layer 'l' is a matrix of size (l,l-1) |
|  | # The Bias is a vectors of size (l,1) |
|  |  |
|  | # Loop through the layer dimension from 1.. L |
|  | layerParams = {} |
|  | for l in range(1,len(layerDimensions)): |
|  | layerParams['W' + str(l)] = np.random.randn(layerDimensions[l],layerDimensions[l-1])\*0.01 # Multiply by .01 |
|  | layerParams['b' + str(l)] = np.zeros((layerDimensions[l],1)) |
|  |  |
|  | return(layerParams) |
|  | return Z, cache |
|  |  |
|  | # 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 = stableSoftmax(Z) |
|  |  |
|  | cache = (forward\_cache, activation\_cache) |
|  | return A, cache |
|  |  |
|  | # Compute the forward propagation for layers 1..L |
|  | # Input : X - Input Features |
|  | # paramaters: Weights and biases |
|  | # hiddenActivationFunc - Activation function at hidden layers Relu/tanh |
|  | # outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | # Returns : AL |
|  | # caches |
|  | # The forward propoagtion uses the Relu/tanh activation from layer 1..L-1 and sigmoid actiovation at layer L |
|  | def forwardPropagationDeep(X, parameters,hiddenActivationFunc='relu',outputActivationFunc='sigmoid'): |
|  | caches = [] |
|  | # Set A to X (A0) |
|  | A = X |
|  | L = int(len(parameters)/2) # number of layers in the neural network |
|  | # Loop through from layer 1 to upto layer L |
|  | for l in range(1, L): |
|  | A\_prev = A |
|  | # Zi = Wi x Ai-1 + bi and Ai = g(Zi) |
|  | #A, cache = layerActivationForward(A\_prev, parameters['W'+str(l)], parameters['b'+str(l)], activationFunc = "relu") |
|  | A, cache = layerActivationForward(A\_prev, parameters['W'+str(l)], parameters['b'+str(l)], activationFunc = hiddenActivationFunc) |
|  | caches.append(cache) |
|  | #print("l=",l) |
|  | #print(A) |
|  |  |
|  | # Since this is binary classification use the sigmoid activation function in |
|  | # last layer |
|  | AL, cache = layerActivationForward(A, parameters['W'+str(L)], parameters['b'+str(L)], activationFunc = outputActivationFunc) |
|  | caches.append(cache) |
|  |  |
|  | return AL, caches |
|  |  |
|  |  |
|  | # Compute the cost |
|  | # Input : Activation of last layer |
|  | # : Output from data |
|  | # : Y |
|  | # :outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | # Output: cost |
|  | def computeCost(AL,Y,outputActivationFunc="sigmoid"): |
|  |  |
|  | if outputActivationFunc=="sigmoid": |
|  | m= float(Y.shape[1]) |
|  | # Element wise multiply for logprobs |
|  | cost=-1/m \*np.sum(Y\*np.log(AL) + (1-Y)\*(np.log(1-AL))) |
|  | cost = np.squeeze(cost) |
|  | elif outputActivationFunc=="softmax": |
|  | # Take transpose of Y for softmax |
|  | Y=Y.T |
|  | m= float(len(Y)) |
|  | # Compute log probs. Take the log prob of correct class based on output y |
|  | correct\_logprobs = -np.log(AL[range(int(m)),Y.T]) |
|  | # Conpute loss |
|  | cost = np.sum(correct\_logprobs)/m |
|  | return cost |
|  |  |
|  | # 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]) |
|  | #print("n=",numtraining) |
|  | #print("no=",numtraining) |
|  | 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 = stableSoftmaxDerivative(dA, activation\_cache,Y,numtraining) |
|  |  |
|  | if activationFunc == 'softmax': |
|  | dW = 1/numtraining \* np.dot(A\_prev,dZ) |
|  | db = 1/numtraining \* 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 |
|  |  |
|  | # Compute the backpropoagation for 1 cycle |
|  | # Input : AL: Output of L layer Network - weights |
|  | # # Y Real output |
|  | # # caches -- list of caches containing: |
|  | # every cache of layerActivationForward() with "relu"/"tanh" |
|  | # #(it's caches[l], for l in range(L-1) i.e l = 0...L-2) |
|  | # #the cache of layerActivationForward() with "sigmoid" (it's caches[L-1]) |
|  | # hiddenActivationFunc - Activation function at hidden layers - relu/sigmoid/tanh |
|  | # # outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | # |
|  | # Returns: |
|  | # gradients -- A dictionary with the gradients |
|  | # gradients["dA" + str(l)] = ... |
|  | # gradients["dW" + str(l)] = ... |
|  |  |
|  | def backwardPropagationDeep(AL, Y, caches,hiddenActivationFunc='relu',outputActivationFunc="sigmoid"): |
|  | #initialize the gradients |
|  | gradients = {} |
|  | # Set the number of layers |
|  | L = len(caches) |
|  | m = float(AL.shape[1]) |
|  |  |
|  | if outputActivationFunc == "sigmoid": |
|  | Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL |
|  | # Initializing the backpropagation |
|  | # dl/dAL= -(y/a + (1-y)/(1-a)) - At the output layer |
|  | dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) |
|  | else: |
|  | dAL =0 |
|  | Y=Y.T |
|  |  |
|  | # Since this is a binary classification the activation at output is sigmoid |
|  | # Get the gradients at the last layer |
|  | # Inputs: "AL, Y, caches". |
|  | # Outputs: "gradients["dAL"], gradients["dWL"], gradients["dbL"] |
|  | current\_cache = caches[L-1] |
|  | gradients["dA" + str(L)], gradients["dW" + str(L)], gradients["db" + str(L)] = layerActivationBackward(dAL, current\_cache, Y, activationFunc = outputActivationFunc) |
|  |  |
|  | # Note dA for softmax is the transpose |
|  | if outputActivationFunc == "softmax": |
|  | gradients["dA" + str(L)] = gradients["dA" + str(L)].T |
|  | # Traverse in the reverse direction |
|  | for l in reversed(range(L-1)): |
|  | # Compute the gradients for L-1 to 1 for Relu/tanh |
|  | # Inputs: "gradients["dA" + str(l + 2)], caches". |
|  | # Outputs: "gradients["dA" + str(l + 1)] , gradients["dW" + str(l + 1)] , gradients["db" + str(l + 1)] |
|  | current\_cache = caches[l] |
|  |  |
|  | #dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackward(gradients['dA'+str(l+2)], current\_cache, activationFunc = "relu") |
|  | dA\_prev\_temp, dW\_temp, db\_temp = layerActivationBackward(gradients['dA'+str(l+2)], current\_cache, Y, activationFunc = hiddenActivationFunc) |
|  | gradients["dA" + str(l + 1)] = dA\_prev\_temp |
|  | gradients["dW" + str(l + 1)] = dW\_temp |
|  | gradients["db" + str(l + 1)] = db\_temp |
|  |  |
|  |  |
|  | return gradients |
|  |  |
|  | # Perform Gradient Descent |
|  | # Input : Weights and biases |
|  | # : gradients |
|  | # : learning rate |
|  | # : outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  | def gradientDescent(parameters, gradients, learningRate,outputActivationFunc="sigmoid"): |
|  |  |
|  | L = int(len(parameters) / 2) |
|  | # Update rule for each parameter. |
|  | for l in range(L-1): |
|  | parameters["W" + str(l+1)] = parameters['W'+str(l+1)] -learningRate\* gradients['dW' + str(l+1)] |
|  | parameters["b" + str(l+1)] = parameters['b'+str(l+1)] -learningRate\* gradients['db' + str(l+1)] |
|  |  |
|  | if outputActivationFunc=="sigmoid": |
|  | parameters["W" + str(L)] = parameters['W'+str(L)] -learningRate\* gradients['dW' + str(L)] |
|  | parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate\* gradients['db' + str(L)] |
|  | elif outputActivationFunc=="softmax": |
|  | parameters["W" + str(L)] = parameters['W'+str(L)] -learningRate\* gradients['dW' + str(L)].T |
|  | parameters["b" + str(L)] = parameters['b'+str(L)] -learningRate\* gradients['db' + str(L)].T |
|  |  |
|  |  |
|  |  |
|  | return parameters |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | # Execute a L layer Deep learning model |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh/sigmoid |
|  | # : learning rate |
|  | # : num of iteration |
|  | # : outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  |  |
|  | def L\_Layer\_DeepModel(X1, Y1, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid", learningRate = .3, num\_iterations = 10000, print\_cost=False):#lr was 0.009 |
|  |  |
|  | np.random.seed(1) |
|  | costs = [] |
|  |  |
|  | # Parameters initialization. |
|  | parameters = initializeDeepModel(layersDimensions) |
|  |  |
|  | # Loop (gradient descent) |
|  | for i in range(0, num\_iterations): |
|  | # Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID. |
|  | #AL, caches = forwardPropagationDeep(X, parameters,hiddenActivationFunc) |
|  |  |
|  | # Compute cost. |
|  | #cost = computeCost(AL, Y) |
|  |  |
|  | # Backward propagation. |
|  | #gradients = backwardPropagationDeep(AL, Y, caches,hiddenActivationFunc) |
|  |  |
|  | ## Update parameters. |
|  | #parameters = gradientDescent(parameters, gradients, learning\_rate) |
|  |  |
|  | AL, caches = forwardPropagationDeep(X1, parameters,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Compute cost |
|  | cost = computeCost(AL, Y1,outputActivationFunc=outputActivationFunc) |
|  | #print("Y1=",Y1.shape) |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, Y1, caches,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate=learningRate,outputActivationFunc=outputActivationFunc) |
|  |  |
|  |  |
|  | # Print the cost every 100 training example |
|  | if print\_cost and i % 1000 == 0: |
|  | print ("Cost after iteration %i: %f" %(i, cost)) |
|  | if print\_cost and i % 1000 == 0: |
|  | costs.append(cost) |
|  |  |
|  | # plot the cost |
|  | plt.plot(np.squeeze(costs)) |
|  | plt.ylabel('cost') |
|  | plt.xlabel('No of iterations (x100)') |
|  | plt.title("Learning rate =" + str(learningRate)) |
|  | #plt.show() |
|  | plt.savefig("fig1",bbox\_inches='tight') |
|  |  |
|  | return parameters |
|  |  |
|  | # Execute a L layer Deep learning model Stoachastic Gradient Descent |
|  | # Input : X - Input features |
|  | # : Y output |
|  | # : layersDimensions - Dimension of layers |
|  | # : hiddenActivationFunc - Activation function at hidden layer relu /tanh/sigmoid |
|  | # : learning rate |
|  | # : num of iteration |
|  | # : outputActivationFunc - Activation function at output - sigmoid/softmax |
|  | #output : Updated weights after 1 iteration |
|  |  |
|  | def L\_Layer\_DeepModel\_SGD(X1, Y1, layersDimensions, hiddenActivationFunc='relu', outputActivationFunc="sigmoid",learningRate = .3, mini\_batch\_size = 64, num\_epochs = 2500, print\_cost=False):#lr was 0.009 |
|  |  |
|  | np.random.seed(1) |
|  | costs = [] |
|  |  |
|  | # Parameters initialization. |
|  | parameters = initializeDeepModel(layersDimensions) |
|  | seed=10 |
|  | # Loop for number of epochs |
|  | for i in range(num\_epochs): |
|  | # Define the random minibatches. We increment the seed to reshuffle differently the dataset after each epoch |
|  | seed = seed + 1 |
|  | minibatches = random\_mini\_batches(X1, Y1, mini\_batch\_size, seed) |
|  |  |
|  | batch=0 |
|  | # Loop through each mini batch |
|  | for minibatch in minibatches: |
|  | #print("batch=",batch) |
|  | batch=batch+1 |
|  | # Select a minibatch |
|  | (minibatch\_X, minibatch\_Y) = minibatch |
|  |  |
|  | # Perfrom forward propagation |
|  | AL, caches = forwardPropagationDeep(minibatch\_X, parameters,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Compute cost |
|  | cost = computeCost(AL, minibatch\_Y,outputActivationFunc=outputActivationFunc) |
|  | #print("minibatch\_Y=",minibatch\_Y.shape) |
|  | # Backward propagation. |
|  | gradients = backwardPropagationDeep(AL, minibatch\_Y, caches,hiddenActivationFunc="relu",outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Update parameters. |
|  | parameters = gradientDescent(parameters, gradients, learningRate=learningRate,outputActivationFunc=outputActivationFunc) |
|  |  |
|  | # Print the cost every 1000 epoch |
|  | if print\_cost and i % 100 == 0: |
|  | print ("Cost after epoch %i: %f" %(i, cost)) |
|  | if print\_cost and i % 100 == 0: |
|  | costs.append(cost) |
|  |  |
|  | # plot the cost |
|  | plt.plot(np.squeeze(costs)) |
|  | plt.ylabel('cost') |
|  | plt.xlabel('No of iterations') |
|  | plt.title("Learning rate =" + str(learningRate)) |
|  | #plt.show() |
|  | plt.savefig("fig1",bbox\_inches='tight') |
|  |  |
|  | return parameters |
|  |  |
|  |  |
|  | # Create random mini batches |
|  | def random\_mini\_batches(X, Y, miniBatchSize = 64, seed = 0): |
|  |  |
|  | np.random.seed(seed) |
|  | # Get number of training samples |
|  | m = X.shape[1] |
|  | # Initialize mini batches |
|  | mini\_batches = [] |
|  |  |
|  | # Create a list of random numbers < m |
|  | permutation = list(np.random.permutation(m)) |
|  | # Randomly shuffle the training data |
|  | shuffled\_X = X[:, permutation] |
|  | shuffled\_Y = Y[:, permutation].reshape((1,m)) |
|  |  |
|  | # Compute number of mini batches |
|  | numCompleteMinibatches = math.floor(m/miniBatchSize) |
|  |  |
|  | # For the number of mini batches |
|  | for k in range(0, numCompleteMinibatches): |
|  |  |
|  | # Set the start and end of each mini batch |
|  | mini\_batch\_X = shuffled\_X[:, k\*miniBatchSize : (k+1) \* miniBatchSize] |
|  | mini\_batch\_Y = shuffled\_Y[:, k\*miniBatchSize : (k+1) \* miniBatchSize] |
|  |  |
|  | mini\_batch = (mini\_batch\_X, mini\_batch\_Y) |
|  | mini\_batches.append(mini\_batch) |
|  |  |
|  |  |
|  | #if m % miniBatchSize != 0:. The batch does not evenly divide by the mini batch |
|  | if m % miniBatchSize != 0: |
|  | l=math.floor(m/miniBatchSize)\*miniBatchSize |
|  | # Set the start and end of last mini batch |
|  | m=l+m % miniBatchSize |
|  | mini\_batch\_X = shuffled\_X[:,l:m] |
|  | mini\_batch\_Y = shuffled\_Y[:,l:m] |
|  |  |
|  | mini\_batch = (mini\_batch\_X, mini\_batch\_Y) |
|  | mini\_batches.append(mini\_batch) |
|  |  |
|  | return mini\_batches |
|  |  |
|  | # Plot a decision boundary |
|  | # Input : Input Model, |
|  | # X |
|  | # Y |
|  | # sz - Num of hiden units |
|  | # lr - Learning rate |
|  | # Fig to be saved as |
|  | # Returns Null |
|  | def plot\_decision\_boundary(model, X, y,lr,fig): |
|  | # Set min and max values and give it some padding |
|  | x\_min, x\_max = X[0, :].min() - 1, X[0, :].max() + 1 |
|  | y\_min, y\_max = X[1, :].min() - 1, X[1, :].max() + 1 |
|  | colors=['black','yellow'] |
|  | cmap = matplotlib.colors.ListedColormap(colors) |
|  | h = 0.01 |
|  | # Generate a grid of points with distance h between them |
|  | xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h)) |
|  | # Predict the function value for the whole grid |
|  | Z = model(np.c\_[xx.ravel(), yy.ravel()]) |
|  | Z = Z.reshape(xx.shape) |
|  | # Plot the contour and training examples |
|  | plt.contourf(xx, yy, Z, cmap="coolwarm") |
|  | plt.ylabel('x2') |
|  | plt.xlabel('x1') |
|  | plt.scatter(X[0, :], X[1, :], c=y, s=7,cmap=cmap) |
|  | plt.title("Decision Boundary for learning rate:"+lr) |
|  | #plt.show() |
|  | plt.savefig(fig, bbox\_inches='tight') |
|  |  |
|  | def predict(parameters, X): |
|  | A2, cache = forwardPropagationDeep(X, parameters) |
|  | predictions = (A2>0.5) |
|  | return predictions |
|  |  |
|  | def predict\_proba(parameters, X,outputActivationFunc="sigmoid"): |
|  | A2, cache = forwardPropagationDeep(X, parameters) |
|  | if outputActivationFunc=="sigmoid": |
|  | proba=A2 |
|  | elif outputActivationFunc=="softmax": |
|  | proba=np.argmax(A2, axis=0).reshape(-1,1) |
|  | print("A2=",A2.shape) |
|  | return proba |
|  |  |
|  |  |
|  | # Plot a decision boundary |
|  | # Input : Input Model, |
|  | # X |
|  | # Y |
|  | # sz - Num of hiden units |
|  | # lr - Learning rate |
|  | # Fig to be saved as |
|  | # Returns Null |
|  | def plot\_decision\_surface(model, X, y,sz,lr,fig): |
|  | # Set min and max values and give it some padding |
|  | x\_min, x\_max = X[0, :].min() - 1, X[0, :].max() + 1 |
|  | y\_min, y\_max = X[1, :].min() - 1, X[1, :].max() + 1 |
|  | z\_min, z\_max = X[2, :].min() - 1, X[2, :].max() + 1 |
|  | colors=['black','gold'] |
|  | cmap = matplotlib.colors.ListedColormap(colors) |
|  | h = 3 |
|  | # Generate a grid of points with distance h between them |
|  | xx, yy, zz = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h), np.arange(z\_min, z\_max, h)) |
|  | # Predict the function value for the whole grid |
|  | a=np.c\_[xx.ravel(), yy.ravel(), zz.ravel()] |
|  |  |
|  | Z = predict(parameters,a.T) |
|  | Z = Z.reshape(xx.shape) |
|  | # Plot the contour and training examples |
|  | #plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral) |
|  | fig = plt.figure() |
|  | ax = plt.axes(projection='3d') |
|  | ax.contour3D(xx, yy, Z, 50, cmap='binary') |
|  | #plt.ylabel('x2') |
|  | #plt.xlabel('x1') |
|  | plt.scatter(X[0, :], X[1, :], c=y, cmap=cmap) |
|  | plt.title("Decision Boundary for hidden layer size:" + sz +" and learning rate:"+lr) |
|  | plt.show() |
|  |  |
|  | def plotSurface(X,parameters): |
|  |  |
|  | #xx, yy, zz = np.meshgrid(np.arange(10), np.arange(10), np.arange(10)) |
|  | x\_min, x\_max = X[0, :].min() - 1, X[0, :].max() + 1 |
|  | y\_min, y\_max = X[1, :].min() - 1, X[1, :].max() + 1 |
|  | z\_min, z\_max = X[2, :].min() - 1, X[2, :].max() + 1 |
|  | colors=['red'] |
|  | cmap = matplotlib.colors.ListedColormap(colors) |
|  | h = 1 |
|  | xx, yy, zz = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h), |
|  | np.arange(z\_min, z\_max, h)) |
|  | # For the meh grid values predict a model |
|  | a=np.c\_[xx.ravel(), yy.ravel(), zz.ravel()] |
|  | Z = predict(parameters,a.T) |
|  | r=Z.T |
|  | r1=r.reshape(xx.shape) |
|  | # Find teh values for which the repdiction is 1 |
|  | xx1=xx[r1] |
|  | yy1=yy[r1] |
|  | zz1=zz[r1] |
|  | # Plot these values |
|  | ax = plt.axes(projection='3d') |
|  | #ax.plot\_trisurf(xx1, yy1, zz1, cmap='bone', edgecolor='none'); |
|  | ax.scatter3D(xx1, yy1,zz1, c=zz1,s=10,cmap=cmap) |
|  | #ax.plot\_surface(xx1, yy1, zz1, 'gray') |

Load\_mnist.Py

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

MNIST.R

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

**Conclusion**

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!