# ECE 763 Computer Vision

## Project-2 Report

Pratik Abhay Bhave

**Face image classification** using a simple neural network to get familiar with steps of training neural networks

Index:

TABLE OF CONTENTS

DATASET ..... 2

TRAIN DATASET ..... 2

TEST DATASET ..... 3

STEP 1: PREPROCESS DATA ..... 3

STEP 2: CHOOSE THE ARCHITECTURE ..... 3

BABYSITTING PROCESS ..... 4

NUMBER OF HIDDEN LAYERS ..... 4

NUMBER OF NEURONS PER LAYER ..... 4

ACTIVATION FUNCTIONS FOR INPUT ..... 5

OPTIMIZER FUNCTIONS ..... 5

LEARNING RATE ..... 5

REGULARIZATION ..... 7

CONCLUSION ..... 8

**Data Set: CELEBA**

**Data set URL:** http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

Data set contains 202599 images with face annotation given in the format of (x1,y1) in the left corner of the image and (width, height) of the bounding box. The annotations were saved in a txt file. Fig 2 shows one of the face image.

Sample format of annotated file

Fig.2 Fig.3

*000001.jpg 95 71 226 313*

*000002.jpg 72 94 221 306*

*000003.jpg 216 59 91 126*

*000004.jpg 622 257 564 781*

*000005.jpg 236 109 120 166*

Fig 1

Using a python script "**Data_Creation.py**" the annotations were imported from "**list_bbox_celeba.txt**" and 11000 images were cropped according to the co-ordinates shown above in Fig 1

The images were then resized to 60x60 as shown in Fig 3, and for all 11000 images, images were re-cropped from left corners with [0:60],[0:60] so as to get 60x60 **Non face** images.

So, in total we have 22000 images, which were saved in a mat file labelled '**total_data.mat**' and uploaded on google drive, for easy extraction of data and faster computation of the code.

Also, data labels are given in '**One Hot**' format, viz, **01** for **face** and **10** for **nonface** (since we have 2 classes, we have 2 bits representing the labels). The labels were also saved in a mat file labelled **'total_label.mat'** and uploaded on google drive for easy extraction **Data division in training and testing data sets:**

**Training Data**

**Train_dataset:** Out of the Dataset of 22000 images, 10000 images of both face and non-face are used for the training purpose, each image flattened out as a 1x10800 vector, thereby making the total dimension of train_dataset as 20000x10800.

**Train_labels:** Labels corresponding to the training set were extracted and stored in train_labels making the total dimension as 20000x2

## Testing Data

**Test_dataset:** Out of the Dataset of 22000 images, 1000 images of both face and non-face are used for the training purpose, each image flattened out as a 1x10800 vector, thereby making the total dimension of test_dataset as 2000x10800.

**Test_labels:** Labels corresponding to the test dataset were extracted and stored in test_labels making the total dimension as 2000x2

**Step1**: Preprocess the data

Following is the comparison of the output of the neural network with input as preprocessed data (normalized data) and input as original data.

The neural network used is the one that provides optimal output, which would be explained in the next pages.

```
In [14]: runfile('C:/Python35/final_code_1.py', wdir='C:/Python35')
Epoch 1 completed out of 10 loss: 1621968.2374267578
Epoch 2 completed out of 10 loss: 5651425.196998596
Epoch 3 completed out of 10 loss: 4403220.397705078
Epoch 4 completed out of 10 loss: 4806454.189952586
Epoch 5 completed out of 10 loss: 3558617.9412612915
Epoch 6 completed out of 10 loss: 2212333.028820038
Epoch 7 completed out of 10 loss: 980074.9575424194
Epoch 8 completed out of 10 loss: 440677.1363372803
Epoch 9 completed out of 10 loss: 270817.61756134033
Epoch 10 completed out of 10 loss: 177135.50148010254
Accuracy: 0.982
```

Fig 4. Output with Pre -Processed input data

```
In [7]: runfile('C:/Python35/windows_f_testing.py', wdir='C:/Python35')
Epoch 1 completed out of 10 loss: 1888197.9881649017
Epoch 2 completed out of 10 loss: 7578949.341114044
Epoch 3 completed out of 10 loss: 6940464.532961845
Epoch 4 completed out of 10 loss: 6490583.317909241
Epoch 5 completed out of 10 loss: 4441808.118598938
Epoch 6 completed out of 10 loss: 2701151.608994961
Epoch 7 completed out of 10 loss: 1425349.558856964
Epoch 8 completed out of 10 loss: 773898.851474762
Epoch 9 completed out of 10 loss: 372090.74017572403
Epoch 10 completed out of 10 loss: 206818.95786476135
Accuracy: 0.965
```

Fig 5. Output without Pre -Processed input data

**Step 2** : Choosing architecture

As mentioned above, I have used a simple multi-layer neural network using tensorflow, with 2 hidden layers, with 2000 and 1000 number of neurons respectively, and the entire project code is written in Python

```
n_nodes_hl1 = 2000
n_nodes_hl2 = 1000
#n_nodes_hl3 = 500
n_classes = 2
batch_size = 128

x = tf.placeholder('float', [None, 10800])
y = tf.placeholder('float')

def neural_network_model(data):
    hidden_1_layer = {'weights':tf.Variable(tf.random_normal([10800, n_nodes_hl1])),
                      'biases':tf.Variable(tf.random_normal([n_nodes_hl1]))}

    hidden_2_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl1, n_nodes_hl2])),
                      'biases':tf.Variable(tf.random_normal([n_nodes_hl2]))}

    output_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl2, n_classes])),
                    'biases':tf.Variable(tf.random_normal([n_classes])),}


    l1 = tf.add(tf.matmul(data,hidden_1_layer['weights']), hidden_1_layer['biases'])
    l1 = tf.nn.leaky_relu(l1)

    l2 = tf.add(tf.matmul(l1,hidden_2_layer['weights']), hidden_2_layer['biases'])
    l2 = tf.nn.leaky_relu(l2)

    output = tf.matmul(l2,output_layer['weights']) + output_layer['biases']
    return output
```

Fig 6. Architecture of Neural Network Used

**Babysitting process**:

There are many constraints and methods that affect the performance of a neural network. I have researched on a few and tried to find the best optimal model for the task of image classification.

1. Number of hidden layers

   There is no finite definition or mathematical formula that advices to use a particular 'n' number of hidden layers. Almost all the research papers that I read suggests that it really depends on many factors, but the prime factor being "Type of classification task". Since our task is a binary classification for faces, a simple neural network containing just one hidden layer also gave a very powerful accuracy of 95%. Since there are many possible combinations possible, it is indeed difficult to find the optimal number in such a small span. After trial and error and playing with the values, I found out that using lot of layers (5, 10) without using any LSTM or convolutional method for a simple neural network deprecates the performance of a neural network (Accuracy ~ 60%). Finally, I found that a **two**-layered model gave me an accuracy of 98.6 %, and hence I proceeded with using it, as shown in Fig 6.

2. Number of neurons per layer

   Just like point 1, this value is also very much a random selection, although research says using decreasing layer of neurons per layer helps a better output. I have used

**2000** and **1000** number of neurons for the 2 hidden layers respectively, as shown in Fig 6

3. Activation Functions for Input

Activation functions are used to keep the output between one and zero range. There are multiple activation functions available, and ReLU is the most popular Activation function used. However, a recently derived function "**Leaky ReLU**" gave a better accuracy and I used it as an activation function. Given below is a table with accuracy

```
l1 = tf.add(tf.matmul(data,hidden_1_layer['weights']), hidden_1_layer['biases'])
l1 = tf.nn.leaky_relu(l1)
#l1= tf.nn.relu(l1)
#l1= tf.nn.tanh(l1)
#l1= tf.nn.sigmoid(l1)
#l1= tf.nn.relu6(l1)
```

Fig 7. Different Activation Functions

| Activation Function | Accuracy |
|---|---|
| ReLU | 96.6 |
| Leaky ReLU | 98.2 |
| Sigmoid | 93.54 |
| Tanh | 90.53 |
| ReLU6 | 92.45 |

Table 1

values obtained for different Activation functions.

4. Optimizer Functions

Our main aim is to reduce the loss or cost of the computations. Just like activation functions, there are plenty of Optimizer Functions available, and optimization in deep neural networks is currently a very active area for research. Given below is a

```
prediction,regularization = neural_network_model(x)
cost = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(logits=prediction,labels=y) )
#cost=tf.nn.sigmoid_cross_entropy_with_logits(_sentinel =None, labels= y, logits= prediction, name=None)
#cost =  tf.reduce_mean(cost + beta*regularization)
optimizer = tf.train.AdamOptimizer(0.01).minimize(cost)
#optimizer = tf.train.AdadeltaOptimizer().minimize(cost)
#optimizer = tf.train.AdagradOptimizer(0.001).minimize(cost)
#optimizer = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
```

Fig 8. Different Optimizer Functions

| Optimization Functions | Accuracy |
|---|---|
| SGD | 82.78 |
| Adam | 98.2 |
| AdaDelta | 70.1 |
| AdaGrad | 96.7 |

Table 2

table with accuracy values for different Optimizer functions.

5. Learning Rate

Probably the most important parameter affecting a neural network, learning rate directly affects the loss of the neural network. As taught by professor in the lectures, I referred to the graph below and played with the learning rate values to understand how the values affect the loss, and thereby the performance of the neural network. Following are the results of neural network for different values of learning rate.

a) Learning rate = 0.01

This is a high learning rate, as compared to the default rate (0.001). Accuracy observed is as shown.

```
In [5]: runfile('C:/Python35/final_code_1.py', wdir='C:/
Python35')
Epoch 1 completed out of 10 loss: 9801094.581542969
Epoch 2 completed out of 10 loss: 128883532.11920166
Epoch 3 completed out of 10 loss: 100028991.35974121
Epoch 4 completed out of 10 loss: 42829642.56906128
Epoch 5 completed out of 10 loss: 29620816.122650146
Epoch 6 completed out of 10 loss: 10679599.59829712
Epoch 7 completed out of 10 loss: 3713037.07220459
Epoch 8 completed out of 10 loss: 862808.714553833
Epoch 9 completed out of 10 loss: 632226.9592285156
Epoch 10 completed out of 10 loss: 390203.3239440918
Accuracy: 0.9515
```

Fig 9 Learning rate = 0.01

b) Learning rate = 0.0001

```
In [7]: runfile('C:/Python35/final_code_1.py', wdir='C:/
Python35')
Epoch 1 completed out of 10 loss: 1061472.0637931824
Epoch 2 completed out of 10 loss: 916969.1907272339
Epoch 3 completed out of 10 loss: 252144.69090270996
Epoch 4 completed out of 10 loss: 143264.19054222107
Epoch 5 completed out of 10 loss: 98310.89493942261
Epoch 6 completed out of 10 loss: 76523.04311466217
Epoch 7 completed out of 10 loss: 60763.391040802
Epoch 8 completed out of 10 loss: 43420.9937667856
Epoch 9 completed out of 10 loss: 33880.20957946777
Epoch 10 completed out of 10 loss: 26350.663047790527
Accuracy: 0.9645
```

Fig 10. Learning Rate = 0.0001

c) Learning rate= 0.00001

```
In [8]: runfile('C:/Python35/final_code_1.py', wdir='C:/
Python35')
Epoch 1 completed out of 10 loss: 11819296.23413086
Epoch 2 completed out of 10 loss: 10647433.240844727
Epoch 3 completed out of 10 loss: 9583656.4085083
Epoch 4 completed out of 10 loss: 8544296.089294434
Epoch 5 completed out of 10 loss: 7520473.272277832
Epoch 6 completed out of 10 loss: 6509367.9841918945
Epoch 7 completed out of 10 loss: 5513970.8779296875
Epoch 8 completed out of 10 loss: 4543053.248352051
Epoch 9 completed out of 10 loss: 3674379.411315918
Epoch 10 completed out of 10 loss: 3058253.669921875
Accuracy: 0.5895
```

Fig 11 Learning rate = 0.00001

This is a low learning rate, and as seen in the output, the loss saturates, and the accuracy is very low, as shown in Fig 11

d) Learning rate = 0.001 (default)

After playing with the learning rate values, I found out the the default learning rate works best for AdamOptimizer.

```
In [14]: runfile('C:/Python35/final_code_1.py', wdir='C:/Python35')
Epoch 1 completed out of 10 loss: 1621968.2374267578
Epoch 2 completed out of 10 loss: 5651425.196998596
Epoch 3 completed out of 10 loss: 4403220.397705078
Epoch 4 completed out of 10 loss: 4806454.189952586
Epoch 5 completed out of 10 loss: 3558617.9412612915
Epoch 6 completed out of 10 loss: 2212333.028820038
Epoch 7 completed out of 10 loss: 980074.9575424194
Epoch 8 completed out of 10 loss: 440677.1363372803
Epoch 9 completed out of 10 loss: 270817.61756134033
Epoch 10 completed out of 10 loss: 177135.50148010254
Accuracy: 0.982
```

Fig 12 Learning Rate = 0.001

Summary of Accuracy corresponding to the learning rate is given below

| Learning Rate | Accuracy |
|---|---|
| 0.01 | 95.15 |
| 0.0001 | 96.45 |
| 0.00001 | 58.95 |
| 0.001(default) | 98.2 |

Table 3

6. Regularization

Regularization parameter contributes to the loss and is beneficial to be used whenever our data is being over-fit to the network. Given below is the comparison of output by using the L2 regularization parameter with different β values.

```
output = tf.matmul(l2,output_layer['weights']) + output_layer['biases']
regularization = tf.nn.l2_loss(hidden_1_layer['weights']) + \
                 tf.nn.l2_loss(hidden_2_layer['weights']) + \
                 tf.nn.l2_loss(output_layer['weights'])
return output,regularization

train_neural_network(x):
prediction,regularization = neural_network_model(x)
beta= 0.001
cost = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(logits=prediction,labels=y) )
cost =  tf.reduce_mean(cost + beta*regularization)
optimizer = tf.train.AdamOptimizer(0.001).minimize(cost)
```

Fig 13 Regularization Calculation

a) β = 0.001

```
In [6]: runfile('C:/Python35/final_code_1.py', wdir='C:/Python35')
Epoch 1 completed out of 10 loss: 3039451.904296875
Epoch 2 completed out of 10 loss: 9044427.72265625
Epoch 3 completed out of 10 loss: 7525649.994140625
Epoch 4 completed out of 10 loss: 8217025.55859375
Epoch 5 completed out of 10 loss: 5970587.84375
Epoch 6 completed out of 10 loss: 4501379.1142578125
Epoch 7 completed out of 10 loss: 3181746.337890625
Epoch 8 completed out of 10 loss: 2433710.6923828125
Epoch 9 completed out of 10 loss: 2195228.705078125
Epoch 10 completed out of 10 loss: 2095365.744140625
Accuracy: 0.976
```

Fig 14 Accuracy when β = 0.001

b) β = 1000

```
In [7]: runfile('C:/Python35/final_code_1.py', wdir='C:/Python35')
Epoch 1 completed out of 10 loss: 1631160656896.0
Epoch 2 completed out of 10 loss: 1276735163904.0
Epoch 3 completed out of 10 loss: 1002438587392.0
Epoch 4 completed out of 10 loss: 788237893120.0
Epoch 5 completed out of 10 loss: 619909245184.0
Epoch 6 completed out of 10 loss: 487179213568.0
Epoch 7 completed out of 10 loss: 382410543360.0
Epoch 8 completed out of 10 loss: 299591065472.0
Epoch 9 completed out of 10 loss: 234150272256.0
Epoch 10 completed out of 10 loss: 182512618752.0
Accuracy: 0.58
```

Fig 15 Accuracy when β = 1000

As seen, a higher term of β leads to a higher loss, which is true sensibly.

**Conclusion:**

Finally, after intensive research and trial and error with values, I have used the following parameters for babysitting my model:

- Number of hidden layers = 2
- Number of neurons per layer= 2000 in layer 1, 2000 in layer 2
- Activation Function: Leaky ReLU
- Learning Rate: 0.001
- Optimization Function: AdamOptimizer
- Regularization: Disabled
- Batch_Size = 128
- Number of Epochs = 10
- Accuracy achieved = 98.2 %

Following is the graph of number of epochs versus Loss

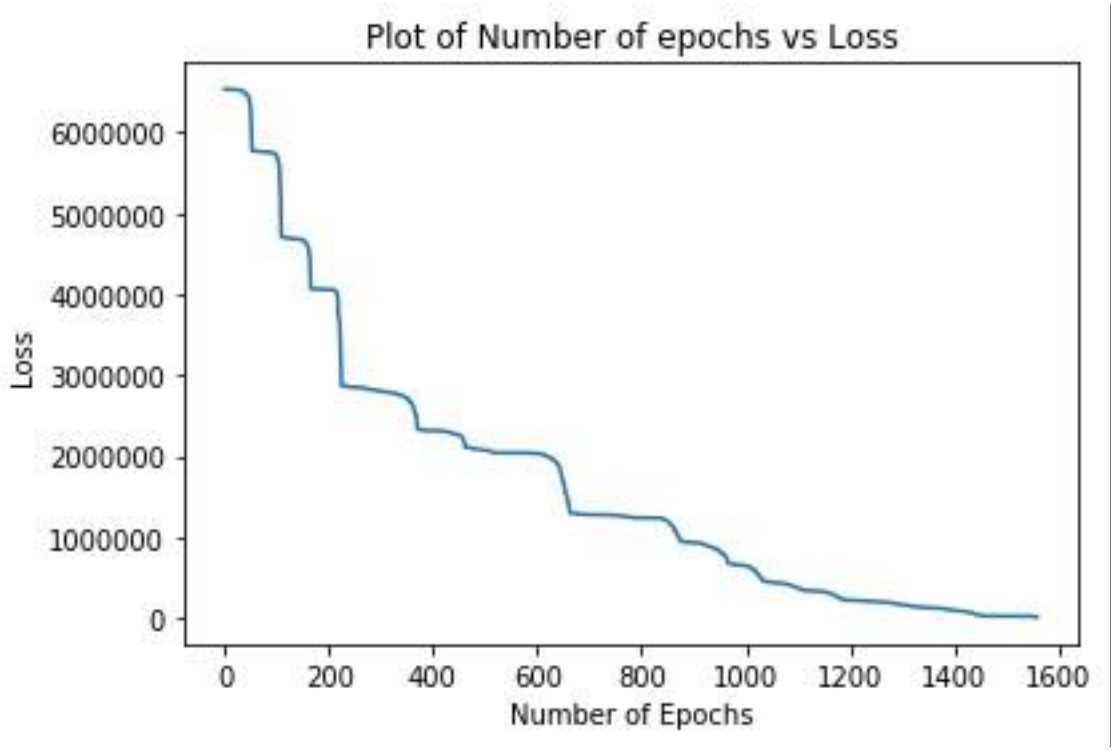


Fig 16