Gesture Classification of Indian Sign Language

A Project Report

submitted by

GAURAV SINGH

EE14B125

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2018

REPORT CERTIFICATE

This is to certify that the report titled **Gesture Classification of Indian Sign Language**, submitted by **Gaurav Singh**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. C.S.Ramalingam
Project Guide
Associate Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai
Date: 11th May 2018

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my guide Dr.C.S.Ramalingam for giving me an opportunity to work under him. Also I would like to thank you for constantly guiding me thoughtfully and efficiently throughout this project, giving me an opportunity to work at my own pace along my own lines, while providing me with very useful directions and insights whenever necessary.

I would also take this opportunity to thank all my friends who have been a great source of motivation and encouragement.

Finally I would also like to thank all of them who have helped me complete my project successfully.

Gaurav Singh
EE14B125
Student
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai
Date: 11th May 2018

TABLE OF CONTENTS

Contents

1 Introduction

There are millions of people around the world who can speak and understand sign language. However, these millions become a tiny amount in front of the billions of people with the inability to understand or speak sign language. Hence, there has always been a necessity as well as the need of coming up with an automated procedure to translate the sign language into worded text.

However, there hasn't been much development in this regard. The reason is because of the absence of a universally spoken sign language thus creating the absence of a common platform on which the work can be done. Also, it is very difficult to integrate the platforms of different sign languages as each one has different grammatical rules as well as the structure of the language. The sign language that has seen the maximum amount of development in this regard is the American Sign Language (ASL). The Indian Sign Language (or also known as the Indo-Pakistani Sign Language), however has seen very little development.

Hence there is a lot of scope for improvement of the Indian Sign Language and in this report we are going to explore the ways of achieving so. But before that, it would be important for us to look into how much work has been done previously on this so that we can develop on the pre-established platform.

2 Previous Work on ISL(Indian Sign Language)

There has been multiple work done on the ISL, some of them include

2.1 RGB based approaches

Rekha et al.[2] work on a vocabulary of 23 static signs and 3 dynamic signs of ISL. They use skin colour segmentation through a Gaussian model to locate hands. Then they use edge orientations (from Principle Curvature based Region Detector) and texture (from Wavelet Packet based Decomposition) as features to train a multiclass SVM and get a recognition rate of 86.3%. However their approach is very slow and has an average recognition time around 55 s.

Table 1: Approaches for ISL

Authors	Dataset/Sensor	Segmentation/features/ recognition method	Recognition Rate (%)
Geeta[1]	static(29, 1450) RGB	B-Spline, key maximum curvature points	∼90
Rekha[2]	static (23, 920) dynamic (3, 66) RGB	Skin color, edge orientations, texture;SVM	77.2
Singha[3]	static (24, 240) RGB	Skin color, Eigen Values; nearest neighbour	97
Bhuyan[4]	static (8, 400) RGB	Skin color, geometric features;nearest neighbour	>90

Singha et al.[3] use Eigen values extracted from segmented hands to classify 24 static signs of ISL using RGB images. They do a nearest neighbour classification using eigen value weighted Euclidean distances and achieve 97% recognition accuracy.

Bhuyan et al.[4] use a skin colour based segmentation procedure to extract hands. A model based approach with geometric features (relative angles between fingers) and Homogeneous Texture descriptors (HTD) is then used to classify static signs according to a nearest neighbour heuristic. They used eight gestures with a training dataset of 400 images and get a high recognition accuracy (above 90%).

2.2 3D based approaches

Van den Bergh and Van Gool et al.[5] (not in reference to ISL, however useful for our approach) use a hybrid approach to track hands by combining RGB and depth data obtained from a ToF camera. They use a face detector (implemented in OpenCV) to locate the face in the RGB image and obtain its distance in the depth image, and for the remaining regions which satisfy a depth threshold, they identify the skin colour using a Gaussian Mixture Model based skin colour model. This enables them to find anything that is extended (like hands) in front of the body towards the camera. After detecting the hands, they use Average Neighbourhood Margin Maximization (ANMM) [22] (approximated using Haarlets) to do matching from their database. They achieved a recognition rate of 99.54% on a vocabulary of six symbols and 350 sample images

3 Dataset used

The dataset (see table 2 for sign names), henceforth referred to as the ISL Dataset, consists of 140 static signs handpicked from ISL. Volunteers were advised to be upright and keep their hands distinct from the body as much as possible. This was done to enable depth images to have a greater level of detail. Right and left hands were not interchangeable in all signs. All volunteers were non-native to sign language. Volunteers were advised to reform the signs after each trial of a sign. This ensured non-duplicacy of the image. Volunteers stood about half a metre from a flat surface. There was a distance of approximately 1 m between the Kinect and the volunteer. The dataset has been selected from the ISL general, technical and banking dictionaries. Apart from complete words, the dataset also has signs for manual fingerspelling (signs for alphabets) . A word may have different variantsparticularly single-handed and two-handed variants. In such cases all have been included. Symbols for single-handed J, single-handed Z and double-handed J are dynamic in nature, so they have not been included. Some Hindi words like Bhangada and Varanasi have also been incorporated. Some images were rejected as the gestures in them were incorrectly performed. The dataset overall consisted of a vocabulary of 140 symbols was collected using 18 subjects, totalling 5041 images.

4 Previous Work done on the dataset

Zafar Ahmed Ansari and Gaurav Harit et al.[6] implemented a functional unobtrusive Indian sign language recognition system using the above dataset. This system proposes a method for a novel, low-cost and easy-to-use application, for Indian Sign Language recognition, using the Microsoft Kinect camera (for obtaining the depth values). In the fingerspelling category of our dataset, they achieved above 90% recognition rates for 13 signs and 100% recognition for 3 signs with overall 16 distinct alphabets (A, B, D, E, F, G, H, K, P, R, T, U, W, X, Y, Z) recognised with an average accuracy rate of 90.68%.

The method employed by them includes pre processing of the image involving feature extraction(they use SIFT algorithm for this) and noise reduction(of depth values of the image) followed by k-means clustering to derive proper classification boundaries. They then proceed to use kNN based procedure to

Table 2: List of signs of the ISL dataset

1.	One	2.	Two	3.	Three
4.	Four	5.	Five	6.	Six
7.	Seven	8.	Eight	9.	Nine
10.	Ten	11.	A	12.	Add
13.	Appreciation	14.	A-SingleHanded	15.	Assistance
16.	B	17.	Bell	18.	Between
19.	Bhangada	20.	Bite	21.	Blow
22.	Bottle	23.	bowl	24.	Boxing
25.	B-SingleHanded	26.	Bud	27.	C
28.	Conservation	29.	Control	30.	C-SingleHanded
31.	D	32.	Density	33.	Deposit
34.	D-SingleHanded	35.	E	36.	Elbow
37.	E-SingleHanded	38.	F	39.	Few
40.	Fine	41.	Friend	42.	F-SingleHanded
43.	G	44.	Ghost	45.	Good
46.	Gram	47.	G-SingleHanded	48.	Gun
49.	H	50.	Handcuffs	51.	Help
52.	Here	53.	Hold	54.	How
55.	H-SingleHanded	56.	I	57.	Intermediate
58.	Iron	59.	I-SingleHanded	60.	It
61.	K	62.	Keep	63.	K-SingleHanded
64.	L	65.	Leaf	66.	Learn
67.	Leprosy	68.	Little	69.	Lose
70.	L-SingleHanded	71.	M	72.	Mail
73.	Me	74.	Measure	75.	Mirror
76.	M-SingleHanded	77.	N	78.	Negative
79.	N-SingleHanded	80.	O	81.	Obedience
82.	Okay	83.	Opposite	84.	Opposition
85.	O-SingleHanded	86.	P	87.	Participation
88.	Paw	89.	Perfect	90.	Potentiality
91.	Pray	92.	Promise	93.	P-SingleHanded
94.	Q	95.	Q-SingleHanded	96.	Quantity
97.	Questions	98.	R	99.	Respect
100.	Rigid	101.	R-SingleHanded	102.	S
103.	Sample	104.	Season	105.	Secondary
106.	Size	107.	Skin	108.	Small
109.	Snake	110.	Some	111.	Specific
112.	S-SingleHanded	113.	Stand	114.	Strong
115.	Study	116.	Sugar	117.	T
118.	There	119.	Thick	120.	Thursday
121.	T-SingleHanded	122.	U	123.	Unit
124.	Up	125.	U-SingleHanded	126.	V
127.	Vacation	128.	Varanasi	129.	V-SingleHanded
130.	W	131.	Warn	132.	Weight
133.	Work	134.	W-SingleHanded	135.	X
136.	X-SingleHanded	137.	Y	138.	You
139.	Y-SingleHanded	140.	Z		

predict the class of the test data point. The pseudo code for kNN classification is given as follows:

Input: Let k be the number of nearest neighbours and D be the set of training samples

For each test sample $z = (x', y')$ do:

- Compute $d(x, x')$, the distance between x' and every example $(x, y) \in D$

- Select $D_z \subseteq D$, the set of k closest training examples to z

$$y' = \underset{v}{argmax} \sum_{(x_i, y_i) \in D_z} I(v = y_i)$$

- End for

Hence kNN being a computationally simpler method, the classification of the images is done on the spot thus giving a real time gesture classification method. However, this modelling was done clearly on the assumption that the user is still and only the valid datasets are considered. To incorporate for a moving user, we would try to employ a new method based on the neural networks(CNN in this case) for the gesture classification.

5 Neural Network Based Approach

Neural Networks have gained a lot of popularity lately. The reason is because they need no pre-processing of data and learn the weights accordingly to minimize the provided loss function. The only challenge in dealing with neural networks is to avoid over-fitting and to optimally design the architecture in order to get best accuracy of the test data.

In this case, we would be focusing on the RGB-D based Convolutional Neural Network as the data-point is a 3-D array with x and y axis representing the pixel's position of the image whereas z axis representing RGB and Depth values associated with each pixel.

But like training of any neural network is preceded by the formation of Dataset, we would like to describe the approach taken to form a proper training and testing dataset. The procedure can be described in the following steps:

- First we have been given the dataset comprising of the images and depth values for each user. We separate them into two folders 'RGB' and 'Depth'
- We then divide the images and depth values into training(Any 16 user's values) and testing(The remaining 2 user's values) datasets. Thus we divide the dataset into training and testing data with a ratio of approximately 90-10. The arrangement of data would look something like this:
  - RGB
    - Training
    - Testing
  - Depth

    - Training
    - Testing

- After partitioning the datasets, we parse through the established dataset while keeping in tabs of the labels associated with each data point. *(Note: The labels exist on the filename and would be useful for defining the training and testing labels as well as data synchronization.).* The code for parsing through the established dataset looks as follows:

Listing 1: Data parsing code

```
classes = ["Train", "Test"]
rgb_values = {}
depth_values = {}
rgb_labels = {}
depth_labels = {}
rgb_values["Train"] = []
rgb_values["Test"] = []
depth_values["Train"] = []
depth_values["Test"] = []
rgb_labels["Train"] = []
rgb_labels["Test"] = []
depth_labels["Train"] = []
depth_labels["Test"] = []

#Loop in order to take in the depth values of the images for both training and testing data
for clas in classes:
    filenames = os.listdir("../Project-Data/Depth/"+clas+"/")
    print filenames
    for fb in filenames:
        print fb
        for k in glob.glob("../Project-Data/Depth/"+clas+"/"+fb+"/*.txt"):
            my_data = genfromtxt(k, delimiter = ' ')
            depth_values[clas].append(my_data)
            names = k.split('/')
            fname = names[5].split('.')
            labels = fname[0].split('-')
            depth_labels[clas].append(labels[1:4])

#Loop in order to take in the RGB values of the images for
```

```
    both training and testing data
for clas in classes:
    filenames = os.listdir("../Project-Data/RGB/"+clas+"/")
    print filenames
    for fb in filenames:
        print fb
        for k in glob.glob("../Project-Data/RGB/"+clas+"/"+fb+"/*.png"):
            image = cv2.imread(k)
            rgb_values[clas].append(image)
            names = k.split('/')
            fname = names[5].split('.')
            labels = fname[0].split('-')
            rgb_labels[clas].append(labels[1:4])
```

- Now, we compare the labels of both the datasets to confirm the data synchronization. The looks like follows:

Listing 2: Data synchronization code

```
val = set(rgb_labels)&set(depth_labels)
if(val != 0):
    print "Data Synchronization Error"
    exit()
```

- Finally after achieving the synchronization, we concatenate to form the training data points and their corresponding labels. The code looks as follows:

Listing 3: Data concatenation code

```
labels = {}
data = {}
data["Train"] = []
data["Test"] = []
labels["Train"] = []
labels["Test"] = []

#Concatenating the RGB values and depth values to form the final data
for clas in classes:
```

```
10      for i in xrange(len(rgb_values[clas])):
11          rgb = np.asarray(rgb_values[clas][i])
12          dpth = np.asarray(depth_values[clas][i])
13          dta = np.dstack(rgb, dpth)
14          data[clas].append(dta)
15
16      labels[clas] = np.asarray(rgb_labels[clas])
17      labels[clas] = labels[clas][1]
18      labels[clas] = labels[clas] - 1
```

- We also take in values and labels of three separate datapoints for future analysis and demonstration of the trained model.

Now, since we have established a valid dataset for training and testing, we proceed into defining the neural network on which this data would be trained. However, before we get into the network architecture, it would be wise to revise all the important teminologies associated with a CNN.

5.1 Convolutional Layer

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting. The CONV layers parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size 5x5x4 (i.e. 5 pixels width and height, and 3 because here eeach pixel had have depth 4, the color channels and the depth value). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. Now, we will have an entire set of filters in each CONV layer (e.g. 32 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the

output volume.

Backpropagation: The backward pass for a convolution operation (for both the data and the weights) is also a convolution (but with spatially-flipped filters).

5.2 Pooling Layer

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged.

Backpropagation: As we know max(x, y) operation has a simple interpretation as only routing the gradient to the input that had the highest value in the forward pass. Hence, during the forward pass of a pooling layer it is common to keep track of the index of the max activation (sometimes also called the switches) so that gradient routing is efficient during backpropagation.

5.3 Fully Connected Layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

6 Architecture of the Network

The architecture of the network here has been derived after multiple trials in order to achieve maximum test accuracy from the model. However, we restricted from trying out a deeper network as the dataset involved limited number of

data points in the training set. The optimal network architecture after multiple trials resulted as follows:

- **Input Layer:** Here the input layer in one sense represents a single data point. However a single data point consists of $A*B*D$ values where A and B represents the image's width and height and D represents the depth or the number of values associated with each pixel. *(Here* $A = 480$*,* $B = 640$ *and* $D = 4$ *)*

  The dimension of the input layer can be altered by changing the $batch\_size$ and this will in turn influence the dimensions of the other hidden layers.

- **Convolutional Layer 1:** After defining the input layer, we now come to defining the first hidden layer of the network. The parameters of the layer are as follows:

  - *Kernel Filter Size:* 5x5
  - *Number of filters:* 32
  - *Activation Function:* ReLU
  - *Padding:* Same(Means we insert zero values to keep the x and y dimension same as that of the previous layer)

  ReLU(Rectified Linear Unit) activation function is zero when x < 0 and then linear with slope 1 when x > 0. ReLU activation is a widely popular activation function as it is computationally simpler and induces non-linearity into the model. The graph looks as follows:

- **Convolutional Layer 2:** This convolutional layer is very much similar to the previous ConV layer. Hence, it the parameters of the layer are as follows:

  - *Kernel Filter Size:* 5x5
  - *Number of filters:* 32
  - *Activation Function:* ReLU
  - *Padding:* Same

- **Pooling Layer 1:** This pooling layer is a max pooling layer with the following parameters:

  - *Filter Size:* 2x2

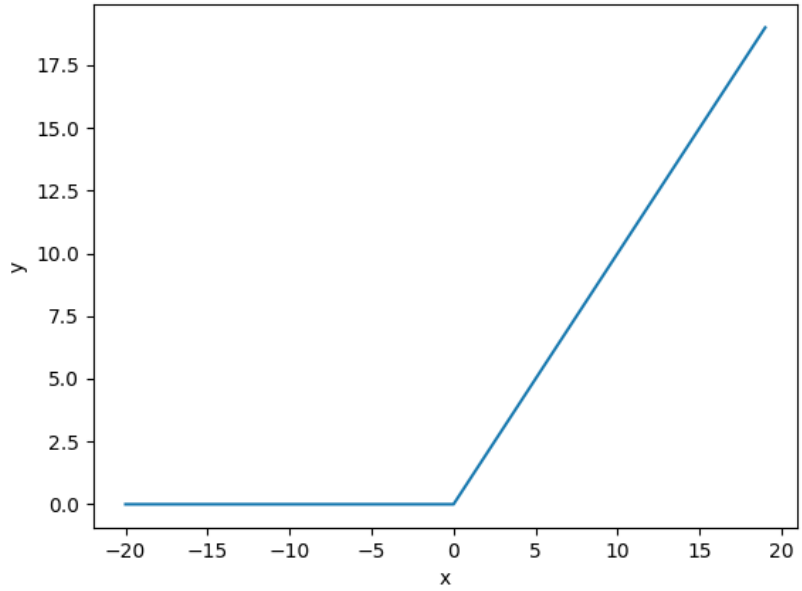

Figure 1: ReLU Function

  - *Strides:* 2
  - *Activation Function:* MAX

- **Convolutional Layer 3:** The first two ConV layers have been used to extract out the features. Now, form here on, each ConV layer would focus on providing weights to each feature in order to achieve accurate classification. The parameters of the third convolutional layer are as follows:
  - *Kernel Filter Size:* 5x5
  - *Number of filters:* 64
  - *Activation Function:* ReLU
  - *Padding:* Same
- **Convolutional Layer 4:**The fourth and final convolutional layer is just as it's previous ConV layer with the following parameters:
  - *Kernel Filter Size:* 5x5
  - *Number of filters:* 64
  - *Activation Function:* ReLU
  - *Padding:* Same
- **Pooling Layer 2:**This pooling layer, just like the previous pooling layer is applied to reduce the number of dimensions and thus reduce computations in the model. The parameters are as follows:

- *Filter Size:* 2x2
- *Strides:* 2
- *Activation Function:* MAX

- **Flattening the pool layer:** We have to define a fully connected dense layer. But before that, we have to flatten the pooling layer to two dimensions [$batch\_size, features$]. Here $features = \frac{A}{4} * \frac{B}{4} * 64$ as each pooling layer removed half of each height and width.

- **Dense Layer and Dropout:** The dense layer is a fully connected layer with a total of 1024 neurons. The activation function used in this layer is again ReLU. To help improve the results of our model, we also apply dropout regularization to our dense layer.

  The dropout rate is specified to be 0.4 or 40%. This means 40% of the elements will be randomly dropped out during training. Also, the dropout would be deactivated during the testing period. The reason of the existence of dropout regularization is to prevent over-fitting the training data.

- **Logits Layer:** The final layer in our neural network is the logits layer, which will return the raw values for our predictions. We create a dense layer with 140 neurons (one for each target class 0 to 139), with linear activation.

  Hence our final output tensor of the CNN, has a shape [$batch\_size, 140$]

The network can visualised through the following image:

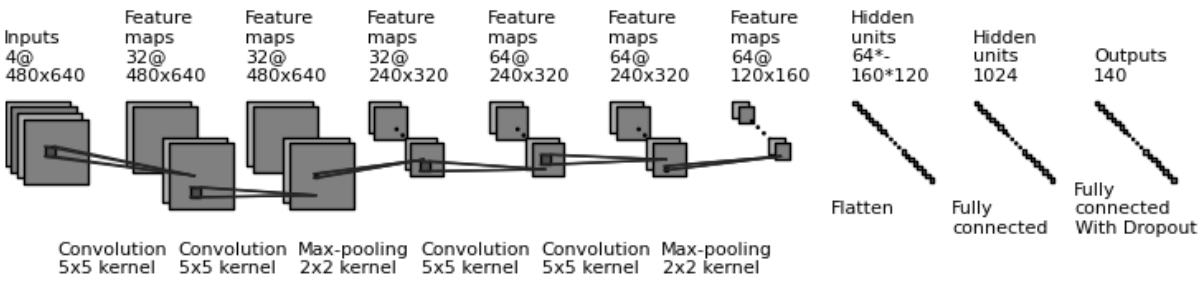

Figure 2: CNN Network

6.1 Generate Predictions:

After deriving the output tensor from the CNN we then proceed to derive two important values from it:

- The **predicted class** for each example. Our predicted class is the element in the corresponding row of the logits tensor with the highest raw value.
- The **probabilities** for each possible target class for each example. We derived the probabilities from our logits layer by applying softmax activation function. The softmax function is described by the following equation:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

where $j = 1, ...K$

6.2 Loss function and training parameters

After defining the final activation function as the softmax function, we use cross entropy as the loss metric.

Now that we have established all the necessary parameters required for training, we perform a total of 20000 epochs with a learning rate of 0.002.

7 Training, Evaluation and Prediction

After describing the model, it is now time to train and evaluate the model. After that, we can then predict the output for some specific input data points and infer the results later and use them to propose a new model to incorporate the robustness.

7.1 Training

During the process the training, we keep track of the loss function to know whether the model is learning or not. In this case we consider users 3 and 15 as the test set and the remaining users as the training set. The loss function as a number of epochs comes out as follows(next page):

Table 3: Training Loss

Epochs	Loss	Epochs	Loss	Epochs	Loss	Epochs	Loss
0	4.811	500	3.363	1000	2.514	1500	1.922
2000	1.466	2500	1.267	3000	1.05	3500	0.926
4000	0.828	4500	0.739	5000	0.652	5500	0.581
6000	0.526	6500	0.473	7000	0.432	7500	0.391
8000	0.354	8500	0.322	9000	0.296	9500	0.263
10000	0.242	10500	0.226	11000	0.202	11500	0.191
12000	0.18	12500	0.177	13000	0.163	13500	0.152
14000	0.141	14500	0.134	15000	0.126	15500	0.114
16000	0.109	16500	0.104	17000	0.103	17500	0.102
18000	0.1	18500	0.097	19000	0.094	19500	0.092

Plotting the value of loss a function of epochs comes out as follows(next page):

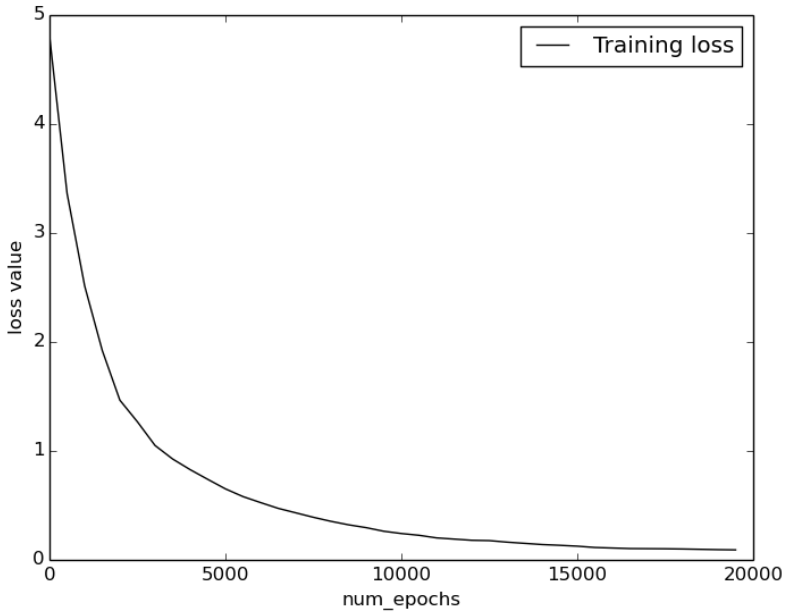

Figure 3: Training Loss

7.2 Evaluation

Clearly from the above figure, we can safely say that the model has learnt properly. However, just achieving a decreasing loss function is not enough. This just proves that the model has learnt well for the given training data set. However, we now might need to validate the model using test data. To make sure that the data is not biased, we performed training and testing of the model three times, each time with different sets of training and testing data. The results came out as follows:

- *Combination 1:* Test Data involved users 17 and 18. Whereas the rest were used for training. The result was:
  `INFO:tensorflow:Saving evaluation summary for step 20000: accuracy = 0.9183, loss = 0.09127`
  `{'loss': 0.09127105, 'global_step': 20000, 'accuracy': 0.91839998}`
- *Combination 2:* Test Data involved users 3 and 15. Whereas the rest were used for training. The result was:
  `INFO:tensorflow:Saving evaluation summary for step 20000: accuracy = 0.8999, loss = 0.10535`
  `{'loss': 0.10535219, 'global_step': 20000, 'accuracy': 0.89997231}`

- *Combination 3:* Test Data involved users 5 and 10. Whereas the rest were used for training. The result was:

```
INFO:tensorflow:Saving evaluation summary for step 20000: ac-
curacy = 0.9055, loss = 0.09738
{'loss': 0.09738444, 'global_step': 20000, 'accuracy': 0.90553288}
```

Keeping these three results into account, we can say that the model gives an accuracy of 90.79% on an average. Where this value was derived by taking the average of the above three combinations.

$$acc = \frac{acc_1 + acc_2 + acc_3}{3}$$

Where acc_i is the accuracy of combination i.

7.3 Prediction

We have considered three unique images to see how the model performs and how accurately can it predict the class of the given test image.

- *Case 1:* We consider the User ID 2 and the image labelled *'USER-2-60-1.png'*. It is given as:

Figure 4: Case 1

This image when entered into the network, gives out the following plot as the output:

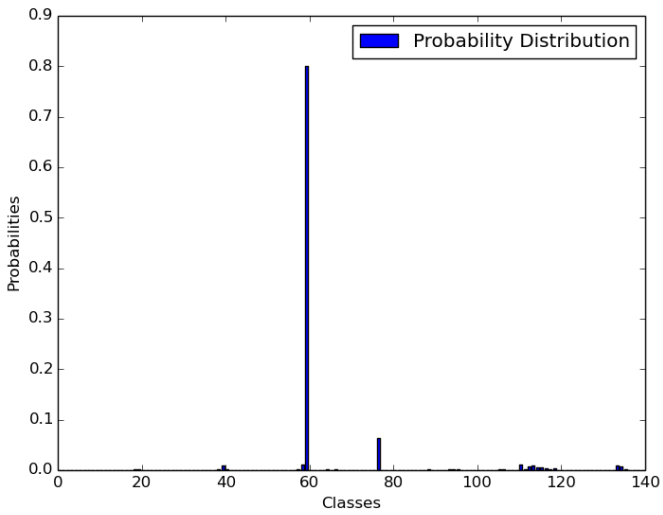

Figure 5: Prob Dist of Case 1

- *Case 2:* We consider the User ID 2 and the image labelled *'USER-2-27-1.png'*. It is given as:

Figure 6: Case 2

This image when entered into the network, gives out the following plot as the output:

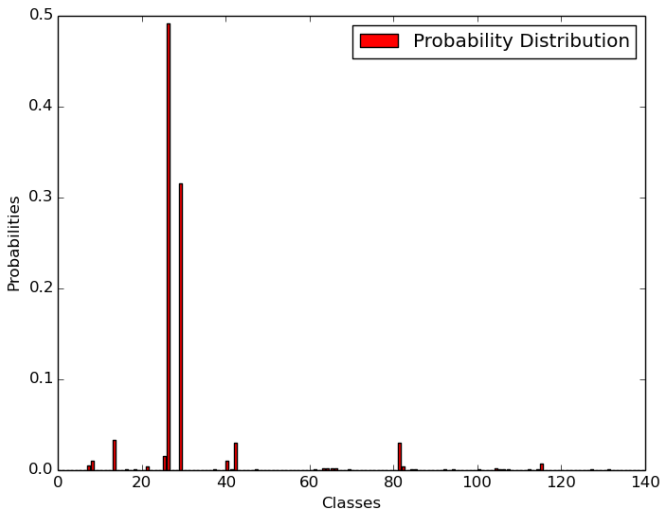

Figure 7: Prob Dist of Case 2

- *Case 3:* We consider the User ID 2 and the image labelled *'USER-2-7-1.png'*. It is given as:

Figure 8: Case 3

This image when entered into the network, gives out the following plot as the output:

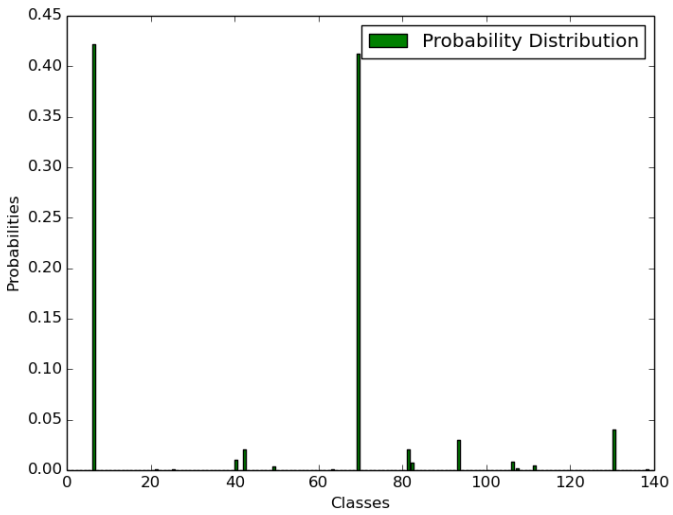

Figure 9: Prob Dist of Case 3

After considering the three cases, we also keep into account the two highest probabilities in each case for future calculations. Here p_k is the highest probability values and k is the corresponding class whereas p_i is the second highest probability value with i being the class associated with it.

Table 4: Two Highest probabilities for each case

	User ID	Label	p_k	k	p_i	i
1	2	60	0.8005	60	0.0642	77
2	2	27	0.4912	27	0.3158	30
3	2	7	0.4217	7	0.4122	70

8 Analysis and Assurance of the model

Before we proceed to analyse and define the assurance of the model, we would first like to look into two terms:

- *Classifiable Image:* An image can be classified by the model and one certain class "can" be predicted as the class of the image
- *Assured class:* Model can only predict the class of the image if it is "assured" that there can be no other class to which the image can be a part of.

Defining these terms is not the only thing necessary. After the definition, it then becomes mandatory to define the parameters associated with each of this terms:

- *Ratio:* The ration takes into account of how much greater is the highest value among the probability distribution(p_k) in comparison to the second highest value(p_i). Hence, to measure the difference, we the ratio of the two values as:
$$ratio = r = \frac{p_k}{p_i}$$
- *Classification filter value or β:* This value finds out all the images unfit for classification and filter them out. Now if the ratio r is less than this value, we deem the image as un-classifiable. Thus if
$$r = \frac{p_k}{p_i} < \beta$$
The image cannot be classified. However, if r is greater than β, we consider the image as classifiable. Thus if
$$r = \frac{p_k}{p_i} > \beta$$
The image is classifiable. We generally define the value of β as close to 1 as possible. Here, in our case, we have defined the value of β as 1.1. Now, after considering the test data set(Users 17 and 18), we find out that out of the 560 test images, we get 532 images as classifiable.
- *Assurance filter value or α:* After the image is declared classifiable, we define another variable which would enable the model to classify the image as one particular class. If the ratio r is less than this value that is:
$$r = \frac{p_k}{p_i} < \alpha$$

We do not classify the image. However, if r is grater than this value that is:

$$r = \frac{p_k}{p_i} > \alpha$$

We classify the image as the class with the highest probability value(i.e. k). The value of α is completely user dependent. Since, we had a dataset consisting of only valid data points, we defined it as a pretty low value that is 1.6

In out test set, out of the 532 classifiable images, we find out that 62.06% can be classified with assurance given our defined value of α. (This also shows that some classes can be more easily classified than the others) We define this value as *maximum achievable accuracy*.

8.1 Maximum Achievable Accuracy

As we have seen that out of the 532 classifiable images, only 62.06% of them can be classified with assurance. Now this quantity can be interpreted as accuracy as ONLY 62.06% are classified as gives out an output of a class.

Thus, the maximum achievable accuracy of the model is given as:

$$\text{Maximum Achievable Accuracy} = \frac{\text{Number of data points with assured class}}{\text{Number of classifiable data points}}$$

However, we use the word "maximum" because the data set used here only involves valid images. Now, if we were to replace a bunch of valid images with invalid images, these invalid images would either be discarded as unclassifiable or as an image with an un-assured class(we have to choose α wisely so that this happens). Either way it would only increase the denominator(or keep it the same), thus reducing the accuracy(or keeping it the same). Now, we can intuitively say that increasing the value of α would only decrease the accuracy of the model as fewer images would be classified, however the robustness of the model will increase as fewer invalid images would get classified. Hence, we arrive at a trade-off where we either choose the accuracy or the robustness of the model.

8.2 Examples

We have considered the three cases above mentioned to see the prediction outputs for the model. Now, we would consider these cases as examples to derive inferences from the parameters defined above.

- *Example 1:*
$$\frac{p_k}{p_i} = \frac{0.8005}{0.0642} = 12.46$$
As the value is greater than α, we can say that it is classifiable and that too the class is '60' or as we have seen from table 2: "It"

- *Example 2:*
$$\frac{p_k}{p_i} = \frac{0.4912}{0.3158} = 1.55$$
As the value is greater less α yet greater than β, we can say that it is classifiable but we cannot assure a class to it.

- *Example 3:*
$$\frac{p_k}{p_i} = \frac{0.4217}{0.4122} = 1.02$$
As the value is less than β, the image is not even classifiable

9 Future Work

We were able to achieve a good enough model for the images. However, we have to consider that the real life datasets are videos. Now, if we were to divide the videos into multiple frames and consider each frame as a data point, we arrive at a input dataset with maximum of the images as invalid(because they represent the intermediate stages of the gestures). Hence, here the previously defined values α and β come into play.

Since we lacked a proper video based dataset of the ISL, we had to restrict our testing to the pre-existing dataset. However, if we were to create a dataset involving lot of intermediate invalid frames, we might be able to arrive at an optimal value of α which would give us a decent accuracy with a good enough robustness on the dataset.

Also, as the CNN is huge and computationally intensive, the testing and prediction can be exported to an external host like AWS, etc to obtain a real time prediction model.

10 References:

[1] Geetha M and Manjusha U (2012). A vision based recognition of indian sign language alphabets and numerals using B-spline approximation. *Int. J. Comp. Sci. Eng. (IJCSE)*

[2] Rekha J, Bhattacharya J and Majumder S (2011). Shape, texture and local movement hand gesture features for Indian Sign Language recognition. *In: 3rd International Conference on Trendz in Information Sciences and Computing (TISC), 2011, pages 3035*

[3] Singha J and Das K (2013). Indian sign language recognition using eigen value weighted euclidean distance based classification technique. *arXiv preprint arXiv:1303.0634*

[4] Bhuyan M, Kar M K and Neog D R (2011). Hand pose identi- fication from monocular image for sign language recognition. *In: 2011 IEEE International Conference on Signal and Image Processing Applications (ICSIPA), pages 378383*

[5] Van den Bergh M and Van Gool L (2011). Combining RGB and ToF cameras for real-time 3D hand gesture interaction. *In: IEEE Workshop on Applications of Computer Vision (WACV), 2011, pages 6672*

[6] Zafar Ahmed Ansari and Gaurav Harit (2016, February). Nearest neighbour classification of Indian sign language gestures using kinect camera. [Online]. Available:
https://link.springer.com/article/10.1007%2Fs12046-015-0405-3

[7] C231n *Convolutional Neural Networks for Visual Recognition* [Online]. Available:
http://cs231n.github.io/convolutional-networks/

[8] Gavin Weiguang Ding (2018, January 1). Drawing ConV net model using python. [Online]. Available:
https://github.com/gwding/draw_convnet

[9] Link to the dataset:
https://github.com/zafar142007/Gesture-Recognition-for-Indian-Sign-Language-using-Kinect/tree/master

[10] Code to draw the image of the CNN:
https://github.com/gwding/draw_convnet

Appendix

A Data parsing and Concatenation code

Majority of the code has already been explained in the report, however the complete code is now presented below.

Listing 4: Code for parsing and concatenating data

```
from __future__ import absolute_import
from __future__ import division
import numpy as np
import glob
import os
from numpy import genfromtxt
import csv
import cv2

#Defining variables to store data for training and testing
classes = ["Train", "Test"]
rgb_values = {}
depth_values = {}
rgb_labels = {}
depth_labels = {}
rgb_values["Train"] = []
rgb_values["Test"] = []
depth_values["Train"] = []
depth_values["Test"] = []
rgb_labels["Train"] = []
rgb_labels["Test"] = []
depth_labels["Train"] = []
depth_labels["Test"] = []

#Loop in order to take in the depth values of the images for both training and testing data
for clas in classes:
    filenames = os.listdir("../Project-Data/Depth/"+clas+"/")
    print filenames
    for fb in filenames:
        print fb
        for k in glob.glob("../Project-Data/Depth/"+clas+"/"+fb+"
```

```
            /*.txt"):
32              my_data = genfromtxt(k,delimiter = ' ')
33              depth_values[clas].append(my_data)
34              names = k.split('/')
35              fname = names[5].split('.')
36              labels = fname[0].split('-')
37              depth_labels[clas].append(labels[1:4])
38
39  #Loop in order to take in the RGB values of the images for both ↩
       training and testing data
40  for clas in classes:
41      filenames = os.listdir("../Project-Data/RGB/"+clas+"/")
42      print filenames
43      for fb in filenames:
44          print fb
45          for k in glob.glob("../Project-Data/RGB/"+clas+"/"+fb+"/*.↩
               png"):
46              image = cv2.imread(k)
47              rgb_values[clas].append(image)
48              names = k.split('/')
49              fname = names[5].split('.')
50              labels = fname[0].split('-')
51              rgb_labels[clas].append(labels[1:4])
52
53
54  #Checking to see if there is a syncronization error in order to ↩
       prevent mis-labelling of the data
55  val = set(rgb_labels)&set(depth_labels)
56  if(val != 0):
57      print "Data Synchronization Error"
58      exit()
59
60  labels = {}
61  data = {}
62  data["Train"] = []
63  data["Test"] = []
64  labels["Train"] = []
65  labels["Test"] = []
66
67  #Concatenating the RGB values and depth values to form the final ↩
       data
```

```
68 for clas in classes:
69     for i in xrange(len(rgb_values[clas])):
70         rgb = np.asarray(rgb_values[clas][i])
71         dpth = np.asarray(depth_values[clas][i])
72         dta = np.dstack(rgb,dpth)
73         data[clas].append(dta)
74 
75     labels[clas] = np.asarray(rgb_labels[clas])
76     labels[clas] = labels[clas][1]
77     labels[clas] = labels[clas] - 1
78 
79 data_pred = []
80 label_pred = []
81 
82 #Using three special cases in order to review the assurance of each↩
       image classification
83 filename1 = "../Project-Data/RGB/Train/user1/USER-2-60-1.png"
84 filename2 = "../Project-Data/Depth/Train/user1/USER-2-60-1.txt"
85 image = cv2.imread(filename1)
86 depth = genfromtxt(filename2,delimiter = ' ')
87 data1_value = np.dstack(image,depth)
88 data1_label = 60-1
89 data_pred.append(data1_value)
90 label_pred.append(data1_label)
91 
92 filename1 = "../Project-Data/RGB/Train/user1/USER-2-27-1.png"
93 filename2 = "../Project-Data/Depth/Train/user1/USER-2-27-1.txt"
94 image = cv2.imread(filename1)
95 depth = genfromtxt(filename2,delimiter = ' ')
96 data2_value = np.dstack(image,depth)
97 data2_label = 27-1
98 data_pred.append(data2_value)
99 label_pred.append(data2_label)
100 
101 filename1 = "../Project-Data/RGB/Train/user1/USER-2-7-1.png"
102 #Compare with 2-70-1
103 filename2 = "../Project-Data/Depth/Train/user1/USER-2-7-1.txt"
104 image = cv2.imread(filename1)
105 depth = genfromtxt(filename2,delimiter = ' ')
106 data3_value = np.dstack(image,depth)
107 data3_label = 7-1
```

```
data_pred.append(data3_value)
label_pred.append(data3_label)
```

B CNN training and testing code

This is the main part of the project, which involves all the network architecture, evaluation as well as prediction of certain specific images. We import data from the previous code in order to train and evaluate the model. This code also produces 'training_loss.txt' as well as 'prob.txt' for plotting in the future.

Listing 5: Code for training and testing CNN network

```
from __future__ import absolute_import
from __future__ import division
import numpy as np
import glob
import os
from numpy import genfromtxt
import csv
import cv2
#Importing data values from the first program
import data_inp
import tensorflow as tf

tf.logging.set_verbosity(tf.logging.INFO)

#Defining the CNN Network
def cnn_model(features, labels, mode):
    n_classes = 140
 # Input Layer
    input_layer = tf.reshape(features["x"], [-1,480,640,4])

  # Convolutional Layer #1
    conv1_layer = tf.layers.conv2d(
    inputs=input_layer,
    filters=32,
    kernel_size=[5, 5],
    padding="same",
    activation=tf.nn.relu)
```

```
28
29    # Convolutional Layer #2
30      conv2_layer = tf.layers.conv2d(
31      inputs=conv1_layer,
32      filters=32,
33      kernel_size=[5, 5],
34      padding="same",
35      activation=tf.nn.relu)
36
37    # Pooling Layer #1
38      pool1 = tf.layers.max_pooling2d(inputs=conv2_layer, pool_size↩
          =[2, 2], strides=2)
39
40      # Convolutional Layer #3
41      conv3_layer = tf.layers.conv2d(
42      inputs=pool1,
43      filters=64,
44      kernel_size=[5, 5],
45      padding="same",
46      activation=tf.nn.relu)
47
48      # Convolutional Layer #4
49      conv4_layer = tf.layers.conv2d(
50      inputs=conv3_layer,
51      filters=64,
52      kernel_size=[5, 5],
53      padding="same",
54      activation=tf.nn.relu)
55
56    # Pooling Layer #2
57      pool2 = tf.layers.max_pooling2d(inputs=conv4_layer, pool_size↩
          =[2, 2], strides=2)
58
59      pool2_flat = tf.reshape(pool2, [-1, 120 * 160 * 256])
60
61    # Dense Layer
62      dense = tf.layers.dense(inputs=pool2_flat, units=1024, ↩
          activation=tf.nn.relu)
63
64    # Add dropout operation; 0.6 probability that element will be ↩
       kept
```

```
65      dropout = tf.layers.dropout(
66          inputs=dense, rate=0.4, training=mode == tf.estimator.↩
               ModeKeys.TRAIN)
67
68    # Logits layer
69      logits = tf.layers.dense(inputs=dropout, units= n_classes)
70
71      predictions = {
72          "classes": tf.argmax(input=logits, axis=1),
73          "probabilities": tf.nn.softmax(logits, name="softmax_tensor↩
               ")
74      }
75      if mode == tf.estimator.ModeKeys.PREDICT:
76          return tf.estimator.EstimatorSpec(mode=mode, predictions=↩
               predictions["probabilities"])
77
78    # Calculate Loss
79      onehot_labels = tf.one_hot(indices=tf.cast(labels, tf.int32), ↩
           depth=n_classes)
80      loss = tf.losses.softmax_cross_entropy(
81          onehot_labels=onehot_labels, logits=logits)
82
83    # Configure the Training Op (for TRAIN mode)
84      if mode == tf.estimator.ModeKeys.TRAIN:
85          optimizer = tf.train.GradientDescentOptimizer(learning_rate↩
               =0.002)
86          train_op = optimizer.minimize(
87          loss=loss,
88          global_step=tf.train.get_global_step())
89          if global_step/500 == 0:
90              fb = open("loss_train.txt","w")
91              fb.write(str(global_step) + " " + str(round(tf.↩
                   reduce_mean(loss),3)) + "\n")
92          return tf.estimator.EstimatorSpec(mode=mode, loss=loss, ↩
               train_op=train_op)
93
94     # Configure for Evaluating Op (for EVAL mode)
95      if mode == tf.estimator.ModeKeys.EVAL:
96          eval_metric_ops = {
97          "accuracy": tf.metrics.accuracy(
98          labels=labels, predictions=predictions["classes"])}
```

```
 99          return tf.estimator.EstimatorSpec(
100          mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
101
102  def main():
103
104   #Setting up train_data
105      train_data = data_inp.data["Train"]
106      train_label = data_inp.labels["Train"]
107
108      #Setting up test_data
109      test_data = data_inp.data["Test"]
110      test_label = data_inp.labels["Test"]
111
112      #Setting up the estimator as the cnn network
113      classifier = tf.estimator.Estimator(
114          model_fn=cnn_model)
115
116    # Set up logging for predictions
117    # Log the values in the "Softmax" tensor with label "↩
        probabilities"
118      tensors_to_log = {"probabilities": "softmax_tensor"}
119      logging_hook = tf.train.LoggingTensorHook(
120          tensors=tensors_to_log, every_n_iter=50)
121
122    # Train the model
123      train_input_fn = tf.estimator.inputs.numpy_input_fn(
124          x={"x": train_data},
125          y=train_label,
126          batch_size=100,
127          num_epochs=None,
128          shuffle=True)
129      classifier.train(
130          input_fn=train_input_fn,
131          steps=20000,
132          hooks=[logging_hook])
133
134     #Test the model
135      test_input_fn = tf.estimator.inputs.numpy_input_fn(
136          x= {"x": test_data},
137          y= test_labels,
138          num_epochs=1,
```

```
        shuffle=False)
    eval_results = classifier.evaluate(input_fn=eval_input_fn)
    print(eval_results)

    #Find specific prob distribution for specific data
    spec_input_fn = tf.estimator.inputs.numpy_input_fn(
        x = {"x": data_inp.data_pred},
        y = data_inp.label_pred,
        num_epochs = 1,
        shuffle = False
        )
    spec_results = classifier.predict(input_fn=spec_input_fn)

    #print spec_results
    f  = open("prob.txt","w")
    f.write(round(spec_results['predictions'],4))


    #Find the prediction vales for training data in order to ↩
       perform future operations(i.e. in the assurance.py)
    pred_fn = tf.estimator.inputs.numpy_input_fn(
        x = {"x":test_data},
        y = test_label,
        batch_size = 1,
        num_epochs = 1,
        shuffle = False
        )
    pred_results = classifier.predict(input_fn=pred_fn)



if __name__ == "__main__":
    tf.app.run()
```

C Maximum Achievable Accuracy Calculation code

After achieving the predictions of each test data point from the model, we then proceed to calculate the maximum accuracy and the number of valid classi-

fiable images in the data set.We do so importing the derived results in the previous code.

Listing 6: Code for calculation of Maximum Acc. and no. of valid images

```
from __future__ import absolute_import
from __future__ import division
import numpy as np
import glob
import os
from numpy import genfromtxt
import csv
import cv2
#Importing probabilities data after parsing the image through the network
import cnn_train_test
import tensorflow as tf

avg = 1/140

#Stores the probabilities of the test data
k = cnn_train_test.pred_results['predictions']
k = np.asarray(k)

#A function to find the second highest element in the array
def getSecondHighest(a):
  hi = mid = lo = 0
  for i in range(0, len(a)):
    x = a[i]
    if ( x > hi):
      lo = mid
      mid = hi
      hi = x
    elif (x > mid):
      lo = mid
      mid = x
    else:
      lo = x
  return mid

#User defined quantities
alpha = 1.6
```

```
beta = 1.1

#Number of test cases
n_cases = k.shape[0]

#Number of classifiable images
n_valid = 0
#Number of images with assured classification
n_assured = 0

#Loop to calculate the above defined quantities by parsing through the test set
for i in xrange(n_cases):
    peak_val = np.amax(k[i])
    second_peak_val = getSecondHighest(k[i])
    ratio = (peak_val-avg)/(second_peak_val-avg)
    if (ratio > alpha):
        n_valid = n_valid+1
        n_assured = n_assured+1
    elif (ratio > beta):
        n_valid = n_valid+1

assured_accuracy = n_assured/n_valid

print "Maximum Assured Accuracy = " + str(round(assured_accuracy,6))

print n_valid
```

D Python Code for plotting

As mentioned above, the second code produces 'training_loss.txt' as well as 'prob.txt' as the results for plotting. Thus we use the below mentioned code to read through the text files and generate necessary plots.

Listing 7: Code for plotting: Pre-Processing

```
import numpy as np
from numpy import genfromtxt
```

```
import matplotlib.pyplot as plt

filename = 'prob.txt'
filename2 = 'loss_train.txt'

k = genfromtxt(filename,delimiter = ',')
t = genfromtxt(filename2,delimiter = ' ')

t = np.asarray(t)
k = np.asarray(k)
x = np.arange(0,140,1)
t = np.transpose(t)
#Defining the width of the bar
width = 1/1.5

#Plotting the probability distribution of the first image
fig, ax = plt.subplots(nrows=1, ncols=1)
ax.bar(x,k[0],width,color='blue')
ax.set_xlabel('Classes')
ax.set_ylabel('Probabilities')
ax.legend(['Probability Distribution'])
fig.savefig('plots/prob_1.png')
plt.close(fig)

#Plotting the probability distribution of the second image
fig, ax = plt.subplots(nrows=1, ncols=1)
ax.bar(x,k[1],width,color='red')
ax.set_xlabel('Classes')
ax.set_ylabel('Probabilities')
ax.legend(['Probability Distribution'])
fig.savefig('plots/prob_2.png')
plt.close(fig)

#Plotting the probability distribution of the third image
fig, ax = plt.subplots(nrows=1, ncols=1)
ax.bar(x, k[2],width,color='green')
ax.set_xlabel('Classes')
ax.set_ylabel('Probabilities')
ax.legend(['Probability Distribution'])
fig.savefig('plots/prob_3.png')
plt.close(fig)
```

```
44
45 #Plotting the training loss value against the number of epochs
46 fig, ax = plt.subplots(nrows=1, ncols=1)
47 ax.plot(t[0], t[1],'k')
48 ax.set_xlabel('num_epochs')
49 ax.set_ylabel('loss value')
50 ax.legend(['Training loss'])
51 fig.savefig('plots/training_loss.png')
52 plt.close(fig)
```