IBM CLOUD BASED NAAN MUDHALVAN PROJECT

TITLE : THE GESTURES RECOGNITION FOR SIGN LANGUAGE(PHASE-5) SUBMISSION PART

TEAM:7

1)K.RAJESH(T.L) - 422421106023

2)A.ALWIN XAVIER - 422421106003

3)T.DHIVAGAR - 422421106009

4)E.SURYA PRAKASH - 422421106029

Abstract

This paper focuses on experimenting with different segmentation approaches and unsupervised learning algorithms to create an accurate sign language recognition model. To more easily approach the problem and obtain reasonable results, we experimented with just up to 10 different classes/letters in the our self-made dataset instead of all 26 possible letters. We collected 12000 RGB images and their corresponding depth data using a Microsoft Kinect. Up to half of the data was fed into the autoencoder to extract features while the other half was used for testing. We achieved a classification accuracy of 98% on a randomly selected set of test data using our trained model. In addition to the work we did on static images, we also created a live demo version of the project which can be run at a little less than 2 seconds per frame to classify signed hand gestures from any person.

Future Distribution Permission

1. Introduction

The problem we are investigating is sign language recognition through unsupervised feature learning. Being able to recognize sign lan- guage is an interesting computer vision problem while , simultaneously being extremely useful for deaf people to interact with people who don't know how to understand American Sign Lan- guage (ASL).

showing the different hand gestures of the ASL alphabet that we wished to classify. The next stepwas to segment out only the hand region from each image and then use this data for unsuper- vised feature learning using an autoencoder, fol- lowed by training a softmax classifier for mak- ing a decision about which letter is being dis- played. Many different segmentation approacheswere tried until we discovered that the skin colorand depth segmentation techniques worked mostconsistently.

2. Methodology

2.1. Problem Statement

2.1.1 Background

The CVPR gesture workshop from 2011 provides a great information on modern gesture recognition models as well as how to incorporate different learning algorithms. There is some past work[1] related to our project that we initially looked at such as segmentation-robust modeling for sign language recognition [3] and sign language and human activity recognition [1], but we ended up using mostly our own approach to sign language recognition. Inspiration for our learning model was drawn from the MNIST handwritten digits recognition problem[2] which also used a similar unsupervised feature learning and classification approach [2]. We also investigated the use of convolutional neural networks for feature learning based on the visual document analysis paper

Dataset

The data used for training and testing came from our own self-made dataset. 1200 samples of each of the 10 signed letters (a, b, c, d, e, f, g, h, i, l) were collected using the Kinect including the corresponding depth data. Each sample consists of a person signing the corresponding letter while facing directly at the Kinect camera. This dataset consists of 6000 images used for training and another 6000 images used for testing. The following is a visualization of some of the unprocessed raw images in the dataset:

The depth data that was collected along with the RGB data overlayed perfectly on top of the original image and was helpful during the segmentation section of the model. An example is shown below:

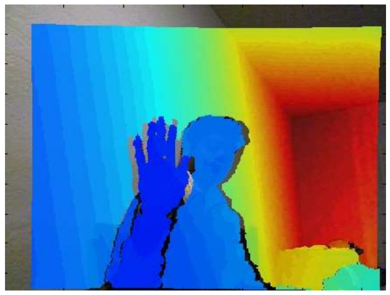


Figure : Depth data overlayed on top of RGB data

However, this approach did not always work since there would be instances where the largest ”object” with distinguishable edges was not the hand (or there was a busy background which clut- tered the output) and therefore segment poorly. We decided to move on to a simpler, but perhaps more accurate method of segmentation using just color.

2.2. Technical Approach

2.2.1 Segmentation Methods

We tried implementing many different versions of quick image segmentation, in particular, hand segmentation.

Edge Segmentation

One of the methods we tried was using a Canny edge detector to find relevant ”objects” in the field of view of the camera. The edges were then dilated, and then all remaining holes in the mask were filled to create a solid, continuous mask. Once this was done, only the largest areas were taken in order to remove all the background clutter objects. This approach makes the simplifying assumption that the biggest objects seen in segmentation are typically of the most interest as well.

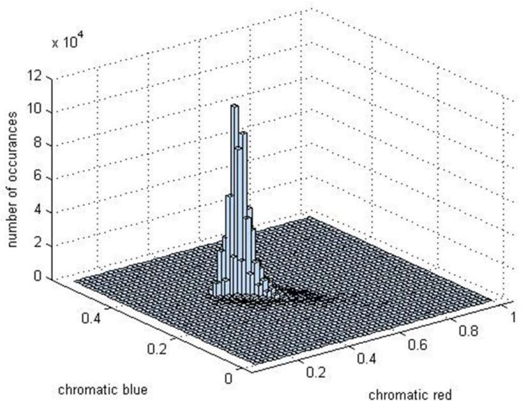
Skin Color Segmentation

We tried out the two approaches for skin segmentation using only color information. The firstapproach involved modeling the skin color bya 2D Gaussian curve and then using this fitted Gaussian to estimate the likelihood of a given color pixel being skin. First, we collected skin patches from 40 random images from the inter- net. Each skin patch was a contiguous brectangular skin area. Skin patches were collected from people belonging to different ethnicities so that our model is able to correctly predict skin areas for a wide variation of skin color. The colors were then normalized as follows :

$$r = \frac{R}{R+G+B}, b = \frac{B}{R+G+B}.$$

The g component is ignored as it is linearly dependent on the other two. The mean and covariance matrix of the 2D Gaussian (with r, b as the axes) is estimated as follows :

$$\text{Mean } m = E[x], \text{ where } x = [r, b]^T$$
$$\text{Covariance } C = E[(x - m)(x - m)^T].$$



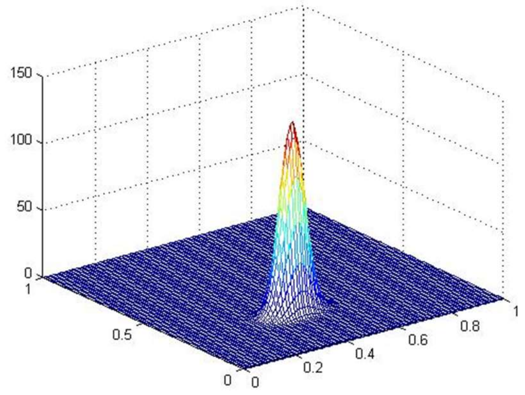


Figure : (Top) Histogram of color distribution for skin patches, (Bottom) Gaussian model fit

With this Gaussian fitted skin color model, the likelihood of skin for any pixel of a given test image can be obtained. If the pixel, has a chromatic pair value of (r, b), then the likelihood of skin for this pixel is given by:

$$Likelihood = e^{[-0.5(x-m)^T C^{-1}(x-m)]}, \text{ where } x = [r, b]^T.$$

Finally, we thresholded the likelihood to classify it as skin or non-skin. However, this approach did not give significantly good or consistent results and failed to detect dimly illuminated parts of skin. These poor results did not meet the quality standard of hand segmentation we needed to get a consistent feature extraction from the learning layer of the model.

The second approach which we used is motivated by the paper [5], in which the authors first transform the image from the RGB space to the YIQ and YUQ color spaces. The authors then compute the parameter

$$\Theta = tan^{-1}(V/U)$$

and combine it with the parameter $\boldsymbol{I}$ to define the region to which skin pixels belong. Specifically, the authors called all pixels with $30 < \boldsymbol{I} < 100$ and $105^o < \Theta < 150^o$ as skin. For our experiments, we tweaked these thresholds a bit, and found that the results were significantly better than our Gaussian model in the previous approach. This might have been because of two reasons:

1. The Gaussian model was trained using data samples of insufficient variety and hence was inadequate to correctly detect skin pixels of darker shades

2. Fitting the model in the RGB space performs poorly as RGB doesnt capture the hue and saturation information of each pixel separately.

Having detected the skin regions quite accurately, we then further filtered out the hand region and eliminated the face/other background pixels that might have been detected by using the corresponding collected depth data. We assumed that in any given frame, the hand was the object of interest and therefore the closest skin-colored object in the camera's view. We then created a secondary dataset out of this segmentation model which consisted of hand gestures for ten letters of the ASL alphabet. Each image was cropped and resized to be a square 32x32 bounded area.

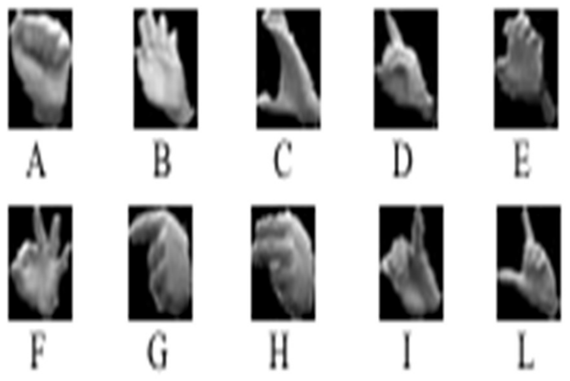


Figure : Visualization of processed database images

Further Segmentation Work

We explored further segmentation methods, but eventually deemed them of less importance than the original skin-color hand segmentation to get an accurate ASL recognition system. One of the other approaches was attempting to properly segment pointed fingers within the detected hand. In order to segment out the fingers, we used convex hull detections to find the possible "fingers" after the hand has already been segmented out. The fingers will ideally be oriented along the direction from the convex hull point to the centroid of the hand as seen in Figure .

However, we never ended up using the detected fingers as part of the classification approach be- cause of the slightly unpredictable detections of convex hulls in a hand image.

Feature Learning and Classification

The extracted data of hand images were fed into an autoencoder in order to perform the actual recognition training part of the project. This stage implements an unsupervised learning algorithm. We feed all the data samples into the sparse autoencoder. The input data from the segmentation block are images of size 32x32 pixels. A sparse autoencoder is chosen initially with an input layer with 32x32 nodes and one hidden layer of 100 nodes. We used L-BFGS to optimize the cost function. This was run for about 400 iterations to obtain estimates of the weights. Now the autoencoder has learnt a set of features similar to edges. A visualization of the learned features can be seen below:

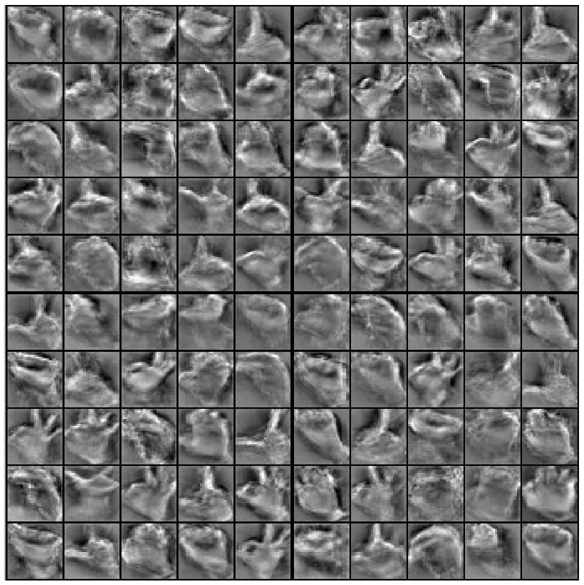


Figure : Visualization of sparse autoencoder features

The next step is to classify the 10 different letters based on the features learnt by the autoencoder training. The output of the hidden layer of the autoencoder is fed into a softmax classifier to now classify the data into 10 categories. The softmax classifier again learns using the L-BFGS optimization function. This algorithm converges after about 40 iterations. We tested the system accuracy by using the remaining 600 images per letter (for a total of 6000 images) as our test set.

An overall view of the system's block diagram can be seen here:

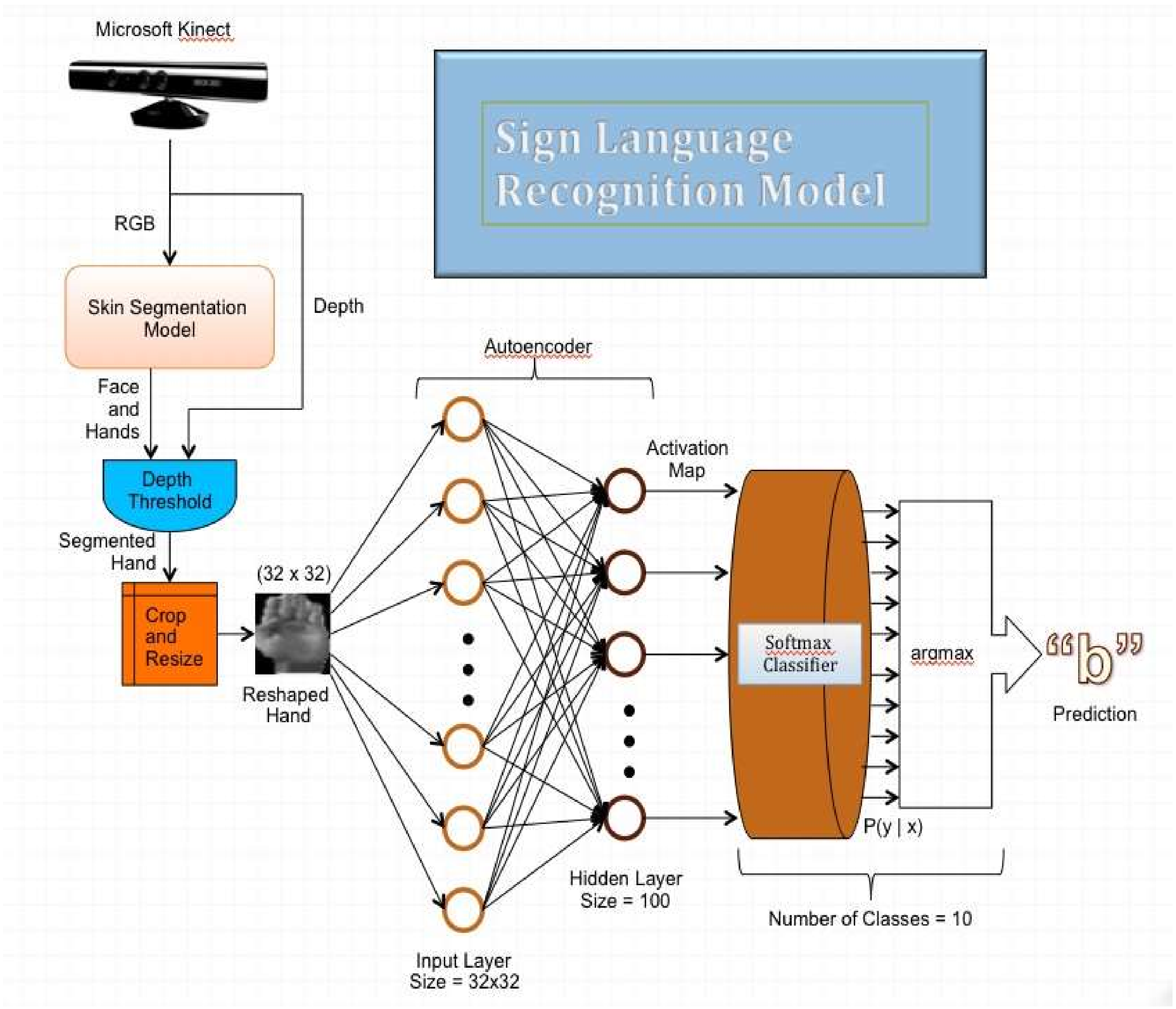


Figure : Block diagram summarizing our approach for sign language recognition

3. Experimental Results and Discussions

In the previous sections, we have mentioned details of our implementation of the hand segmentation, unsupervised feature learning and classification sub-blocks. In this section, we report the performance of our system through tables and figures. Our primary evaluation metric is based on classification accuracy. Given an unsegmented image of a person signing a letter, we want to see what the accuracy of our model is in classifying/predicting the signed letter in the image. Achieving a classification accuracy around 98% similar to many MNIST digit recognition scores is desired.

As a preliminary diagnostic, we plotted a learning curve showing the training error and the test error as a function of the size of the training set. The following plot shows the learning curve we obtained:

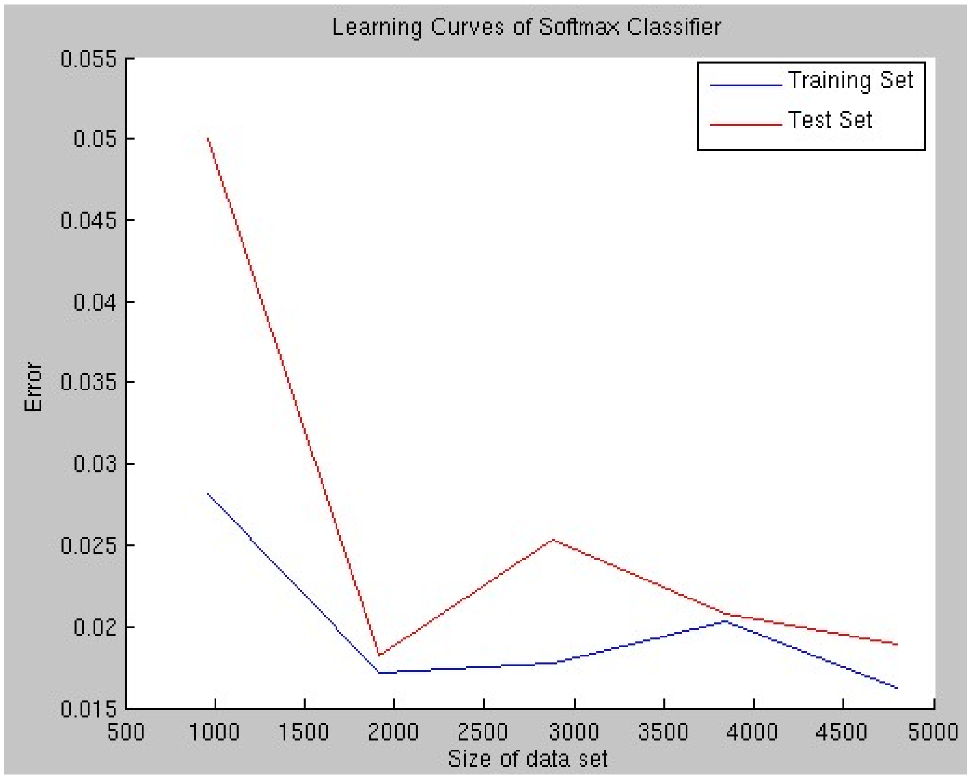
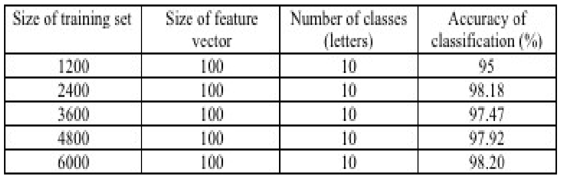


Figure : Learning curve of sign language recognition system

In our milestone report, we had used 50 hidden units for our autoencoder. Analyzing our learning curve, we observe that the training error and the test error are close to each other (except for one aberration at training set size 3000), and even the training error is more than 1.5%, which is somewhat high. Hence suspecting that we might be in the high bias region, we decided to increase the size of our features by increasing the number of hidden units of our autoencoder to 100. Our final classification accuracy on the test set achieved using this 100 length feature vector was 98.2%.The following table summarizes all our implementation details and reports the accuracy obtained :



Size of training set	Size of feature vector	Number of classes (letters)	Accuracy of classification (%)
1200	100	10	95
2400	100	10	98.18
3600	100	10	97.47
4800	100	10	97.92
6000	100	10	98.20

Table 1: Classification Accuracy Evaluation Results on Test Set

The accuracy results that we obtained are comparable to the accuracy of digit recognition on the MNIST dataset and therefore we believe that our approach works relatively well.

As a finishing step to our project, we have successfully created a real time implementation of our entire system, so that hand gestures made in front of the Kinect connected to our computer directly displayed the image captured by the kinect, the segmented hand gesture and the output of our classifier, which is one of the ten letters in our dataset. The evaluation process takes less than 2 seconds per frame. The following figures show screenshots of our real-time implementation and the results obtained. In each screenshot, the original image is shown with the result of the segmentation and the predicted result to the right of it.

4. Conclusion and Future Work

In this project, we have implemented an automatic sign language gesture recognition system in real-time, using tools learnt in computer vision and machine learning with python. We learned about how sometimes basic approaches work better than complicated approaches.

```
from skimage import transform
from skimage import data
import matplotlib.pyplot as plt
import os
import numpy as np
from skimage.color import rgb2gray
import random
import tensorflow as tf
```

In [2]:

```
def load_data(data_directory):
    directories = [d for d in os.listdir
(data_directory)
                   if os.path.isdir(os.p
ath.join(data_directory, d))]
    labels = []
    images = []
    for d in directories:
        label_directory = os.path.join(d
ata_directory, d)
        file_names = [os.path.join(label
_directory, f) for f in os.listdir(label
_directory)]
        for f in file_names:
            images.append(data.imread(f)
)
            labels.append(ord(d))
    return images, labels

ROOT_PATH="../input/project"
train_data_directory=os.path.join(ROOT_P
ATH, "train")

images, labels=load_data(train_data_dire
ctory)
```

In [3]:

```
images_array = np.array(images)
labels_array = np.array(labels)

# Print the number of `images`'s elements
print("Total number of images:",images_a
rray.size)
# Count the number of labels
print("Total No of classes:",len(set(lab
els_array)))
print("Label Array: ",[chr(X) for X in s
et(labels)])
```

```
Total number of images: 4852
Total No of classes: 24
Label Array:  ['A', 'B', 'C', 'D', 'E'
, 'F', 'G', 'H', 'I', 'K', 'L', 'M', '
N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U',
 'V', 'W', 'X', 'Y']
```

In [4]:

```
# Determine the (random) indexes of the ima
ges that you want to see
hand_signs = [12,45,65,35]

# Fill out the subplots with the random ima
ges that you defined
for i in range(len(hand_signs)):
    plt.subplot(1, 4, i+1)
    plt.axis('off')
    plt.imshow(images[hand_signs[i]])
    plt.subplots_adjust(wspace=0.5)

plt.show()
```

In [5]:

```
# Determine the (random) indexes of the ima
ges
hand_signs = [300, 1250, 2650, 3000]

# Fill out the subplots with the random ima
ges and add shape, min and max values
for i in range(len(hand_signs)):
    plt.subplot(1, 4, i+1)
    plt.axis('off')
    plt.imshow(images[hand_signs[i]])
    plt.subplots_adjust(wspace=0.5)
    plt.show()
    print("shape: {0}, min: {1}, max: {2}"
.format(images[hand_signs[i]].shape,

        images[hand_signs[i]].min(),

        images[hand_signs[i]].max()))
```

```
shape: (1088, 1920, 3), min: 19, max: 18
7
```

```
shape: (480, 640, 3), min: 0, max: 217
```

```
shape: (480, 640, 3), min: 0, max: 218
```

```
shape: (1088, 1920, 3), min: 0, max: 2
24
```

In [6]:

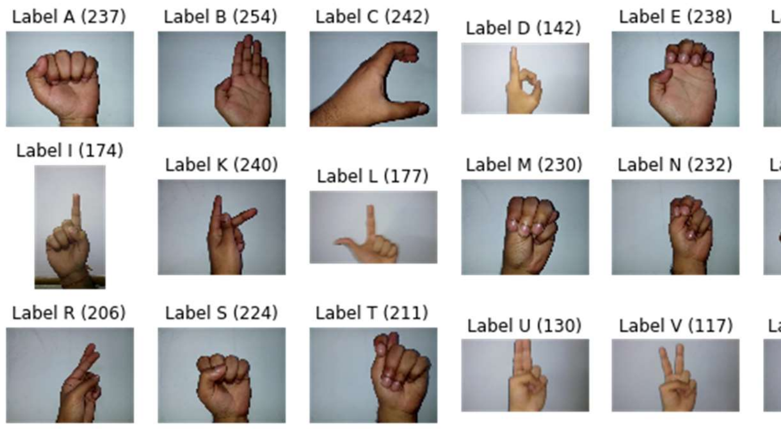
```
# Get the unique labels
unique_labels = set(labels)

# Initialize the figure
plt.figure(figsize=(15, 15))

# Set a counter
i = 1

# For each unique label,
for label in unique_labels:
    # You pick the first image for each l
abel
    image = images[labels.index(label)]
    # Define 64 subplots
    plt.subplot(8, 8, i)
    # Don't include axes
    plt.axis('off')
    # Add a title to each subplot
    plt.title("Label {0} ({1})".format(c
hr(label), labels.count(label)))
    # Add 1 to the counter
    i += 1
    # And you plot this first image
    plt.imshow(image)

# Show the plot
plt.show()
```



Feature Extraction

In [7]:

```
# Resize images
images32 = [transform.resize(image, (28, 2
8,3)) for image in images]
images32 = np.array(images32)
```

```
/opt/conda/lib/python3.6/site-packages/s
kimage/transform/_warps.py:84: UserWarni
ng: The default mode, 'constant', will b
e changed to 'reflect' in skimage 0.15.
  warn("The default mode, 'constant', wi
ll be changed to 'reflect' in "
```

Image Conversion to Grayscale

In [8]:

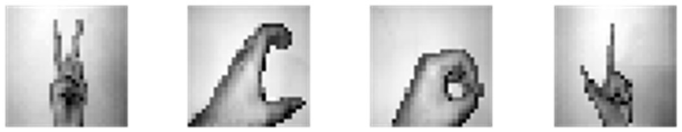
```
images32 = rgb2gray(np.array(images32))
```

In [9]:

```
for i in range(len(hand_signs)):
    plt.subplot(1, 4, i+1)
    plt.axis('off')
    plt.imshow(images32[hand_signs[i]], cm
ap="gray")
    plt.subplots_adjust(wspace=0.5)

plt.show()

print(images32.shape)
```



```
(4852, 28, 28)
```

Model

In [10]:

```
x = tf.placeholder(dtype = tf.float32, sha
pe = [None, 28, 28])
y = tf.placeholder(dtype = tf.int32, sha
pe = [None])
images_flat = tf.contrib.layers.flatten(
x)
logits = tf.contrib.layers.fully_connect
ed(images_flat, 100, tf.nn.relu)
loss = tf.reduce_mean(tf.nn.sparse_softm
ax_cross_entropy_with_logits(labels = y,
 logits = logits))
train_op = tf.train.AdamOptimizer(learni
ng_rate=0.001).minimize(loss)
correct_pred = tf.argmax(logits, 1)
accuracy = tf.reduce_mean(tf.cast(correc
t_pred, tf.float32))

print("images_flat: ", images_flat)
print("logits: ", logits)
print("loss: ", loss)
print("predicted_labels: ", correct_pred
)
```

```
images_flat:  Tensor("Flatten/flatten/
Reshape:0", shape=(?, 784), dtype=floa
t32)
logits:  Tensor("fully_connected/Relu:
0", shape=(?, 100), dtype=float32)
loss:  Tensor("Mean:0", shape=(), dtyp
e=float32)
predicted_labels:  Tensor("ArgMax:0",
shape=(?,), dtype=int64)
```

In [11]:

```
sess = tf.Session()

sess.run(tf.global_variables_initializer
())

for i in range(201):
        print('EPOCH', i)
        _, accuracy_val = sess.run([trai
n_op, accuracy], feed_dict={x: images32,
 y: labels})
        if i % 10 == 0:
            print("Loss: ", loss)
        print('DONE WITH EPOCH')
```

```
EPOCH 0
Loss:  Tensor("Mean:0", shape=(), dtyp
e=float32)
DONE WITH EPOCH
EPOCH 1
DONE WITH EPOCH
EPOCH 2
DONE WITH EPOCH
EPOCH 3
DONE WITH EPOCH
EPOCH 4
DONE WITH EPOCH
EPOCH 5
DONE WITH EPOCH
EPOCH 6
DONE WITH EPOCH
EPOCH 7
DONE WITH EPOCH
EPOCH 8
DONE WITH EPOCH
EPOCH 9
DONE WITH EPOCH
EPOCH 10
Loss:  Tensor("Mean:0", shape=(), dtype=
float32)
DONE WITH EPOCH
EPOCH 11
DONE WITH EPOCH
EPOCH 12
DONE WITH EPOCH
EPOCH 13
DONE WITH EPOCH
EPOCH 14
DONE WITH EPOCH
EPOCH 15
DONE WITH EPOCH
EPOCH 16
DONE WITH EPOCH
EPOCH 17
DONE WITH EPOCH
EPOCH 18
DONE WITH EPOCH
EPOCH 19
DONE WITH EPOCH
EPOCH 20
Loss:  Tensor("Mean:0", shape=(), dtype=
float32)
DONE WITH EPOCH
EPOCH 21
DONE WITH EPOCH
EPOCH 22
DONE WITH EPOCH
EPOCH 23
DONE WITH EPOCH
EPOCH 24
DONE WITH EPOCH
EPOCH 25
DONE WITH EPOCH
EPOCH 26
```

```
DONE WITH EPOCH
EPOCH 27
DONE WITH EPOCH
EPOCH 28
DONE WITH EPOCH
EPOCH 29
DONE WITH EPOCH
EPOCH 30
Loss:  Tensor("Mean:0", shape=(), dtyp
e=float32)
DONE WITH EPOCH
EPOCH 31
DONE WITH EPOCH
EPOCH 32
DONE WITH EPOCH
EPOCH 33
DONE WITH EPOCH
EPOCH 34
DONE WITH EPOCH
EPOCH 35
DONE WITH EPOCH
EPOCH 36
DONE WITH EPOCH
EPOCH 37
DONE WITH EPOCH
EPOCH 38
DONE WITH EPOCH
EPOCH 39
DONE WITH EPOCH
EPOCH 40
Loss:  Tensor("Mean:0", shape=(), dtyp
e=float32)
DONE WITH EPOCH
EPOCH 41
DONE WITH EPOCH
EPOCH 42
DONE WITH EPOCH
EPOCH 43
DONE WITH EPOCH
EPOCH 44
DONE WITH EPOCH
EPOCH 45
DONE WITH EPOCH
EPOCH 46
DONE WITH EPOCH
EPOCH 47
DONE WITH EPOCH
EPOCH 48
DONE WITH EPOCH
EPOCH 49
DONE WITH EPOCH
EPOCH 50
Loss:  Tensor("Mean:0", shape=(), dtype=
float32)
DONE WITH EPOCH
EPOCH 51
DONE WITH EPOCH
EPOCH 52
DONE WITH EPOCH
EPOCH 53
DONE WITH EPOCH
EPOCH 54
DONE WITH EPOCH
EPOCH 55
DONE WITH EPOCH
EPOCH 56
DONE WITH EPOCH
EPOCH 57
DONE WITH EPOCH
EPOCH 58
DONE WITH EPOCH
EPOCH 59
DONE WITH EPOCH
EPOCH 60
Loss:  Tensor("Mean:0", shape=(), dtype=
float32)
DONE WITH EPOCH
EPOCH 61
DONE WITH EPOCH
EPOCH 62
DONE WITH EPOCH
EPOCH 63
DONE WITH EPOCH
EPOCH 64
DONE WITH EPOCH
EPOCH 65
DONE WITH EPOCH
EPOCH 66
DONE WITH EPOCH
EPOCH 67
DONE WITH EPOCH
EPOCH 68
DONE WITH EPOCH
EPOCH 69
DONE WITH EPOCH
EPOCH 70
Loss:  Tensor("Mean:0", shape=(), dtype=
float32)
DONE WITH EPOCH
EPOCH 71
DONE WITH EPOCH
EPOCH 72
```

```
DONE WITH EPOCH
EPOCH 73
DONE WITH EPOCH
EPOCH 74
DONE WITH EPOCH
EPOCH 75
DONE WITH EPOCH
EPOCH 76
DONE WITH EPOCH
EPOCH 77
DONE WITH EPOCH
EPOCH 78
DONE WITH EPOCH
EPOCH 79
DONE WITH EPOCH
EPOCH 80
Loss:  Tensor("Mean:0", shape=(), dtyp
e=float32)
DONE WITH EPOCH
EPOCH 81
DONE WITH EPOCH
EPOCH 82
DONE WITH EPOCH
EPOCH 83
DONE WITH EPOCH
EPOCH 84
DONE WITH EPOCH
EPOCH 85
DONE WITH EPOCH
EPOCH 86
DONE WITH EPOCH
EPOCH 87
DONE WITH EPOCH
EPOCH 88
DONE WITH EPOCH
EPOCH 89
DONE WITH EPOCH
EPOCH 90
Loss:  Tensor("Mean:0", shape=(), dtyp
e=float32)
DONE WITH EPOCH
EPOCH 91
DONE WITH EPOCH
EPOCH 92
DONE WITH EPOCH
EPOCH 93
DONE WITH EPOCH
EPOCH 94
DONE WITH EPOCH
EPOCH 95
DONE WITH EPOCH
EPOCH 96
DONE WITH EPOCH
EPOCH 97
DONE WITH EPOCH
EPOCH 98
DONE WITH EPOCH
EPOCH 99
DONE WITH EPOCH
EPOCH 100
Loss:  Tensor("Mean:0", shape=(), dtype=
float32)
DONE WITH EPOCH
EPOCH 101
DONE WITH EPOCH
EPOCH 102
DONE WITH EPOCH
EPOCH 103
DONE WITH EPOCH
EPOCH 104
DONE WITH EPOCH
EPOCH 105
DONE WITH EPOCH
EPOCH 106
DONE WITH EPOCH
EPOCH 107
DONE WITH EPOCH
EPOCH 108
DONE WITH EPOCH
EPOCH 109
DONE WITH EPOCH
EPOCH 110
Loss:  Tensor("Mean:0", shape=(), dtype=
float32)
DONE WITH EPOCH
EPOCH 111
DONE WITH EPOCH
EPOCH 112
DONE WITH EPOCH
EPOCH 113
DONE WITH EPOCH
EPOCH 114
DONE WITH EPOCH
EPOCH 115
DONE WITH EPOCH
EPOCH 116
DONE WITH EPOCH
EPOCH 117
DONE WITH EPOCH
EPOCH 118
DONE WITH EPOCH
EPOCH 119
```

```
DONE WITH EPOCH
EPOCH 120
Loss:  Tensor("Mean:0", shape=(), dtyp
e=float32)
DONE WITH EPOCH
EPOCH 121
DONE WITH EPOCH
EPOCH 122
DONE WITH EPOCH
EPOCH 123
DONE WITH EPOCH
EPOCH 124
DONE WITH EPOCH
EPOCH 125
DONE WITH EPOCH
EPOCH 126
DONE WITH EPOCH
EPOCH 127
DONE WITH EPOCH
EPOCH 128
DONE WITH EPOCH
EPOCH 129
DONE WITH EPOCH
EPOCH 130
Loss:  Tensor("Mean:0", shape=(), dtyp
e=float32)
DONE WITH EPOCH
EPOCH 131
DONE WITH EPOCH
EPOCH 132
DONE WITH EPOCH
EPOCH 133
DONE WITH EPOCH
EPOCH 134
DONE WITH EPOCH
EPOCH 135
DONE WITH EPOCH
EPOCH 136
DONE WITH EPOCH
EPOCH 137
DONE WITH EPOCH
EPOCH 138
DONE WITH EPOCH
EPOCH 139
DONE WITH EPOCH
EPOCH 140
Loss:  Tensor("Mean:0", shape=(), dtyp
e=float32)
DONE WITH EPOCH
EPOCH 141
DONE WITH EPOCH
EPOCH 142
DONE WITH EPOCH
EPOCH 143
DONE WITH EPOCH
EPOCH 144
DONE WITH EPOCH
EPOCH 145
DONE WITH EPOCH
EPOCH 146
DONE WITH EPOCH
EPOCH 147
DONE WITH EPOCH
EPOCH 148
DONE WITH EPOCH
EPOCH 149
DONE WITH EPOCH
EPOCH 150
Loss:  Tensor("Mean:0", shape=(), dtype=
float32)
DONE WITH EPOCH
EPOCH 151
DONE WITH EPOCH
EPOCH 152
DONE WITH EPOCH
EPOCH 153
DONE WITH EPOCH
EPOCH 154
DONE WITH EPOCH
EPOCH 155
DONE WITH EPOCH
EPOCH 156
DONE WITH EPOCH
EPOCH 157
DONE WITH EPOCH
EPOCH 158
DONE WITH EPOCH
EPOCH 159
DONE WITH EPOCH
EPOCH 160
Loss:  Tensor("Mean:0", shape=(), dtype=
float32)
DONE WITH EPOCH
EPOCH 161
DONE WITH EPOCH
EPOCH 162
DONE WITH EPOCH
EPOCH 163
DONE WITH EPOCH
EPOCH 164
DONE WITH EPOCH
EPOCH 165
```

```
DONE WITH EPOCH
EPOCH 166
DONE WITH EPOCH
EPOCH 167
DONE WITH EPOCH
EPOCH 168
DONE WITH EPOCH
EPOCH 169
DONE WITH EPOCH
EPOCH 170
Loss:  Tensor("Mean:0", shape=(), dtyp
e=float32)
DONE WITH EPOCH
EPOCH 171
DONE WITH EPOCH
EPOCH 172
DONE WITH EPOCH
EPOCH 173
DONE WITH EPOCH
EPOCH 174
DONE WITH EPOCH
EPOCH 175
DONE WITH EPOCH
EPOCH 176
DONE WITH EPOCH
EPOCH 177
DONE WITH EPOCH
EPOCH 178
DONE WITH EPOCH
EPOCH 179
DONE WITH EPOCH
EPOCH 180
Loss:  Tensor("Mean:0", shape=(), dtyp
e=float32)
DONE WITH EPOCH
EPOCH 181
DONE WITH EPOCH
EPOCH 182
DONE WITH EPOCH
EPOCH 183
DONE WITH EPOCH
EPOCH 184
DONE WITH EPOCH
EPOCH 185
DONE WITH EPOCH
EPOCH 186
DONE WITH EPOCH
EPOCH 187
DONE WITH EPOCH
EPOCH 188
DONE WITH EPOCH
EPOCH 189
DONE WITH EPOCH
EPOCH 190
Loss:  Tensor("Mean:0", shape=(), dtype=
float32)
DONE WITH EPOCH
EPOCH 191
DONE WITH EPOCH
EPOCH 192
DONE WITH EPOCH
EPOCH 193
DONE WITH EPOCH
EPOCH 194
DONE WITH EPOCH
EPOCH 195
DONE WITH EPOCH
EPOCH 196
DONE WITH EPOCH
EPOCH 197
DONE WITH EPOCH
EPOCH 198
DONE WITH EPOCH
EPOCH 199
DONE WITH EPOCH
EPOCH 200
Loss:  Tensor("Mean:0", shape=(), dtype=
float32)
DONE WITH EPOCH
```

Evaluation

In [12]:

```
# Pick 10 random images
sample_indexes = random.sample(range(len(i
mages32)), 10)
sample_images = [images32[i] for i in samp
le_indexes]
sample_labels = [labels[i] for i in sample
_indexes]

# Run the "predicted_labels" op.
predicted = sess.run([correct_pred], feed_
dict={x: sample_images})[0]

# Print the real and predicted labels
print(sample_labels)
print(predicted)
```

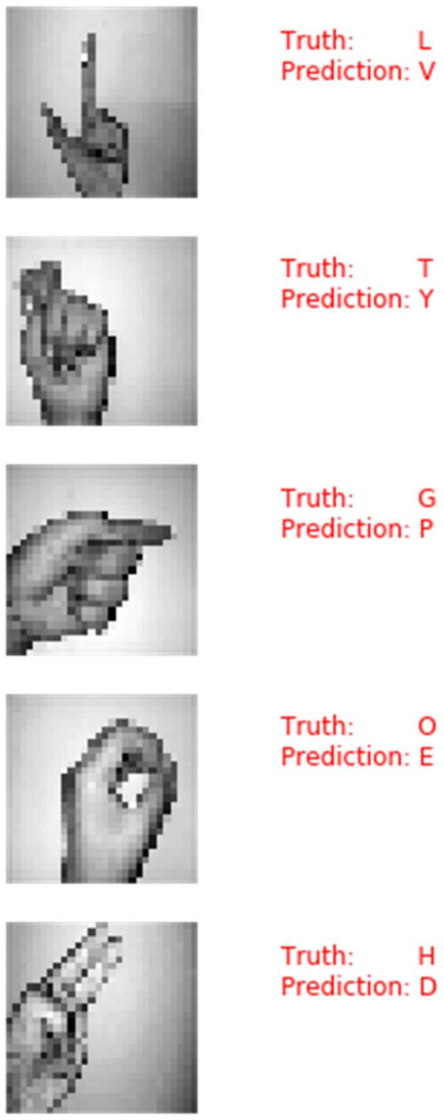
```
[76, 86, 84, 73, 71, 79, 79, 80, 72, 65]
[86 86 89 73 80 65 69 80 68 65]
```

In [13]:

```
# Display the predictions and the ground tr
uth visually.
```

```
fig = plt.figure(figsize=(10, 10))
for i in range(len(sample_images)):
    truth = sample_labels[i]
    prediction = predicted[i]
    plt.subplot(5, 2,1+i)
    plt.axis('off')
    color='green' if truth == prediction
 else 'red'
    plt.text(40, 10, "Truth:        {0}\
nPrediction: {1}".format(chr(truth), chr
(prediction)),
             fontsize=12, color=color)
    plt.imshow(sample_images[i],cmap='gr
ay')

plt.show()
```



```
   In [14]:
sess.close()
```

THANK

YOU !...