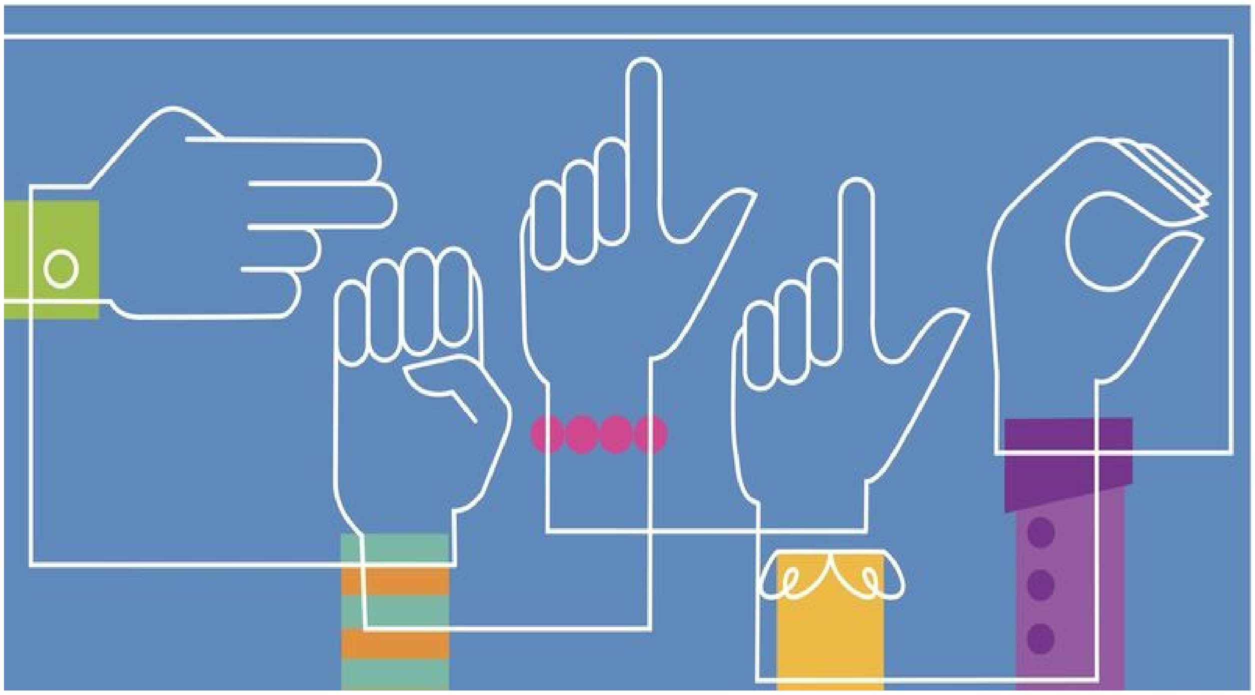


Poster Credit: https://www.virtualreality-news.net

Sign Language Translation Tool using Computer Vision and Deep Learning

30.11.2019

Borneel Bikash Phukan
Roll no. 1728207
B. Tech. (Computer Science & System Engineering)
Kalinga Institute of Industrial Technology
Bhubaneswar, Odisha

Table of Content

Introduction

Out of the entire 7.7 billion people of the world as of April 2019, it is estimated that 466 million people in the world are deprived of the ability to hear. It has been found that people who are deaf are also prone to being mute throughout their lives, although there are certain exceptions. Deaf people often can lip read but in order to communicate they do it through Sign Languages. Hand Gesture dependent Sign Language is a special type of conversation tool that is well understood by the deaf community but it is not well known amongst the general audience. This creates a huge communication gap between the general people and the deaf community all across the globe. This is the same reason why any deaf individual is treated differently than a normal able-bodied individual. So in order to break this stereotype, it is high time we create software which enables the proper translation of the Sign Language so that it can be understood by the people with no such disability. This project demonstrates one such technology made using Deep Convolutional Neural Network and Gesture Recognition.

Scope

This project covers one such domain of AI using Deep Learning Technologies. This project shall help a computer to analyze the hand gesture of a human being and determine the alphabet or symbol being represented using Deep Neural Networks. The “Sign Language Translation Tool” uses Convolutional Neural Network model (a Deep Learning Algorithm) to collect huge amount of data, split it into training, validation and test sets, train itself using the training dataset pool, calculate the weights, and apply the calculated weights on a validation set before moving on to the final test set.

Prerequisite Knowledge

In order to ensure proper understanding of this SRS, it is recommended that the reader has a basic idea on Machine Learning and Deep Learning Algorithms and a few hands-on experiences with tools like OpenCV, Keras, and Tensorflow.

P.S: This project has been made using JetBrains PyCharm Community Edition 2019.

Basic Knowledge Requirements

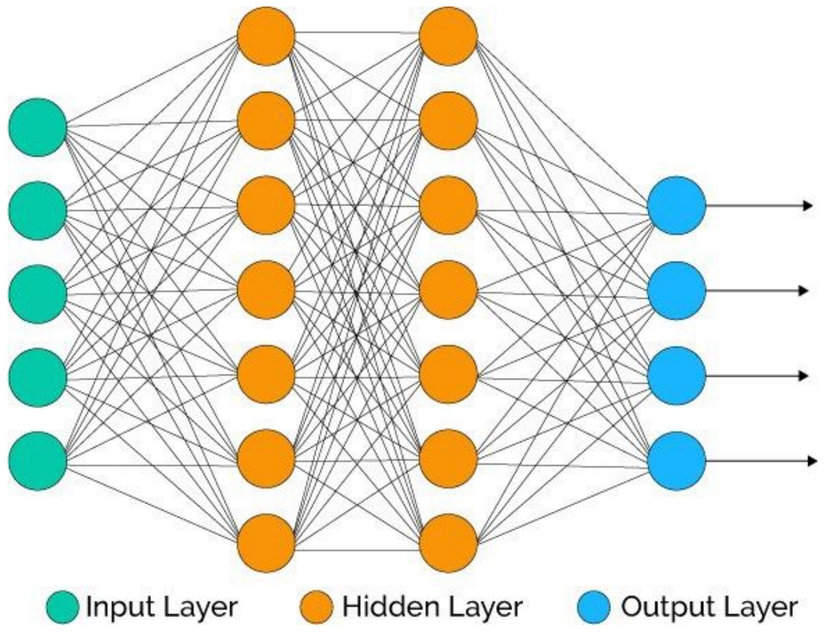
In order to understand how this software model works, we first need to have a certain basic idea about the Convolutional Neural Network being used here, the tools being used and brief knowledge of Python programming knowledge. Some of the main points of focus in the entire documentation are:

1. Neural Network in Deep Learning
2. Convolutional Neural Network
3. Toolkit Specification (OpenCV & Keras)
4. MNIST Sign Language Dataset

1. What is a Neural Network?

To understand what a Convolutional Neural Network is, we need to first understand what a Neural Network is. A neural network is a series of various layers of algorithms that aim to recognize the relationship amongst the data through a method that mimics the way how a human brain operates. The basic building block of this model is called a 'node' which is a replica of an actual neuron, the basic building block of the Human Nervous System.

A Neural Network has an input layer, a hidden layer consisting of many layers of nodes, and the final output layer. The nodes are nothing more than an Activation function, a mathematical function that performs a mathematical task on a piece of input data manipulates it to find a relationship between the input and the output data.



How a Neural Network Model gets trained?

Every neural network needs to be trained on a segment of real data before being put to real use. It needs to have a particular combination of weights that corresponds to the highest accuracy of the Neural Network. This change or optimization of weights is mapped in the gradient descent graph, and a particular weight is assumed to be suitable if the gradient descent graph reaches a particular minimum.

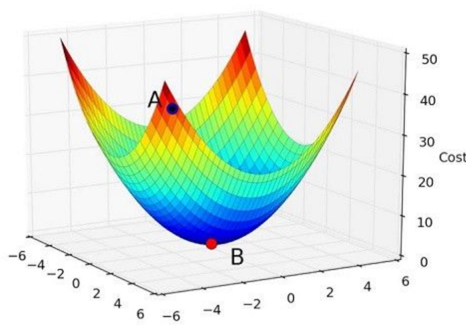


Fig. Gradient Descent Mapping. B is the point of Minima.

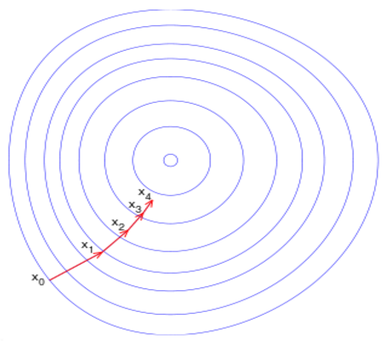


Fig. Moving of Weights towards the central minima during the training phase

Forward Propagation

The entire process of feeding data into the Artificial Neural Network's input layer and getting processed layer after layer until being outputted through the output layer is known as Forward Propagation. The ultimate result of the forward propagation is called the Loss Function. The loss function needs to be eliminated to zero if we need the full accuracy of the model. Hence we need to compute using Backward Propagation.

Backward Propagation

Once we produce the loss function, the output elements are fed back into the network and propagated backward until we get the end result as the optimized weights of the Neural Network.

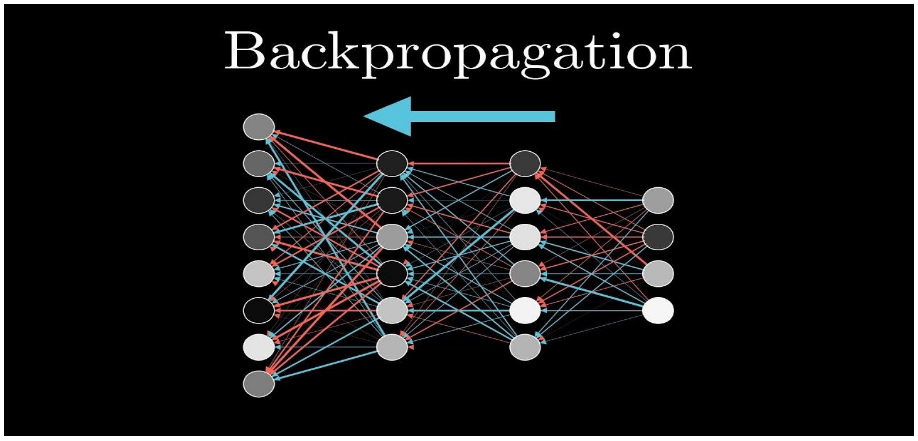


Fig. Backpropagation in Neural Network

What after training?

After the training phase is over, the model is operated with the validation set data, which ensures that the model works perfectly under extreme conditions. Finally, after this phase is over, the model is further tested with the test and accuracy and loss curve of the output are computed. Higher the accuracy and lower the loss, more favourable our model is for deployment.

2. Convolutional Neural Network

CNN is almost the same as the Artificial neural network. But the only difference is that CNN uses certain different and fixed layers represented by blocks. It is mostly used in cases that involve Images and Video inputs, including real-time camera input.

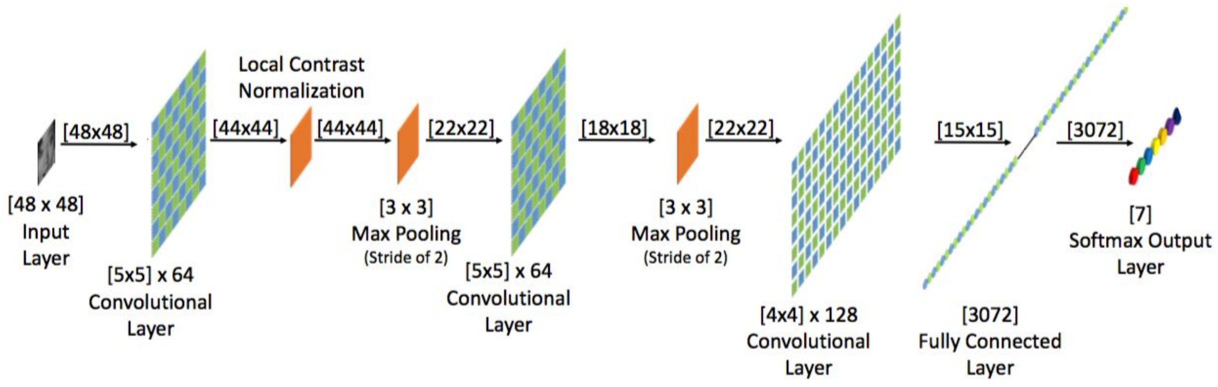

[48x48]
[48 x 48] Input Layer
[5x5] x 64 Convolutional Layer
Local Contrast Normalization
[44x44]
[44x44]
[3 x 3] Max Pooling (Stride of 2)
[22x22]
[5x5] x 64 Convolutional Layer
[18x18]
[3 x 3] Max Pooling (Stride of 2)
[22x22]
[4x4] x 128 Convolutional Layer
[15x15]
[3072] Fully Connected Layer
[3072]
[7] Softmax Output Layer

Fig. Layers used in a sample CNN

A CovNet uses filters to analyze these image inputs. Training of a CovNet means, training these filters to accurately detect objects in the image and assign correct weights to each filter.

Image Reduction using Convolutional Layer

The first step of the model is to reduce the size of the image input by means of a matrix filter called convolution, which contains predefined values. The image, also being divided into the matrix, has values assigned in each cell. The summation of the product of these values with that of convolution are added together to create one value of the output matrix, called the convolved feature. The convolved feature is completed. If the image is in RGB mode, this process is done with every channel of the image. The number of cells that the filter will move depends on the 'stride' of the filter.

Pooling Layer

To reduce the computational power required for processing the convolved feature, the spatial size must be reduced. Hence a Pooling Layer comes into play. In the context of this project, we are using the Max Pooling technique, a technique that returns the maximum value from the portion of the image covered by the kernel. It also provides noise reduction and dimensional reduction of the convolved feature, making it more efficient.

The convolution layer and the pooling layer are continuously used one layer after another until we get the most suitable convolved feature matrix for our CovNet.

Fully Connected Layer

The matrix that we had got at the end of the final Convolution Layer, are flattened and arranged in a linear fashion, and a ReLu (Rectified Linear Unit) activation is applied on each cell (or node). This will bring the value between 0 and z, a unit used in the activation function.

This flattened layer on the Convolution Neural Network is fed into an Artificial Neural Network, followed by which feed forwarding and backpropagation takes place, generating some dominant features and low-level features.

Once features generated are slightly distinguishable, the dominant features are classified using a Softmax Classification Layer. These features are the ones that had been detected using the Convolutional Neural Network Model.

3. Toolkit Specifications

Keras

Keras is an open-source neural network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, R, Theano, or PlaidML. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible, the same reason for which this is being used for building this particular project model.

OpenCV

OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision. In simple language, it is a library used for Image Processing. It is mainly used to do all the operations related to Images. It also supports multiple languages from ranging from Python, Java, C++, Android SDK, and even C. Here we shall be using OpenCV for the purpose of detecting the hand gestures. Plus, for the purpose of proper tuning of the input frames being detected, we shall add an image tuning feature, which shall be discussed subsequently.

4. MNIST Sign Language Dataset Specification

The Sign Language Dataset is a huge collection of images of hand gestures each signifying an alphabet from A to Z. It consists of 2000 images in the form of .csv dataset of a particular hand gesture corresponding to an alphabet. First of all, the huge .csv file is converted into a recognizable image file, followed by splitting the dataset into training and test sections. For the purpose of this project, we are dividing each of these 2000 images of a particular alphabet into 250 test images and 1750 training images, after processing as per the convenience.

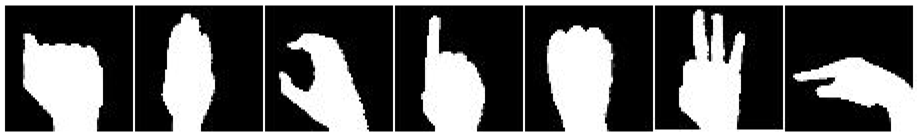


Fig. Sample Converted Images from MNIST .CSV Data

Code Description and Analysis

1. Convolutional NN Model File (model.py)

Step1: First we import all the libraries and the components that shall be required in the creation of the neural network.

```
from keras import optimizers
from keras.models import Sequential
from keras.layers import Convolution2D
from keras.layers import MaxPool2D
from keras.layers import Flatten
from keras.layers import Dense, Dropout
from keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
```

Step2: Next we create the Convolutional Neural Network model which shall be trained to detect the hand gestures and label it for every english alphabet. The structure of the neural network is given below:

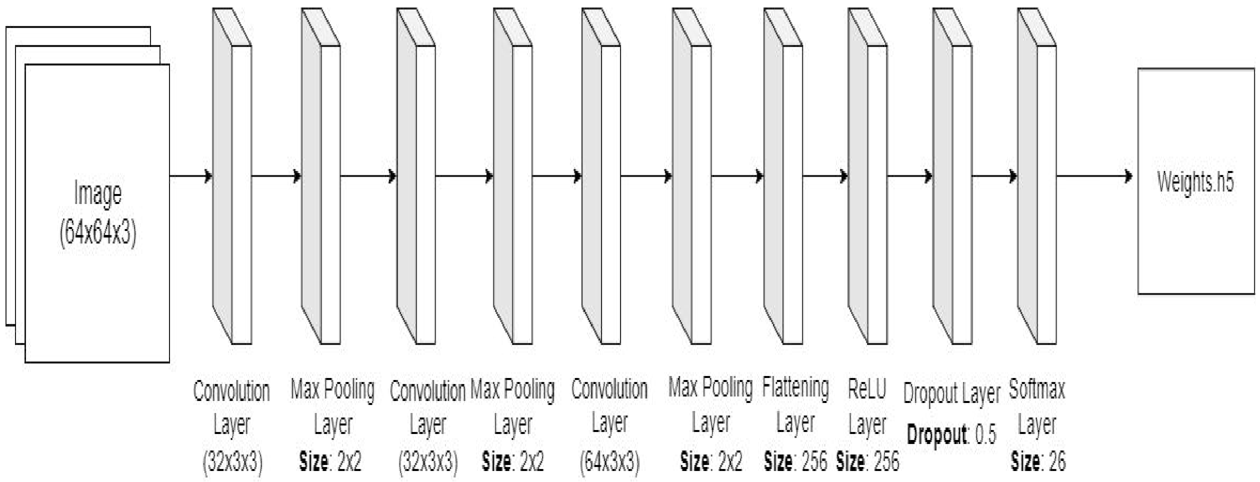


Image (64x64x3)
Convolution Layer (32x3x3)
Max Pooling Layer Size: 2x2
Convolution Layer (32x3x3)
Max Pooling Layer Size: 2x2
Convolution Layer (64x3x3)
Max Pooling Layer Size: 2x2
Flattening Layer Size: 256
ReLU Layer Size: 256
Dropout Layer Dropout: 0.5
Softmax Layer Size: 26
Weights.h5

Fig. Convolutional Neural Network Model for Gesture Recognition

The code written for the corresponding model is given below:

```
model = Sequential()
model.add(Convolution2D(32, 3, 3, input_shape=(64, 64, 3),
activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Convolution2D(32, 3, 3, activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Convolution2D(64, 3, 3, activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(26, activation='softmax'))
model.compile(optimizer=optimizers.SGD(lr=0.05),
loss='categorical_crossentropy', metrics=['accuracy'])
```

Step3: Now we need to specify the hyperparameters for the model created. The list of hyperparameters and its corresponding values are given below:

No.	Hyperparameter	Values
1.	Training Images	1750
2.	Test/Validation Images	250
3.	Training Image Size	64x64
4.	Test Image Size	64x64
5.	Epochs	25
6.	Steps/Epoch	800
7.	Validation Steps	6500

The code for setting the hyperparameters are given below:

```
train_data = ImageDataGenerator(rescale=1. / 255,
shear_range=0.2, zoom_range=0.2, horizontal_flip=True)
test_data = ImageDataGenerator(rescale=1. / 255)
training_data = train_data.flow_from_directory('data/train',
target_size=(64, 64), batch_size=32, class_mode='categorical')
test_data = test_data.flow_from_directory('data/test',
target_size=(64, 64), batch_size=32, class_mode='categorical')
```

```
final_model = model.fit_generator(training_data,
steps_per_epoch=800, epochs=25, validation_data=test_data,
validation_steps=6500)
model.save('Weights.h5')
```

Step 4 (optional): Once the model has been trained and the weights has been saved in the Weights.h5 file, two important graphs are plotted to generate a rough estimate about the performance of the model:

1. Accuracy vs epochs Graph
2. Loss vs epochs Graph

The code for plotting the graph are given below (using matplotlib.pyplot):

```
#Plotting the mean Accuracy per epochs graph
print(final_model.history['acc'])
plt.plot(final_model.history['acc'])
plt.plot(final_model.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

#Plotting the mean Loss per epochs graph
plt.plot(final_model.history['loss'])
plt.plot(final_model.history['loss_val'])
plt.title('model loss')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

2. Live testing file (camera.py)

Step 1: First I imported the libraries that are required for processing the video feed and loading the model weights.

```
import cv2
import numpy as np
from keras.models import load_model

model = load_model('Weights.h5')
image_x, image_y = 64, 64
```

Step 2: Now the function for making the prediction is defined. This function shall be responsible for taking in the video feeds, convert it into frames of images, convert again into an image matrix, and extract the features by comparison with the loaded weights.

```
def prediction():
   import numpy as np
   from keras.preprocessing import image
   test_image = image.load_img('test.png',
target_size=(64,64))
   test_image = image.img_to_array(test_image)
   test_image = np.expand_dims(test_image, axis=0)
   result = model.predict(test_image)

   if result[0][0] == 1:
       return 'A'
   elif result[0][1] == 1:
       return 'B'
   elif result[0][2] == 1:
       return 'C'
   elif result[0][3] == 1:
       return 'D'
   elif result[0][4] == 1:
       return 'E'
   elif result[0][5] == 1:
       return 'F'
   elif result[0][6] == 1:
       return 'G'
   elif result[0][7] == 1:
       return 'H'
   elif result[0][8] == 1:
       return 'I'
   elif result[0][9] == 1:
       return 'J'
   elif result[0][10] == 1:
       return 'K'
   elif result[0][11] == 1:
       return 'L'
   elif result[0][12] == 1:
       return 'M'
   elif result[0][13] == 1:
       return 'N'
   elif result[0][14] == 1:
       return 'O'
   elif result[0][15] == 1:
        return 'P'
    elif result[0][16] == 1:
        return 'Q'
    elif result[0][17] == 1:
        return 'R'
    elif result[0][18] == 1:
        return 'S'
    elif result[0][19] == 1:
        return 'T'
    elif result[0][20] == 1:
        return 'U'
    elif result[0][21] == 1:
        return 'V'
    elif result[0][22] == 1:
        return 'W'
    elif result[0][23] == 1:
        return 'X'
    elif result[0][24] == 1:
        return 'Y'
    elif result[0][25] == 1:
        return 'Z'
```

Step 3: Now to define and configure the video feeds and the image processing. The images that were extracted from MNIST Sign Language .csv data, after processing were converted into a clear binary image of black and white.

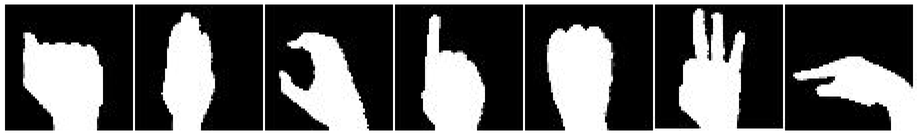


Fig. Final processed image

The input data acquired from the device camera may be subjected to several external interference or noises. The input visuals are apparently converted to HSV mode (Hue, Saturation, Value) in order to match the image data above. This apparent conversion along with reducing the external noises is done using a trackbar methodology.

In this method, we set up three parameters according to HSV image criteria.

No.	Channel	Definition	Range
1.	Hue	It represents the colour type being retrieved from the camera.	0-255
2.	Saturation	It represents the vibrancy of the colour of the image retrieved. Lower the saturation, more faded the image data retrieved.	0-255
3.	Value	This represents the brightness of the image.	0-255

In order to extend flexibility, I have added a trackbar for the lower bound and upper bound respectively.

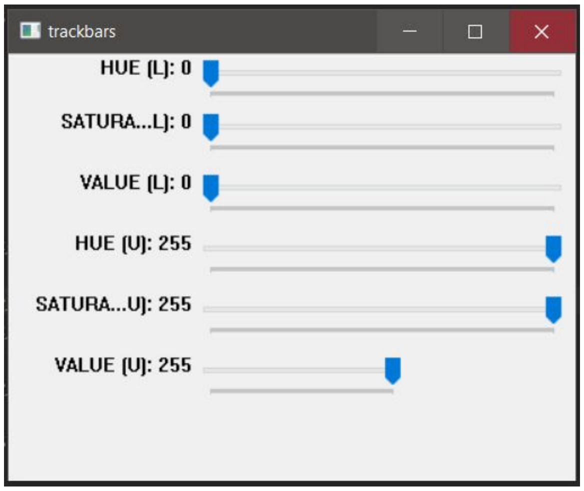


Fig. Sample Screenshot of the Trackbars

The code written for this are as follows:

```
video = cv2.VideoCapture(0)

def nothing(x):
   pass

#Creating trackbars
cv2.namedWindow("trackbars")
cv2.resizeWindow("trackbars", 400, 300)
cv2.createTrackbar("HUE (L)", "trackbars", 0, 255, nothing)
cv2.createTrackbar("SATURATION (L)", "trackbars", 0, 255,
nothing)
cv2.createTrackbar("VALUE (L)", "trackbars", 0, 255, nothing)
cv2.createTrackbar("HUE (U)", "trackbars", 255, 255, nothing)
```

```
cv2.createTrackbar("SATURATION (U)", "trackbars", 255, 255,
nothing)
cv2.createTrackbar("VALUE (U)", "trackbars", 255, 255,
nothing)

cv2.namedWindow("Output")
image_text = ''

#Defining the hsv image mask (what computer sees)
while True:
   ret, frame = video.read()
   frame = cv2.flip(frame, 1)
   hue_l = cv2.getTrackbarPos("HUE (L)", "trackbars")
   sat_l = cv2.getTrackbarPos("SATURATION (L)", "trackbars")
   val_l = cv2.getTrackbarPos("VALUE (L)", "trackbars")
   hue_u = cv2.getTrackbarPos("HUE (U)", "trackbars")
   sat_u = cv2.getTrackbarPos("SATURATION (U)", "trackbars")
   val_u = cv2.getTrackbarPos("VALUE (U)", "trackbars")

   image = cv2.rectangle(frame, (425, 100), (625, 300), (0,
255, 0), thickness=5, lineType=8, shift=0)

   low_bound = np.array([hue_l, sat_l, val_l])
   high_bound = np.array([hue_u, sat_u, val_u])
   image_crop = image[102:298, 427:623]
   hsv = cv2.cvtColor(image_crop, cv2.COLOR_BGR2HSV)
   mask = cv2.inRange(hsv, low_bound, high_bound)

   cv2.putText(frame, image_text, (30, 400),
cv2.FONT_HERSHEY_PLAIN, 1.5, (0, 255, 0))
   cv2.imshow("Test Output", frame)
   cv2.imshow("HSV Output", mask)

#Saving the output data
   image_name = "test.png"
   save_image = cv2.resize(mask, (image_x, image_y))
   cv2.imwrite(image_name, save_image)
   image_text = prediction()

#Press 'Esc' to quit
   if cv2.waitKey(1) == 27:
       break

video.release()
cv2.destroyAllWindows()
```

Performance Analysis

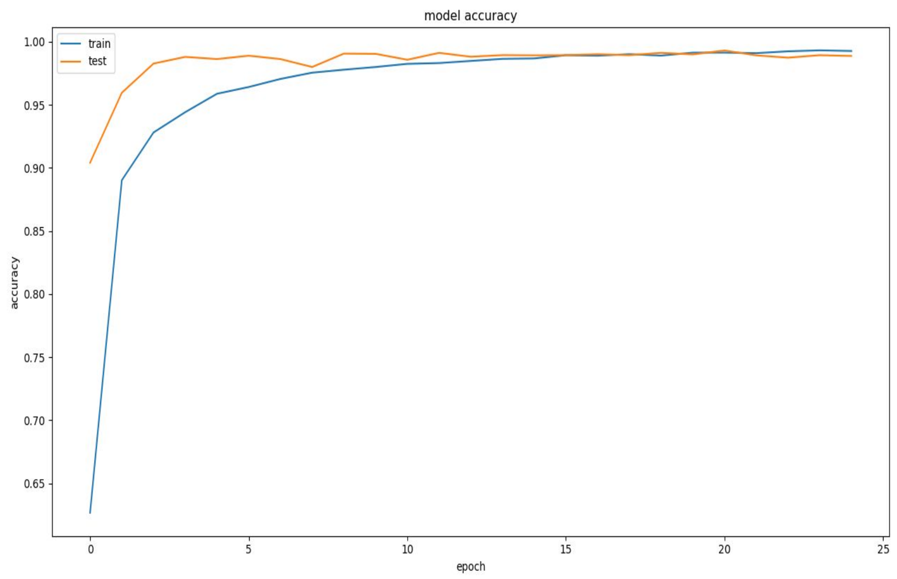


Fig. Accuracy per Epochs Graph (**≈99.23%**)

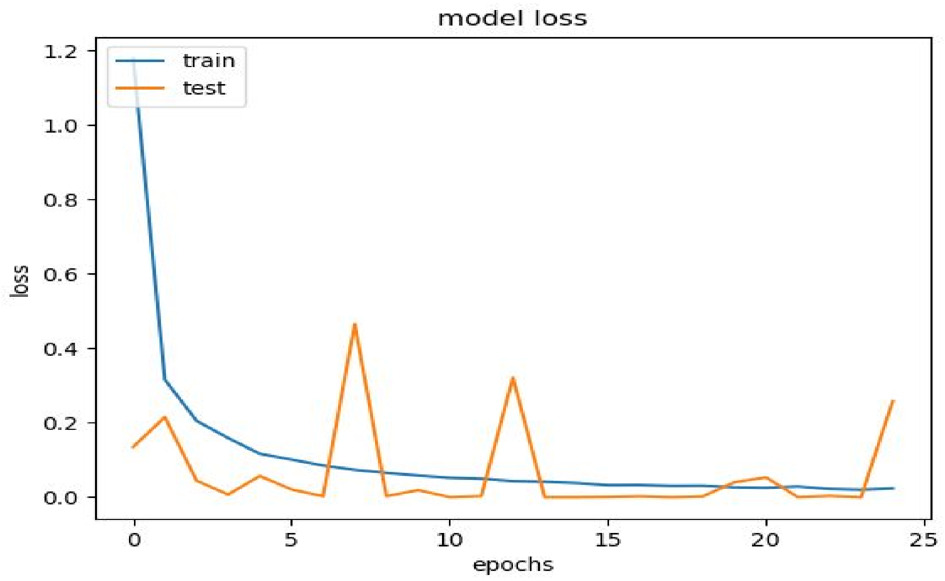
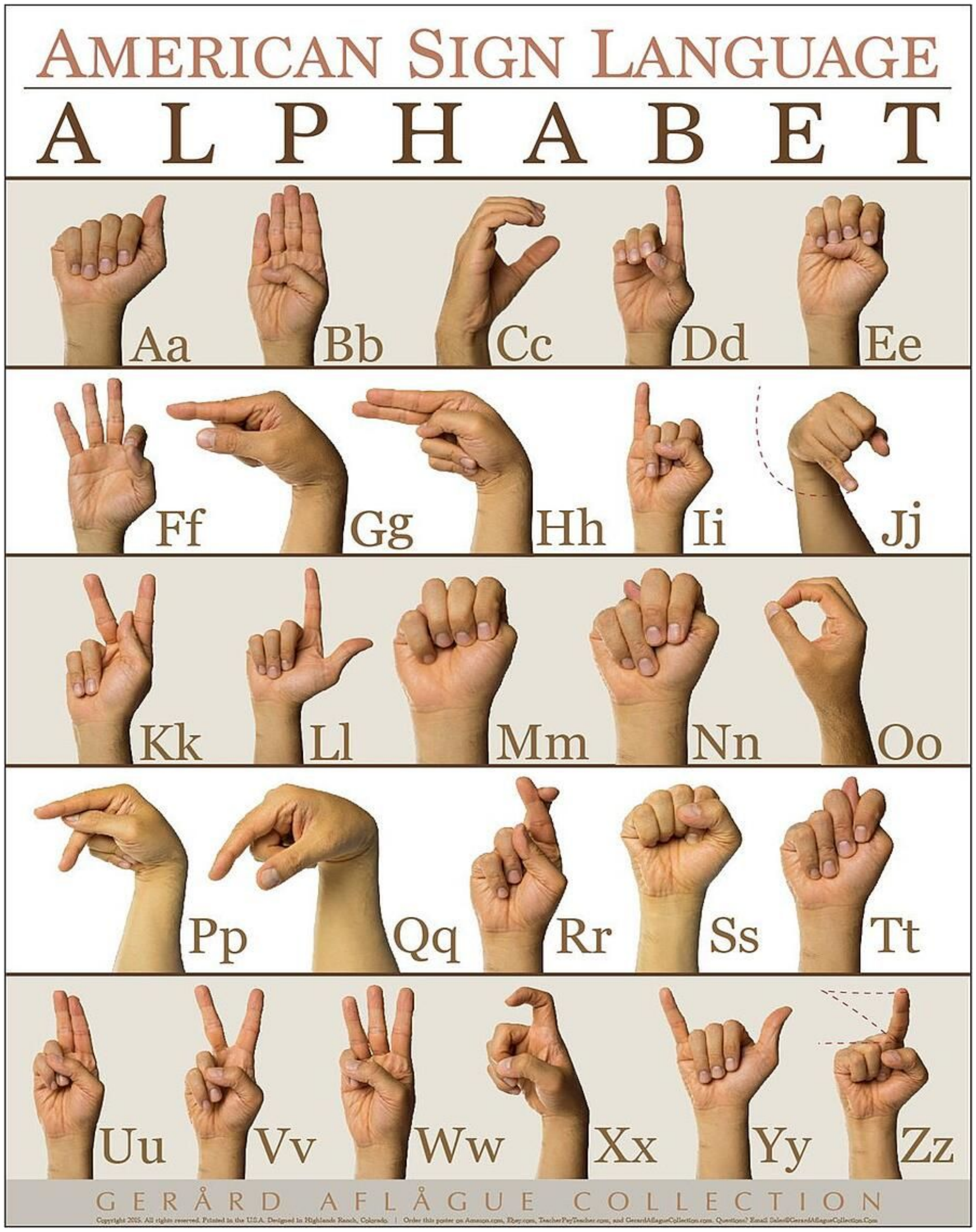


Fig. Loss per Epochs Graph (negligible)

Sign Language Reference Chart


AMERICAN SIGN LANGUAGE
ALPHABET
Aa
Bb
Cc
Dd
Ee
Ff
Gg
Hh
Ii
Jj
Kk
Ll
Mm
Nn
Oo
Pp
Qq
Rr
Ss
Tt
Uu
Vv
Ww
Xx
Yy
Zz
GERÅRD AFLÅGUE COLLECTION
Copyright 2015. All rights reserved. Printed in the U.S.A. Designed in Highlands Ranch, Colorado. | Order this poster on Amazon.com, Ebay.com, TeacherPayTeacher.com, and GerardAflagueCollection.com. Questions? Email Sales@GerardAflagueCollection.Com

References

No.	Resource	Link
1.	Keras Documentation Home Page	https://keras.io/
2.	Computer Vision Tutorial by Udemy	https://www.udemy.com/course/computer-vision-a-z/
3.	Machine A-Z Tutorial by Udemy	https://www.udemy.com/course/machinelearning/
4.	MNIST Sign Language Dataset from Kaggle	https://www.kaggle.com/datamunge/sign-language-mnist
5.	Stack Overflow Community	https://stackoverflow.com/
6.	Theory in the document	https://towardsdatascience.com

Conclusion

As a conclusion, it can be stated that with the help of this sign language translation tool, the sign language translation need not require any expertise from a trained translator. Also, it can be concluded that despite the existing data which has been taken from the MNIST Sign Language Dataset, we can also explicitly add certain data from our own, thus manipulating and training the model to support some more gestures, thereby increasing functionalities.