INTELLIGENT ROBOTICS I - PROJECT: LEARNING PROSTHESIS

Abhiraj Abhaykumar Eksambekar

abhiraj@pdx.edu

932343978

TABLE OF CONTENTS:

1) INTRODUCTION 4

2) ENVIRONMENT 5

2.1) Software 5

2.1.1) Visual Studio Code 5

2.1.2) Jupiter Notebook 5

2.1.3) Python 5

2.2) Hardware 5

2.2.1) Arduino Nano 6

2.2.2) Raspberry Pi 4 6

2.2.3) Sunflower PCA 9685 7

2.2.4) Futaba S3003 7

2.2.5) Raspberry Pi Camera V2 8

3) DATASET FOR TRAINING TESTING AND VALIDATION 9

3.1) Creating of Dataset 9

3.2) Training Dataset 10

3.3) Validation Dataset 10

3.4) Testing Dataset 10

4) CONVOLUTION NEURAL NETWORK MODEL 11

4.1) CNN Layers 11

4.2) MOBILE_NET_V1 12

5) PREPROCESSING IMAGES 13

5.1) Preprocess for training 13

5.2) Preprocess for prediction 14

6) TRAINING PROCESS 15

7) H5 TO TFLITE CONVERSION 18

8) PREDICTING IMAGES WITH TRAINED MODEL 19

9) TESTING THE MODEL 20

9.1) Test set 1- 200 Images from Main Training Dataset 20

9.2) Test set 2- 2000 Images Simple Set 21
9.3) Test set 3- 2000 Images Complex Set 22
9.4) Test set 4- 2000 Images Noisy Set 23
9.5) With Live Video 25
10) INMOOV ROBOT ARM 29
10.1) Structure 29
10.2) Controlling Arm with Arduino Code: 29
10.3) Controlling Arm with Python Script 30
11) DETECTION AND REPLICATION 31
12) CONCLUSION AND FUTURE POSSIBILITIES 33
12.1) Improving Arm Structure 33
12.2) Improving Classification 33
12.3) Other Methods of Controlling the Arm 33
12.4) Conclusion 33

1) INTRODUCTION

As we move into the future, it is necessary to find ways for humans to enable them to do tasks which may be impossible for them to do due to physical limitations. Missing limbs can limit humans to do specific tasks but cannot stop them. Therefore, my project deals with studies and a simple implementation of prosthetic robotic arm controlled with gestures provided by the user. The gestures by users will be taken input from camera feed and similar gesture will be replicated on to the robotic arm. Machine learning and neural networks, the algorithm can be trained to detect and classify different objects. There are 2 main types of training methods namely- supervised learning and unsupervised learning. Each method can be used for different type of application. More will be discussed of these methods in training section.

It is also important to have a good mechanical design of the arm. A good design provides more degrees of freedom. It also ensures the arm and fingers can perform tasks as good as a human arm. The mechanical design should also ensure that the structure is strong and none of its moments restrict moment of other part. For example, it can happen that screw in finger is so long that when finger next to it is unable to move due to the long screw obstructing its motion.

In this project I try to replicate my hand gestures on a prosthetic robotic arm by showing the gestures on camera.

2) ENVIRONMENT

2.1) Software

2.1.1) Visual Studio Code

Visual Studio Code is used to write the main python script and for Arduino code as well. Platform IO in a module found in Visual Studio Code which is used to download the Arduino code into the respective device. It can also be used for serial monitor. Python modules available in VS code provide more functionality and efficient way to write a python script.

2.1.2) Jupiter Notebook

Jupiter Notebook is an open-source web application that allows user to create documents containing live code. It enables the user to write small code snippets and test them individually. It helps in easy debugging of the python script.

2.1.3) Python

Scripts for training a machine learning model, replication the final gestures on to arm and other supporting scripts for this project are written in python. It provides packages necessary for training the model and for other hardware modules in use. Training and supporting scripts for this project run on python 3.8.6. And the final script to classify and replicate runs on python 3.7.3.

2.2) Hardware

All the python scripts and Arduino code are written on laptop having 16GB RAM and 6GB GPU. GPU helps in training the machine learning model faster.

2.2.1) Arduino Nano

Figure 1 Arduino Nano

Arduino Nano is a small bread-board friendly board based on ATMEGA328. It has similar functionalities to that of Arduino UNO only difference being small in size, compact and a smaller number of pins. It has onboard FTDI FT232RL serial communication. This is used for USB to serial communication with the microcontroller and even for programming the microcontroller.

2.2.2) Raspberry Pi 4

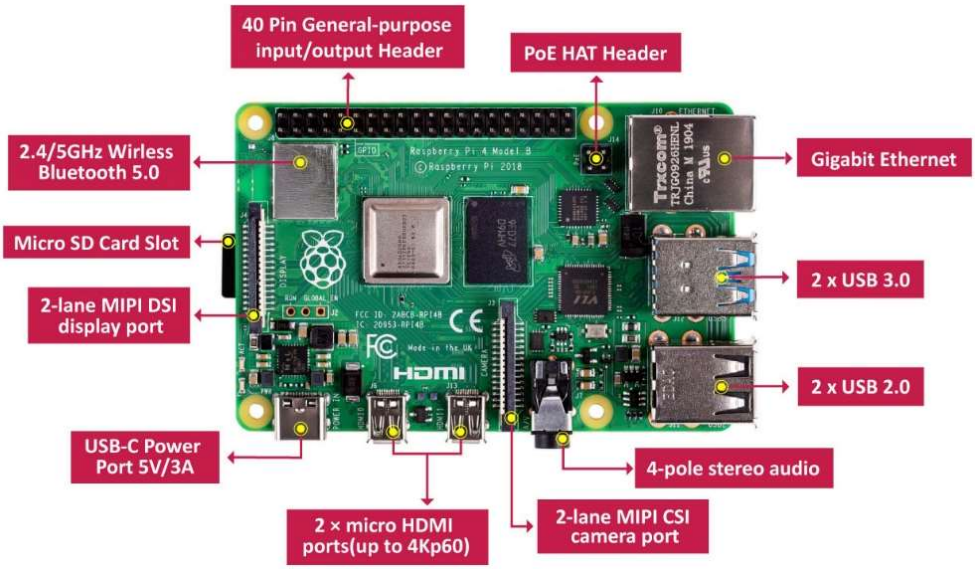

Figure 2 Raspberry Pi 4

Raspberry pi is a tiny computer that can be used for learning programming and projects. It has GPIO pins which can be used to control external hardware. This project uses Raspberry Pi 4 with 4GB RAM. It has 4 USB ports, 2 micro-HDMI ports, 1 Ethernet port and 1 USB C port. It has onboard port to interface compatible camera. This board uses quad-core 64bit SoC processor running at 1.5GHz. The

python scripts are run on this device. GPIO pins on Raspberry Pi are used to control servo control module.

2.2.3) Sunflower PCA 9685

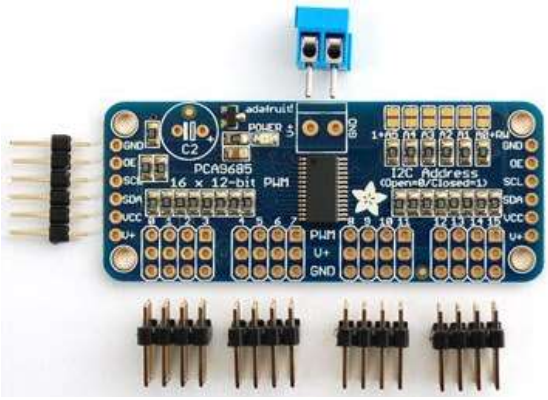

Figure 3 Sunflower PCA 9685

Sunflower PCA 9685 is 16-channel servo driver module. It is used to control the servos used to control the servo motors used to move the fingers to destined position. This module is programmed with I2C communication. Each channel for servo can be set to have different duty cycle. Therefore, each servo connected to PCA module can be controlled separately.

2.2.4) Futaba S3003

Figure 4 Futaba S3003

Futaba S3003 is a servo motor with operating range 4.8-6V. It can rotate from 0-180 degrees. Initial test of motors was made to check the step size of these motors. From official documentation and testing, it was noted that 2% duty cycle corresponded to 0-degree angle. And 7.5% duty cycle corresponds to 180 degrees.

2.2.5) Raspberry Pi Camera V2

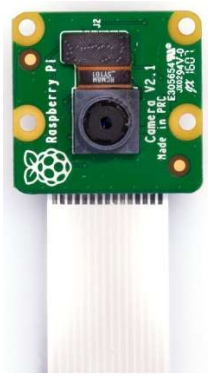

Figure 5 Raspberry Pi Cam V2

Raspberry Pi Cam V2 is 8-Megapixels. It supports 1080p30, 720p60 and VGA90 video modes.

3) DATASET FOR TRAINING TESTING AND VALIDATION

3.1) Creating of Dataset

For this project I created my own dataset containing images of 10 different gestures. The 10 gestures are- Zero, One, Two, Three, Four, Five, Thumbs Up, Rock Sign, Spiderman Sign, Ok Sign. To capture these gestures, I wrote a simple python script which guides the user to show the required gesture and then start capturing it. This script saves the gesture images in a directory named after the user and in the corresponding gesture sub-folder. To run this script, the user needs to have opencv python library installed on his/her machine. Once installed, the user only needs to run the script normally and follow the instructions given in runtime.

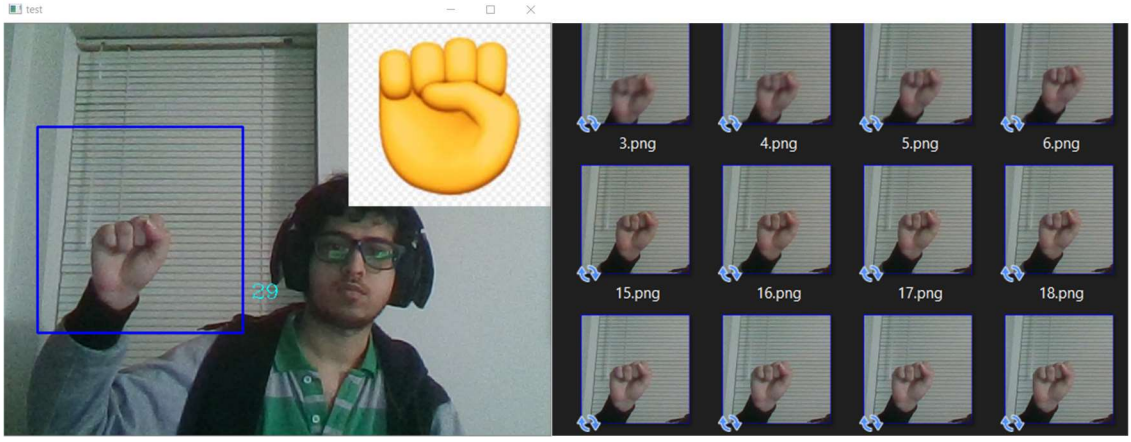

Figure 6 Dataset Generation

The left figure shows the capturing process, and the right figure shows the saved images. In the left image above, we can see the gesture to be captured is displayed at right corner of the window and corresponding gesture is done by the user inside the blue box. Thus, all the gestures images are captured and saved in the respective folders. This script captures 200 images of one gesture and there are 10 gestures in total. Therefore, for one user, there are 2000 gestures captured and saved in total. This script was given to friends, family and classmates to expand the training set of which only 8 volunteers helped with expanding the dataset. Therefore, the dataset had 16000 images in total.

3.2) Training Dataset

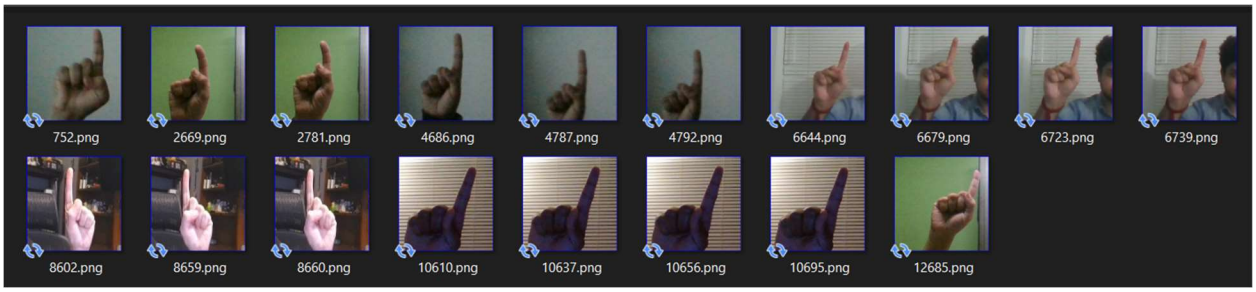

Figure 7 Training Set Showing Different Hands and Background

The main dataset of 16000 images was divided into 3 different datasets. The training dataset contains major portion of main dataset. This dataset is used to train the CNN to classify required gestures. From the 16000 images, the 12757 images were used for training. Having a variety and a bigger dataset helps in training the model to classify required classes more accurately. The above dataset shows different types of hands for same gesture used for training the CNN model.

3.3) Validation Dataset

The validation dataset has 3000 gesture images taken from the main dataset. This dataset validates that the training of model has no problems. An overfitting problem arises when the model gets trained so perfectly that it is able to classify objects from given training set but fails when new images are asked to classify. With the help of validation set, it is possible to find any possible overfitting problem.

3.4) Testing Dataset

Out of the 16000 images in main dataset, 200 images were kept aside to test the trained model. Training helps to understand whether the model is able to predict the known gestures correctly. Apart from these 200 images, another 3 testing datasets were created for rigorous testing of the trained model. These testing model includes simple 10 gesture model with hand gestures with less noise, no translation, and no rotation. The next test set had hand gestures with no noise, and lot of hand translation and rotation. The last test set has lot of noise and lot of rotation and translation of hand. The noise includes, too much background light from different sources. All these test sets were tested, and confusion matrix was created for analysis.

4) CONVOLUTION NEURAL NETWORK MODEL

4.1) CNN Layers

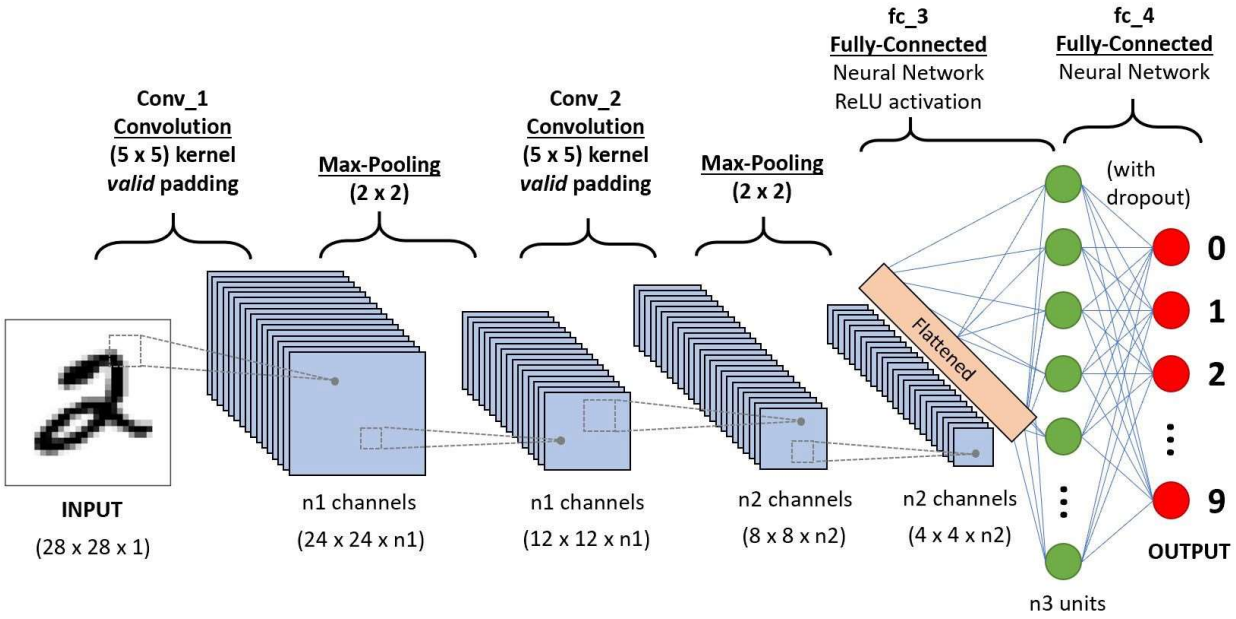

Figure 8 Convolutional Neural Network

The CNN model consists of convolution layer, max pooling layers and zero padding layers. These layers extract features from the training images. The convolution layer is usually a 2x2 filter which convolves the input n-dimension array. This process tries to reduce noise so that the feature extraction can be done more efficiently.

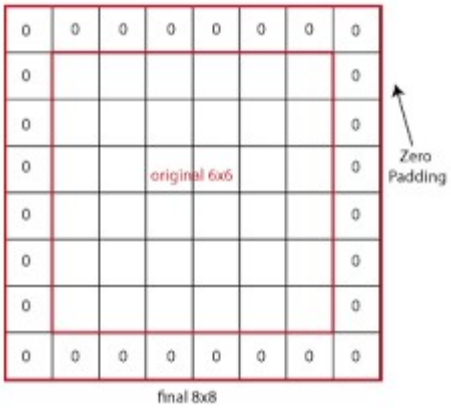

Figure 9 Zero Padding

The zero-padding layer pads the edges of the input with zeros. Input array dimensions is increased by 1x1.

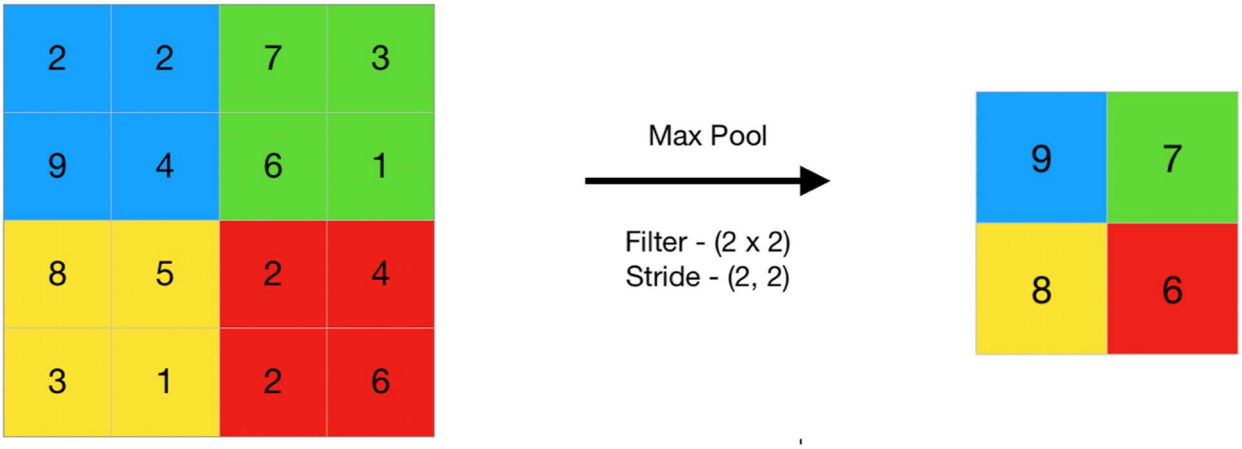

Figure 10 Max Pooling

The maxpooling layer is a filter which convolves the filter with input and extracts max values from the convolving area. It extracts feature from the input.

4.2) MOBILE_NET_V1

I am using the mobile net model available in keras application package. This model is an efficient CNN model for small mobile like devices. It consumes less disk space and is faster than bigger size models. It consumes 16 MBs of disk space. Its top-1 accuracy is 70.4% and top-5 accuracy is 89.5%. This means that this model is likely to have the correct classification at its 1st position about 0.704 times and similarly, top-5 accuracy means, that this model is 0.895 times likely to have the correct classification in its first 5 prediction rankings for a given input image.

MobileNet has parameter size of 4,253,864. And has 88 fully connected layers. This model is pretrained to detect 1000 different classes like animals, faces, hands and some objects. For this project I only require the model to classify 10 different gestures, therefore, I modify the output layer to classify only 10 gesture and train this new model to classify required gesture.

The main reason for selecting MobileNet CNN is that it is small in size and can process faster. Fast processing is required since the classification of live video will be running on a Raspberry Pi 4 which has less computation speed. After modifying the layers, the final CNN model used for training and classification has input of 224x224 size and 10 output neurons. Summary of the CNN can be found in training model report file.

5) PREPROCESSING IMAGES

Preprocessing image involves the image to be processed to reduce noise and unnecessary features. This helps in faster operations on the image. Main preprocessing is done using the function tf.keras.applications.mobilenet.preprocess_input(). This function is specified for MobileNet CNN model. Apart from this, more processing is done a little differently for training the model and for prediction of images.

5.1) Preprocess for training

```
In [3]: #Generating Batches for training and validating
        #Batches are created after preprocessing the images in folder with processing
         function for mobile net
        trainBatch= ImageDataGenerator(rotation_range= 40, zoom_range=[0.15,1.4], prep
        rocessing_function= tf.keras.applications.mobilenet.preprocess_input).flow_fro
        m_directory(directory= trainDir, shuffle=True, target_size=(224,224), batch_si
        ze = 32)
        validBatch= ImageDataGenerator(rotation_range= 40, zoom_range=[0.15,1.4], prep
        rocessing_function= tf.keras.applications.mobilenet.preprocess_input).flow_fro
        m_directory(directory= validDir, shuffle=True, target_size=(224,224), batch_si
        ze = 32)

        Found 12757 images belonging to 10 classes.
        Found 3000 images belonging to 10 classes.
```

Figure 11 Code Snippet- Preprocessing using ImageDataGenerator

To train the CNN model, ImageDataGenerator method is used. This method is available in keras preprocessing package. This method is used with flow_from_directory. It takes input argument as a directory which has classes separated in sub directories. Other arguments used are batch_size which creates batches of images from input directory of given batch size. Target_size resizes the input image to required dimension. Training the model runs for defined number of epochs. Epochs can be considered as steps. In each individual epoch the model will be trained with all batches generated created using all images in the folder. Benefit of using ImageDataGenerator is that it generates a different set of batches for each epoch according to parameters and arguments set in the method. The shuffle argument will shuffle the images in the directory each time it generates the batches. Rotation_range will randomly rotate the image each epoch. Similarly, zoom_range will zoom in or zoom out randomly as per the range defined. Using this method, model can be trained to be more generalized as it will be trained same image but different set of features each time.

5.2) Preprocess for prediction

In [5]:

```
#preprocessing the image, Same as done while training the model
def preProcess(image):
    reszImg = cv2.resize(image, (224, 224))
    reszImg= cv2.cvtColor(reszImg, cv2.COLOR_BGR2RGB)
    preProsImg = tf.keras.applications.mobilenet.preprocess_input(reszImg)
    reshapImg = preProsImg.reshape(-1,224,224,3)
    return reshapImg
    pass
```

Figure 12 Code Snippet- Preprocessing for Prediction of Image

Same preprocessing function is used during prediction but since prediction is done for only single image, ImageDatasetGenerator is not used. Instead, custom function is created to process the single image. This function initially resizes the image 224x224, which is the input size of the CNN model. ImageDatasetGenerator loads the image in RGB format and opencv loads the image in BGR format, and the training of CNN was done for RBG format. Therefore, the BGR format of OpenCV is converted to RGB format. Converted RGB format is preprocessed using the preprocess function for MobileNet CNN. After applying main preprocessing to the image, it is reshaped to 4-dimension array which is compatible with the MobileNet input. This reshaped image is passed to trained CNN model for it to predict the gesture in the image.

6) TRAINING PROCESS

Training the model means to model is fed with images with known labels. The model predicts some output according to its current weights and biases. The training function calculates loss depending how badly it predicted the image and updates weights and biases interconnecting the neurons in two consecutive layers.

After generating batches and downloading and modifying the MobileNet CNN model, it needs to be trained with training set so that the new model is able to classify the training images and any new images correctly. For this the model is compiled using the following command.

```
In [8]: #Compile the model
        model.compile(optimizer= tf.keras.optimizers.Adam(lr=0.0001), loss='categorica
        l_crossentropy', metrics= ['accuracy'])
```

Figure 13 Code Snippet- Compile Model

This command configures the model with losses matrices. Optimizer decides learning rate, also called as step size. Lower the learning rate, lower steps the training model takes towards the required output. The loss function will minimize the loss by summing all individual losses. Categorical cross entropy is used so since there are more than two neurons in the output layer. The metrices creates two local variables that are used to compute, the frequency with predicted labels matches true labels.

```
In [9]: #Fine Tune the model to predict requred classes
        for i in range(0,25):
            print("Epoch Completed: {}".format(i*10))
            model.fit(x= trainBatch, validation_data= validBatch, epochs= 10, verbose=
        2)

            #Save the model
            model.save('model\mainModel{}.h5'.format(i))
```

Figure 14 Code Snippet- Training Model

To start training of the model, fit method is used. This takes input training batches, validation batches, epochs and verbose. The above snippets train the model for 10 epochs and then it saves the trained model. This training and saving methods run in a for loop for 25 times. Training the model for huge number of epochs can train the model near accurate. Training the model takes too much time and computation power. During training period, a loss in power or any failure in computer can cause the models training to be lost and will require the model to be trained from start. Therefore, the for loop ensures it saves models timestamps after training it for every 10 epochs. And thus, int the end, the model gets trained for 250 epochs. Increasing or decreasing the epochs can drastically affect the

training of the model. Having a smaller number of epochs might keep the model untrained. If we set a huge number for epoch, it is possible that model gets trained very good. But setting a very big value will consume too much time to train the model. Therefore, having appropriately enough value is necessary. The trained model is saved every time in .h5 format.

```
Epoch Completed: 0
Epoch 1/10
WARNING:tensorflow:Callbacks method `on_train_batch_end` is slow compared to
the batch time (batch time: 0.0312s vs `on_train_batch_end` time: 0.1250s). C
heck your callbacks.
399/399 - 194s - loss: 0.8252 - accuracy: 0.7358 - val_loss: 0.3533 - val_acc
uracy: 0.8907
Epoch 2/10
399/399 - 197s - loss: 0.2691 - accuracy: 0.9149 - val_loss: 0.1884 - val_acc
uracy: 0.9397
Epoch 3/10
399/399 - 195s - loss: 0.1889 - accuracy: 0.9387 - val_loss: 0.1254 - val_acc
uracy: 0.9573
Epoch 4/10
399/399 - 188s - loss: 0.1409 - accuracy: 0.9556 - val_loss: 0.1132 - val_acc
uracy: 0.9637
Epoch 5/10
399/399 - 190s - loss: 0.1181 - accuracy: 0.9628 - val_loss: 0.1145 - val_acc
uracy: 0.9620
Epoch 6/10
399/399 - 190s - loss: 0.1014 - accuracy: 0.9669 - val_loss: 0.0820 - val_acc
uracy: 0.9690
Epoch 7/10
399/399 - 189s - loss: 0.0941 - accuracy: 0.9697 - val_loss: 0.0747 - val_acc
uracy: 0.9733
Epoch 8/10
399/399 - 189s - loss: 0.0818 - accuracy: 0.9740 - val_loss: 0.0644 - val_acc
uracy: 0.9777
Epoch 9/10
399/399 - 190s - loss: 0.0762 - accuracy: 0.9727 - val_loss: 0.0665 - val_acc
uracy: 0.9783
Epoch 10/10
399/399 - 188s - loss: 0.0719 - accuracy: 0.9761 - val_loss: 0.0611 - val_acc
uracy: 0.9780
Epoch Completed: 10
Epoch 1/10
399/399 - 193s - loss: 0.0696 - accuracy: 0.9771 - val_loss: 0.0534 - val_acc
uracy: 0.9797
```

Figure 15 Training Output Epochs Initial Stages

The above image shows training output for 1st 11 epochs. After completing 10 epochs, the model is saved and then again, the training continues for another 10 epochs. Each epoch output gives a loss and accuracy for training set and for validation set. Initially the model has a huge loss of 0.8252 and accuracy of 0.7358 and similarly for validation with 0.3533 and 0.8907 loss and accuracy. The model trains itself for the training loss and updates the weights and biases of each neural network. As seen in the figure, the accuracy starts rising and the loss starts getting lower as it progresses to train for more epochs.

```
Epoch 6/10
399/399 - 202s - loss: 0.0053 - accuracy: 0.9980 - val_loss: 0.0055 - val_acc
uracy: 0.9977
Epoch 7/10
399/399 - 206s - loss: 0.0065 - accuracy: 0.9976 - val_loss: 0.0062 - val_acc
uracy: 0.9980
Epoch 8/10
399/399 - 205s - loss: 0.0061 - accuracy: 0.9978 - val_loss: 0.0115 - val_acc
uracy: 0.9967
Epoch 9/10
399/399 - 205s - loss: 0.0108 - accuracy: 0.9966 - val_loss: 0.0029 - val_acc
uracy: 0.9993
Epoch 10/10
399/399 - 205s - loss: 0.0118 - accuracy: 0.9962 - val_loss: 0.0125 - val_acc
uracy: 0.9963
```

Figure 16 CNN Model Train Last Epochs

The above figure shows the last 4 epochs of training. Over 250 epochs, the loss decreased to 0.0118 and the increased to near accuracy of 0.9962. The validation loss is also low, and its accuracy is also near accurate. This tell that the model is trained properly and does not have the overfitting problem.

7) H5 TO TFLITE CONVERSION

Prediction of an images takes lot of computation power. On high end device, computation happens quick and any delays are barely recognized. But in low end devices, high computation power is not available and therefore, on these devices it takes more time to process and classify the image using the CNN model. To have faster prediction, the model can be converted to tflite format. This format converts the detailed model to only weights and biases of the layers. Tflite format reduces the disk space consumed by the model by half.

```
# Convert the model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model.
with open('model.tflite', 'wb') as f:
  f.write(tflite_model)
```

Figure 17 Code Snippet- Convert keras model to tflite

Conversion of keras model is done by loading the keras model in variable "model". Using from_keras_model.convert, the original keras model can be converted into tflite compatible format. This format is saved by simple file write syntax.

```
In [20]: interpreter= tf.lite.Interpreter(model_path= 'model/mainModel.tflite')
         interpreter.allocate_tensors()
         input_details= interpreter.get_input_details()
         output_details= interpreter.get_output_details()
```

Figure 18 Code Snippet- Load tflite model

A tflite model can be loaded in the code suing the commands shown in the above image. It stores, input and output tensors in separate variables. All this makes the prediction process faster.

8) PREDICTING IMAGES WITH TRAINED MODEL

Prediction can of the image can be done using the keras h5 model or tflite model. Preprocessing is required in both cases to before passing the image for prediction to the trained CNN model. Predictions are done as per explained in previous sectins.

```
#     interpreter.set_tensor(input_details[0]['index'], processFrame)
#     interpreter.invoke()
#     pred= interpreter.get_tensor(output_details[0]['index'])

    pred= model.predict(processFrame)
```

Figure 19 Code Snippet- Prediction using trained model in tflite and keras format

The commented part of the code shows prediction of image done using the tflite model. The uncommented command is used to predict using the keras h5 model. Both these commands give a value ranging from 0-1 for each individual neuron in output layer. The neurons in output layers corresponding to each individual gesture label. The value assigned to each neuron indicates that how likely the image can be classified to the corresponding label of the neuron.

The neuron having highest value and its corresponding label is considered as the label predicted by the trained CNN model.

9) TESTING THE MODEL

```
testDir1 = "Set_Similar_to_Train"
testDir2 = "Abhiraj_Simple_Test"
testDir3 = "Abhiraj_Different_Background_Test"
testDir4 = "Abhiraj_Extreme_Translation_Test"
testDir= [testDir1,testDir2,testDir3,testDir4]
```

Figure 20 Code Snippet- Test Directories

For testing the trained model, directories are set containing different training sets. Images from all the directories and its subfolders are loaded and passed on to the CNN model for prediction. A confusion matrix is created based on the prediction done by the CNN model and actual labels.

In [11]:

```
#Start predicting the test folders and update corresponding Confusion matrix
#Load Test Folder
for i in range(0,4):
    print(testDir[i])
    #gesture in corresponding test directory
    for subs in os.listdir(testDir[i]):
        print(subs)
        #Load images in gesture folder and predict and update confussion matrix
        for images in os.listdir(os.path.join(testDir[i], subs)):
            img = cv2.imread(os.path.join(testDir[i], subs, images))
            prosimg = preProcess(img)
            pred = getPredict(prosimg, interMain, inputMain, outputMain)
            predArray[i][dectFolder[subs]][np.argmax(pred[0])] = predArray[i][dectFolde
r[subs]][np.argmax(pred[0])] + 1
    print("")
```

Figure 21 Code Snippet- Predicting each image in all folders one by one

The above code snippet loads image from the labeled folders and preprocesses it and predicts it. The prediction is appended to the confusion matrix. This confusion matrix is plotted using seaborn package.

9.1) Test set 1- 200 Images from Main Training Dataset

Figure 22 Sample Test Set 1

This testing set consists of 20 images in each gesture folder. It is taken from the main dataset having gestures of all volunteers.

```
In [13]:
```

```
#Confussion Matrix for test set 1
confMat(predArray[0], lables)
```

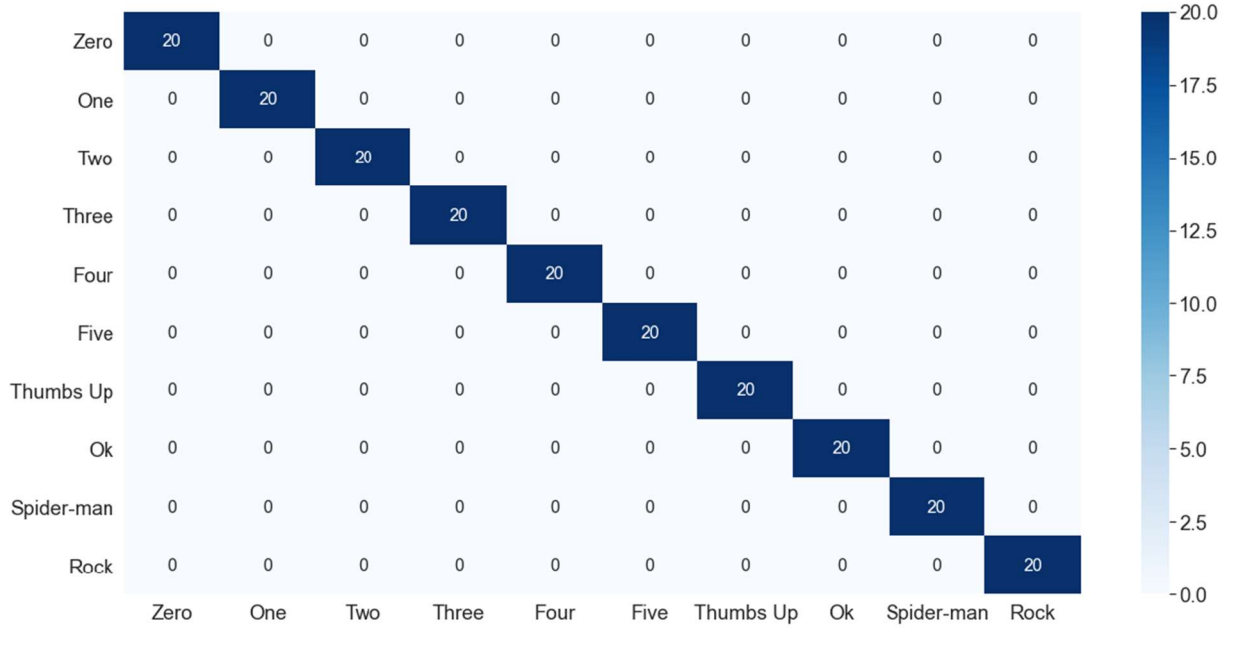

Figure 23 Confusion Matrix for Test Set 1

The above image shows that the CNN model was trained perfectly such that it can detect images related to original dataset perfectly.

9.2) Test set 2- 2000 Images Simple Set

Figure 24 Test Set 2

This test set contains images of gestures with less translation or rotation and less noise. This set tries to clearly display each individual gesture so that model should not have trouble classifying it. This dataset was not used for training the CNN model.

```
In [14]:

#Confussion Matrix for test set 2
confMat(predArray[1], lables)
```

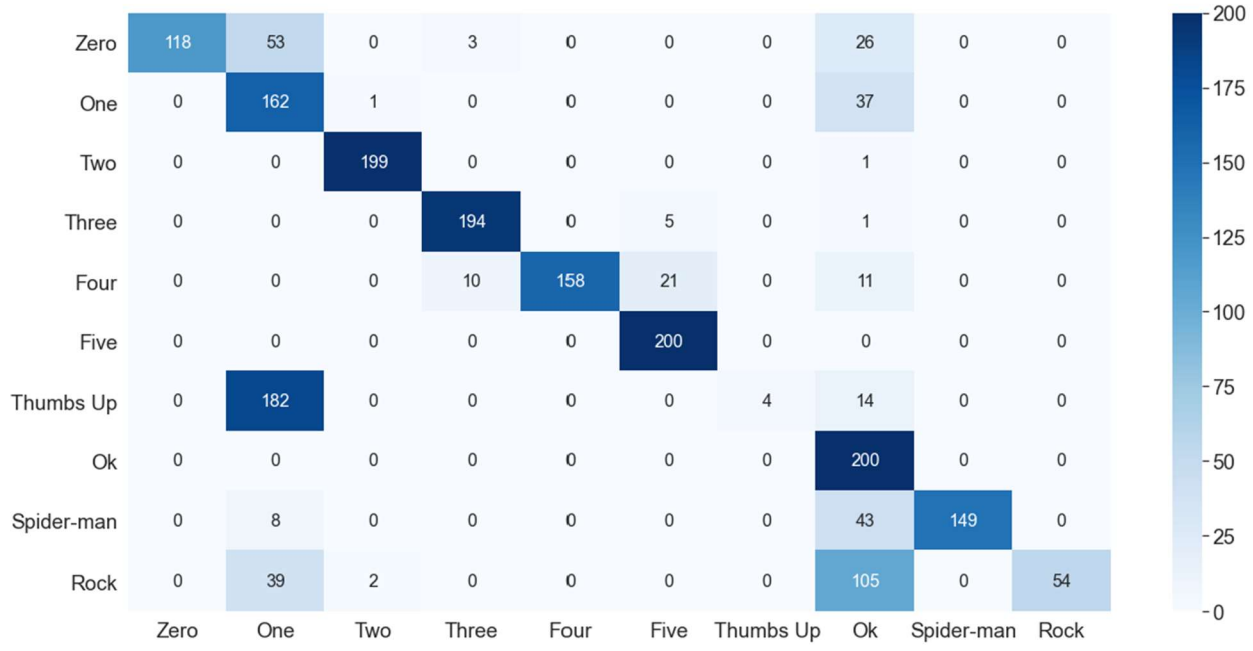

Figure 25 Confusion Matrix for Test Set 2

The confusion matrix shows that the model did very good job predicting gestures- one, two, three, four, five, ok and spider-man correctly. These gestures were predicted correctly for more than 75%. This model did fair job predicting zero gesture. It only predicted around 60% of zero gestures correctly. The model failed to predict gesture thumbs-up and rock for more than 75%. 182 images out of 200 of thumbs up were predicted as one. And 105 images of rock was predicted as ok.

9.3) Test set 3- 2000 Images Complex Set

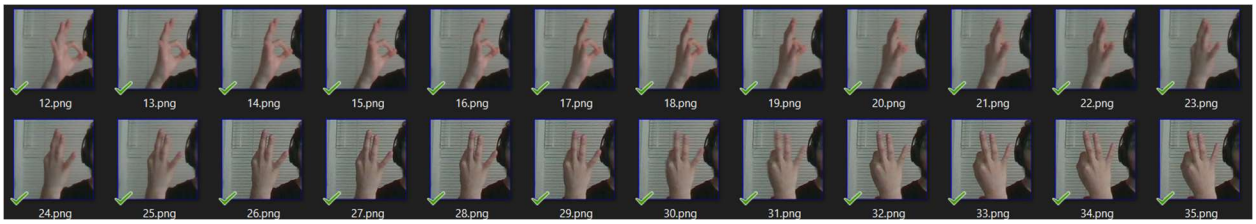

Figure 26 Test Set 3

This test set aims to test the trained model for high translation of hand gestures. As seen from the above image, the ok gesture is rotation to extreme showing back of the hand.

There is high possibility that the model may not work as required but helps to understand the accuracy of the trained model.

```
In [16]:
#Confussion Matrix for test set 4
confMat(predArray[3], lables)
```

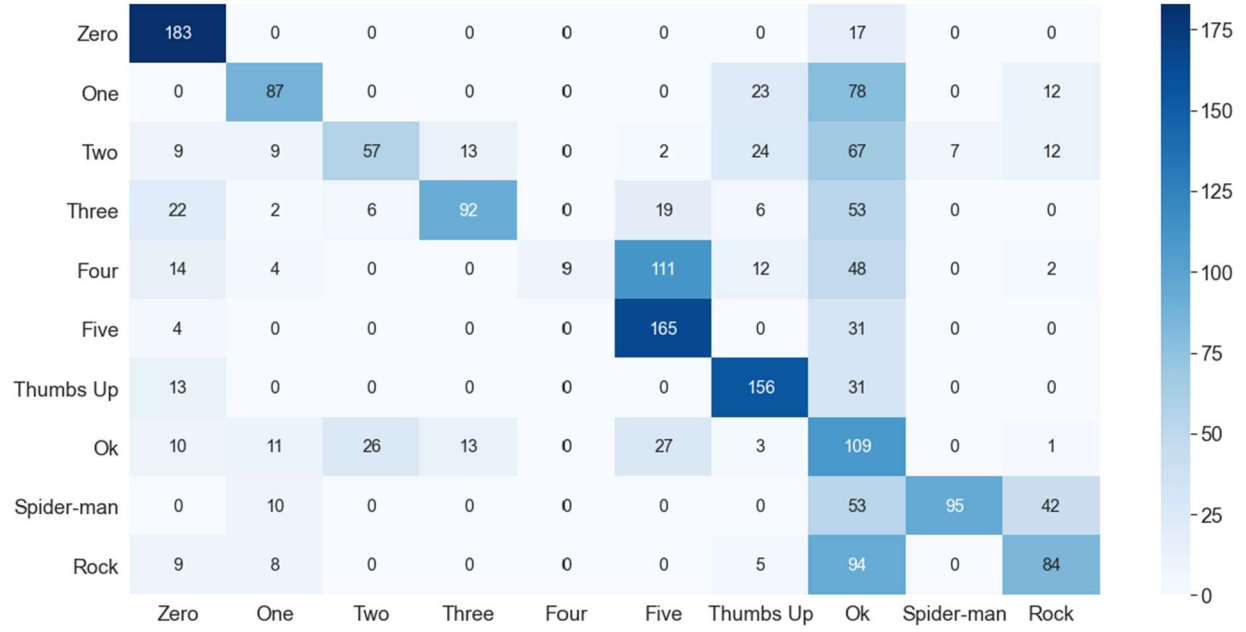

Figure 27 Confusion Matrix for Test Set 3

From the confusion matrix, it is seen that the model has trouble predicting most of the images but performs acceptable in prediction gestures zero, five, thumbs up and ok. Other gestures predictions were near 50% or less than 10% for some gesture. But it performed much better than expected.

9.4) Test set 4- 2000 Images Noisy Set

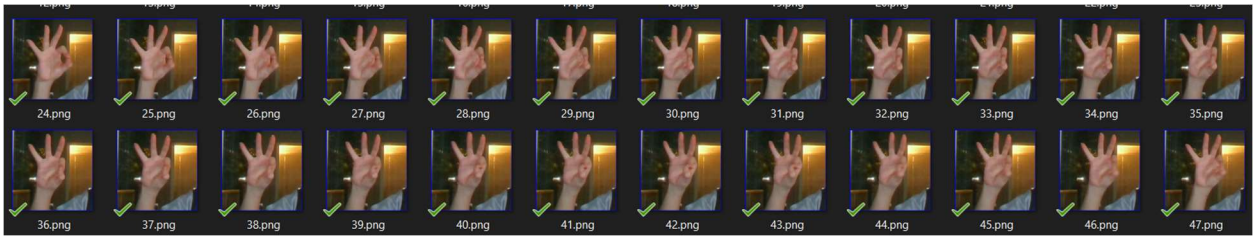

Figure 28 Test Set 4

This test set has images with noisy background. As shown in the above image, there is yellow light source in the background. The model was tested for its performance with

new and noisy background. The trained model is supposed to perform horrible predicting this dataset.

In [15]:

```
#Confussion Matrix for test set 3
confMat(predArray[2], lables)
```

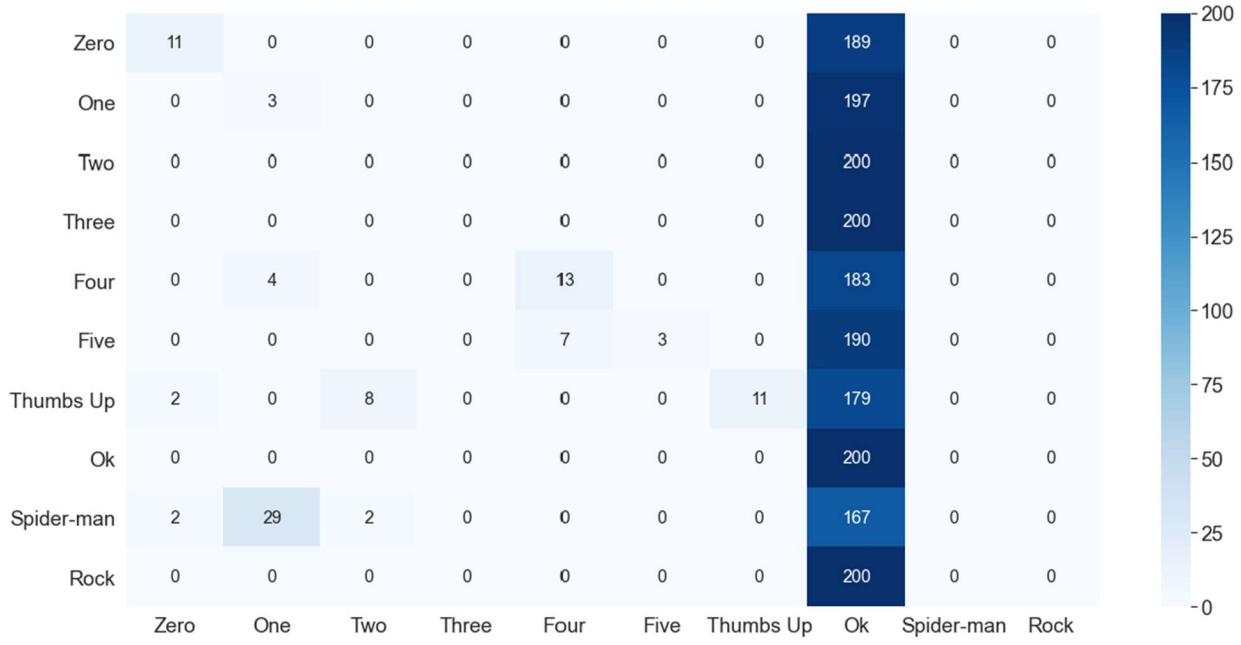

Figure 29 Confusion Matrix for Test Set 4

As seen in the confusion matrix, the model is not able to predict any of the images correctly. It predicted almost all images as thumbs up. It was expected for the model not to be able to predict the images correctly. To improve this, training of the model should have images from different noisy background. This can help the CNN model to extract more relevant features to the required gesture class.

9.5) With Live Video

Testing Model with live video

In [19]:
```
#The value in the bracket can change as per the camera in use
vid= cv2.VideoCapture(1)
```

In [20]:
```
interpreter= tf.lite.Interpreter(model_path= 'model/mainModel.tflite')
interpreter.allocate_tensors()
input_details= interpreter.get_input_details()
output_details= interpreter.get_output_details()
```

In [21]:
```
while True:
    ret, frame= vid.read()
    if ret==0:
        print("No Camera Detected, try changing port number")
        break

    shape_fr = frame.shape
    start_pt = (int(shape_fr[1]/16),int(shape_fr[0]/4))
    end_pt = (int(shape_fr[1]/2-shape_fr[1]/16),int(shape_fr[0] - shape_fr[0]/
4))

    roi = frame[int(shape_fr[0]/4):int(shape_fr[0] - shape_fr[0]/4),int(shape_
fr[1]/16):int(shape_fr[1]/2-shape_fr[1]/16)]
    processFrame= preProcess(roi)
    frame = cv2.rectangle(frame, start_pt, end_pt, (255,0,0), 2)

#     interpreter.set_tensor(input_details[0]['index'], processFrame)
#     interpreter.invoke()
#     pred= interpreter.get_tensor(output_details[0]['index'])

    pred= model.predict(processFrame)

    gest = lables[np.argmax(pred[0])]
    cv2.putText(frame, gest, (50,50), cv2.FONT_HERSHEY_COMPLEX, 1, (0,255,0),
2)
    cv2.imshow('Video', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

vid.release()
cv2.destroyAllWindows()
```

Figure 30 Code Snippet- Testing CNN Model with Live Video

The trained model can be tested with live video as well. The first line selects and sets the camera to be used to get live feed.

In the while loop, the code runs until user presses ‘q’ to exit the loop. The first line takes image from the camera. Region of interest is clipped and stored in another variable. The ROI is passed to preprocess function. The processed image is acquired and passed on to model for prediction. The model then predicts some value. Depending

on the prediction the trained CNN model has made, corresponding gesture is printed on the screen indicating that our trained model has predicted that gesture.

To test the model more rigorously, I tested it by trying it to predict gestures present in real time video feed. I used the last part of the code snippet to test the model with video feed.

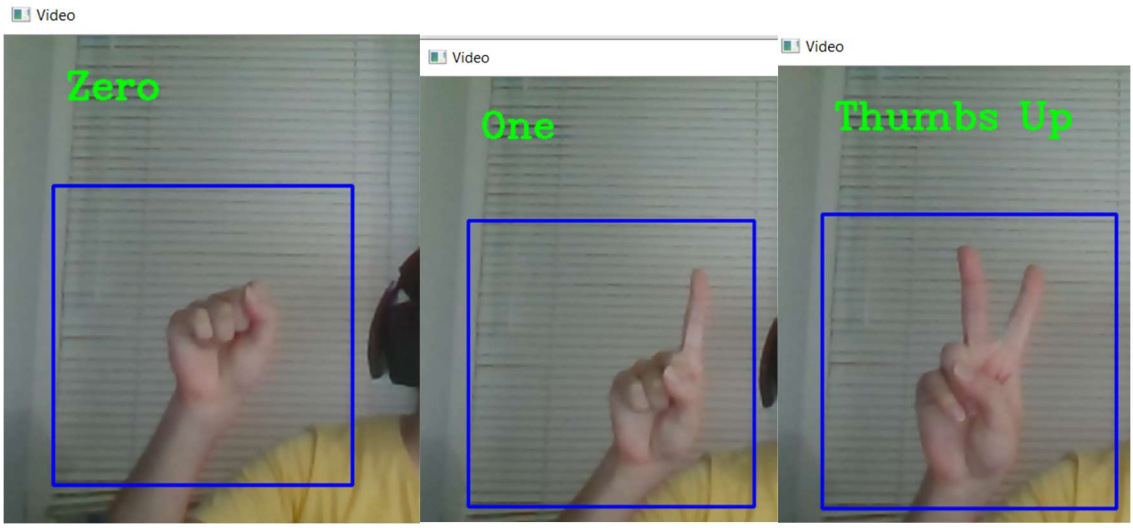

Figure 31 Prediction in Live Video -1

From above images we can see that the model is unable to predict the gesture two properly. While testing of gesture one, the model was able to predict it correctly about 70% of the time. The model was always correctly predicting gesture zero.

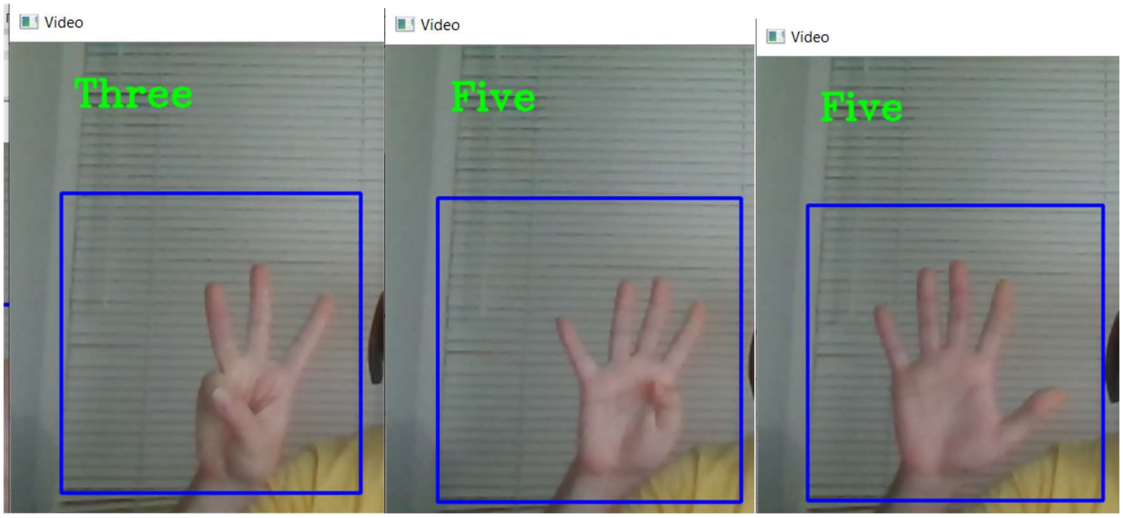

Figure 32 Prediction in Live Video -2

This trained model always correctly predicts gesture five but has a little trouble predicting gesture 4. About 50% of the time, gesture four is predicted properly. Model is trained to classify gesture 3 correctly.

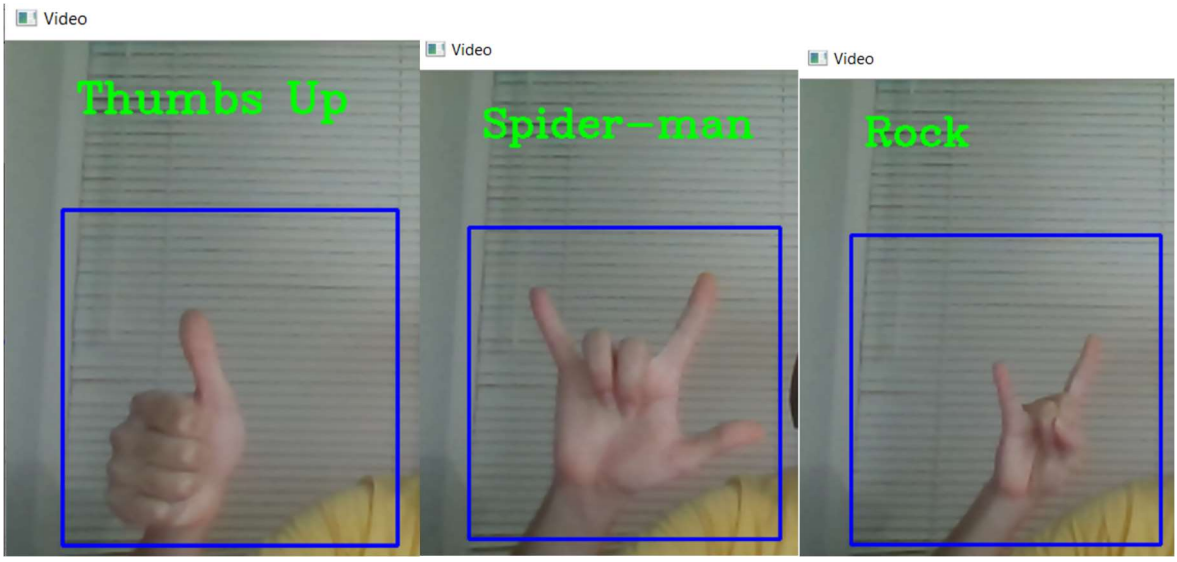

Figure 33 Prediction in Live Video -3

The CNN model also classifies the correct gesture thumbs up at all times. Model is good enough that is can differentiate between spiderman and rock gesture although they are very similar. Model sometimes mis predicts the rock gesture for spider man but most of the time it does a good job of predicting it correctly.

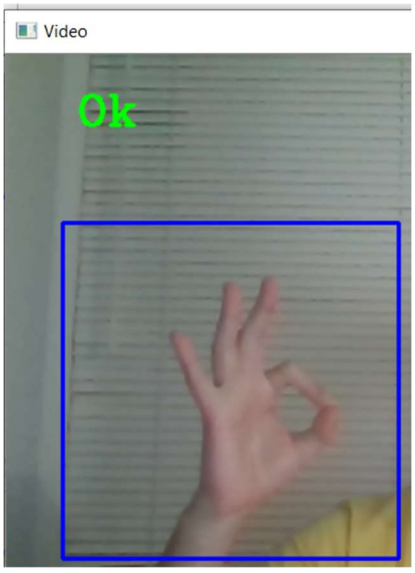

Figure 34 Prediction in Live Video -4

Model also always predict the Ok gesture correctly. The trained CNN model is a good way to classify different gestures. With a huge dataset with variety of images, a CNN model is always trained to predict almost all gestures correctly.

10) INMOOV ROBOT ARM

10.1) Structure

The arm is taken from the INMOOV robot. It has 6 Futaba S3003 servo motors. 5 motors control each individual finger, and the 6th motor controls the rotation of the palm. 5 motors are placed at the elbow part of the arm and the motor controlling fist movement is located near the fist itself. The motors controlling the fingers are connected to the fingers with strings. These stings stretch and curl in and out to fold or open the finger. The problem with finger is only complete rotation of servo show good gestures, any rotation in between has the fingers to be loose and show unwanted and unpredicted behavior of the finger.

10.2) Controlling Arm with Arduino Code:

```
#include <Servo.h>

//Object of all servos created.
Servo index;    //D2
Servo middle;   //D5
Servo ring;     //D3
Servo little;   //D6
Servo thumb;    //D4
Servo hand;     //D7
```

Figure 35 Code Snippet- Arduino Servo Objects

Servo library is available for Arduino to precisely control servos. This library provides servo class and related methods to rotate the servo. Here I created different servo objects for each finger.

```
//This function rotates the motor from 0-120 degrees for testing purpose
void movementTest(Servo *finger){
        for (size_t i = 0; i < 120; i++)
        {
                Serial.println(i);
                finger->write(i);
                delay(150);
        }
        finger->write(0);
}
```

Figure 36 Code Snippet- Finger movement from open to close

For self-test, the above function writes angle ranging from 0-120 degrees to each individual finger one by one. This code closes the finger to test its working. This Arduino code simply tests working and controlling the servos and finger motion and required rotation.

10.3) Controlling Arm with Python Script

Package named adafruid_pca9685 is available in python. This package helps to write commands to the PCA9685 module which will set desired frequencies and duty cycle for each servo motor individually. Using this package, I wrote a class for hand to calculate values to pass to the module to set the desired frequencies. It has set of variables corresponding to each individual motor angle. User assigns required angle to the corresponding servo variable and calls writeMod method to write the values to the motor.

```
def writeMod(self):
    self.angle = self.index
    self.f1.duty_cycle = int(self.calcValue())
    self.angle= self.middle
    self.f2.duty_cycle = int(self.calcValue())
    self.angle= self.ring
    self.f3.duty_cycle = int(self.calcValue())
    self.angle= self.little
    self.f4.duty_cycle = int(self.calcValue())
    self.angle= self.thumb
    self.f5.duty_cycle = int(self.calcValue())
    self.angle= self.fist
    self.f6.duty_cycle = int(self.calcValue())
    pass
```

Figure 37 Code Snippet- Calculate values to rotate servo motor to desired location

This writeMod method acquires angles to be written to the motor. Then it calculates duty cycle value using another calcValue method of this class. The calculated value is passed to the motor using the method duty_cycle provided by the adafruit package.

```
def five(self):
    self.index = 0
    self.middle = 0
    self.ring = 0
    self.little = 0
    self.thumb = 0
    pass

def thumbsup(self):
    self.index = 180
    self.middle = 180
    self.ring = 180
    self.little = 180
    self.thumb = 0
    pass
```

Figure 38 Code Snippet- Class Object to set angles for each gesture

There are more methods in this class similar to the ones in the above image. These methods assign angle values to the motor corresponding to its finger.

11) DETECTION AND REPLICATION

The trained CNN model is used with the motor control script. Depending on prediction done by the CNN model, the motor control part replicates the predicted gesture on to the robot arm using the methods class to set required angles to each finger and write the values to the motor.

```
predFrame, prosImg = preProcess(roiFrame)
prediction = getPredict(predFrame, interpreter1, input_details1, output_details1)

labels1 = ["Zero","One","Two","Three","Four","Five","Thumbs Up","Ok","Spider-man","Rock
gest = labels1[np.argmax(prediction[0])]

## Show Detected gesture in video window frame
cv2.putText(frame, gest, (50,50), cv2.FONT_HERSHEY_COMPLEX, 1, (0,255,0), 2)
cv2.imshow('Video', frame)
cv2.imshow('Prossed Image', prosImg)

## Select final gesture after detecting 5 consecutive same gesture
if preGest is not gest:
    predCount = 0
    preGest = gest
    pass
else:
    predCount = predCount + 1
    if predCount >= 5:
        finalGesture = gest
        print(finalGesture)
        pass
```

```
## Write the gestures to the motor
if finalGesture == 'Zero':
    hand.zero()
elif finalGesture == "One":
    hand.one()
elif finalGesture == "Two":
    hand.two()
elif finalGesture == "Three":
    hand.three()
elif finalGesture == "Four":
    hand.four()
elif finalGesture == "Five":
    hand.five()
elif finalGesture == "Thumbs Up":
    hand.thumbsup()
elif finalGesture == "Spider-man":
    hand.spiderman()
elif finalGesture == "Rock":
    hand.rock()
elif finalGesture == "Ok":
    hand.ok()
hand.writeMod()
pass
```

Figure 39 Code Snippet- Gesture detection and Replication

Live video frames are captured using the opencv videocapture methods. Preprocessing and prediction for the frame is done using the method described in previous sections. Label of the prediction is identified. A counter is set to record 5 consecutive predictions of the same gesture. During runtime, there can be fast change in hand gestures and the model will produce different prediction for these gestures. The fast change will cause the motors change its angles very fast and can cause damage to the motor. Therefore, the counter ensures stability in gesture before it is replicated onto the arm.

After achieving stable prediction, corresponding method of class to set fingers to corresponding gesture is called. Once angels are set, writeMod method is used to write the values to servo motors.

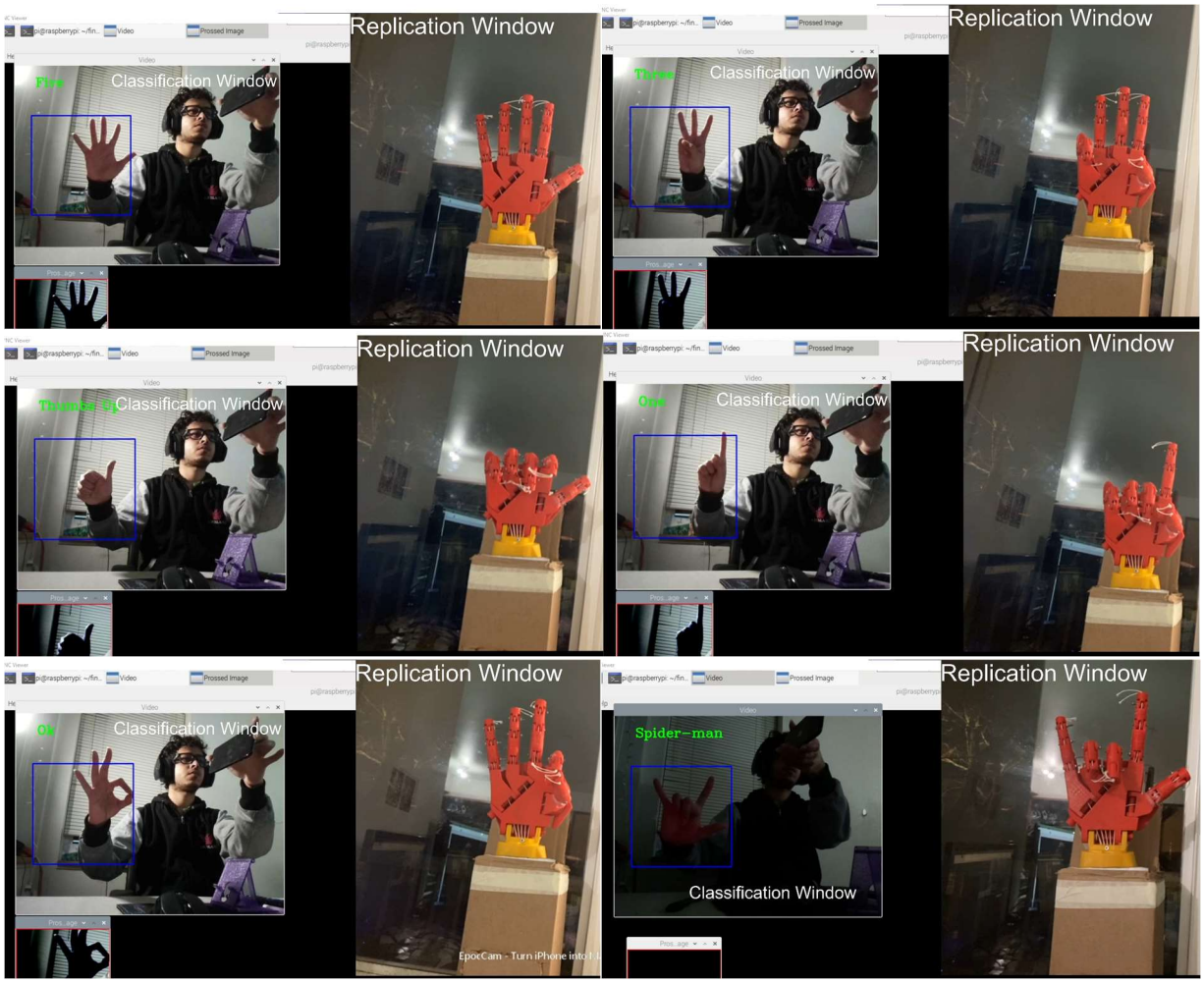

Figure 40 Result of Detection and Replication

The result images show two windows. User input gestures and the corresponding prediction done by CNN model is shown in the left window. The right window shows the same gesture replication done on the robotic arm. The results also showed that there were some gestures like zero and Rock which were not recognized properly and working prediction was replicated on the arm. Out of 10 gestures, model was able to predict 7 gestures correctly for most of the time and those gestures were also replicated correctly on the arm.

12) CONCLUSION AND FUTURE POSSIBILITIES

12.1) Improving Arm Structure

The string on the motor slip out of the ring which causes loose tension in the string and therefore failure in moving the fingers accurately. Any middle values of servo motor angles do not keep the string tight which causes the fingers not to have precise movement. This can possibly be corrected by connecting springs to back of the finger. It will ensure that the string will have enough tension at all motor angles and thus provide middle values of rotation as well.

12.2) Improving Classification

A twostep classification can be implemented to possibly improve the classification using current dataset. From confusion matrix, we can find the gestures that are being mis predicted. When first model mis predict a gesture, the image is passed to a second model which is trained to classify between actual gesture and mis predicted gesture. Thus, provide a better classification result.

12.3) Other Methods of Controlling the Arm

Instead of image processing to recognize gestures from user and to replicate it on the arm, signals from brain activity can also be used to control the arm. Knowing the areas of brain responsible for moving the actual hand can help to map it to control the robot arm as well. This will surely help in developing good prosthetic limbs.

12.4) Conclusion

In this project, I was able to classify 10 different gestures almost accurately by training and using Convolutional Neural Network Model. The movements of fingers were studied, and servo motor control was done with two different methods. Arduino was used for understanding the motor relation with fingers. Motor control in python was done to replicate the gestures detected during run time.