2020

Sign Language Recognizer

FYP REPORT 6 July, 2020

Authors: Muhammad Shoaib 2016-UET-NML-CS-31

Mahad Asif 2016-UET-NML-CS-10

Supervisor: Dr. Junaid Akhtar

Co-supervisor: Dr. Bacha Rehman

Department of computer science

Namal Institute, Mianwali

TABLE OF CONTENTS

INTRODUCTION

The project mainly belongs to deep learning (sub-domain of machine learning) involving neural networks to develop a predictive model trained on the collected dataset with its interactive android user interface in the form an android app. The aspect of the project is related to sign language used by the **deaf** and the **mute** people. The system will use the visual hands dataset based on American Sign Language (most commonly used sign language worldwide) and interpret this visual data into textual information on the word-level to be displayed on an android screen. While running an app, each meaningful text getting translated from its corresponding hand gesture will be displayed on the mobile screen. This will make the normal person able to communicate with that disabled person while recording the video of that disabled person distant less than a meter from him/her.

PROBLEM STATEMENT

Sign language is very important for the deaf and the mute people to communicate both with the normal people and with themselves. We as normal people tend to ignore the importance of sign language which is the mere source of communication for the deaf and the mute communities. These people are facing major downfalls in their lives because of these disabilities or impairments leading to their unemployment, severe depression, and several other symptoms. One of the services they are using for communication or for us to talk to them are the sign language interpreters. But, hiring these interpreters is very costly and therefore, cheap solution is required for resolving this big problem of these massive unfortunate people. A thorough analysis and survey to find a way to make these disabled communicable with themselves and the normal people have led to the breakthrough of the Sign Language Recognition System. The system targets to recognize the sign language and translate it into text for some meaningful communication.

OBJECTIVE

Our aim is to develop a sign language recognition system capable enough to translate the most commonly expressed hand gestures used by deaf or a dumb person into textual data. To make these disabled people communicable is our prime objective.

LITERATURE VIEW

In this detailed section, we have stated all the required work of machine learning and android domains learned so far for the development of this system. The concepts and practical guidelines include the basic understanding of sign language, some machine learning-related useful concepts, usage of deep

learning APIs and several python built-in packages, cloud services, several ways to deploy machine learning model on android and web, image annotation tools etc.

SIGN LANGUAGE

Sign Language is a natural language that serves as a pre-dominant gesturing language for the deaf and the mute communities. There are about 200-300 dialects of sign languages throughout the world today and out of them, some are evolving very rapidly amongst groups of deaf and mute children and adults. American Sign Language is one of the most predominant and the most commonly used language. Therefore, we are also working on the American Sign Language (ASL) because of its popularity and flexibility.

Sign language is a set of structured gestures and there are different types of gestures in the sign language. Considering the American Sign Language about which we have studied as yet, each gesture has some assigned meaning and strong rules of context and grammar may be applied to make the recognition tractable. The flexibility of the ASL allows it to incorporate different body parts movements. Different body parts involved in ASL makes the language more comprehensible, complete and flexible. The body parts involved are:

- Hand gestures
- Lips movement
- Facial expressions
- Arms movement
- Whole-body movement
- Shoulder movement
- Other body parts may also involve in rare cases

The body parts mentioned above play a vital role in making the ASL more comprehensive and meaningful. The most significant part that plays the most crucial role in making the gestures more expressive is the hands' movement. Hands movement involves fingerspelling which is the representation of letters of the writing system and also the numeral systems using merely the hands. ASL possesses a set of 26 signs termed as the American Manual Alphabet which can be used to drive the words of the English language. These 26 signs of English alphabets are interpreted through fingerspelling. But, how can we express the meaning through fingerspelling is shown below:

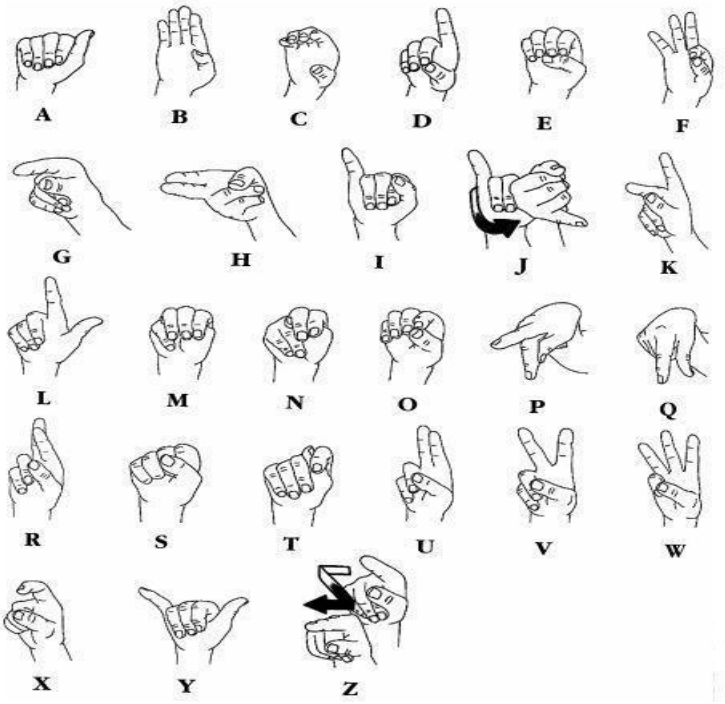


FIGURE 1: AMERICAN SIGN LANGUAGE ALPHABETS

STATIC AND DYNAMIC HAND GESTURES

More importantly from our project's perspective, hand gestures are classified into two categories: static and dynamic. The static hand gesture is a particular hand configuration and pose represented by a single image. The dynamic hand gesture is a moving gesture represented by a sequence of images like 'J' and 'Z' in the above figure. But in this project, we are working merely on the static hand gestures.

CONVOLUTIONAL NEURAL NETWORKS

We have studied the convolutional neural networks (CNNs) and how they have become more efficient, fast and robust as compared to the standard and simple artificial neural networks (ANNs) composed of dense layers. Convolutional Neural Networks involve the convolutional layers incorporating some hyper-parameters: padding, strides, kernel size, etc. The fast and efficient feature extraction is done by these layers using some other useful operations (Max Pooling, Dropout, etc.) to exterminate the trivial feature information from the matrices passed through this convolutional process. How the information is processed in these hidden layers and how some hyper-parameters affect the performance of the model by altering their values on hit and trial basis is what we have learned in CNNs.

OpenCV

OpenCV is an open-source computer vision library provided by python which aims to provide useful functions for image processing. We have learned how to use the core operations provided by the library e.g. reading, writing, displaying the images, drawing shapes on the images, demarcating the object detected in the whole image through bounding boxes, drawing convex hull through contours (boundary points of object of interest), applying bit-wise operations in required conditions, conversion, merging and separating of various color space channels and several other main functions as well. In our project, we are reading, writing and rescaling the images and videos using this library.

Transfer Learning

Transfer learning refers to the situation when the knowledge learned in one task or domain is reused to improve the generalization in other settings. From machine learning's perspective, it can be defined as reusing the saved weights of any pre-trained model to improve the accuracy or to train your own model. In order to use the weights of any pre-trained model e.g. VGG16, inceptionV3, we have changed the input layer, output layer and have added some fully-connected layers at the bottom of these ready-made architectures to make the model more effective and more suitable for use in accordance with the nature of our dataset. We have used some pre-trained models provided by *Keras.applications* package in our case of sign language recognition. The pre-trained models we have used are:

- VGG16
- InceptionV3
- ResNet

Statistical evaluation of these models is done in later sections for clarification about which one of these best suits our use case.

Object Detection APIs

There are several object detection APIs available publicly. These APIs incorporates several pre-trained models trained on their corresponding different visual datasets. These APIs were made open-source to be used by data scientist, machine learning or deep learning engineers to solve the object detection problems which is the main aspect of computer vision using the deep learning approaches. There are two of the most famous object detection APIs which we encountered to solve our problem:

- TensorFlow Object Detection API
- You Only Look Once (YOLO)

Public Image Datasets

There are several image datasets which were made available for public use for preparing object detection models and for other image processing tasks as well. We tried two the pre-trained models trained on these two datasets:

COCO

Is a large-scale object detection, segmentation and captioning dataset. COCO contains about 80 different types of most commonly used objects. It contains about 1.5 million object instances and more importantly to note that some pre-trained models available in the TensorFlow object detection API and some YOLO models are also trained on this dataset.

IMAGENET

ImageNet is a large collection of annotated images publicly made available for computer vision research. This large-scale collection of images is a critical resource for analyzing, training and testing the machine learning algorithms. There are round about 14 million images and 1000 categories in this dataset and this dataset is also used for large-scale visual recognition challenge competitions. The pre-trained models provided by *keras.applications* python package are also called complex functional models because these functional models are trained on the ImageNet dataset. These pre-trained models are capable enough to classify any image that falls into these categories of images.

CONSTRAINTS AND LIMITATIONS

There are some constraints we have imposed from the user's perspective since we are doing this project on the most basic level due to lack of expertise, time, and resources. Therefore, we tend to be more specific in some aspects:

LANGUAGE-SPECIFIC

Sign language is not a universal language, each country has its own sign language and even regions have their own dialects. But we are focusing on the well-developed and complete natural language and that is American Sign Language. Creating a proper sign language (ASL-American Sign Language in this case) translator is not the desired result at this point. This would incorporate advanced grammar, syntax structure understanding of the system which is expectedly outside the scope of this project. Moreover, ASL dictionaries contain 5000 to 6000 different signs in them and since we cannot interpret all these signs into a textual format, therefore, we aim to translate the most common signs used in everyday life.

It is important to note that ASL also includes the movements of other body parts as well but we are focusing on merely the hand gestures.

ENVIRONMENT-SPECIFIC

We are developing our system in specified environments since our system is not so flexible to produce good results in all the environments. Our system would be able to show flexibility in balanced lighting conditions e.g. a person sitting in a room using a laptop under balanced lighting conditions. High illuminance of light reflecting on the user's one side of hand will probably cause the system to produce

incorrect predictions. Therefore, the intensity of light should be balanced all over the place where a user is operating this system.

User's Distance from a camera

Based on the efficiency of the algorithm used in backend, produced results limit the distance of the disabled person from a camera. So, in our case, the machine learning algorithm is limiting the person to be distant at max half meter from a camera.

Limited Dataset and Resources

Due to the limited dataset and hardware resources, the project's limitations are defined accordingly. This has greatly reduced the scope of the project to more basic level.

Workflow

Building a machine learning model involves several trials and errors. Specifically, for those who are novices at the concept constantly tweak and alter their algorithms and models. During this time to make some right selection for the algorithm to use for building the machine learning model, challenges arise with handling data and tuning the architecture of the model. Therefore, the whole project workflow is divided into some iterations and each iteration is following the general steps that are required in any machine learning project. The following iterations are describing the tools and technologies that have been tried in building a real-time sign language recognizer.

Iteration 1

The iteration involves the idea of developing the sign language recognizer based upon merely the hands dataset. This iteration does not include the detection of hand in the whole image incorporating a person rather only to develop a model capable enough to recognize the hand sign through the hand image. The workflow involved during this iteration is divided into detailed general steps as below:

Data Collection

For most computer vision projects, the dataset is the most basic requirement and a quality dataset should possess consistency, diversity amongst the images (no redundancy), noiseless and good in number.

Most people start their sign language journey with the alphabets which is the most basic level of any sign language to learn. We have made the selection of American Sign Language (ASL) so, there would surely be the fingerspelling of ASL alphabets. In our case, there was no consistent and healthy dataset available on the internet so we did compile our own dataset. The dataset consists of images of American Manual Alphabets incorporating 26 different signs from a to z. Each alphabet has its own distinct gesture possessing its own structure and orientation.

The dataset was made in three different environments: room, sunlight, and fog. For the sake of clarification about the images, some sample images are as:

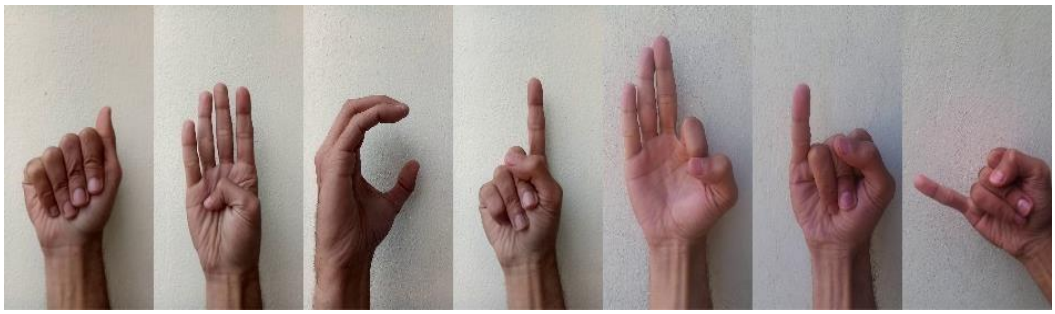


FIGURE 2: HAND SIGNS

For some storage scheme, we created 26 folders entitled from A to Z and each folder consists of its own 1200 images of signs incorporating 400 fog, 400 sunlight and 400 room images. Subsequently, there are 1200*26 images in the dataset. The directories containing these folders are treated as training and validation datasets for the model which will be explained later in this section.

HOW WE HAVE COLLECTED THE DATASET?

We made 1:00-1:25 minute video of each sign using our smartphone's camera in three environments as stated before. There are 26 videos in each environment. From these 1-minute videos, we have extracted 400-500 frames using the OpenCV python library.

DATA SPLITTING

The data used to build the final model comes from multiple datasets. There are three different purposeful datasets for any computer vision project to analyze, make some comparisons and improve the performance of the model. In particular, these three different types of datasets are used in different stages of the creation of any machine learning model. These three distinct datasets are stated below:

TRAINING DATASET

The training set is a dataset on which the model is trained for learning weights or features. Initially, the model is fitted on the training dataset and in our case specifically, 80 percent of the whole dataset is used for training dataset which is approximately 23000 images.

VALIDATION DATASET

The model is fitted on the validation dataset for the unbiased evaluation of itself during training. It validates the performance of the model that how well the model is learning its weights before it is used for the real-time testing on the testing dataset. In our scenario of sign language recognition, we used 20 percent of the dataset equivalent to 7600 images.

TEST DATASET

After the completion of training and validation phenomena, the test dataset is used for testing the model and to measure the goodness of how well the model is trained. For this, 780 samples are used for the test dataset since there are 30 test images for each sign alphabet. This self-generated test set was created in order to measure the model's ability to generalize. More importantly, this test set is not collected from the 30800 images.

DATA PRE-PROCESSING

Data pre-processing involves the transformation applied to the data before feeding it to the model for training. As mentioned in the preceding section that videos made from the smartphone's camera were possessing full-HD (1920*1080) quality which is of high resolution. In order to keep the computations less complex and fast, the extracted 1920*1080 images of different gestures were rescaled into 120*72 following the same proportion of dimensionality. The images of hand gestures of 26 corresponding alphabets were captured with white wall as a background which seems to be nearly noiseless and consistent. Apparently, hand in all the images is of merely one user which is not good because in this case, the dataset should constitute multiple user's hands including various skin colors, sizes, structure, and shape. In short words to summarize the above discussion, these 30800 images were rescaled and labeled for inputs into the model to prepare them for the succeeding phenomena.

KERAS FOR PRE-PROCESSING

Programmatically speaking, Keras provides a fast, short and efficient way to read images from the directory and convert into a typical Keras object format compatible for training, testing and prediction. This is all done using the ImageDataGenerator built-in class provided by Keras. This class automatically configures all the operations and then instantiates its own generator object of augmented image batches. The instantiated generator objects are then used as input parameters in the Keras model method that is fit_generator function used for training. Specifically, in our case, this package is used for the following pre-processing operations:

- Rescaling of the images (120, 72)
- Dataset splitting (training: 80%, validation: 20%)
- Defining batch size (equals 32)
- Defining the class mode (categorical, in this case)

TRAINING

The deep learning model is trained using the convolutional neural network architecture based on the dataset gathered as yet. Since the project requires the classification of the images, CNN is a go-to architecture. The neural network takes in input as multiple training and validation images which are processed in the convolutional hidden layers using the weights that are adjusted during the training phenomena. Then, the model spits out the prediction at the last output layer of the architecture. During all this complex training process, there's no need to specify what patterns our model should learn during

the transmissions in the hidden layers because neural network has the capability to learn on its own. This is all that what is going to interpret in this detailed section.

MODEL ARCHITECTURE

The model used is VGG16 which is a functional model that has allowed us to create a more flexible model for recognizing different signs of corresponding different hand gestures on the alphabetic level. The internal structure of model is described in the next section:

ANATOMY OF VGG16

The name VGG-16 came from the fact that it consists of 16 hidden layers including 16 Convolutional layers and 2 dense (fully-connected) layers along with Max Pooling layers and Activation layers. The architecture is divided into different blocks entitled as Conv-1, Conv-2, Conv-3, Conv-4 and Conv-5. Each block incorporates several convolutional layers and one Max Pooling layer at the end. Conv-1 consists of two convolutional layers and 64 filters (kernels or feature detectors), Conv-2 also consists of two convolutional layers with 128 filters, Conv-3 contains 4 convolutional layers with each constituting 128 filters and Conv-4 and Conv-5 are also made of 4 convolutional layers each and have 512 filters. The model is using a filter size of 3 by 3 in all the convolutional layers globally. Pictorially, the architecture can be shown below:

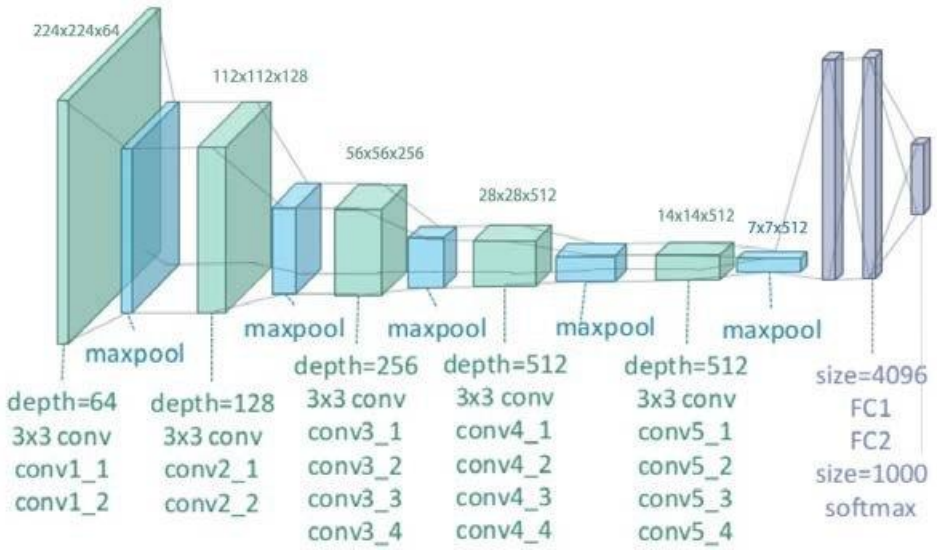


FIGURE 3: VGG CONVOLUTIONAL ARCHITECTURE

The architecture shown is an absolute VGG16 architecture. But, intuitively with some predictive and statistical analysis, some modifications had to be made in the architecture to make it usable and more suitable for the use case. Changes made are described in next detailed section.

CHANGES MADE IN VGG16

Intuitively, some dense layers were added at the bottom of this architecture just before the last output layer and after the dense layer of 4096 nodes to make this architecture look more feasible and balanced to the scenario accordingly. The 5 dense layers added are regular layers of 512, 256, 256, 128

and 128 neurons or nodes. Changes were made in the input and the output layers according to the nature of the gathered dataset. By default, the built-in VGG package provided by Keras accepts (224, 224, 3) input shape by default indicating the height, width, and channels of the input images on the first layer. Accordingly, to the case, modifications made in input shape parameter on the input layer are (120, 72, 3) and thus, reducing the complexity of architecture because the size of the input matrix is reduced and increasing the speed of mathematical computation of underlying activities involved during the training of the model. In the case of the output layer, it is mentioned earlier that VGG-16 is pre-trained on 1000 classes. Therefore, the output layer is containing 1000 nodes but our storyline acquires the output layer to have 26 nodes. Since the American Manual Alphabet involves 26 different signs or hand gestures representing 26 English alphabets. Thereby, to make the model compatible and consistent with the case, reduction was made in the count of output nodes from 1000 to 26. Shortly, these are the major changes made to the VGG-16 model to make it usable and intuitively, the best-modified VGG-16 for the scenario.

MODEL COMPILATION

Once, the network or architecture for the model to train is defined, compilation is done before the training begin. Compilation transforms the architecture consisting of a sequence of several layers filled up with some hyper-parameters into a series of matrices ready for transformation depending on how Keras API configures this underlying computation in the backend. In short words, compilation configures and prepares the model for training. Compilation requires certain number of parameters to be specified which includes the optimization algorithm to train the network and loss function used to evaluate the model and this loss value is minimized by the optimization algorithm. Considering the case, stochastic gradient descent (SGD) is applied for optimization and 'categorical_crossentropy' as a loss function because the model is based on categorical outputs. The model compiled to make it ready for the training phenomena is represented in the succeeding section.

MODEL SUMMARY

The filled-up prepared model architecture for precise summarization is shown as:

Layer (type)	Output shape	parameters #
input_1 (InputLayer)	[(None, 120, 72, 3)]	0
block1_conv1 (Conv2D)	(None, 120, 72, 64)	1792
block1_conv2 (Conv2D)	(None, 120, 72, 64)	36928
block1_pool (MaxPooling2D)	(None, 60, 36, 64)	0
block2_conv1 (Conv2D)	(None, 60, 36, 128)	73856
block2_conv2 (Conv2D)	(None, 60, 36, 128)	147584
block2_pool (MaxPooling2D)	(None, 30, 18, 128)	0
block3_conv1 (Conv2D)	(None, 30, 18, 256)	295168
block3_conv2 (Conv2D)	(None, 30, 18, 256)	590080
block3_conv3 (Conv2D)	(None, 30, 18, 256)	590080
block3_pool (MaxPooling2D)	(None, 15, 9, 256)	0
block4_conv1 (Conv2D)	(None, 15, 9, 512)	1180160
block4_conv2 (Conv2D)	(None, 15, 9, 512)	2359808
block4_conv3 (Conv2D)	(None, 15, 9, 512)	2359808
block4_pool (MaxPooling2D)	(None, 7, 4, 512)	0
block5_conv1 (Conv2D)	(None, 7, 4, 512)	2359808
block5_conv2 (Conv2D)	(None, 7, 4, 512)	2359808
block5_conv3 (Conv2D)	(None, 7, 4, 512)	2359808
block5_pool (MaxPooling2D)	(None, 3, 2, 512)	0
global_average_pooling2d_1	(None, 512)	0
dense_1 (Dense)	(None, 512)	262656
dense_2 (Dense)	(None, 256)	131328
dense_3 (Dense)	(None, 256)	65792
dense_4 (Dense)	(None, 128)	32896
dense_5 (Dense)	(None, 128)	16512
dense_6 (Dense)	(None, 26)	3354

FIGURE 4: MODEL SUMMARY (VGG16)

The architecture is already discussed in the preceding section but there are some other important concepts to take into brief analysis. But there are some other parameters as well in the figure: output shape of the resultant matrix after each layer and the number of parameters present in each layer. Considering that we know how the convolutional process takes place at the convolutional layers of the given model. It is important to note that the size of the output matrix is decreasing with increasing dimensionalities after the Max Pooling operation takes place. This is due to the fact that dimensionalities of the output matrices are indicating the number of feature detectors or kernels used in the convolutional layer. Max Pooling is the cause for the reduction in the size of the output matrices. Max Pooling extracts the important features or weights which model needs intuitively and exterminates other sets of numbers or weights considering them as trivial information from the matrix. The number of parameters involved in the Max Pooling layer is 0 and it is due to the general fact that Max Pooling does not possess any weights. It rather decreases the parameters in the convolutional layers. Therefore, some people call it an operation of feature extraction, not a layer. It is important to note some types of parameters learned before and after the training:

NON-TRAINABLE PARAMETERS: are the already learned weights by the pre-trained model and we are using these learned sets of numbers having the count: 14714688.

TRAINABLE PARAMETERS: are the connections of dense layers added by us, are having the count equal to 512538.

MODEL INITIATION FOR TRAINING

At this stage, the model instance is fitted to the *fit_generator* function to start the training process. This function trains the model for a fixed number of epochs (iterations on a dataset) using training and validation generators incorporating the pre-processed images and some other required attributes. The model had been trained for 13 epochs in almost 3 hours and took 481 steps to complete one epoch. There are certain approaches to analyze the trained model and makes some improvements in it.

Keras provides the capability to register callbacks when training the deep learning model. One of the most important callbacks that are for all deep learning models is the history callback. It records all the statistical analysis during each epoch and this includes training loss and training accuracy after trained on the trained dataset and validation loss and accuracy after cross-validated on the validation dataset. History object is returned after the completion of the 3-hours long training process. We have utilized the graphical approach representing the whole training history of the model with epochs count on the x-axis and other parameters on the y-axis using the metrics stored in the history object. These parameters include validation loss, validation accuracy, training loss, and training accuracy. The following graphs are providing useful indications about how our model is trained. We can see from graphs that accuracies in the first graph and losses in the second graph are converging towards each other instantly just after the second epoch completion. There are no signs of overfitting or under fitting because lines are converging smoothly in a horizontal manner towards the end in the second half which is an indication that the model has learned weights or important features extracted from the images quietly well. The behavior can be seen graphically as follows:

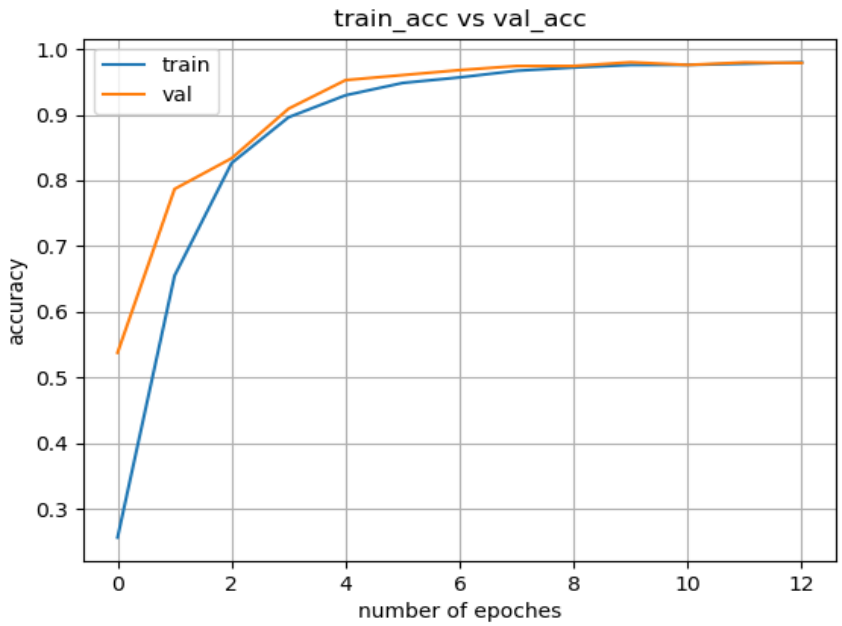


FIGURE 5: VGG ACC GRAPH

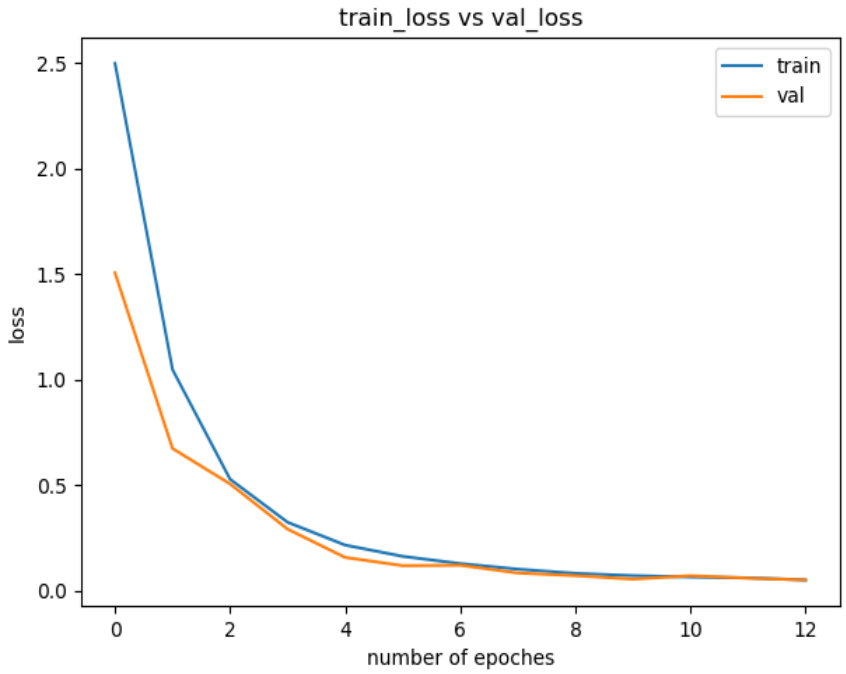


FIGURE 6: VGG LOSS GRAPH

After having the graphical analysis on the validation performance of the model, the next general step is to test the model on unforeseen data. This random and unforeseen evaluation will test the real

intelligence of the model that how well it has learned from the input information. This unforeseen behavior is explained in the next section.

TESTING

After the model is trained, testing is required to measure the real performance of the model on unseen data which the model has not encountered yet. Different programmatical approaches are available to test the performance of the trained model. The statistical evaluation is done using two methods and that are confusion matrix and classification report. These two approaches are explained in detail in next two succeeding sections.

CONFUSION MATRIX

To describe the performance of the classification model on a test dataset, confusion matrix is used as shown under:

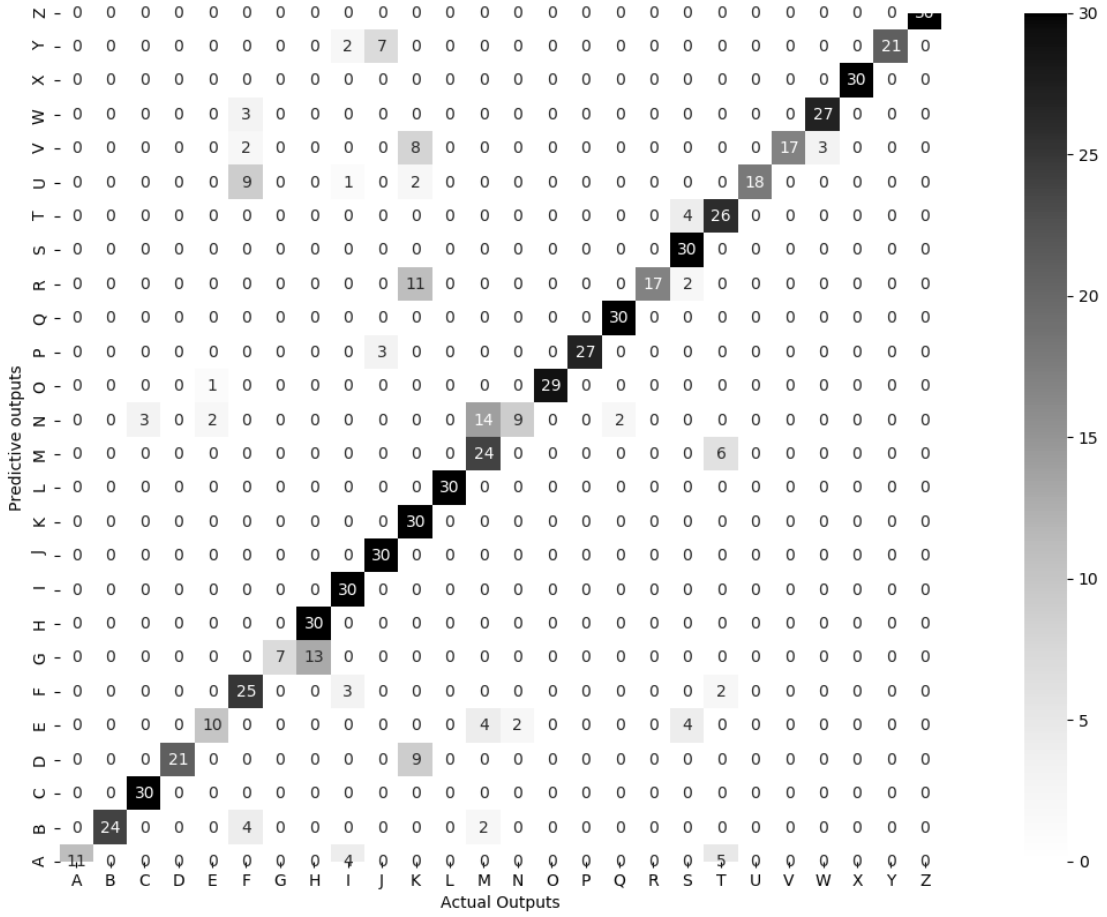


FIGURE 7: VGG CONFUSION MATRIX

The above error matrix or table allows the statistical visualization of the testing performance of the trained model. Actual outputs and predictive outputs are plotted as x-axis and y-axis. It is allowing easy identification of confusion between the classes e.g. one class: N is commonly labeled as the other class: M thereby, confusion between these two classes is very high: 14. The black color is an indication of an ideal case (no confusion) while white color is indicating the worst case (high confusion). Statistically, there are some signs which are predicted ideally representing full black diagonal boxes. As it is explained earlier that there are a total of 30 test images for each sign alphabet for predictions. Therefore, alphabets possessing an ideal testing result are containing 30 in their corresponding black boxes diagonally positioned in the matrix. The goodness of prediction of any sign class is proportional to the dimness of back color. There is a total of 10 ideal cases of categorical predictions having 0 confusion rate, classes performing badly (less than 50%) have the count of 10 are an indication of high confusion rate and other predictions can be termed as normal confusion rates.

CLASSIFICATION REPORT

The classification report is showing the representation of the main classification matrix on a per-class basis. This visual report is giving better and deeper intuition about the behavior of the classifier showing the functional weaknesses of the trained model in many analytical aspects.

	precision	recall	f1-score	support
A	1.00	0.55	0.71	20
B	1.00	0.80	0.89	30
C	0.91	1.00	0.95	30
D	1.00	0.70	0.82	30
E	0.77	0.50	0.61	20
F	0.58	0.83	0.68	30
G	1.00	0.35	0.52	20
H	0.70	1.00	0.82	30
I	0.75	1.00	0.86	30
J	0.75	1.00	0.86	30
K	0.50	1.00	0.67	30
L	1.00	1.00	1.00	30
M	0.55	0.80	0.65	30
N	0.82	0.30	0.44	30
O	1.00	0.97	0.98	30
P	1.00	0.90	0.95	30
Q	0.94	1.00	0.97	30
R	1.00	0.57	0.72	30
S	0.75	1.00	0.86	30
T	0.67	0.87	0.75	30
U	1.00	0.60	0.75	30
V	1.00	0.57	0.72	30
W	0.90	0.90	0.90	30
X	1.00	1.00	1.00	30
Y	1.00	0.70	0.82	30
Z	1.00	1.00	1.00	30
accuracy			0.82	750
macro avg	0.87	0.80	0.80	750
weighted avg	0.87	0.82	0.81	750

FIGURE 8: VGG CLASSIFICATION REPORT

Here, support column is showing the count of test images as per class e.g. all classes constitute a total of 30 test images except 'A', 'E' & 'G' which constitute 20 classes each. F1-score is the mean of precision and recall such that the best score is 1.0 and the worst is 0.0. The ability of the classifier to find all positive instances (correct predictions) is defined by the recall column numerically. In recall column, 10 classes (C, H, I, J, K, L, Q, S, X, & Z) which possess 1.00 recall value are the outcomes predicted ideally by our model as was the case visualized in confusion matrix. The report is showing the testing accuracy to be 82 percent which is the real predictive result of our classifier.

ITERATION 2

Now, this iteration incorporates the goal for developing the sign language recognizer able to classify the hand gestures on word level as level. This is the major difference between the iteration 1 and iteration 2 and the difference is all about upgrading from the alphabetic level to word level. The comprehensibility and flexibility of an American Sign Language allowed us to enhance the machine learning

model to much better level to make it more usable, attractive and communicable for the disabled ones. The workflow of this iteration is same as of iteration 1, involving general steps required to develop any particular machine learning model. The steps in detail are as under:

DATA COLLECTION

In preceding iteration, dataset collection was made at alphabetic level but in this iteration, we made 8-10 class collection at word level. Most commonly used words were considered for collection that are easy distinguishable in their pose, structure and orientation when compared to each other.

As described in the sign language section about static and dynamic gestures and how these two types of gestures differ from each other. Because it is important to note that the 10 words collected here are all static. This implies that there exist some words in ASL that are describable using merely static images. There are no dynamic gestures considered in whole project because dynamic gestures classification requires some more computationally intensive technologies and architectures which goes beyond the scope of the project. So, the words collected are: 'give', 'have', 'mute', 'need', 'not', 'we', 'will', 'you' and 'are'.

The images collected from this dataset possess some plain and complex backgrounds. The plain backgrounds involve blue, green, orange, purple and some random backgrounds incorporating multiple obstacles. To get some more clarity, here's some sample images below:

FIGURE 9: NOT SIGN SAMPLES

All the images are incorporating the same sign and that is 'not'. Images are marginally distinguishable from each other regarding texture, color, background and as well as skin color. It is important to note that these collected images are of merely one human. But, difference in environment is making a clear impact on skin. These images are quite different as compared to previous iteration images which were containing only one type of background but now, videos are made in random environments to make the model more robust and predict in diverse environments as well.

For some storage scheme, 26 folders were created entitled from A to Z and each folder consists of its own 1600 images of signs incorporating 300 images in blue, purple, green, orange and some images in complex or random making up to the total of 1600 images for each class. Subsequently, there are

1200*9 images in the dataset. The directories containing these folders are treated as training and validation datasets for the model which will be explained later in this section.

How we have collected the dataset?

We made 1:00-1:25 minute video of each sign using our smartphone's camera in diverse environments as stated before. Nine input videos were made for each environment as there are 9 classes. From these 1-minute videos, 200-300 frames were extracted from each video to avoid the replication in the dataset. Because replicated images usually make the model over fit when there is very low dissimilarity index in the corresponding pixels when two different images are compared with each other. Thus, total collection was made up to 34200 number of images.

Data Splitting

As stated in the previous iteration phase that data splitting is done to predict the model's efficiency and robustness. Again traditionally, dataset is divided into three purposeful divisions as:

Training Dataset

In this iteration, 80 percent of the whole dataset is used for training dataset which is approximately 27360 images.

Validation Dataset

The remaining 20 percent dataset is all involved in validation phase which is done during the training phase. The 20 percent of the images becomes 34600 – 27360 = 6840.

Test Dataset

For the evaluation in the testing phase, some images (frames) were extracted from the input videos which weren't included in either training or validation sets. 200 images were collected from all the environments for each class and this made up to total 1800 testing images.

Data Pre-processing

As the videos were recorded from the smartphone's camera possessing full HD (1920 * 1080) images and to process these images by passing through convolutional neural networks is computationally intensive and will be very time-consuming. So, to make the training process consume less time and make learning process faster, some pre-processing was needed to be done. These 1920 * 1080 images were scaled down to 320 * 180 and thus reducing the size of the matrices (images). The reduction in the size of the matrices greatly reduces the number of parameters and complexity (multiplication and addition) of the architecture.

Some pre-processing facilitation is provided by Keras as described in current section of preceding iteration about data augmentation, splitting and batch sizing. Same pre-processing is done here with same batch size of 32, splitting of (80-20), rescaling is explained latterly and class mode as 'categorical' as we are solving categorical classification.

TRAINING

In the training phase, different keras-provided pre-trained functional models were tried and these models include VGG16, Mobile Net and InceptionV3. These models are pre-trained on ImageNet dataset mentioned in the literature review section. Because these models learned vast amount of knowledge (weights) including 1000 of different classes before therefore, it proved easy for them to learn these 9 hand signs that are easily distinguishable from each other.

Model	Size	Accuracy	Parameters	Depth
Mobile Net	16MB	0.895	4,253,864	88
VGG16	528MB	0.901	138,357,544	23
InceptionV3	92MB	0.937	23,851,784	159

FIGURE 10: MODEL COMPARIONS

The table is indicating that Mobile Net is the most light-weight convolutional architecture but not as accurate as InceptionV3 and VGG16 because of its low complexity and depth. Here, depth refers to the topological depth of the network in terms of activation layers, batch normalization layers, pooling layers, drop out layers etc. but not as complex in mathematical computation as VGG16 which including highest number of parameters. Third column is showing the performance results on ImageNet dataset and InceptionV3 is on the top among these. So, we used these famous and experienced convolutional architectures specifically for our case because of their popularity in solving the multi-class recognition problems.

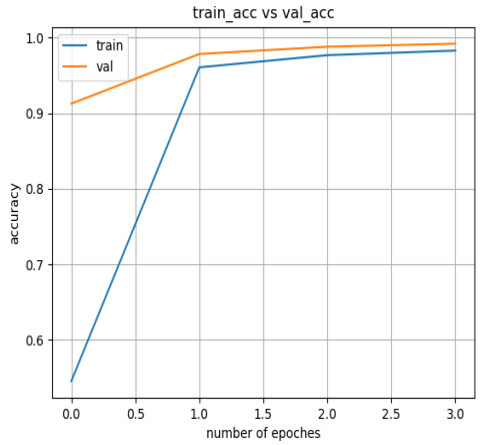
CHANGES MADE IN ARCHITECTURES

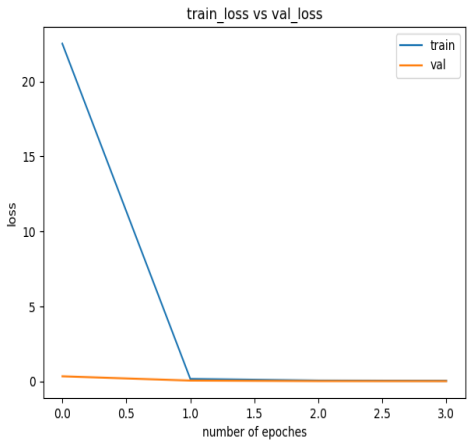
Similar changes were made in all three convolutional architectures to make the network more optimized and suitable to this specific use case. The structure of all three architectures were converted from functional to sequential, dense layer of 256 nodes was added just before the 'softmax' output layer and parameter of number of classes at the output layer was changed from 1000 to 9 as we custom-trained the models at 9 hand gestures classes.

TRAINING RESULTS

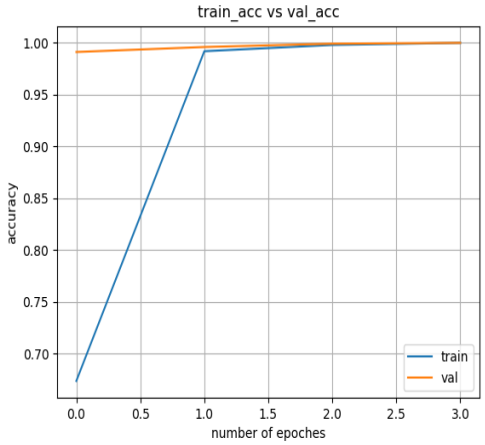
It took 4 epochs for each model for training phenomena and each epoch containing 31 steps while passing the batch size of 32 through the network. After completion, the training results were really good by all three models. The training and validation results of three architecture are represented graph-ically below:

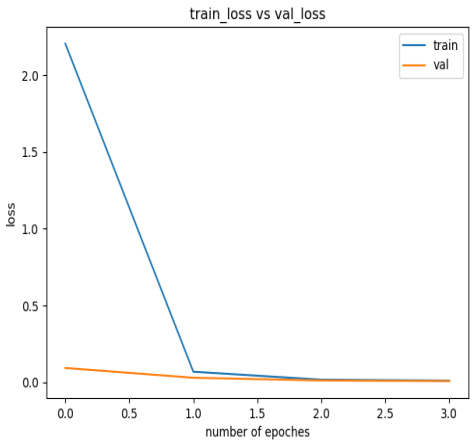
INCEPTIONV3



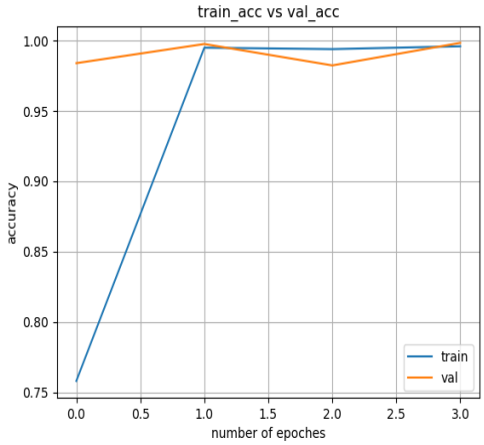


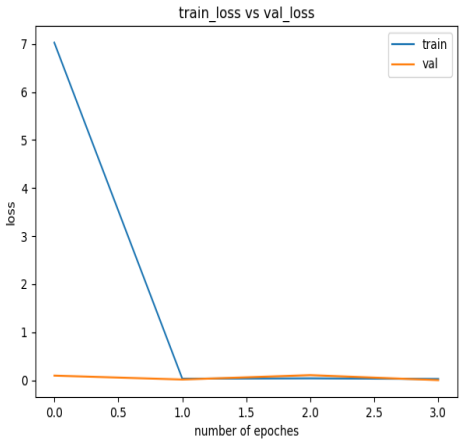
VGG16





MOBILE NET





The graphs are showing the training and validation results comparisons among MobileNet, InceptionV3 and VGG16. There's not major difference between their results because in all three cases, training and validation accuracies are surging to 98> percent and loss is less than 0.2. Noticeably, models have learned really fast on almost 28000 training images provided to them.

TESTING

For the testing procedure, all the three models were tested on the test set incorporating 200 images for each class. All models performed really well with 94> testing accuracies. As per-class analysis and evaluation, confusion matrices of the models are shown below:

INCEPTIONV3 (CONFUSION MATRIX AND CLASSIFICATION REPORT)

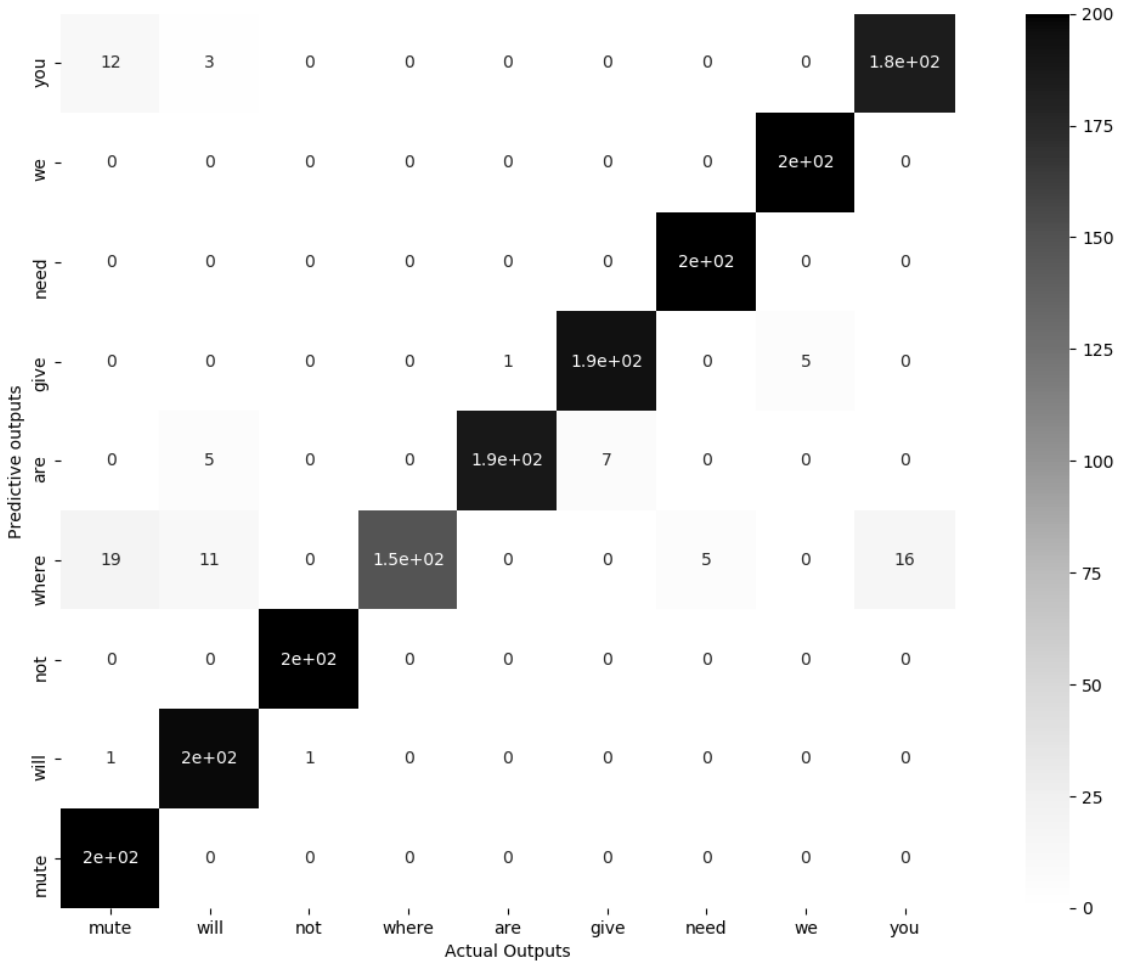


FIGURE 11: CONFUSION MATRIX OF INCEPTIONV3

As per-class analysis, it is important noted that InceptionV3 is failed to discriminate between 'mute' and 'where' up to 10 percent and 'where' and 'will' up to 5 percent and that is negligible. Per-

class performance on all the other signs has been impressive and the testing accuracy is surging to 95 percent as shown in the classification report. As the report is an indication that there are 200 images per class and ‘not’, ‘give’, ‘are’, ‘where’ and we are giving almost 100 percent accuracy.

**********CLASSIFICATION REPORT**********

	precision	recall	f1-score	support
mute	0.63	1.00	0.77	200
will	1.00	0.33	0.50	200
not	1.00	1.00	1.00	200
where	0.99	1.00	0.99	200
are	1.00	1.00	1.00	200
give	1.00	1.00	1.00	200
need	1.00	0.98	0.99	200
we	1.00	1.00	1.00	200
you	0.94	1.00	0.97	200
accuracy			0.92	1800
macro avg	0.95	0.92	0.91	1800
weighted avg	0.95	0.92	0.91	1800

FIGURE 12: CLASSIFICATION REPORT OF INCEPTIONV3

VGG16 (Confusion matrix and Classification report)

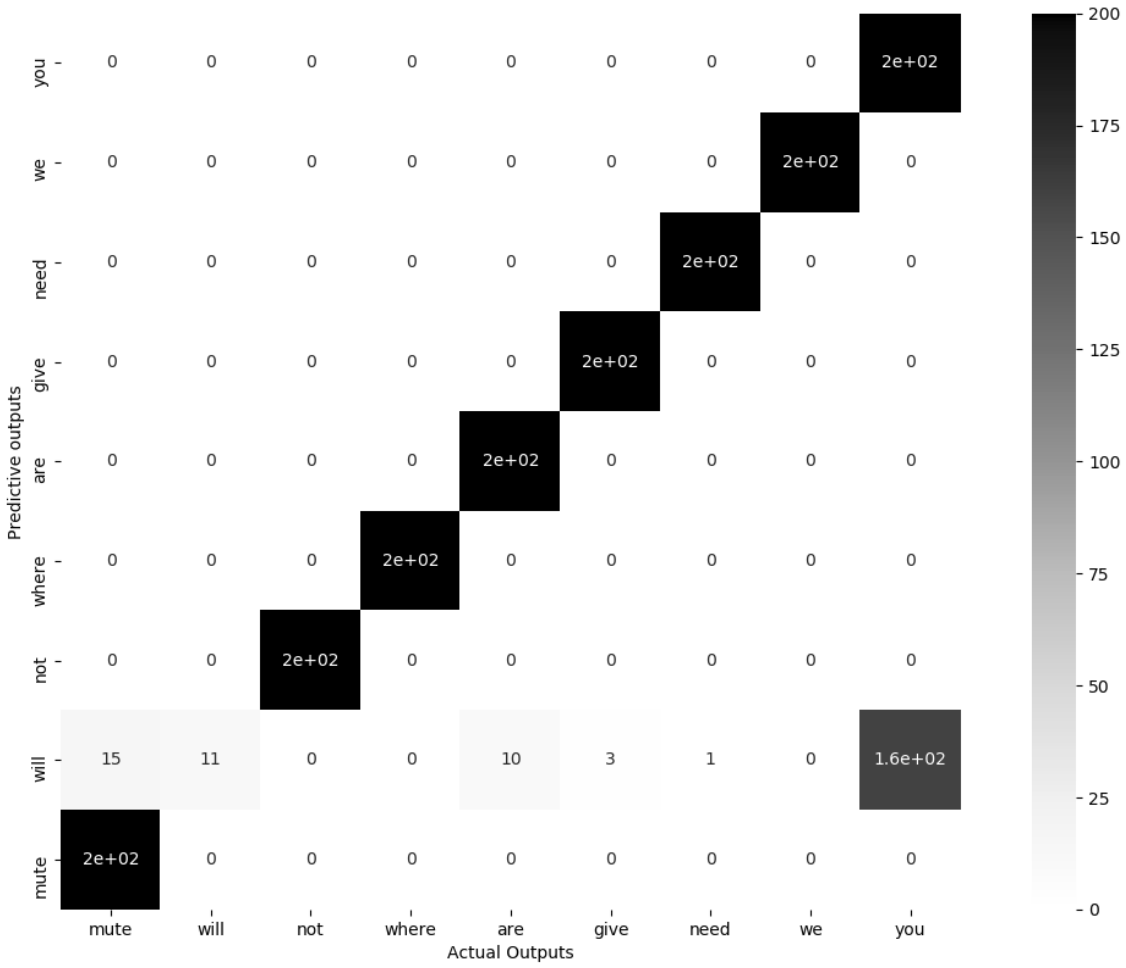


FIGURE 13: CONFUSION MATRIX OF VGG16

As per-class analysis, VGG16 has also been impressive on the testing set. The confusion matrix is indicating that there's some significant difference inn performance regarding 'will' hand sign as compared to all other hand signs. VGG16 is greatly confused to find the difference between 'you' and 'will'. It is worth considering that except 'will', VGG16 had outclassed on remaining hand classes. It can also be analyzed statistically in the following classification report that 'will' has the f1-score of 0.10 which is marginally less than the other f1-scores of remaining classes. But, VGG16 is giving the overall accuracy of 94 percent which is quite healthy.

**********CLASSIFICATION REPORT**********

	precision	recall	f1-score	support
mute	0.93	1.00	0.96	200
will	1.00	0.06	0.10	200
not	1.00	1.00	1.00	200
where	1.00	1.00	1.00	200
are	0.95	1.00	0.98	200
give	0.99	1.00	0.99	200
need	1.00	1.00	1.00	200
we	1.00	1.00	1.00	200
you	0.56	1.00	0.71	200
accuracy			0.90	1800
macro avg	0.94	0.90	0.86	1800
weighted avg	0.94	0.90	0.86	1800

FIGURE 14: CLASSIFICATION REPORT OF VGG16

MOBILE NET (CONFUSION MATRIX AND CLASSIFICATION REPORT)

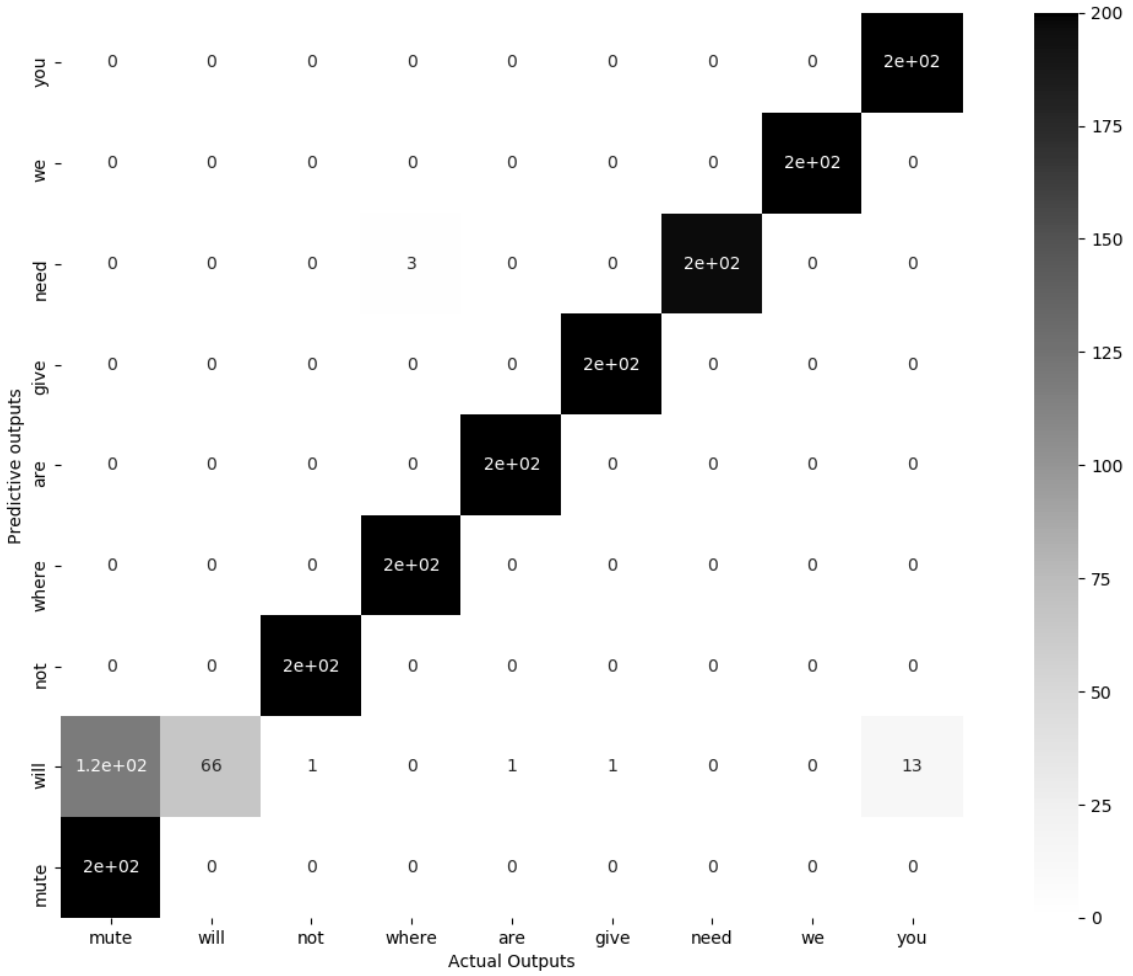


FIGURE 15: CONFUSION MATRIX OF MOBILE NET

As per-class predictive analysis vs actual outputs, MobileNet also performed really well and it is important to note that alike VGG16, MobileNet is also confused in finding the dissimilarity of 'will' class with 'you' and 'mute'. But, overall performance of MobileNet is also cool surging to the accuracy of 96 percent and that is far higher than IncpetionV3 and VGG16. Statistically, performance measures of MobileNet are shown below through classification report.

```
******************CLASSIFICATION REPORT**********************
              precision    recall  f1-score   support

        mute       0.86      1.00      0.93       200
        will       0.91      0.99      0.95       200
         not       1.00      1.00      1.00       200
       where       1.00      0.74      0.85       200
         are       0.99      0.94      0.97       200
        give       0.97      0.97      0.97       200
        need       0.98      1.00      0.99       200
          we       0.98      1.00      0.99       200
         you       0.92      0.93      0.92       200

    accuracy                           0.95      1800
   macro avg       0.96      0.95      0.95      1800
weighted avg       0.96      0.95      0.95      1800
```

FIGURE 16; CLASSIFICATION REPORT OF MOBILENET

CONCLUSION

So far, we tested three famous keras-provided pre-trained functional models on challenging hand gestures dataset containing exactly 27360 training images and 6480 validation images. All of these models performed with 95 percent testing accuracies. But, the winner in this case was MobileNet model with the highest testing accuracy of 96 percent, followed by InceptionV3 with 95 percent and at last VGG16 with 94 percent.

ITERATION 3

After producing better performances using different custom-made convolutional models, Keras-provided functional models, the most widely used TensorFlow Object Detection API was tried further in this iteration to make the model work in real-time. This iteration has some major difference when compared to preceding iterations. This iteration doesn't involve just the problem of hand classification merely through the input hand images but also hand detection problem which is s prior step to hand gesture recognition. Because the dataset is now changed from hand images to fully-framed person images which is explained in detail in the data collection phase. Next section provides the slight overview of this API about what makes this API more reliable and productive in performance.

TensorFlow Object Detection API

TensorFlow provided models are able to detect multiple objects within an image with bounding boxes. The API has been trained on the COCO dataset (Common objects in context), containing around 300K images of 90 most commonly used objects.

FIGURE 17: COCO CLASSES

The API provides several pre-trained models trained on the COCO dataset that provides a trade-off between speed of execution and accuracy in placing the bounding boxes on right region of interest in the whole images. The following table provides the bird's eye view about the pre-trained models available in the API GitHub repository.

Model name	Speed	COCO mAP	Outputs
ssd_mobilenet_v1_coco	fast	21	Boxes
ssd_inception_v2_coco	fast	24	Boxes
rfcn_resnet101_coco	medium	30	Boxes
faster_rcnn_resnet101_coco	medium	32	Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	slow	37	Boxes

FIGURE 18: TABLE OF TF MODELS

Here, mAP stands for 'mean average precision' is the product of precision and recall on drawing and detecting the bounding boxes. It provides a great idea for how sensitive the model is in recognizing the object of interests and how well for the avoidance of false alarms. Consequently, higher the mAP score, more accurate the model is but this high accuracy comes at the cost of speed of execution in recognizing the objects.

TENSORFLOW MODEL SELECTION

After having a brief analysis on TensorFlow-provided pre-trained models and now, considering the use case, it is really important to make the selection for the model to use for hand gestures recognition. The correct selection is never made without considering the available hardware resources and the dataset in machine learning project. So, depending upon the quality of the dataset and hardware resource, decision was made to try the most light-weight model (SSD MobileNet V1). These SSD models suits well when there is insufficient availability of resources: dataset and hardware. Because remaining TensorFlow-provided pre-trained possess some complex architectures and usually, these complex architectures need more parameters for learning to train at the specified task. Secondly, when there's no GPU or cloud service available, it becomes very difficult for the laptop or any personal computer to train faster-rcnn like complex models.

DATA COLLECTION

In all previous iterations, dataset was containing the images of merely the hands but in this iteration, there are fully-framed images of a person either mute or deaf, involving his/her hand gesturing. Object detection problem is evolved due to some significant alteration in the dataset. So, the nature of the dataset requires the model to first detect the hand and localize the hand gestures through the bounding box in the whole image incorporating a disable one. After the region of interest is localized with the bounding box, hand classification is made further. Consequently, the problem now requires a multi-step approach involving hand detection with hand classification. Therefore, for this problem, TensorFlow Object Detection API is taken into account to execute this two-step process.

DATASET CLASSES

In this iteration, 10-class data collection is made by collecting the images containing a gesturing person. Also, in this iteration, most commonly used words are considered that are easily differentiated when compared with each other. All collected words are static in their nature thus, representable only by a single frame or image. So, the collected words are: 'bathroom', 'give', 'mute', 'will', 'not', 'where', 'we', 'are', 'meeting' and 'together'.

DATASET IMAGES

The collected images are not consistent in nature with respect to backgrounds. The images are possessing some more diverse and complex backgrounds. These can be termed as tough images for the model to learn because some images involve occlusions as well. As, this dataset also involves human body as well and there are some chances that hand pose may come in front of the face resulting in skin color in the background as well.

To keep the data stored in an organized way and to make it usable for the labelling process, all the images are stored in merely one folder. One other folder is created that is used for storing the annotation results of all images along with other formatted required for the conversion phase. There are total 12730 images in which each class constitutes around 1230-1280 images.

How we have collected the dataset?

The dataset is collected in the similar way as in the previous iteration.

Data Annotation

With all the images gathered, it's the phase to label the desired object in every image. There are various image annotation tools available and, in this case, the chosen one is Visual Object Tagging Tool (VoTT). The annotation process is basically drawing the bounding boxes around the object of interest in an image. The label process automatically creates the. JSON file that describes the object in an image. In this case, there's only one object in every image so, the number of .json files created are equal to the total number of images in the whole dataset. The JSON file includes the image and bounding box information in specified format. The information contains name, id, path, format, size (width, height) of the image file and coordinates of bounding box drawn manually, class of the hand gesture and some other formatted information introduced by the annotation tool. VoTT also provides the facility to split the dataset into train/validation samples after all the images have gone through labelling process. The dataset splitting done by the tool is explained in the next section.

Data Splitting

In this section, dataset is divided into two parts: train and validation sets. The splitting is done automatically when the program runs to train the model. No images are included in the testing phase because testing is done in real-time in this final iteration considering the number of frame detections per FPS value which will be explained later in succeeding sections. So, the statistics of training and validation sets are: 80 percent and 20 percent

Data Conversion

There are different dataset conversions that need to be done to prepare the data for the training phenomena. Because TensorFlow Object Detection API accepts the training and validation datasets in specified formats. First, the images json data is used to create .csv files containing all the data for the training and testing images. Train.csv and test.csv are generated through this conversion. These two csv files are now converted into .tfrecord formatted files as train.tfrecord and test.tfrecord which is the only acceptable format by TensorFlow models to train using TensorFlow object detection API. As a result, the data is now fully prepared to input to TensorFlow SSD model for the training process.

Training

Some specific training pipeline is needed to be followed to train the model using TensorFlow Object Detection API. So, to train the model using API, some other files were also necessary other than formatted record dataset files. These include model configuration file and labels file. Model configuration file is used to configure the training and the evaluation process. This configuration file includes the parametric information of the model. At high level, the config file is divided into five blocks:

1. The initial block is model_config file defines the information about what type of model will be trained and this involves the meta-architecture, feature extractor etc.

2. The train_config decides what parameters should be used for the training phenomena. Some of these parameters are stochastic gradient descent, input pre-processing and feature extractors or kernels initialization parameters.
3. The eval_config block defines what set of performance measurements will be reported while training. These evaluation measurements include different set of matrices.
4. The train_input_config decides what dataset the model should be trained on, involves the paths of training dataset record file and classes file. This block provides the information about the training set while training.
5. The eval_input_config block includes the information of the evaluation or validation dataset files like their paths and formats. Simply, defines on which dataset the model will be trained on.

The most basic monomers of model configuration file are defined above and other detailed information about parameters are also included in these segments.

As, we are using pre-trained SSD model in our case and there are different variants of SSD models available in the API like SSD_MobileNet_V1, SSD_Inception_V2, SSD_ResNet_101 etc. But we made the selection of SSD model incorporating MobileNet_V1 convolutional architecture. The MobileNet convolutional architecture is the most light-weight model available in this API. Depth-wise separable convolution is used in this architecture to reduce the model size and complexity. As, we are developing an android app and so, this model is useful for mobile and embedded vision applications. Smaller model size includes fewer number of parameters and smaller complexity includes fewer multiplication and additions. The next section summarizes the high-level view of this architecture.

MODEL ARCHITECTURE

Here's the summary of the MobileNet V1 architecture:

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

FIGURE 19: MOBILENET ARCHITECTURE

It is noted that Batch Normalization (BN) and RELU are applied each convolution to eradicate the trivial information involved and to reduce the size of the matrices. Doing this, greatly reduces the computational complexity and time to train the model rapidly.

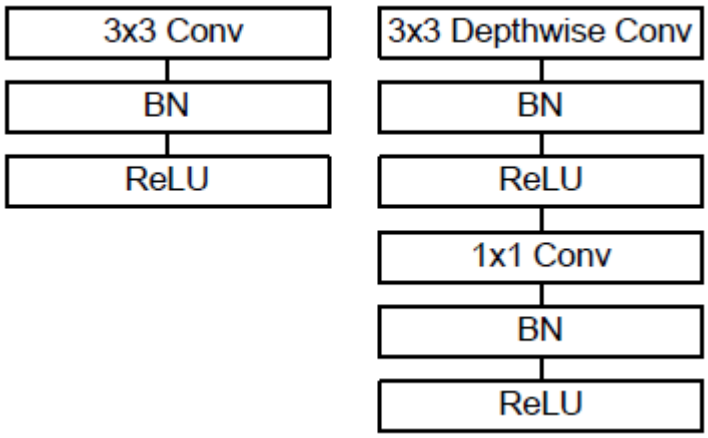


FIGURE 20: BATCH NORMALIZATION AND RELU LAYERS

The above diagram is showing the standard convolution and depth-wise convolution using Batch Normalization (BN) and RELU activation function.

CHANGES MADE IN MOBILENET

For custom training using TensorFlow Object Detection API, custom changes were needed and these changes were made in the model configuration file. For training the model on a custom dataset, num_classes parameter value is altered from 90 to 10 as we are training on merely 10 hand signs. Secondly, paths and names of dataset and label files of were altered to train on custom dataset. Thirdly, training steps were reduced from 20000 to 15000 to reduce some time. So, these were the few changes which were needed to be made for some custom training and the requirements.

TIME AND HARDWARE FOR TRAINING

Time was also the major reason for why we made the selection of SSD model. For training the model, it took 30-35 hours for completion. There were 15000 steps to be covered in training phenomena and for each 100 steps, it took 600s-800s. Due to limited hardware resources, training was taking too much time and laptop used for training incorporates 16 GB ram, core i7, 256 GB SSD with no GPU. While on the other hand in case of faster-rcnn which are relatively more complex models as compared to SSD models were taking 1200s-1400s for each 100 steps. As a result, minimum 6-8 days are needed to custom-train faster-rcnn on this hardware and therefore, we gave up on this heavy selection.

MODEL INTEGRATION WITH ANDROID

Now, the model is fully trained and prepared for integration with android. But, before final integration, some steps are crucial to make the machine learning trained model loadable by java interpreter in the android system.

MODEL CONVERSION

Model conversion from frozen graph (.pb format) into .tflite format is the final step for integration with android. Because .tflite format is the only acceptable format for the deployment of machine learning models in android mobile devices. This conversion is beneficial as it makes the model file lightweight and also makes the android app occupy less space on an android mobile device and perform recognition process at high speed.

The above conversion is performed by TensorFlow lite converter and generates a TensorFlow lite flat buffer file (.tflite). The conversion supports several formats like frozen graphs (.pb) files generated by YOLO or TensorFlow Object Detection API, saved model files (.h5 or .pkl) generated by keras API and other concrete functions. But in our case, we are following the conversion explained in start of preceding paragraph.

MODEL QUANTIZATION

Moreover, there are two ways to perform the conversion. This is an additional feature provided by the TensorFlow API to make model occupy less space on a client device so that app may run smoothly. Because, as we all know that mobile devices often have a limited memory or computational power. So, various optimization techniques can be applied to make the model work best on these light devices. Currently, TensorFlow lite supports optimization via quantization. Quantization reduces the precision of the model's parameters which is by default are 32-bit floating point numbers. This 32-bit floating point format is converted into 8-bit integer format and thus, results in the smaller model size and faster computation. The quantization greatly helps in reducing the inference time that is needed for one prediction. On the contrary, the main disadvantage of quantization is that it somehow affects the accuracy of the model but not greatly. It is important to note that the model file in our case is not quantized.

Along with model (.tflite) file, labels (.txt) file is also required in the assets folder of an android app project. In our case, the model (.tflite) file is occupying 22.5MB and when integrated with client android device, app is occupying 80MB of space. Consequently, there are certain two main files which are used to integrate the machine learning models with android which have been discussed clearly in this section.

TESTING

To test the model intelligence before integration with android, 1 min video was recorded including all 10 hand signs using smartphones camera at 30 FPS. Frames were extracted from the video using OpenCV and doing some other image pre-processing and model was applied on these video frames for predictions. At average, model was detecting 15/30 frames per second.

On the same recording video but now, using object tracking algorithms provided by OpenCV library, 20+/30 frames per second were being detected. Several object tracking algorithms were applied but every object tracking algorithm has its own pros and cons in terms of speed and accuracy. Some trackers were performing really well but at the cost of low execution speed and some were performing at an average rate (20-30 FPS) but with high accuracy. In our case depending upon this trade-off between

speed and accuracy, Boosting and KCF were performing really well considering both performance measures.

After detecting the initial frame by the model, trackers get hold of next subsequent frames based upon the prior bounding box information of the previous frame. If tracker is not able to detect any of the next subsequent frames, model is again applied on that untracked frame. In case if both are not able to make the detection in next 50 subsequent frames then output is clarified that there's no desired hand sign for detection.

Now in case when the model is integrated with android and app is ready to operate now in real-time, one important factor that was needed to be considered and that was inference time. Inference time is the time taken by the model for one prediction on an unforeseen data. Inference time mainly depends on the quality of the client device. Comparisons between the inference time were made using different android mobile devices.

- 80ms – 200ms (8 – 12 FPS)

  OPPO A5 2020, 3GB RAM, 64 ROM, Octa-core CPU

- 250ms – 320ms (6 – 8 FPS)

  Huawei Y9 Prime 2019, 4GB RAM, 128GB ROM, Octa-core

- 400ms – 600ms (2 – 4 FPS)

  Huawei Y6 Prime 2019, 2GB RAM, 32GB ROM, Quad-core

It is important to remember that the model is not quantized in these above cases. If quantized, can lead to much better speed.

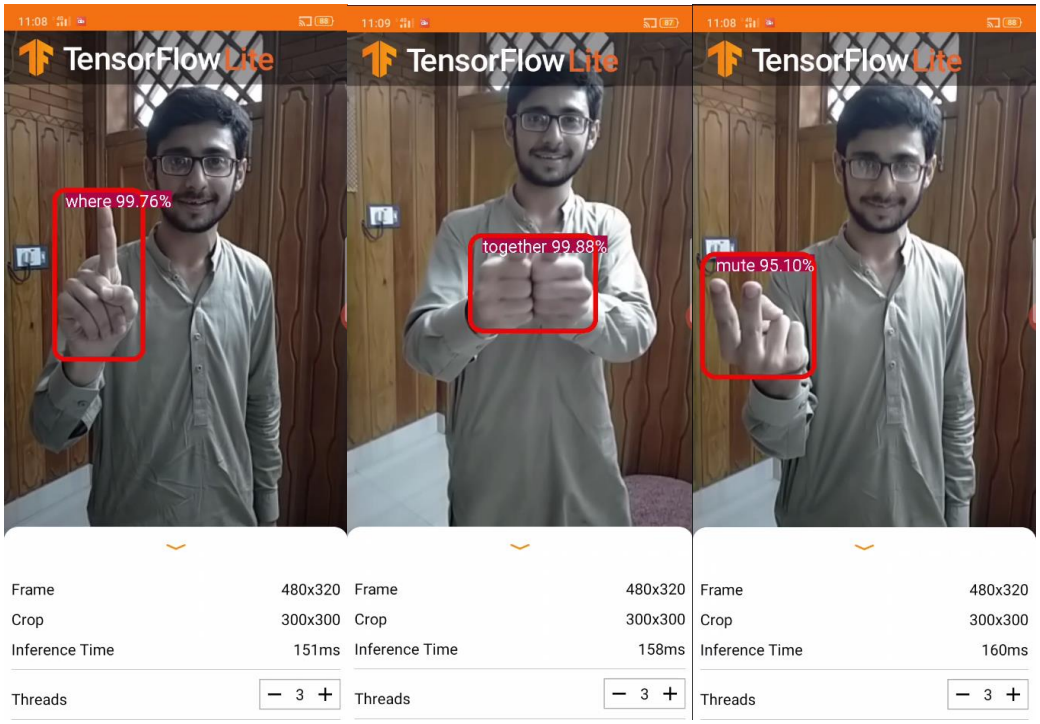


FIGURE 21: DETECTED HAND SIGNS IN ANDROID IMAGES

The above activity of app is showing the predicted hand gestures output by the SSD model. The app consists of only this functional camera activity and nothing else. When non-sign language user runs an app, app simply accesses the smartphones camera and starts making detections of desired hand poses of sign language user who is either deaf or mute. This is the overall functionality of Sign Language Recognition App capable enough to translate the hand signs into textual information understandable by an app user who is non-sign language user.

FINAL CONCLUSION

So, we chosen several custom-made and pre-trained models and tried on various custom-made datasets. Most of them performed quite well and in the final stage extending our project to two-step process: 1) hand detection and 2) hand gesture classification. TensorFlow Object Detection was used to solve both problems. Hand classification problem was resolved by detecting not just one hand but various classes of hand gestures. Actually, SSD detector was used in solving the multi-class recognition problem using TensorFlow Object Detection API. This prevented us to add some classifier to out project in the final stage.

REFERENCES

REFERENCES

http://link.springer.com/10.1007/s00138-019-01043-7. (n.d.).

http://www.state.me.us/dep/rwm/oilconveyance/pdf/hwtconveyanceform.pdf. (n.d.).

https://ieeexplore.ieee.org/document/8451657/. (n.d.).

https://linkinghub.elsevier.com/retrieve/pii/S1877050917320720. (n.d.).

https://machinelearningmastery.com/applied-deep-learning-in-python-mini-course/. (n.d.).

https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69?gi=61477d538d67. (n.d.).

https://www.irojournals.com/iroiip/V2/I2/01.pdf. (n.d.).

https://www.python36.com/mnist-handwritten-digits-classification-using-keras/. (n.d.).

https://www.research.manchester.ac.uk/portal/files/54571713/FULL_TEXT.PDF. (n.d.).

https://www.tensorflow.org/datasets/catalog/coco. (n.d.).