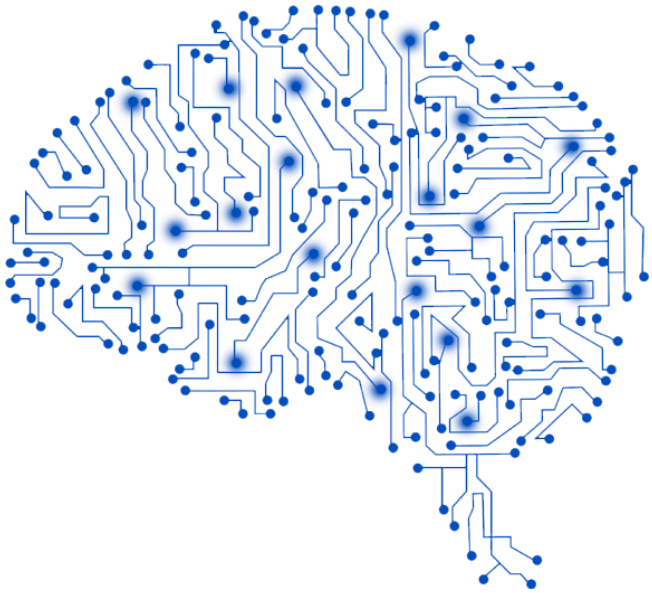
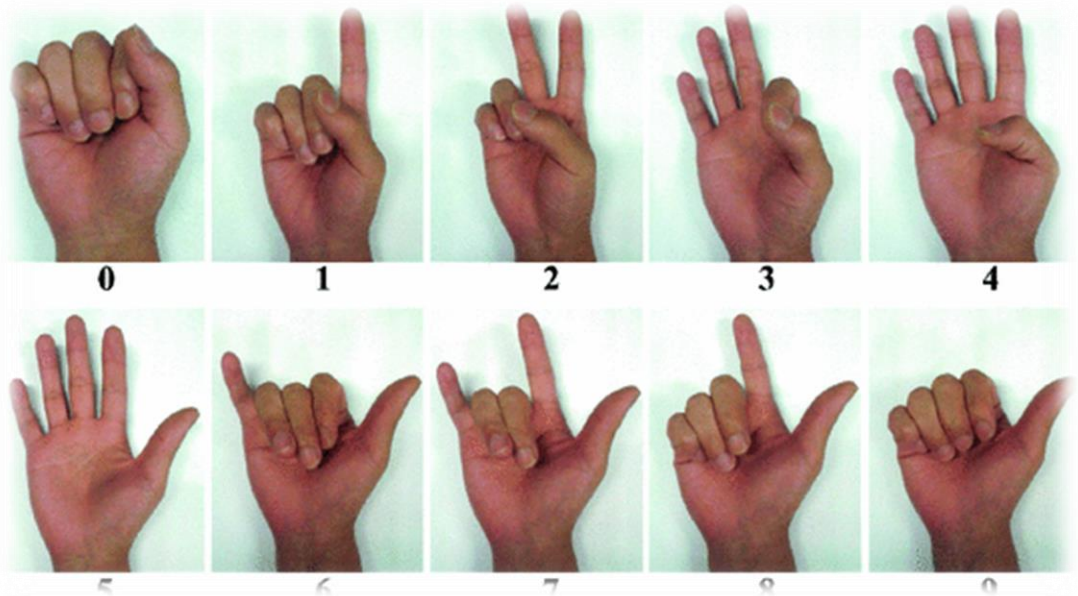
REAL TIME HAND GESTURE TRACKING USING DEEP LEARNING

ABHISHEK KUMAR RANJAN
193310005
M.Tech-CSRE



June 7,2020

MENTOR: KRITI AGARWAL



INDEX

Introduction:

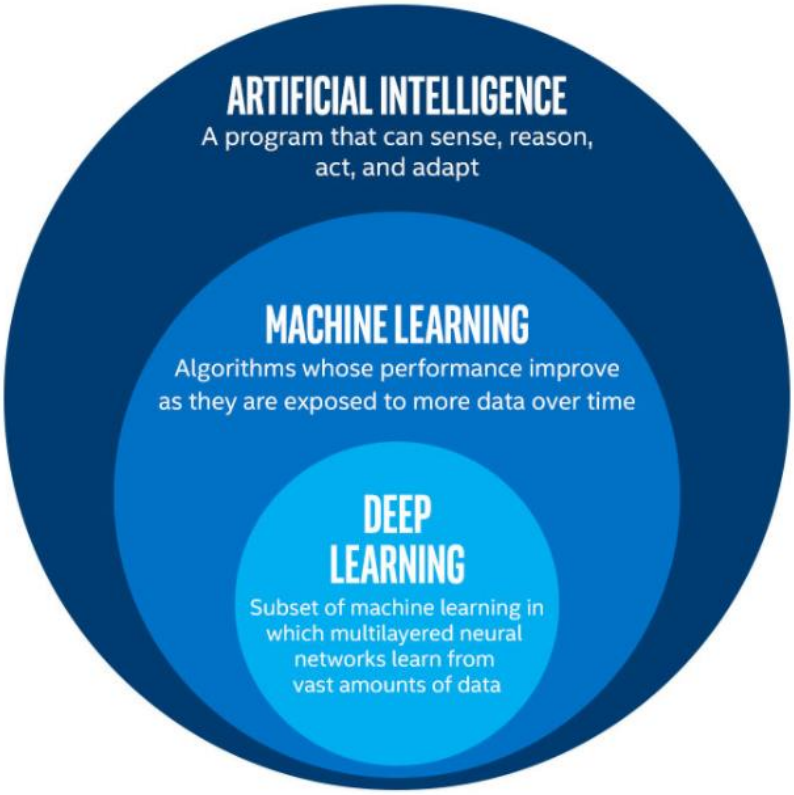


Fig-1: AI,ML,DL visual explanation

Deep Learning is a subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks.

Deep learning architectures such as deep neural networks, deep belief networks, recurrent neural networks and convolutional neural networks have been applied to fields including computer vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation, bioinformatics, drug design, medical image analysis, material inspection and board game programs, where they have produced results comparable to and in some cases surpassing human expert performance.

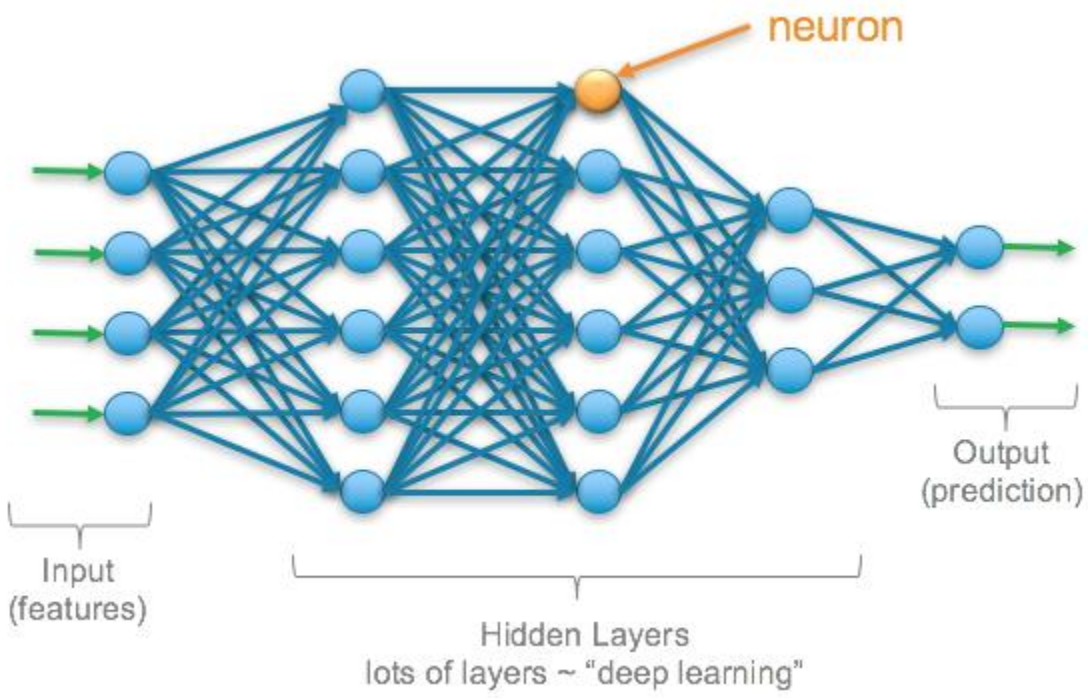


Fig-2: A visual representation of Artificial Neural Network

CNN:

As my project is gesture tracking so it deals with images, it is computer vision problem so CNN i.e. ***Convolutional Neural Network*** is used here.

CNN is now the go-to model on every image related problem. The main advantage of CNN compared to its predecessors is that it automatically detects the important features without any human supervision. For example, given many pictures of cats and dogs it learns distinctive features for each class by itself.

CNN is also computationally efficient. It uses special convolution and pooling operations and performs parameter sharing. This enables CNN models to run on any device, making them universally attractive.

A CNN model can be thought as a combination of two components: feature extraction part and the classification part. The convolution + pooling layers perform feature extraction. For example given an image, the convolution layer detects features such as two eyes, long ears, four legs, a short tail and so on. The fully connected layers then act as a classifier on top of these features, and assign a probability for the input image being a dog.

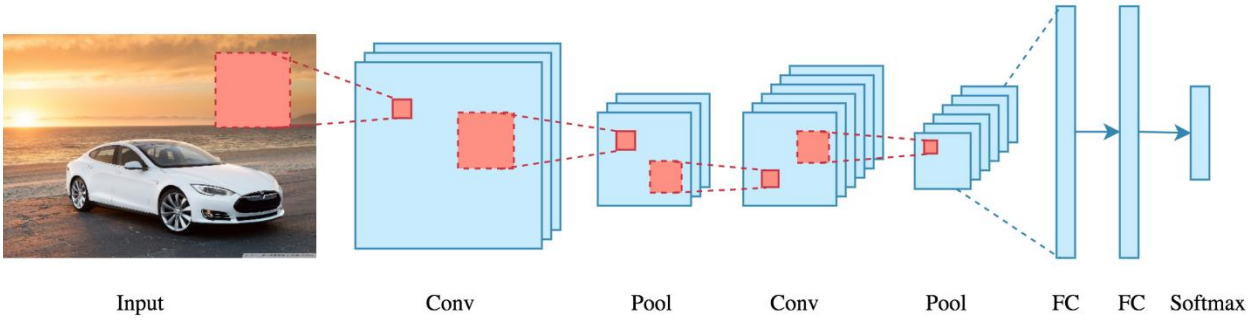


Fig-3: A series of operations done over image to train classifier to generate Desired model

Architecture Of ConvNets:

An image as 3D representation. An image has RGB color channels and each channel has 2D array of pixels. In convolutional neural network, we feed in the image and the network gives us desired output from the output layer. Different layers that are involved in convolutional neural network are as follows:

1. **Convolutional Layer:**
   These layers are the basic building blocks of convolutional neural network. A convolutional layer consists of numerous kernels/filters that can be learned by training the network. These kernels are then applied over input and they perform the convolutions. Each layer has its own kernel having some special dimension. Each filter in convolution layers can be understood as extracting some features from input image.

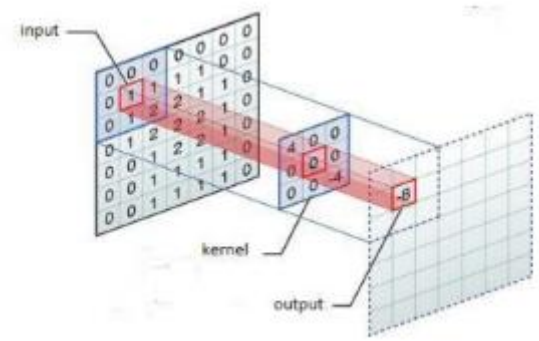


Fig-4: Convolution operation

We give an image as input of some size and depth of 3, but as we progressively moves further size of output in a particular layer decreases but depth increases. So there are 3 hyperparameters that control the size of the output volume: the depth, stride, and zero-padding.

- First, the depth of the output volume is a hyperparameter: it corresponds to the number of filters we would like to use, each filter try to learn for something different in the input. For example, if the first Convolutional Layer takes as input the raw image, the different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color.
- Second, we must specify the stride with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 then filter jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.
- Sometimes it will be convenient to pad the input volume with zeros around the border. The size of this zero-padding is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes (most commonly it is used to exactly preserve the spatial size of the input volume so the input and output width and height are the same).

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

n_{in}: number of input features
n_{out}: number of output features
k: convolution kernel size
p: convolution padding size
s: convolution stride size

2. **Pooling Layer:**

Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting.Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively.

- ***Average Pooling:***
  An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region.
- ***Max Pooling:***
  A max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region.

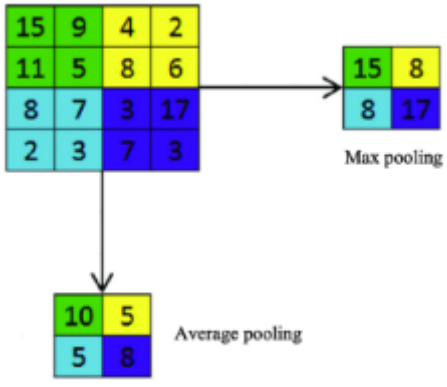


Fig-5: Max. & Avg. pooling

3. **<u>Fully Connected Layer</u>:**
   Neurons in a fully connected layer have full connections to all activations in the previous layer, just as in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset. The only difference between FC and CONV layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters. At the end of FC layer softmax layer results in actual classification.

4. **<u>Non-Linearity Layer</u>:**
   Basically, this is not a layer but is explicitly written as layer. It adds a non-linearity to the output of each layer so as to help gradient descent. It is also known as activation function. Some of the activation functions used often are shown below--

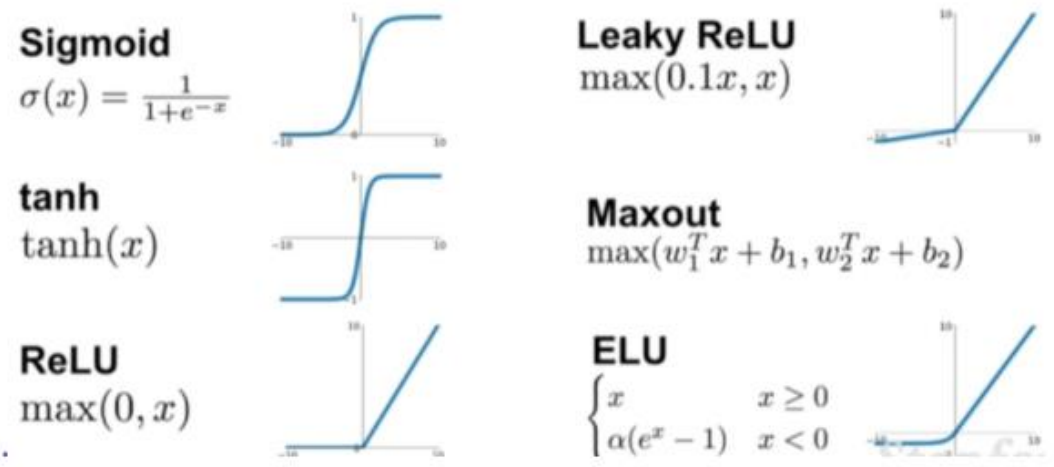


Fig-6: various activation functions

Execution using OpenCV:

1. **OpenCV:**
   OpenCV (Open Source Computer Vision Library) is a library of programming functions mainly aimed at real-time computer vision.
   It provides wide variety of inbuilt functions to enhance computer vision task.

Fig-6: OpenCV logo

2. **Algorithm:**
   Program using OpenCV in python was developed to detect simple hand gesture from first principle. Here CNN or any machine learning concept was not used, a simple mathematical and geometrical concept was used to detect hand gesture. Algorithm as follows---

   - First an interface was setup to access live stream of mobile camera using IP-webcam. Mobile camera was used for capturing better clarity images.
   - ROI(region of interest) was created as a region where hand need to be placed and that part only will be processed.
   - Image segmentation- to get binary image of hand and background, Gaussian blur- to reduce noise was applied over ROI.
   - Contour to hand was obtained and using contour number of fingers was detected. All the contour will make acute angle in between fingers, by counting number of acute angles number of fingers or gesture was reported.

3. Results and Inference:

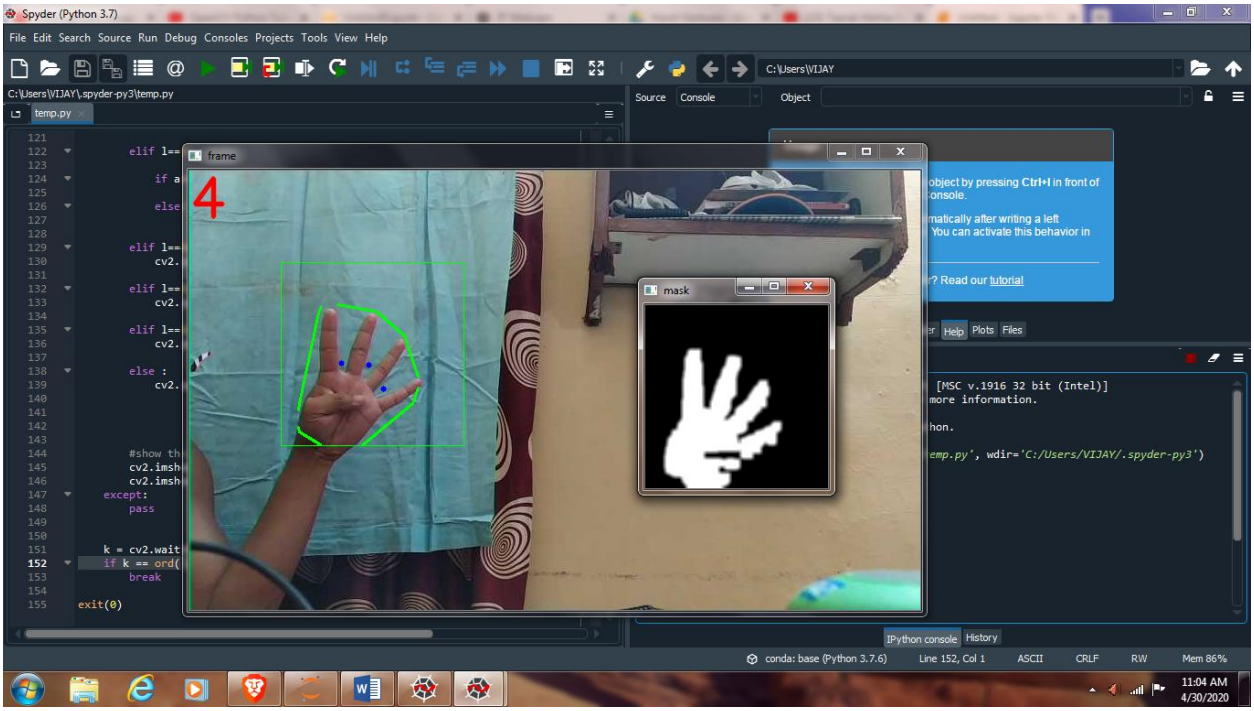


Fig-7a

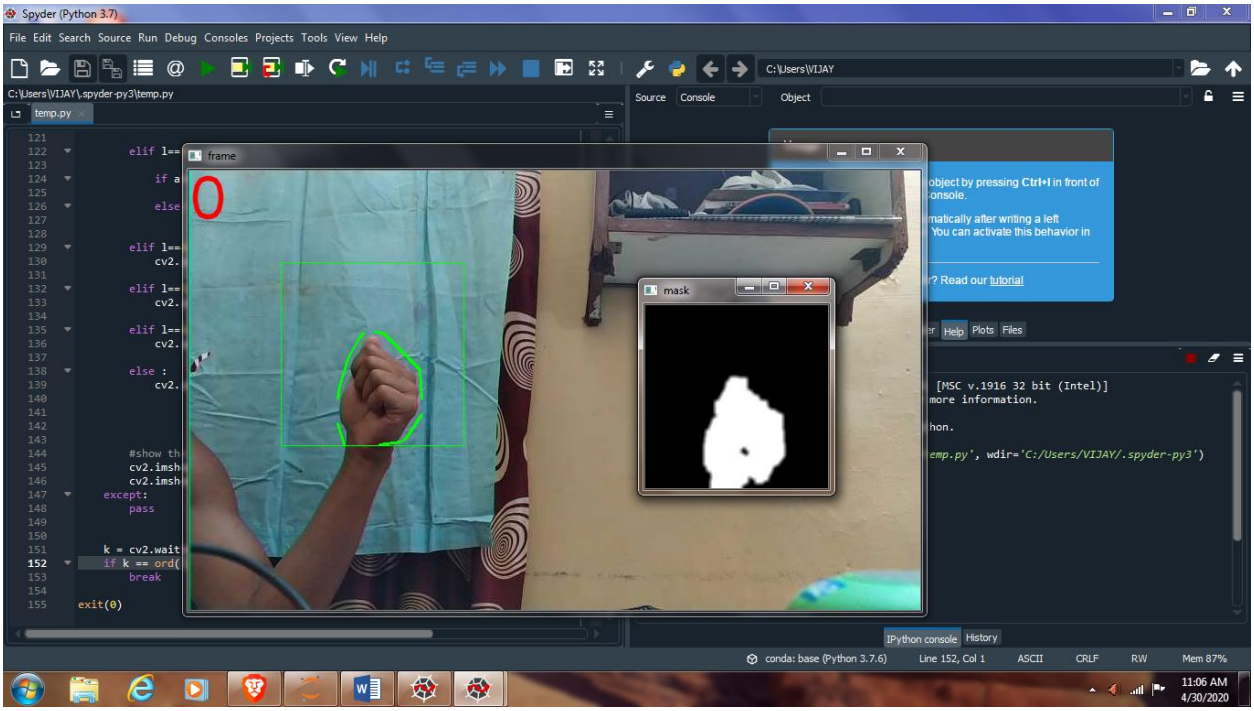


Fig-7b

Fig-7a & 7b: Sample images of real-time gesture tracking

- Here in this case gesture was detected using purely logic of mathematics, no image processing was used.
- This method is not much reliable as a range of skin color was defined in code which may or may not work for all.
- We need to manually place hand in ROI , it's results vary if background contrast is not good are some of the drawback of this algorithm.

Execution using CNN in tensorflow:

1. **Tensorflow:**
   TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. We will use inbuilt functions in tensorflow to build our CNN model.

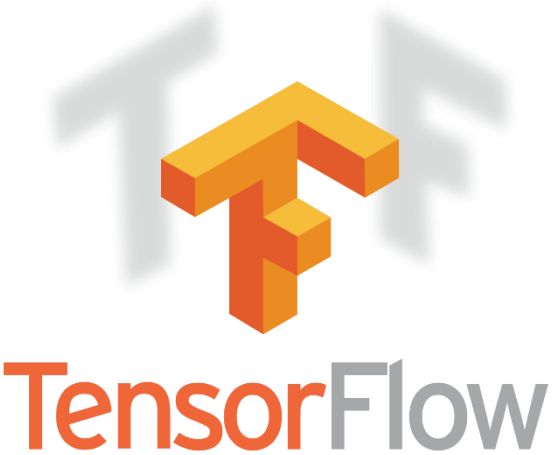


Fig-8: TensorFlow

2. **Keras:**
   Keras is TensorFlow's high-level API for building and training deep learning models. It's used for fast prototyping, state-of-the-art research, and production.

3. Google colab's numerous advantage:

- Program was written in Google colab as it offers numerous advantages.
- It is just like jupyter notebook so its interface and environment is very simple to use
- It saves our work directly into Google drive automatically so easy to use.
- No need to install or update any IDE, packages over local system as it runs over Google server.
- It connect us to a virtual machine which enable us to use high end hardware configuration for free. Like in normal mode about 12gb ram and 100gb HDD.
- Various modes of execution are offered like Normal, GPU,TPU—
  - Normal mode—used for normal execution. Configurations offered are 12gb RAM and 100gb HDD.
  - GPU—It means graphical processing unit. It is used when we require heavy computation like training neural network is heavy computation intensive. GPU mode reduce this time drastically. Configurations offered are 12gb RAM and 60gb HDD and Tesla K80 GPU.
  - TPU- It means tensor processing unit. It offers even better performance.
- Google offer this much that also free of cost. It helps a lot to students and research community as not everyone have high end system.
- If offers local runtime option also to connect computer to local computer to access local files and run program using local hardware.
- It is very easy to share our work with other, one can see and even edit the work if access is given.
- It can be mounted to Google drive to access or store files.
- Files can be directly pushed to GitHub repository very easily.
- Another important feature which was highly helpful in this project was that it can directly download Kaggle dataset using an API tool into virtual machine which can be used for learning models. Irrespective of internet speed it downloads dataset into virtual machine from kaggle servers with speed of about 100 Mbps.

Fig-9: Google Colaboratory

4. **<u>Important prerequisite for understanding tensorflow mode:</u>**
    - data1--https://www.kaggle.com/gti-upm/leapgestrecog
    - data2--https://www.kaggle.com/evernext10/hand-gesture-of-the-colombian-sign-language
    - model1--self-generated model
    - model2--fine tuning VGG16 model

So two datasets were used to generate 2 models for each dataset.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 124, 124, 32)      832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 62, 62, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 60, 60, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 30, 30, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 28, 28, 128)       73856
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 12, 12, 128)       147584
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 6, 6, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4608)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 4608)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               589952
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 832,010
Trainable params: 832,010
Non-trainable params: 0
```

Fig-10: self-generated model

Layer (type)	Output Shape	Param #
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
dense (Dense)	(None, 10)	40970

Total params: 134,301,514
Trainable params: 40,970
Non-trainable params: 134,260,544

Fig-11: fine-tuned VGG-16 model

5. **Fine tuning:**

First, we cut off the final set of fully connected layers (i.e., the “head” of the network where the class label predictions are returned) from a pre-trained CNN (typically VGG, ResNet, or Inception). We then replace the head with a new set of fully connected layers with random initializations.

From there, all layers below the head are frozen so their weights cannot be updated (i.e., the backward pass in back propagation does not reach them).

We then train the network using a very small learning rate so the new set of fully connected layers can learn patterns from the previously learned CONV layers earlier in the network — this process is called allowing the FC layers to “warm up”.

Optionally, we may unfreeze the rest of the network and continue training.

Applying fine-tuning allows us to utilize pre-trained networks to recognize classes they were not originally trained on.

It is highly helpful when we don’t have any clue about how to build model we can take existing model and play with it by varying parameters to build our own model.

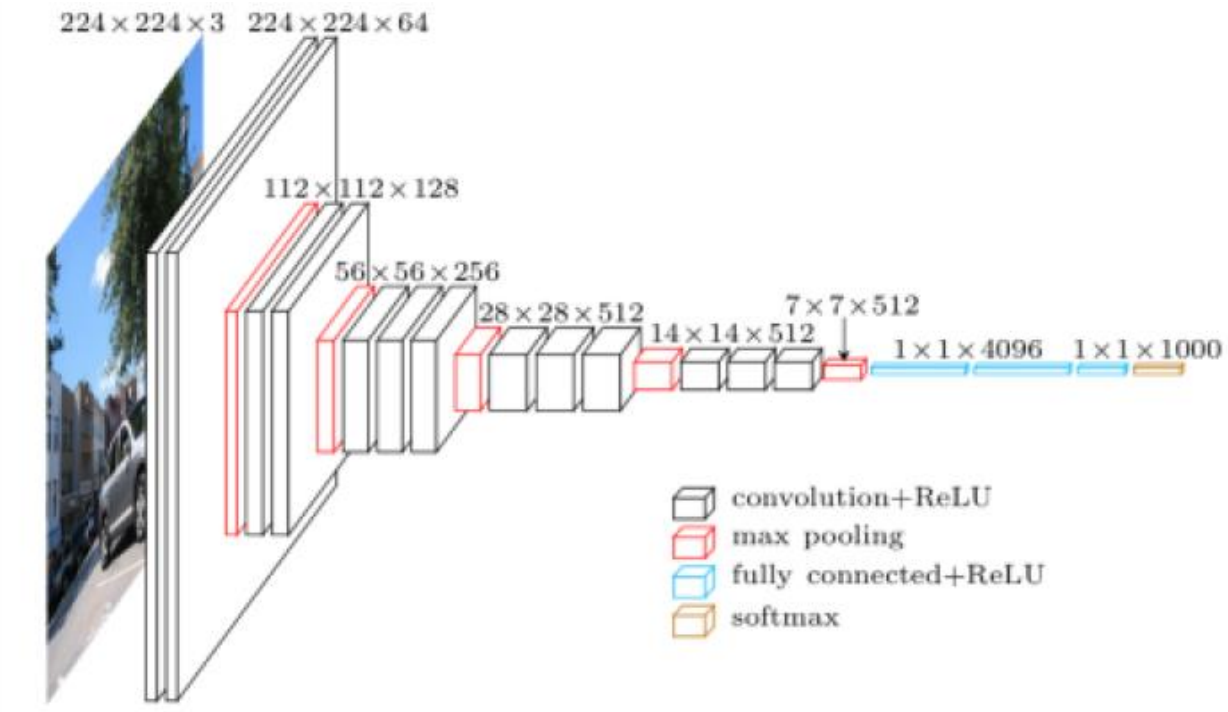
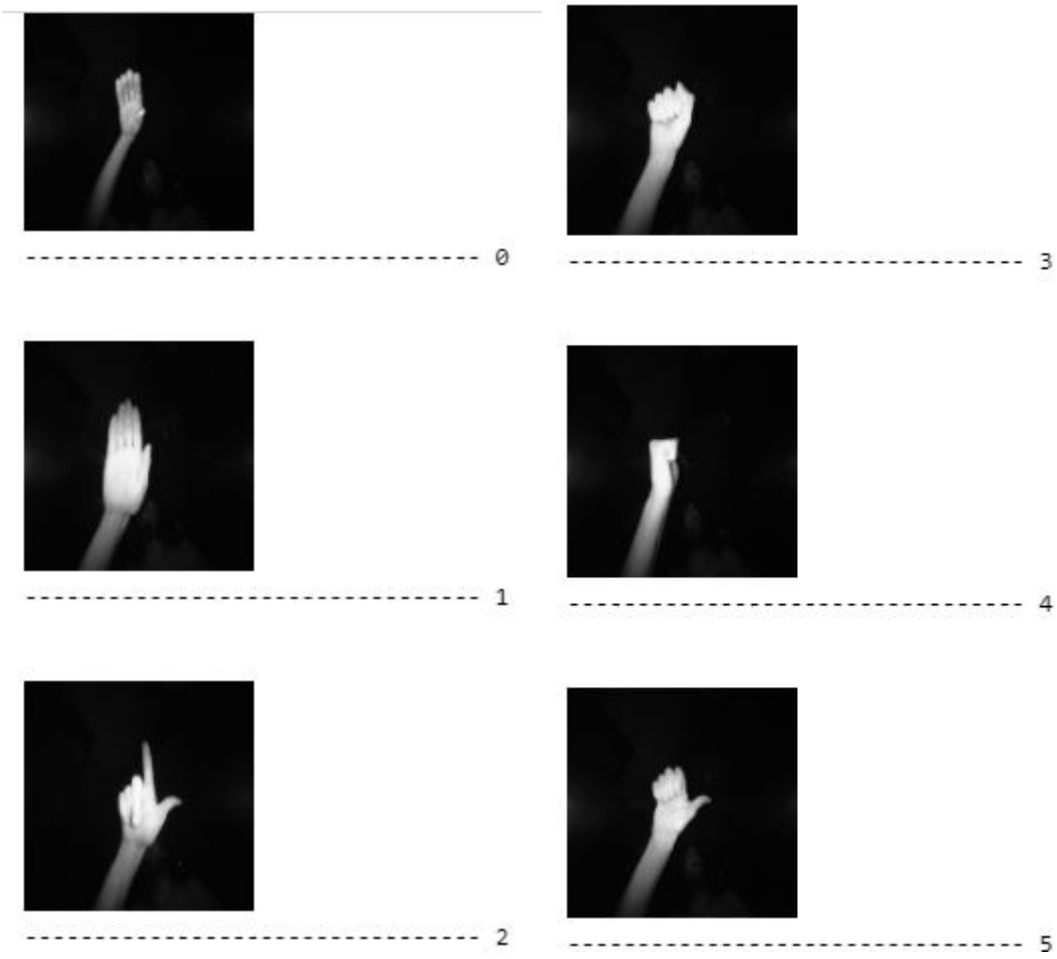


Fig-12: VGG-16 model

6. ALGORITHM:

- First kaggle was integrated with Google colab and datasets were downloaded to virtual machine using kaggle API.
- Name of all the data image including its directory structure was stored into a variable.
- Using above variable whole image data and its labels were stored in x and y variables.
- Train and test split.
- CNN model is built using sequential command and then trained using training data.
- Then model is saved into drive so that it can be loaded easily in future and can be used directly without training again.
- Validation, accuracy and confusion matrix id reported using test data.
- Then model is tested for images given by user using webcam.



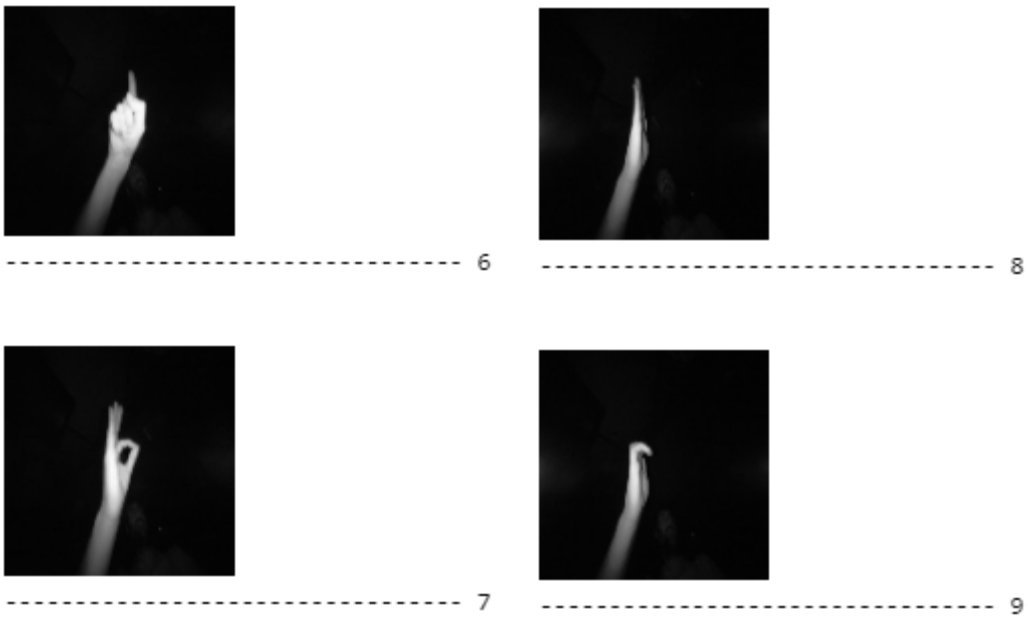
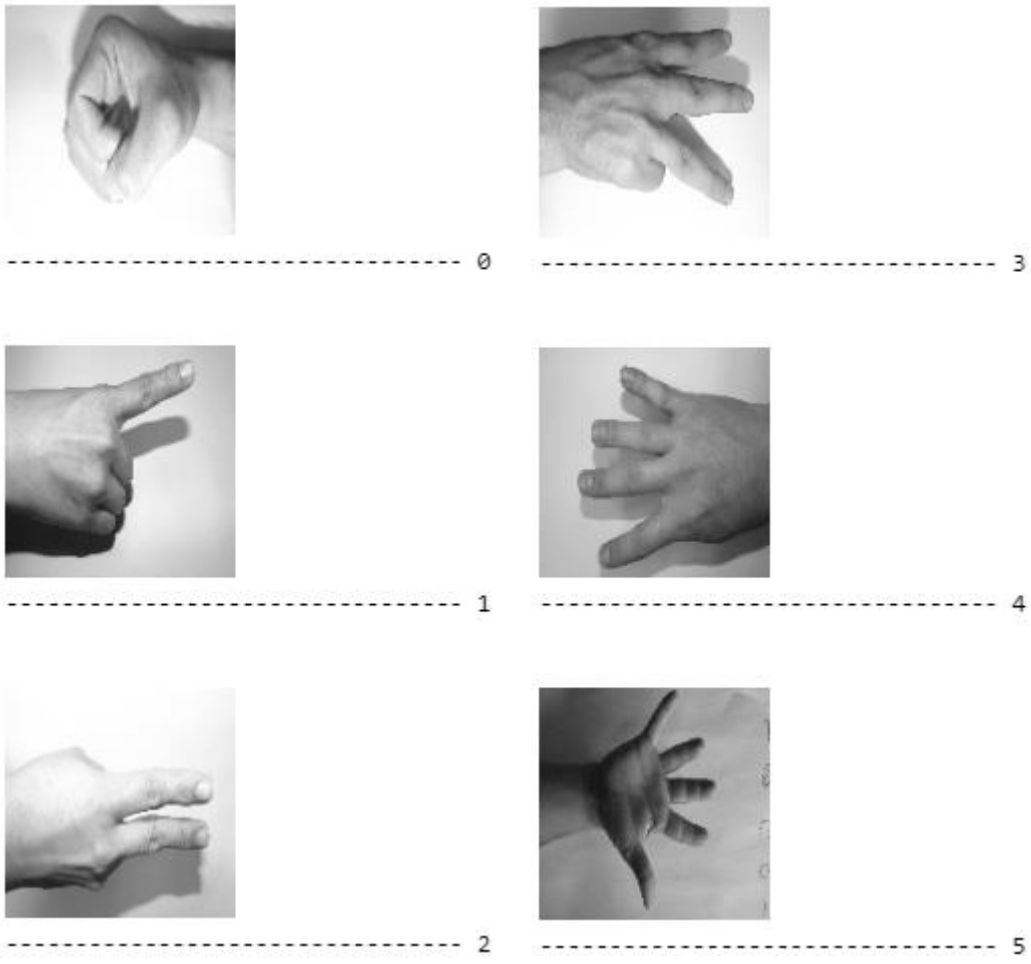


Fig-13: dataset-1 sample data



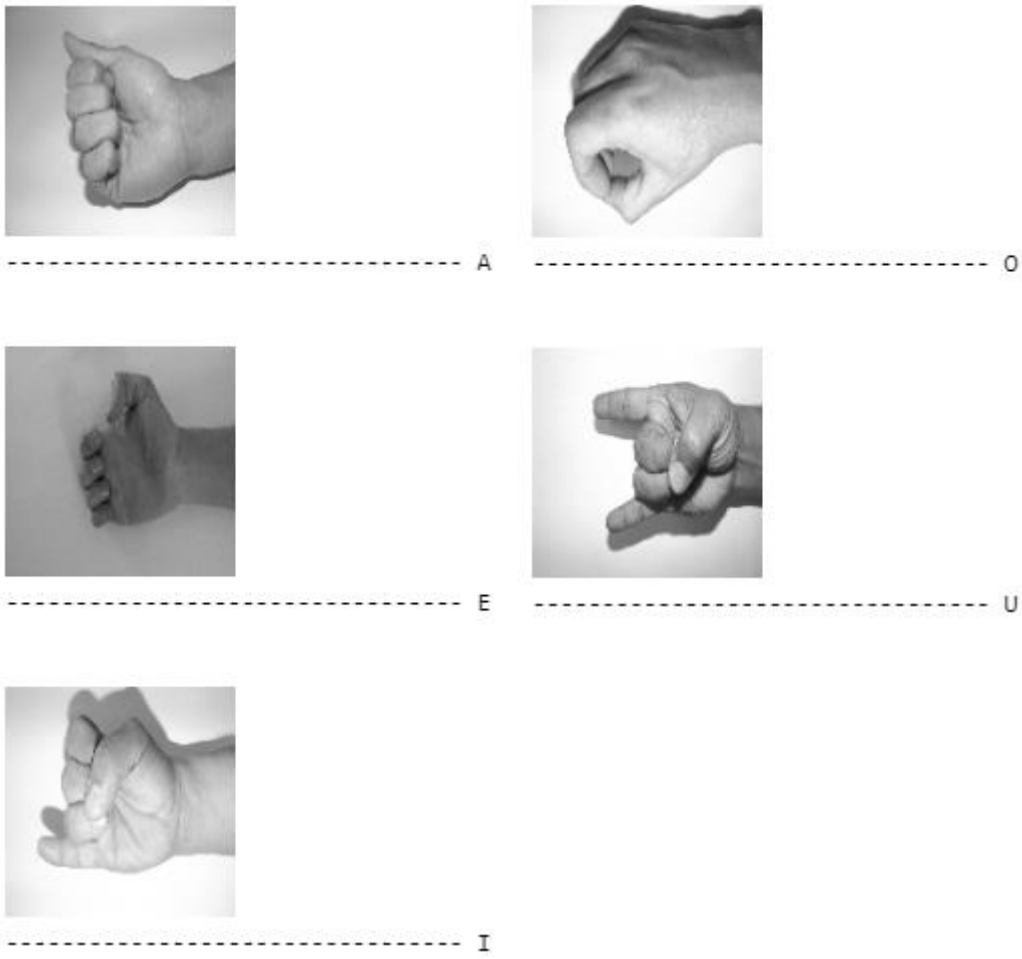


Fig-14: dataset-2 sample data

7. RESULTS:

- Model1 over data1

```
Train on 14000 samples, validate on 6000 samples
Epoch 1/5
 - 5s - loss: 0.0306 - accuracy: 0.9906 - val_loss: 0.0183 - val_accuracy: 0.9965
Epoch 2/5
 - 5s - loss: 0.0137 - accuracy: 0.9958 - val_loss: 0.0181 - val_accuracy: 0.9978
Epoch 3/5
 - 5s - loss: 0.0154 - accuracy: 0.9957 - val_loss: 0.0053 - val_accuracy: 0.9990
Epoch 4/5
 - 5s - loss: 0.0087 - accuracy: 0.9975 - val_loss: 0.0066 - val_accuracy: 0.9987
Epoch 5/5
 - 5s - loss: 0.0126 - accuracy: 0.9966 - val_loss: 0.0039 - val_accuracy: 0.9993
```

Fig-15a

	0	1	2	3	4	5	6	7	8	9
0	618	0	0	0	0	0	0	0	0	0
1	0	611	0	0	0	0	0	0	0	0
2	0	0	596	0	0	0	0	0	0	0
3	0	0	0	600	0	0	0	0	0	0
4	0	0	0	0	605	0	0	0	0	0
5	0	0	0	3	0	618	0	0	0	0
6	0	0	0	0	0	0	586	0	0	0
7	0	0	0	0	0	0	0	568	0	0
8	0	0	0	0	0	0	0	0	591	0
9	0	0	0	0	0	0	0	0	1	603

Fig-15b :confusion matrix

Fig-15a & 15b: Result of model1 over data1

- Model2 over data1

```
Epoch 1/5
219/219 - 46s - loss: 0.1186 - accuracy: 0.9687 - val_loss: 0.0101 - val_accuracy: 0.9993
Epoch 2/5
219/219 - 45s - loss: 0.0067 - accuracy: 0.9996 - val_loss: 0.0041 - val_accuracy: 0.9997
Epoch 3/5
219/219 - 45s - loss: 0.0032 - accuracy: 0.9998 - val_loss: 0.0024 - val_accuracy: 1.0000
Epoch 4/5
219/219 - 45s - loss: 0.0017 - accuracy: 1.0000 - val_loss: 0.0017 - val_accuracy: 0.9998
Epoch 5/5
219/219 - 45s - loss: 0.0012 - accuracy: 1.0000 - val_loss: 0.0012 - val_accuracy: 1.0000
```

Fig-16a

	0	1	2	3	4	5	6	7	8	9
0	618	0	0	0	0	0	0	0	0	0
1	0	611	0	0	0	0	0	0	0	0
2	0	0	596	0	0	0	0	0	0	0
3	0	0	0	600	0	0	0	0	0	0
4	0	0	0	0	605	0	0	0	0	0
5	0	0	0	0	0	621	0	0	0	0
6	0	0	0	0	0	0	586	0	0	0
7	0	0	0	0	0	0	0	568	0	0
8	0	0	0	0	0	0	0	0	591	0
9	0	0	0	0	0	0	0	0	0	604

Fig-16b: confusion matrix

Fig-16a & 16b: Result of model2 over data1

- Model1 over data2

```
Train on 2322 samples, validate on 996 samples
Epoch 1/16
 - 1s - loss: 0.0119 - accuracy: 0.9953 - val_loss: 2.1392 - val_accuracy: 0.7279
Epoch 2/16
 - 1s - loss: 0.0143 - accuracy: 0.9961 - val_loss: 2.4635 - val_accuracy: 0.7139
Epoch 3/16
 - 1s - loss: 0.0353 - accuracy: 0.9914 - val_loss: 2.3778 - val_accuracy: 0.7018
Epoch 4/16
 - 1s - loss: 0.0366 - accuracy: 0.9901 - val_loss: 2.9215 - val_accuracy: 0.7118
```

Fig-17a

	0	**1**	**2**	**3**	**4**	**5**	**A**	**E**	**I**	**O**	**U**
0	48	0	0	0	0	0	2	2	2	30	0
1	1	80	7	3	2	1	1	0	3	0	1
2	0	13	58	8	5	0	2	0	2	0	3
3	1	4	18	57	14	0	3	0	1	1	2
4	0	2	3	15	59	10	0	0	0	0	2
5	0	0	0	4	5	83	2	0	0	0	0
A	4	0	0	0	1	1	81	2	2	1	0
E	3	0	0	0	0	1	8	69	2	3	0
I	0	4	0	0	1	0	4	1	73	1	5
O	23	0	0	0	0	0	5	3	2	51	0
U	0	4	1	2	0	2	1	1	5	0	69

Fig-17b: confusion matrix

Fig-17a & 17b: Result of model1 over data2

- Model2 over data2

```
Epoch 1/25
37/37 - 8s - loss: 1.9233 - accuracy: 0.3816 - val_loss: 1.3700 - val_accuracy: 0.5201
Epoch 2/25
37/37 - 7s - loss: 1.0434 - accuracy: 0.6283 - val_loss: 1.1601 - val_accuracy: 0.5904
Epoch 3/25
37/37 - 7s - loss: 0.7884 - accuracy: 0.7339 - val_loss: 1.0297 - val_accuracy: 0.6335
Epoch 4/25
37/37 - 7s - loss: 0.7011 - accuracy: 0.7545 - val_loss: 1.0576 - val_accuracy: 0.6255
Epoch 5/25
37/37 - 7s - loss: 0.5691 - accuracy: 0.8140 - val_loss: 0.9729 - val_accuracy: 0.6355
```

Fig-18a

	0	**1**	**2**	**3**	**4**	**5**	**A**	**E**	**I**	**O**	**U**
0	52	1	0	1	0	1	2	0	1	26	0
1	1	85	8	4	0	0	0	0	0	1	0
2	1	12	52	18	2	0	1	0	0	0	5
3	1	3	13	58	17	3	1	0	0	0	5
4	0	0	3	21	46	17	1	0	0	0	3
5	0	0	0	2	8	83	0	0	0	0	1
A	3	2	0	1	0	4	76	1	4	0	1
E	5	1	0	1	0	2	4	71	1	0	1
I	4	2	0	3	2	0	4	5	50	1	18
O	39	3	0	0	0	0	0	0	1	40	1
U	0	0	4	3	0	1	0	0	1	0	76

Fig-18b: confusion matrix

Fig-18a & 18b: Result of model2 over data2

8. Validation of model over webcam captured image:

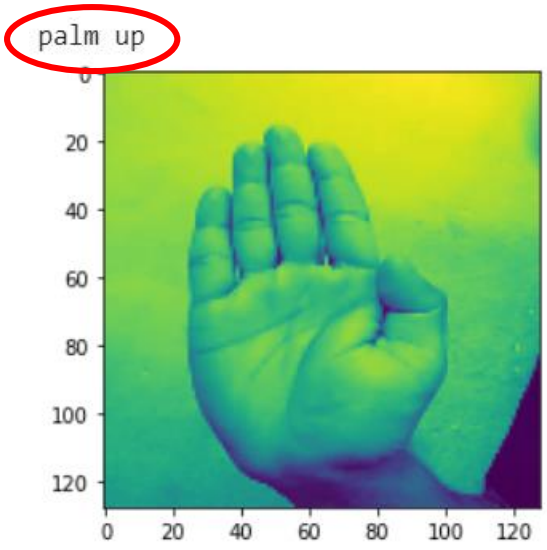


Fig-19a

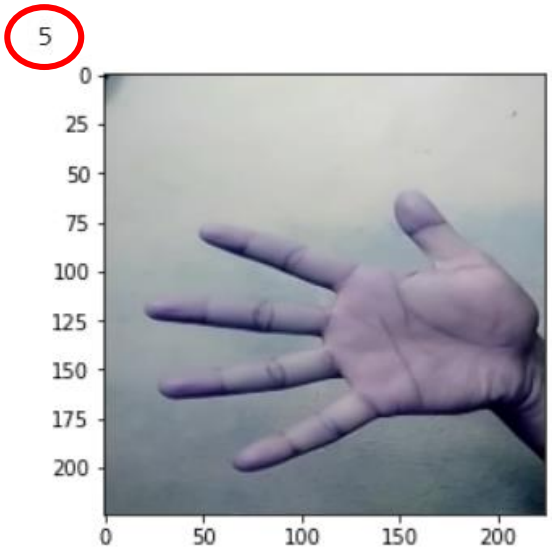


Fig-19b

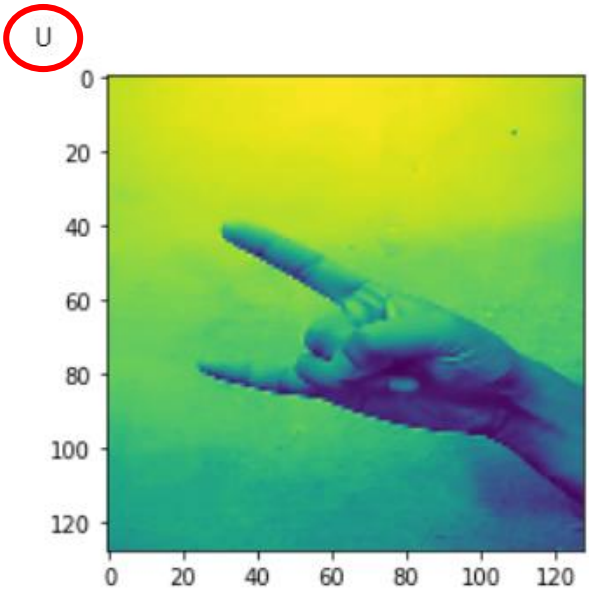


Fig-19c

Fig-19a & 19b & 19c: Classification results of webcam images

9. Results comparison using GPU in colab:

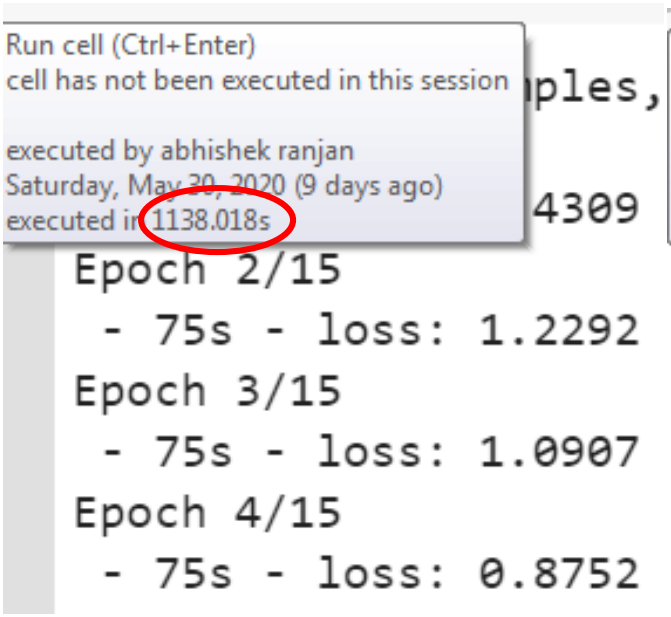


Fig-20a

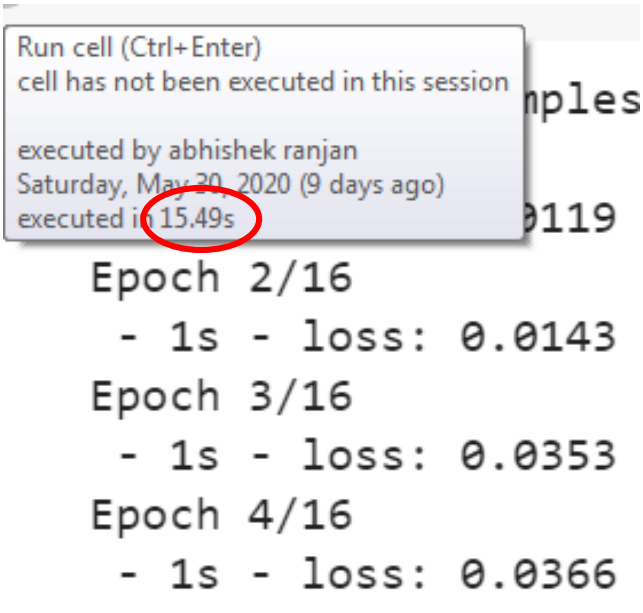


Fig-20b

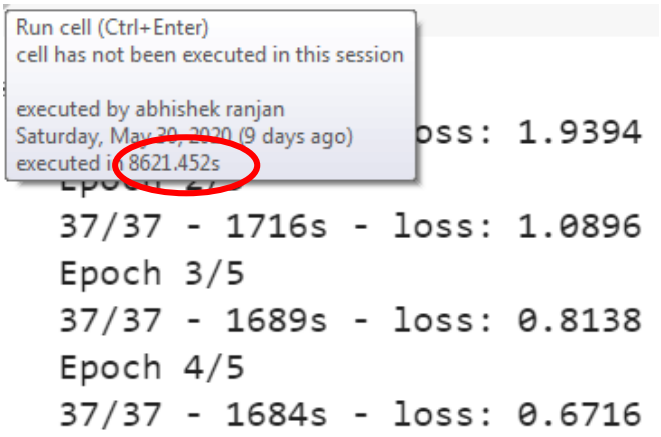


Fig-20c

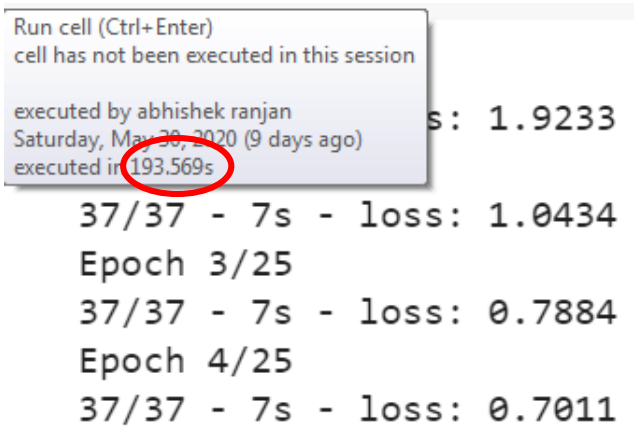


Fig-20d

Fig-20a & 20b: Time comparison in running model 1
Fig-20c & 20d: Time comparison in running model 2

It can be seen above that how powerfully GPU mode of colab enhances the results and saves lots of time. It is like boon as not everyone have access to such high end systems.

10. INFERENCES and FUTURE SCOPE OF WORK:

- In case of data1 we get very high accuracy about 99% in both the models and about 100% for validation set.
- In case of data2 we get 99% for model1 and 80% for model2 but validation accuracy is about 70% in both case.
- It was observed that data2 was little bit ambiguous and had similar images in 2 classes so accuracy obtained is less in both cases.
- After testing data over webcam image it was observed that most of the time result was not accurate so high accuracy obtained is just a result of overfitting.
- GPU mode is amazing to work with
- Overfitting elimination can be the future scope of work.
- Realtime tracking was not possible as code was running on virtual machine and to have effect of real time tracking we need to upload data at very high speed in Google server which is not possible, it can be also added in future scope of work.

Acknowledgement:

First and foremost, I would like to thank my mentor, KRITI AGARWAL, whose enthusiasm, patience and knowledge on this topic motivated me a lot. Proper guidance about resources was very helpful as there are plenty of resources so selecting right one becomes very important part in learning process.

I would also like to thank the Maths and Physics club, IIT Bombay for this wonderful initiative. It helped me learn a lot about the topic I love and helped me spend lockdown in a fruitful way.

REFERENCES:

- https://towardsdatascience.com/
- https://miro.medium.com/max/1400/1*uulvWMFJMidBfbH9tMVNTw@2x.png
- https://miro.medium.com/max/631/1*TiORvHgrJPme_lEiX3olVA.png
- https://miro.medium.com/max/578/1*ToPT8jnb5mtnikmiB42hpQ.png
- https://www.coursera.org/learn/neural-networks-deep-learning
- https://www.coursera.org/learn/convolutional-neural-networks
- https://colab.research.google.com/notebooks/intro.ipynb

MY WORK REPOSITORY:

All work including all notebooks can be found in github repository—
https://github.com/ranjanABHI/Hand_Gesture_Recognition