Neural Networks Group Case Study: Hand Gesture Recognition Project

**Problem Statement**:

This project involves building a 3D Convolutional Neural Network (CNN) to correctly recognize hand gestures by a user to control a smart TV.

The objective of this projects is to build a hand gesture recognition model that can be hosted on a camera installed in a smart TV that can understand 5 gestures.

The gestures are continuously monitored by the webcam mounted on the TV. Each gesture corresponds to a specific command:

- Thumbs up: Increase the volume

- Thumbs down: Decrease the volume

- Left swipe: 'Jump' backwards 10 seconds

- Right swipe: 'Jump' forward 10 seconds

- Stop: Pause the movie

**About the Dataset:**

The training data consists of a few hundred videos categorised into one of the five classes. Each video (typically 2-3 seconds long) is divided into a sequence of 30 frames(images). These videos have been recorded by various people performing one of the five gestures in front of a webcam - similar to what the smart TV will use.

The videos have two types of dimensions - either 360x360 or 120x160 (depending on the webcam used to record the videos).

Data Source : <https://drive.google.com/uc?id=1ehyrYBQ5rbQQe6yL4XbLWe3FMvuVUGiL>

**Neural Network Architectures Used:**

For analysing videos using neural networks, two types of architectures are used commonly.

1. **Convolutions + RNN**

One is the standard CNN + RNN architecture in which you pass the images of a video through a CNN which extracts a feature vector for each image, and then pass the sequence of these feature vectors through an RNN.  
The conv2D network will extract a feature vector for each image, and a sequence of these feature vectors is then fed to an RNN-based network. The output of the RNN is a regular softmax (for a classification problem such as this one).

In place of generic RNN, LSTM and GRU has been used in our experiments.

The image network will just give some feature representation but the LSTM/GRU will be able to decipher the sequence information to classify them as one of the class.

Then dense layer output can be fed in sequence to LSTM/GRU to get the desired output.

An LSTM has 4 gates, while GRU has 3 gates. Using GRU will significantly reduce the training times as it needs to compute values for 3 gates and its performance is at par with the LSTMs.

Another advantage here is we can use the **transfer learning** here, since the state of the art networks are already available for the image classification, we can easily use the already trained weights of resNet or VGGNet and then we can use those networks to transform these images to give the image representation (eg dense layer output).

The dense layer which will be used will be standard models like resnet, VGGnet etc.

1. **3D convolutional network**.

The other popular architecture used to process videos is a natural extension of CNNs - a 3D convolutional network.

Just like in 2D conv, you move the filter in two directions (x and y), in 3D conv, you move the filter in three directions (x, y and z). In this case, the input to a 3D conv is a video (which is a sequence of 30 RGB images). If we assume that the shape of each image is 100x100x3, for example, the video becomes a 4-D tensor of shape 100x100x3x30 which can be written as (100x100x30)x3 where 3 is the number of channels. Hence, deriving the analogy from 2-D convolutions where a 2-D kernel/filter (a square filter) is represented as (fxf)xc where f is filter size and c is the number of channels, a 3-D kernel/filter (a 'cubic' filter) is represented as (fxfxf)xc (here c = 3 since the input images have three channels). This cubic filter will now '3D-convolve' on each of the three channels of the (100x100x30) tensor.

**Data Ingestion Pipeline and Custom Generator:**

As we already know, in most deep learning projects you need to feed data to the model in batches. This is done using the concept of **generators.**

Creating data generators is probably the most important part of building a training pipeline. Although libraries such as Keras provide built-in generator functionalities, they are often restricted in scope and you have to write your own generators from scratch. For example, in this problem, you need to feed *batches of videos*, not images. Similarly, in an entirely different problem such as 'music generation,' you may need to write generators which can create batches of audio files.

In this project, we have written our own **batch data generator** using the **Python’s generator functions**. A Python generator object requires very less memory as compared to a function which is of primary importance in deep learning models.

Generators have huge advantages of performance/memory/execution time for very large datasets. Also, we have better readability and has all features available with the python native objects.

The generator **yields a batch of data** and 'pauses' until the fit\_generator calls next(). Note that in Python-3, this functionality is implemented by \_next\_().

This is based on the concept of the lazy evaluation.

Yield statement in generator returns one value at a time. Generator helps us to bring that amount of data into memory to process stuffs. This helps us to do the batch wise gradient descent on a model.

We use our own custom data generator not the in-built image data generator which is available with the Keras. The reason is we have variety of data from multiple sources like text, images, csv files, audio etc.

Otherwise it might takes 100s of Gigs of memory for processing using ImageNet etc using the normal Keras.fit() .

**About the Experiments Performed:**

Please refer the attached spreadsheet for the list of the experiments that were performed to arrive at the best model for the given problem statement:



Many a times during the training process the model’s validation loss will not go down even after few epochs, or sometimes even it might go up. We say that the gradient updates have hit the plateau and they are not moving towards the minima.

When we hit the plateau during the network training then we start reducing the learning rate. Sometimes higher learning rate will keep the optimisation process shuffling around the minima but not going towards the minima. So, we have reduced the initial learning rate to 0.0002 for Adam optimizer in our experiments wherever needed.

Above attached file has details for each model like model type, number of images, image size selected, number of parameters, batch size, number of epochs, training time, results of model performance and decision/explanation for each of the model.

**Final Model Selected:**

The model is getting saved for all the experiment for each epoch in the form of .h5 file in the disk. The final model chosen is model-00020-0.03254-0.98793-0.15436-0.95000.h5

This model is based on the transfer learning with GRU and re-training the weights. The pre-trained model of MobileNet is used for this purpose with weights of ImageNet.

Since our dataset is kind of subset of the ImageNet dataset, then we are going to transfer the knowledge of this model onto our datasets.

In Keras, MobileNet resides in the applications module. Keras offers out of the box image classification using MobileNet if the category you want to predict is available in the [ImageNet categories](https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a).

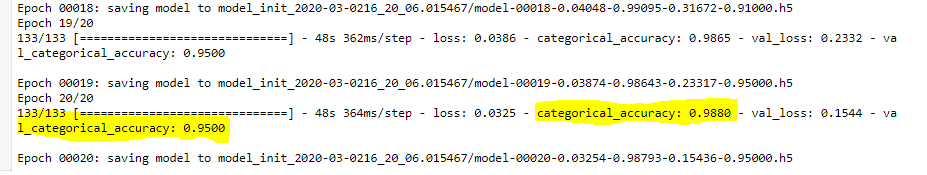
Now since the ImageNet dataset does not have exact dataset available as per our problem statement, we have not feezed the layers that we are using for our final model and we have re-trained the weights in our transfer learning.

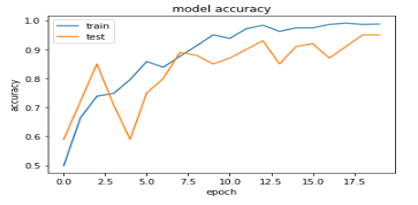
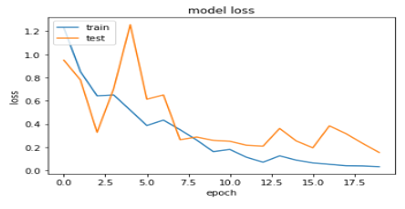
‘Fine Tuning’, generally, is when we freeze the weights of all the layers of the pre-trained neural networks (on dataset A [e.g. ImageNet]) except the penultimate layer and train the neural network on dataset B [e.g. Fashion-MNIST], just to learn the representations on the penultimate layer. We usually replace the last (softmax) layer with another one of our choice

In other words, we are not using the Fine Tuning as fine-tuning is used when the dataset used to train the pre-trained model is very similar to or the same as the new dataset which is not in our case.

for layer in base\_model.layers:

# trainable has to be false in order to freeze the layers  
 layer.trainable = False # or True



**We were able to get the excellent validation accuracy of 95 % for this selected model built using the transfer learning of MobileNet without freezing/fine-tuning.**

**References:**

1. [Jeff Knup's blog: 'Yield' and Generator Functions](https://jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/)
2. [Corey Schafer (YouTube video): Generator functions](https://www.youtube.com/watch?v=bD05uGo_sVI)
3. <https://keras.io/preprocessing/image/>
4. <https://github.com/fchollet/deep-learning-models/releases/download/v0.6/mobilenet_1_0_224_tf_no_top.h5>
5. <http://www.image-net.org/>
6. <https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a>