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.

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.

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.

**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:



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 .keras file in the disk. The final model chosen is **finalModel.keras**, it is based on **model-00019-0.01445-0.99698-0.19829-0.90000.weights.h5** weights

This model is based on the CNN and LSTM architecture. We tried training model with Convolution 3D, but were not getting good results.

Architecture involves 5 CNN layers and 1 LSTM layer at the end to handle nature of sequential image as the actual input is expected to be a video.

This model is selected because:

-- Training Accuracy : 99%, Validation Accuracy : 90%

-- Number of Parameters(1,002,085) less according to other models performance

-- Model is not overfitting

-- Learning rate gradually decreacing after 12 Epoch

- Final set of data parameters to be used

-- image\_width = 84

-- image\_height = 84

-- img\_idx = [0,4,10,12,14,16,17,18,19,20,21,22,23,24,25,26,28,29]

-- number\_ofinput\_frames = len(img\_idx)

-- batch\_size = 32

-- num\_epochs = 25

-- learning\_rate = 0.001

The best weights of CNN-LSTM: model\_init\_2024-04-0521\_44\_18.948914/model-00019-0.01445-0.99698-0.19829-0.90000.weights.h5. We considered this weight for model testing, Let's have look at the performance below

Trend of model accuracy with epochs:

A graph of a graph

Description automatically generated

Loss trend

A graph of a graph with blue and orange lines

Description automatically generated

**We were able to get the excellent validation accuracy of 90 % for this selected model**