**Deep Learning Case Study - Gesture Recognition**

**Team** – Snigdha Chakraborty, Priyanshi Pandey, Rakesh Patra

**Problem Statement**

As a data scientist at a home electronics company which manufactures state of the art smart televisions. We want to develop a cool feature in the smart-TV that can recognize five different gestures performed by the user which will help users control the TV without using a remote.

* 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.

**Data Understanding**

The training data consists of a few hundred videos categorized 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.

**Objective**

Our task is to train different models on the 'train' folder to predict the action performed in each sequence or video and which performs well on the 'Val' folder as well. The final test folder for evaluation is withheld - final model's performance will be tested on the 'test' set.

**Types of architectures suggested are:**

1. 3D Convolutional Neural Network (CONV3D)

3D convolutions are a natural extension to the 2D convolutions you are already familiar with. 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 100 x 100 x 3, for example, the video becomes a 4D tensor of shape 100 x 100 x 3 x 30 which can be written as (100 x 100 x 30) x 3 where 3 is the number of channels. Hence, deriving the analogy from 2D convolutions where a 2D kernel/filter (a square filter) is represented as (f x f) x c where f is filter size and c is the number of channels, a 3D kernel/filter (a 'cubic' filter) is represented as (f x f x f) x c (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 (100 x 100 x 30) tensor

1. CNN + 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).

**Generator**

This is one of the most important part of the code. In the generator, we are going to pre-process the images as we have images of 2 different dimensions (360 x 360 and 120 x 160) as well as create a batch of video frames. The generator should be able to take a batch of videos as input without any error. Steps like cropping, resizing and normalization should be performed successfully.

**Data Pre-Processing**

1. Cropping and Resizing of Images – It is mainly done to ensure that the NN only recognizes the gestures effectively rather than focusing on the other background noise present in the image.
2. Normalization of Images – Normalizing the RGB values of an image can at times be a simple and effective way to get rid of distortions caused by lights and shadows in an image.
3. Augmentation – For hit and trial we performed data augmentation by **slightly** **rotating** the preprocessed images of the gestures to make our model more generalizable and to achieve better accuracy. [*Point to note – As we did not observe any drastic change in accuracy or loss, Augmentation was dropped from final model*. *Moreover, due to addition of augmented data the total time to build a model tangibly increased.]*

**Model Building and Training**

**Architecture I - CONV3D**

**Mechanism**: 3D convolutions are a natural extension to the 2D convolutions in which case, the input to a 3D conv is a video (which is a sequence of 30 RGB images in this case). In 3D conv, we move the filter in three directions (x, y and z)

**Detailed Experiments and Analysis**

1. We Started with batch size equal to total number of images in the train data set.

**Outcome**: Model not trainable – Getting Resource exhausted exception

1. We reduced the batch size to half the total number of images in the train data set.

**Outcome**: Model not trainable – Getting Resource exhausted exception

1. We further reduced the batch size to 64 and build a simple conv3d network with 4 layers with kernel size of 3\*3 with Adam as optimizer and with an image resolution of 84x84, dropouts after convolutional and FC layers.

**Outcome**: Validation accuracy was too low at 0.18. The model started underfitting. The validation loss was considerably higher than the training loss and same with accuracies

1. Removed dropouts after Convolutional layer and retain after FC layer. Also use batch normalization after every convolutional layer.

**Outcome**: Model performance i.e. accuracy improved a little to 0.24 but val\_loss was still high ~0.9 Not an efficient model and need further improvement.

1. Reduced image size to 50x50 and tried to build the same model with 10 samples per frame

**Outcome**: The number of trainable parameters was around 9 digits. We experimented with different number of feature maps, and then reduced the number of feature map such that number of trainable parameters was around 8 digits.

Tried to run for 30 epochs but still the model was not trainable after doing the above changes

1. The number of trainable parameters was around 8 digits. Reduced the number of feature maps and the number of trainable parameters was reduced to 6 digits and increased the dropout.

**Outcome**: The model was trainable, but the accuracy was still low ~0.30

1. Experimented with different model configurations, parameters and various iterations and combinations of batch sizes, image dimensions, filter sizes, padding and stride length. We also played around with different learning rates and ReduceLROnPlateau was used to decrease the learning rate if the monitored metrics (val\_loss) remains unchanged in between epochs.

**Outcome**: The accuracy started improving and turned out to be 0.57 but was still low. The change in accuracy may be due to below reasons -

•Cropping the images were leading to loss of information as some images had the main person at one extreme side of the frame instead of being at the center.

•Finally, chose 120\*120 as that was the smaller of the image sizes provided to us. Resizing to a size bigger than the smallest image sizes might have distorted the image or made it too grainy to be interpreted correctly.

•Normalization of the images were also done by dividing the dimensions by 255.

Further tuning of hyperparameter needed

1. Continued use of batch normalization after every convolutional layer with Adam and added L2 regularization in the FC layer. Retained the dropouts in FC

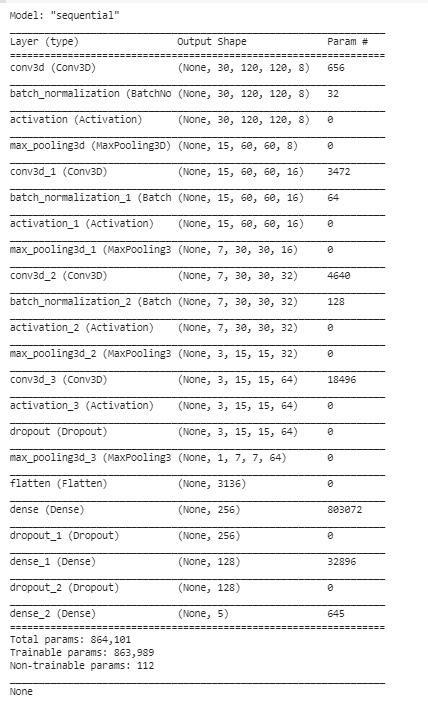
**Outcome**: Accuracy did not improve and remained at 0.57

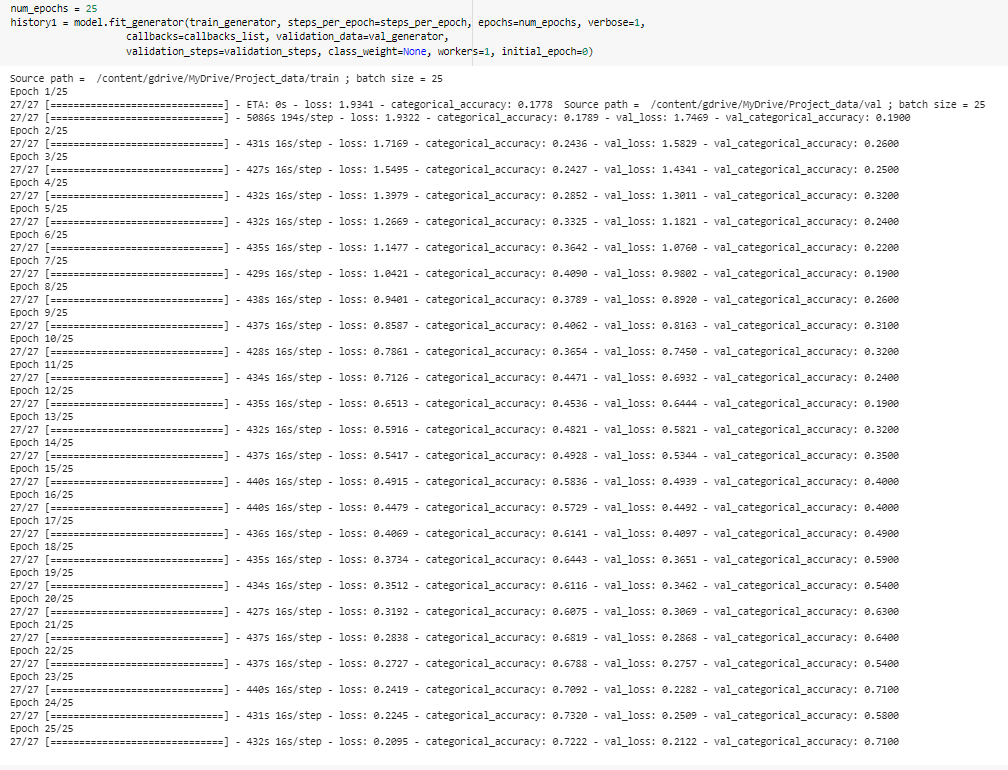
1. Tried with more nodes per conv3D layer starting with 16 and going up to 128 (in addition to the Dense layers)

**Outcome**: The number of trainable parameters grew very large but model accuracy did not improve significantly.

1. To decrease the validation loss, we experimented with different Optimizer like Nadam, sgd.
2. Finally, the optimizer SGD was chosen to generate the final model with activation function as Relu, image size 120x120 and samples per frames = 30

**Outcome**: The erratic behavior of the training accuracy and loss reduced. The training and validation losses started going down with each epoch (screenshot of model summary and last few epoch runs below)

Training and Validation accuracy was 0.72 – 0.71 respectively and loss with ~0.2, best performance among all the models tried till now. 



**Architecture II - CNN - RNN**

**Mechanism**: Transfer Learning with VGG16 Model that feeds into a Recurrent Neural Network. 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.

**Detailed Experiments and Analysis**

1. Simple CNN-RNN model (Vanilla Model)

**Outcome**: As the input to RNN was incorrect we got incompatible exception – “*Input 0 is incompatible with layer lstm\_1: expected ndim=3, found ndim=2”*

1. Inputs to the RNN (LSTM) was reshaped to be [samples, time steps, features]. We flattened the output from the Maxpool layers of the CNN and reshaped it and then fed it to the RNN. We specified the timesteps to be 30 while initializing the Simple layer.

**Outcome**: Invalid Argument exception while running the model for Ablation experiment. Expected input was to be of 4 parameters whereas generator was passing an extra param – number of frames per sample. It just needs the image shapes.

1. Changed the generator function code by adding an If-Else logic to check if the architecture is Conv3D or CNN-RNN. Changes done are: -

**Outcome**: For the CNN-RNN model, the batch data generator outputs data of shape (batch\_size, no\_of\_rows, no\_of\_cols, no\_of\_channels) along with the batch labels.

1. The accuracy was very low with SGD. Highest validation accuracy obtained was 0.25 with a high loss

**Outcome**: The model was not able to generalize the behavior to extract features from images properly

1. Removed dropouts from the initial layers and kept them only for the layers with 64 and more nodes and used Aam

**Outcome**: The accuracy was still low

1. We substituted GRU layer in-place of the Simple RNN layer and added a L2 regularization param of 0.01, added back a dropout of 0.5 to the GRU layer.

**Outcome**: The validation accuracy improved to ~0.48 and the runtime of each epoch increased compared to simple RNN due to increase in number of parameters.

*[Point to Note: We did not try LSTM due to increase in number of parameters compared with GRU as it has more number of gates opposed to GRU.]*

1. Same as CONV3D we tried different optimizers like Nadam, Adam and SGD in CNN-RNN.
   1. With Nadam we used lr=0.002, beta\_1=0.9, beta\_2=0.999, epsilon=1e-08, schedule\_decay=0.004,
   2. With SGD we used lr=0.01, decay=1e-6, momentum=0.9, nesterov=True and
   3. With Adam we used default params.
   4. We also used ReduceLROnPlateau with following params (monitor='val\_loss', factor=0.2, patience=2, cooldown=1, verbose=1)

**Outcome**: Though Nadam increased validation accuracy while training the entire sample (~0.61-0.66), the validation loss was high.

SGD gave good validation loss of around ~0.3 though the validation accuracy was around ~0.55-0.6. We decided to use SGD for the next model trials as it is more generic and simpler optimizer as opposed to Nadam

Adam which were more advanced and sometimes would fail to reach the optimal solution if not used with caution.

1. The ReduceLROnPlateau function helped a little in achieving better metrics. We experimented with its params by modifying the factor to 0.5 and adding min\_lr=0.0001 but they did not improve accuracy, so we decided to go with factor=0.2 without specifying a min learning rate.
2. CNN with transfer learning VGG16 with pre trained weights fed into GRU layers.

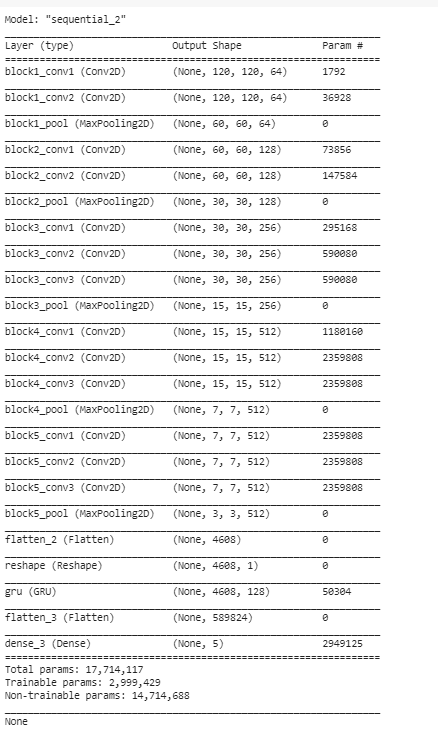
**Outcome**: The model started overfitting. We got categorical accuracy close to 1 (~0.97), a high loss of about ~0.88 with increasing validation loss of around ~1.04 and decreasing validation categorical accuracy of ~0.65- 0.67

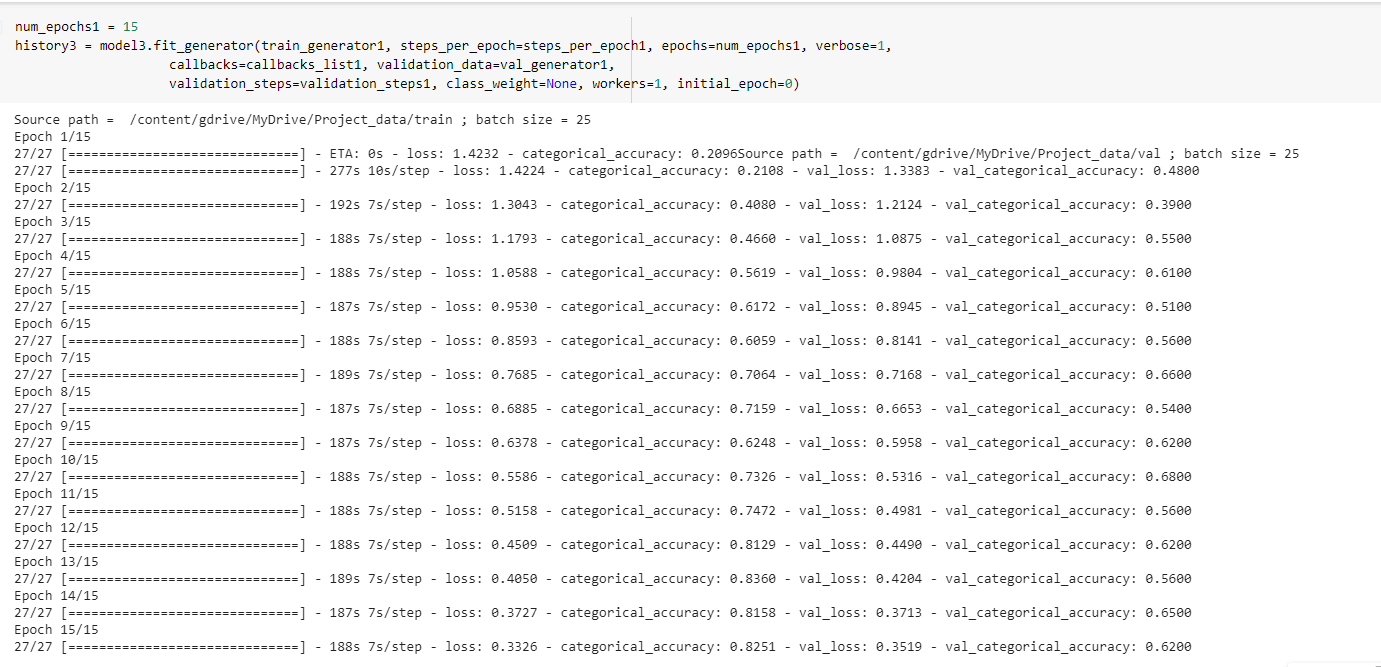
1. Added recurrent\_regularizer to GRU layer with L2=0.01

**Outcome**: The model stopped overfitting and both validation and training losses started going down with each epoch.

1. Testing VGG16\_GRU model with full sample (SGD with lr=0.01, decay=1e-6, momentum=0.9, nesterov=True).

**Outcome**: The model became more stable and validation loss started going down. We ran 15 epochs validation and training loss came down to 0.3. We got a great categorical accuracy of ~0.82 and validation categorical accuracy of ~0.62. We can say that the model accuracy will increase with additional epochs.





**Final Model**

After doing all the experiments, we finalized Final Model – CONV3D, which performed well.

Reason:

* Training Accuracy: 71%, Validation Accuracy: 71%
* Trainable Parameters 863,989 less according to other models’ performance
* Learning rate gradually decreasing after some Epochs

**General Observation**

* It was observed that as the Number of trainable parameters increase, the model takes much more time for training.
* Batch size ∝ GPU memory / available compute. A large batch size can throw GPU Out of memory error, and thus here we had to work with batch size till we were able to arrive at an optimal value of the batch size which our GPU could support
* Increasing the batch size greatly reduces the training time but this also has a negative impact on the model accuracy. This made us realize that there is always a trade-off here on basis of priority - If we want model to be ready in a shorter time span, opt for larger batch size else choose lower batch size for model accuracy.
* Increasing the batch size also impacts the memory while executing model checkpoint
* Early stopping helped in overcoming the problem of overfitting which our initial version of model was facing.
* Early stopping was used to put a halt at the training process when the val\_loss would start to saturate / model’s performance would stop improving.
* Transfer learning boosted the overall accuracy of the model.