Problem Statement:

Imagine you are working as a data scientist at a home electronics company which manufactures state of the art smart televisions. You want to develop a cool feature in the smart-TV that can recognise five different gestures performed by the user which will help users control the TV without using a remote.

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

Gesture	Action
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.

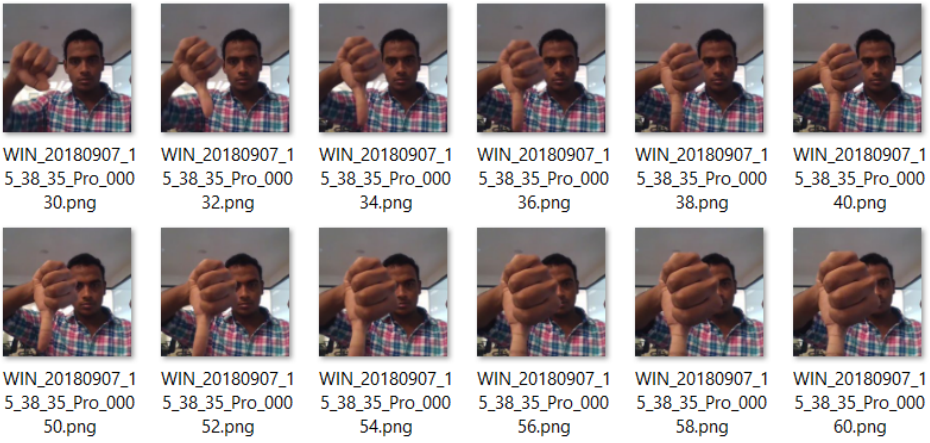
About Dataset:

Dataset Overview: The dataset for the project comprises several hundred videos, each belonging to one of five distinct classes. These classes represent different gestures. Each video is relatively short, typically lasting 2 to 3 seconds. The videos are further divided into a sequence of 30 frames or images, capturing the progression of the gesture over time.

Gestures and Classes: The dataset's videos showcase individuals performing gestures in front of a webcam. These gestures are categorized int

five specific classes. Each class represents a distinct action or motion, such as raising a thumb, lowering a thumb, swiping left, swiping right, and stopping

Variety and Source: The videos in the dataset are captured by different individuals, resulting in a diverse range of appearances, backgrounds, and execution styles. The videos emulate the interaction one might have with a smart TV, making them relevant for the intended application, possibly involvin gesture-based control of a TV interface.



Project Objective :

This endeavor aims to construct a sophisticated model designed for the recognition of five distinct hand gestures. The primary objective involves training a model utilizing the data contained within the 'train' directory, with a specific emphasis on achieving commendable performance across the 'validation', adhering to established practices in the realm of machine

learning. It is important to note that the 'test' folder has been reserved exclusively for the purpose of evaluation, where the efficacy of the finalized model will be rigorously assessed.

Architectures suggested for model training :

1. **Convolutions + 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).

2. **3D Convolutional 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 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.

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 we have to write our own generators from scratch.

For example, in this problem, we are expected to feed batches of videos, not images. 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.

Architecture Explanation:

Exploring Model Configurations and Hyper-parameters:

- During our project, we engaged in multiple rounds of experimentation, tweaking parameters like batch sizes, image dimensions, filter sizes, padding, and stride lengths.
- Our aim was to find an equilibrium between efficient training and extracting the most meaningful features from our data.

Adaptive Learning Rate Management with ReduceLROnPlateau:

- To tackle the challenge of learning rate tuning, we employed the ReduceLROnPlateau technique.
- We monitored the validation loss and dynamically adjusted the learning rate, reducing it if no progress was observed across epochs.
- This strategy allowed us to handle the nuances of learning rate dynamics, leading to more effective convergence.

Optimizers: SGD, Adam, Adagrad, Adadelta:

- We experimented with both Stochastic Gradient Descent (SGD) and Adam optimizers.
- Our choice leaned towards Adam due to its capacity to mitigate parameter variance, which significantly improved our model's accuracy.
- Due to our computational limitations, we refrained from exploring Adagrad and Adadelta optimizers, which tend to have longer convergence times.

Combating Overfitting with Batch Normalization, Pooling, and Dropout:

- As our model's complexity grew, we noticed signs of overfitting creeping in.
- We responded by introducing techniques like Batch Normalization, pooling layers, and dropout to counteract these effects.
- We closely monitored how our model's validation accuracy lagged behind training accuracy and strategically incorporated these layers.

Controlled Training Progression using Early Stopping:

- To ensure effective training, we adopted the early stopping technique.
- It allowed us to halt the training process under specific circumstances.
- We stopped training when validation loss showed signs of plateauing or when the model's performance reached a ceiling, preventing overtraining and fostering better generalization.

Observations:

Architecture	Model	Stats
CNN3D	1	**Total params:** 357,541 **Loss**: 0.279128 **Categorical Accuracy**: 0.906486 **Validation loss** : 0.651742 **Val categorical accuracy**: 0.76 **Learning rate :** 0.001
	2	**Total params: 2,218,501** **Loss**: 0.454797 **Categorical Accuracy**: 0.835596 **Validation loss** : 0.399113 **Val categorical accuracy**: 0.83 **Learning rate :** 0.001
	3	**Total params: 2,218,501** **Loss**: 0.254739 **Categorical Accuracy**: 0.92006 **Validation loss** : 0.459508 **Val categorical accuracy**: 0.8 **Learning rate :** 0.0002
	4	**Total params: 357,541** **Loss**: 0.21489 **Categorical Accuracy**: 0.924585 **Validation loss** : 0.44819 **Val categorical accuracy**: 0.8 **Learning rate :** 0.001
CNN2D_LTSM	1	**Total params: 680,357**

		Loss: 0.372279 **Categorical Accuracy**: 0.868778 **Validation loss** : 0.765292 **Val categorical accuracy**: 0.7 **Learning rate :** 0.0002
	2	**Total params: 2,572,677** **Loss**: 0.180251 **Categorical Accuracy**: 0.948718 **Validation loss** : 0.774845 **Val categorical accuracy**: 0.71 **Learning rate :** 0.0002
	3	**Total params: 4,916,869** **Loss**: 0.585273 **Categorical Accuracy**: 0.755656 **Validation loss** : 0.978482 **Val categorical accuracy**: 0.64 **Learning rate :** 0.001
	4	**Total params: 1,838,501** **Loss**: 0.337761 **Categorical Accuracy**: 0.892911 **Validation loss** : 0.876419 **Val categorical accuracy**: 0.68 **Learning rate :** 0.0002
	5	**Total params: 680,357** **Loss**: 0.752631 **Categorical Accuracy**: 0.707391 **Validation loss** : 0.905341 **Val categorical accuracy**: 0.63 **Learning rate :** 0.001
	6	**Total params: 1,000,101**

		Loss: 0.185548 **Categorical Accuracy**: 0.944193 **Validation loss** : 0.646062 **Val categorical accuracy**: 0.8 **Learning rate :** 0.0002
	7	**Total params: 51,125** **Loss**: 0.588716 **Categorical Accuracy**: 0.760181 **Validation loss** : 0.530007 **Val categorical accuracy**: 0.8 **Learning rate :** 0.000008
CNN2D_GRU	1	**Total params: 904,357** **Loss**: 0.932555 **Categorical Accuracy**: 0.612368 **Validation loss** : 0.919931 **Val categorical accuracy**: 0.53 **Learning rate :** 0.001
	2	**Total params: 1,444,261** **Loss**: 0.823208 **Categorical Accuracy**: 0.662142 **Validation loss** : 0.774209 **Val categorical accuracy**: 0.6 **Learning rate :** 0.001

Based on our observations, let's analyze the best model:

Model 2 (CNN3D): Among the CNN3D models, Model 2 stands out with a validation accuracy of 83%, which is the highest among CNN3D models. It also has a reasonable number of parameters.

Model 2 (CNN2D_LTSM): This model has a relatively high training accuracy (94.87%) and a validation accuracy of 71%. The number of parameters is moderate.

Model 6 (CNN2D_LTSM): Model 6 has a high training accuracy (94.42%) and the highest validation accuracy of 80%. The number of parameters is relatively moderate.

Model 4 (CNN2D+LSTM): This model shows a good balance between training and validation accuracies with reasonable parameters.

Conclusion:

As per my analysis, I have considered below two model as best fit i.e.

Model 2:

- Architecture: CNN3D
- Total params: 2,218,501
- Training Loss: 0.454797, Training Accuracy: 83.56%
- Validation Loss: 0.399113, Validation Accuracy: 83%
- Learning Rate: 0.001

Model 4:

- Architecture: CNN3D

- Total params: 357,541
- Training Loss: 0.21489, Training Accuracy: 92.46%
- Validation Loss: 0.44819, Validation Accuracy: 80%
- Learning Rate: 0.001

Now, upon reevaluation, considering our preference for minimizing parameters, minimizing overfitting, and having the least validation loss, Model 2 from the CNN3D architecture might indeed be the better fit:

Reasons for Choosing Model 2:

- Validation Accuracy and Loss: Model 2 achieves a higher validation accuracy (83%) and a slightly lower validation loss (0.399113) compared to Model 4.
- Overfitting: While both models demonstrate minimal overfitting, Model 2's training and validation accuracies are closer, indicating a slightly better generalization.
- Parameters: While Model 4 has fewer parameters, Model 2's parameter count (2,218,501) is still reasonable and might not significantly impact inference times.