Hand Gesture Recognition for a Smart TV Interaction System

Case Study by Venkat Lata and Deepthi Shreeya

Figure 1: Tony Stark using Hand Gesture to find the eigen value of the particle in the inverted Mobius strip model for Time Travel. Pic Credits: Tony Stark Figures Out Time Travel Scene | Avengers: Endgame (2019) Movie

Introduction

AI-powered hand gesture recognition technology uses computer vision and deep learning to interpret and respond to human hand movements, creating seamless, touch-free interaction with devices. This technology enhances user experience in smart TVs, VR, automotive systems, and healthcare by translating gestures into commands. As AI advances, gesture recognition is becoming more accurate and responsive, driving intuitive and immersive human-computer interaction.

Problem Statement

A home electronics company which manufactures state of the art smart televisions wants 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:

1. Thumbs up: Increase the volume
2. Thumbs down: Decrease the volume
3. Left swipe: 'Jump' backwards 10 seconds
4. Right swipe: 'Jump' forward 10 seconds
5. Stop: Pause the movie

Data Understanding

The training dataset comprises 663 videos, each categorized into one of the five gesture classes. Similarly, the validation dataset includes 100 videos, also classified into one of the five gesture categories. Each video, typically 2-3 seconds long, is segmented into **30 frames** (images). The videos feature different individuals performing gestures in front of a webcam, similar to the target smart TV environment.

The dataset reveals two primary video resolutions:

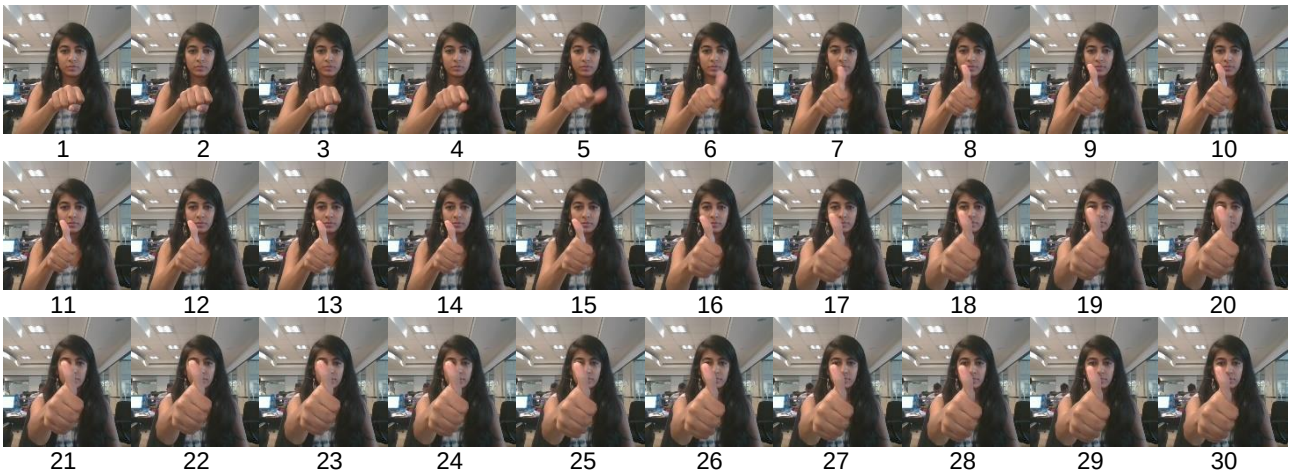
- **360x360 pixels**
- **120x160 pixels**

To ensure uniformity, pre-processing is required to standardize video dimensions. The aspect ratios are different for the two frame sizes. The 120x160 pixel frames need cropping to achieve square dimensions like 360x360 pixels frame, while the 360x360 pixel frames, being of higher resolution, should be resized to 120x120 pixels.

The provided CSV file contains metadata for each video, including:

- **Subfolder name** – Location of 30 frames
- **Gesture label** – Name of the gesture
- **Numeric label** – Values between 0 and 4, corresponding to each gesture class

Example: The subfolder named 'train/WIN_20180907_16_15_33_Pro_Thumbs Up_new' contains the following 30 sequential frames representing the '**thumbs up**' gesture for volume increase.



Dataset Link: https://drive.google.com/uc?id=1ehyrYBQ5rbQQe6yL4XbLWe3FMvuVUGiL

Data Ingestion Pipeline – Building a Custom Data Generator

Data ingestion is a critical component of any data-driven project. It ensures that raw data from various sources is collected, processed, and stored in a format suitable for analysis. The process of building a custom data generator serves as a foundational element of the data ingestion pipeline. Designed to scale horizontally, the generators can manage large volumes of video data without compromising performance. Validation rules are embedded within the generator to maintain data integrity.

In this case study, we used generators for frame transformation by cropping, resizing, and normalizing the images, as well as for relevant batch generation of the historical video frames.

Model Architectures: 3D Convolution Networks and CNN-RNN Stack

After acquiring and understanding the dataset, the next step involves experimenting with different architectures to develop the gesture recognition model. For analysing videos using neural networks, two types of architectures are used commonly.

1. 3D Convolutional Network (Conv3D)

3D convolutional networks are an extension of 2D CNNs. While 2D convolutions move the filter across two dimensions (x, y), 3D convolutions expand this movement to three dimensions (x, y, z). This approach processes video data by convolving over spatial and temporal dimensions.

Input Representation:

- A video comprising 30 RGB images of size 100x100x3 becomes a 4D tensor: 100x100x3x30
- This can be represented as: (100x100x30)x3

Filter Representation:

- In 2D: (f x f) x c (filter size f, channels c)
- In 3D: (f x f x f) x c

The cubic filter convolves over each channel of the input tensor, capturing spatial and temporal information essential for video-based classification tasks.

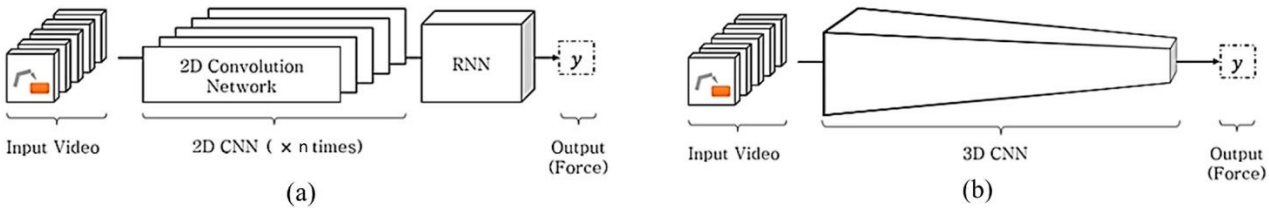


Figure 2 - (a) Heterogenous network structure in the Convolutional Neural Network (CNN) + Long Short-Term Memory (LSTM) method (b) homogeneous network structure in 3D CNN method. Pic Credits - An Efficient Three-Dimensional Convolutional Neural Network for Inferring Physical Interaction Force from Video by by Dongyi Kim, Hyeon Cho, Hochul Shin, Soo-Chul Lim and Wonjun Hwang,

2. Conv2D + RNN Stack

Another effective approach is to combine CNNs with RNNs. This architecture extracts spatial features using a 2D CNN and passes them through an RNN to capture temporal dependencies.

Workflow:

- **CNN** extracts feature vectors from each frame.
- **RNN** (LSTM or GRU) processes the sequence of feature vectors.
- **Softmax** activation classifies the gesture based on the final output.

Preferences:

- **Transfer Learning** – Pre-trained CNN models like VGG16, ResNet50, and Inception can be fine-tuned for gesture recognition rather than custom CNNs.
- **GRU over LSTM** – GRUs are computationally efficient, with fewer gates and parameters compared to LSTMs.

Experiments performed for the Case Study

Hyper-parameters used:

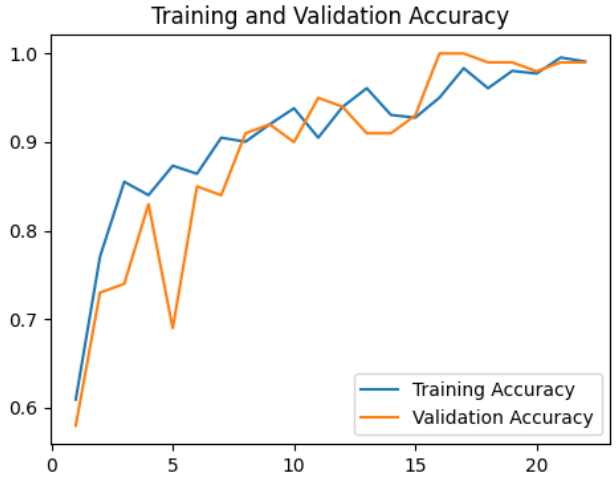
- Batch-size = 10
- Learning Rate = 0.01
- ReduceLROnPlateau patience = 3 and EarlyStopping patience = 5

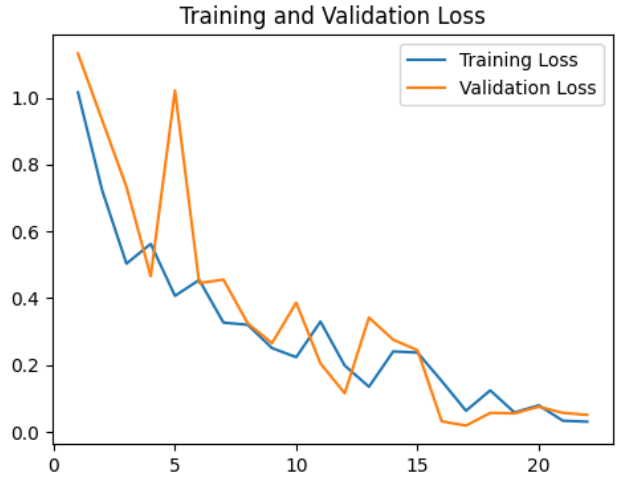
	Model	Features	Result	Decision + Explanation
1	**Conv3D**	3 Conv. Layers = (16, 32, 64) Flatten Layer 1 Dense layer 1 Dropout layer Img size = 128x128 px	Train acc. = 90% Valid. acc. = 85% Best model = Epoch 7 Epochs = 11	Model is **overfitted** and unable to learn the general patterns. We need to Increase the no. of dropout (to reduce overfitting) and convolutional layers (to improve learning).
2	**Conv3D**	4 Conv. Layers = (16, 32, 64, 128) Average Pooling Layer 2 Dense layers 2 Dropout layers Img size = 128x128 px	Train acc. = 83.5% Valid. acc. = 88% Best Model = Epoch 23 Epochs = 28	Model is **well-fitted** and **stable**, but we can push further to improve the train and validation accuracies by increasing the no. of conv. Layers or the no. of neurons.
3	**Conv3D**	6 Conv. Layers = (16, 32, 64x2, 128x2) Flatten Layer 2 Dense layers 4 Dropout layers Img size = 128x128 px	Train acc. = 22% Valid. acc. = 27% Best Model = Epoch 14 Epochs = 17	Model is **underfitted** and is not able to learn properly. We must revert back to 4 convolutional layers with more no. neurons.
4	**Conv3D**	4 Conv. Layers = (32, 64, 128, 256) Average Pooling Layer 2 Dense layers 2 Dropout layers Img size = 128x128 px	Train acc. = 96.5% Valid. acc. = 95% Best Model = Epoch 28 Epochs = 33	Model is **well-fitted** and stable. Increasing the no. of neurons **improved accuracy by 7%**. Further increasing the no. of neurons will lead to more no. of parameters. Hence changing the model architecture to improve accuracy even further.
5	**EfficientNetB0 + LSTM**	Img size = 64x64 px	Train acc. = 98% Valid. acc. = 96% Best Model = Epoch 9 Epochs = 14	Model is **well-fitted.** Validation accuracies and loss is highly fluctuating for the initial epochs but the models corresponding to the later epochs are highly stable and well-fitted. Since the no. of parameters in the pre-trained models is high we should try to reduce it by using GRU layer instead of using a LSTM layer for sequence learning.

6	**EfficientNetB0 + GRU** (removed from notebook)	Img size = 64x64 px	Model showing ResourceExhausted Error	We need to reduce the size of images even further but that will cause loss of information and produce an imperfect model, hence we will try another pretrained model.
6	**MobileNetV2 + LSTM**	Img size = 96x96 px	Train acc. = 99% Valid. acc. = 99% Best Model = Epoch 22 Epochs = 23	Model is **stable, well-fitted** and has a high train and validation accuracies, low losses. In order to reduce the no. of parameters even further we should use GRU layer instead of LSTM.
7	**MobileNetV2 + GRU**	Epochs = Img size = 96x96 px	Train acc. = 99% Valid. acc. = 99% Train loss = 0.0385 Valid. loss = 0.0568 Best Model = Epoch 21 Epochs = 22	Parameters: 2,782,469 Final Model = model-00021-0.03319-0.99548-0.05676-0.99000.keras The model is very **stable and well-fitted**. It has the least number of parameters at 99% accuracy.

Final Model

Model 7 Results:





1. Validation Accuracy is moderately fluctuating for early epochs but as the no. of epochs increase the fluctuations subside significantly, implying a **well-fitted** model.

2. The Validation Loss also shows a similar trend as the validation accuracy graph.

3. Epoch 21 is the best model with Train Accuracy = 99% and Validation Accuracy = 99%. Even with a lesser no. of parameters we got a good accuracy.

4. Using MobileNetV2 + GRU has resulted in a very stable model due to an adequate no. of parameters.

Hence the final model is

model-00021-0.03319-0.99548-0.05676-0.99000.keras

Model: "sequential"

Layer (type)	Output Shape	Param #
time_distributed MobileNetV2 (TimeDistributed)	(None, 30, 3, 3, 1280)	2,257,984
time_distributed_1 Global Average Pooling (TimeDistributed)	(None, 30, 1280)	0
gru (GRU)	(None, 128)	541,440
dense (Dense)	(None, 128)	16,512
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 5)	645

Total params: 8,381,521 (31.97 MB)

Trainable params: 2,782,469 (10.61 MB)

Non-trainable params: 34,112 (133.25 KB)

Optimizer params: 5,564,940 (21.23 MB)