1. Dataset : Jester

https://www.qualcomm.com/developer/software/jester-dataset

Overview

The Jester dataset is designed for training machine learning models to recognize human hand gestures, particularly in the context of human-computer interaction. It enables the development of responsive and accurate gesture recognition systems capable of distinguishing between subtle differences in gestures.

Content:
The dataset consists of **148,092 labeled video clips** of individuals performing a variety of hand gestures in front of a camera or webcam.

Classes:
There are **27 distinct gesture classes**

Dataset Split:
The dataset is divided into three parts to facilitate model training and evaluation:

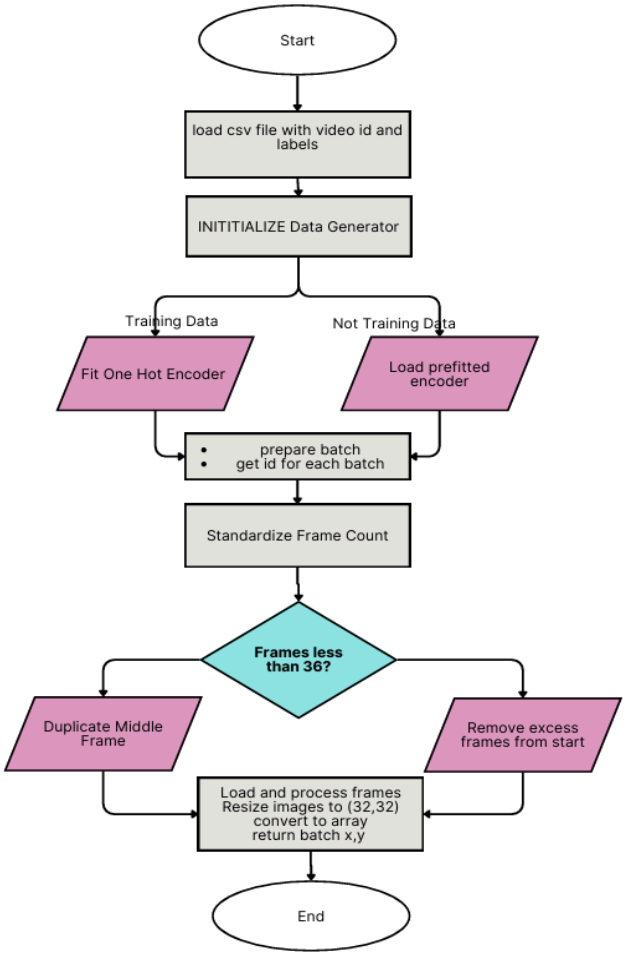
- **Training Set:** 118,562 clips
- **Validation Set:** 14,787 clips
- **Test Set (without labels):** 14,743 clips

Quality and Format:

- **Image Quality:** Each video clip is represented as a series of JPG images, with a height of 100 pixels and a variable width depending on the content.
- **Frame Rate:** Videos were extracted at **12 frames per second**, and the number of JPGs per clip varies based on the length of the original video.
- **File Structure:** The dataset is provided in a TGZ archive format, split into parts with a maximum size of 1 GB. The total download size is approximately **22.8 GB**. Each directory within the archive corresponds to a single video, and filenames are sequentially numbered starting from `1.jpg`.

The dataset was created with contributions from over **1,300 unique crowd actors**, ensuring a diverse representation of gestures.

2. Preprocessing Steps for Gesture Recognition Model



1. **Load and Filter Annotations:**
    - Import necessary libraries.
    - Define a list of gestures (`gesture_list`) that you want to keep for your training and validation datasets.
    - Load the training annotations from the CSV file (`jester-v1-train.csv`) using Pandas.
    - Filter the DataFrame to keep only the rows where the labels are in the `gesture_list`.
    - Save the filtered DataFrame to a new CSV file (`new_jester_train.csv`).
    - Repeat the same process for validation data by loading `jester-v1-validation.csv` and saving it as `new_jester_val.csv`.
2. **(Not done, can be implemented in future) Apply Data Augmentation:**

    - Implement data augmentation techniques (like rotation, flipping, and zooming) to enhance the training dataset, increasing its diversity and helping the model generalize better.(was not done)
3. **Create a Data Generator for Batch Processing:**
    - Define a `DataGenerator` class that inherits from `tf.keras.utils.Sequence`. This class should:
        - Initialize the DataFrame, batch size, image dimensions, number of frames, and channel count.
        - Use one-hot encoding for the labels and store the encoder for later use.
        - Implement methods to:
            - Get the length of the dataset.
            - Generate batches of data, ensuring frames are standardized to a specific count per sample.

Initialization (`__init__` method)

- Initializes the generator with key parameters such as `batch_size`, `image_dim`, `frames_count`, and others.
- Loads the annotations from the specified CSV file and applies one-hot encoding to the labels using `OneHotEncoder`.
- Prepares the data for training or validation by loading the appropriate encoder based on whether the generator is for training or validation.

Length of Dataset (`__len__` method):

- Calculates and returns the total number of batches based on the size of the DataFrame and the batch size.

Batch Generation (`__getitem__` method):

- Retrieves the indexes for the current batch, collects the corresponding IDs, and calls the data generation method to create the batch data and labels.

Epoch Handling (`on_epoch_end` method):

- Shuffles the data indexes after each epoch to ensure that the model sees the data in a different order during each training pass.

Data Generation (`__data_generation` method):

- Initializes an empty array for the batch of images and labels.
- Iterates through the list of IDs, loading and preprocessing the corresponding frames (images).

- Resizes and converts each image to an array, ensuring that the images are in the correct format for the model.
- One-hot encodes the labels for the batch and returns both the batch of frames and the encoded labels.

Standardizing Frame Count (`standardize_frame_count` method):

Ensures that each sample contains a fixed number of frames (`frames_count`).

If a sample has fewer frames, duplicates frames from the middle to fill the gap. If there are too many frames, it removes excess frames.

Overview:

First, we loaded the CSV file containing image IDs and labels.

The DataGenerator is initialized with various parameters.

There's a check to determine if it's training data:

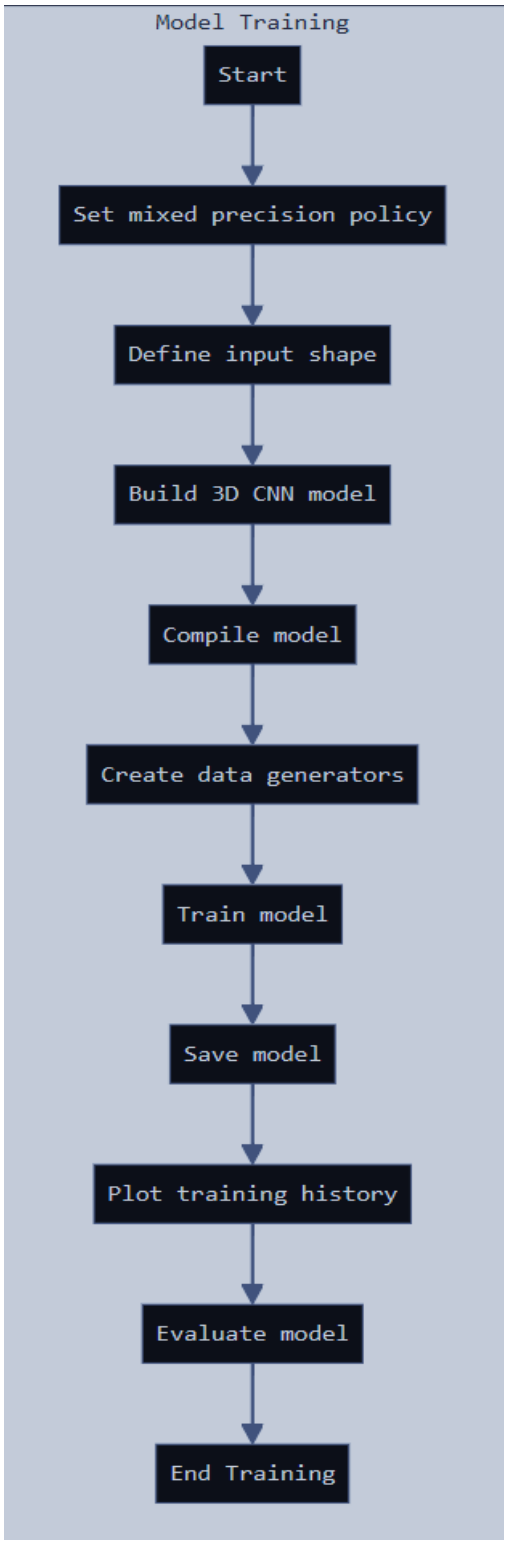
- If yes, a new OneHotEncoder is fitted on the labels and saved.
- If no, a pre-fitted encoder is loaded.

For each batch:

- File paths for each ID in the batch are retrieved.
- The frame count is standardized:
    - If there are fewer frames than required, the middle frame is duplicated.
    - If there are more frames than required, excess frames are removed from the start.
- Each frame is then loaded, resized, and converted to an array.

Finally, the labels are one-hot encoded, and the batch (X, y) is returned.

3. Model Training



Use of Mixed precision: It uses both 16-bit and 32-bit floating-point numbers in computations, improving speed and reducing memory usage. Layers like convolutions,weignts use 16-bit precision, while critical layers like batch normalization, loss values use 32-bit for stability. Although there's some reduction in precision, model accuracy remains stable. This technique is effective with **NVIDIA GPUs** equipped with **Tensor Cores**. On CPUs it has very less benefit.

Refer:
https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html

Input shape : (36,32,32,3)
(36, 32, 32, 3) refers to:

- 36 frames (like a video sequence).
- Each frame is 32x32 pixels.
- 3 represents the RGB color channels.

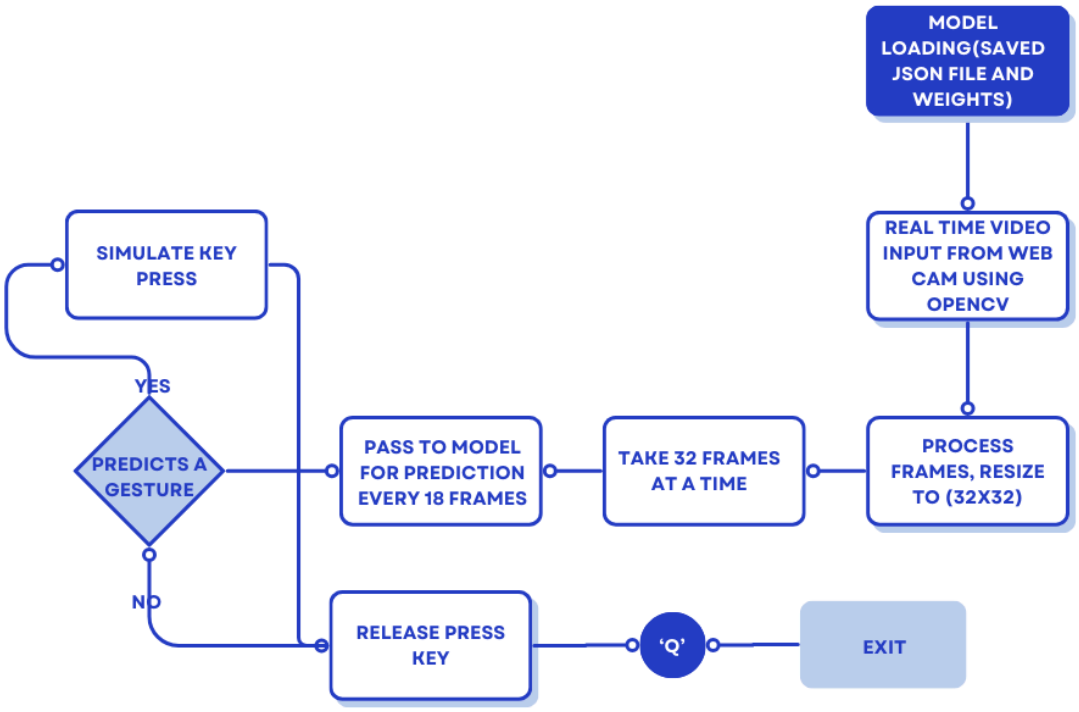
(Tried using grayscale, i.e 1 channel,performance was low)

(Tried passing 16 frames together instead of 36, i.e ,performance was low, nan error)

Create DataGenerators:
preprocessing(see above diagram)

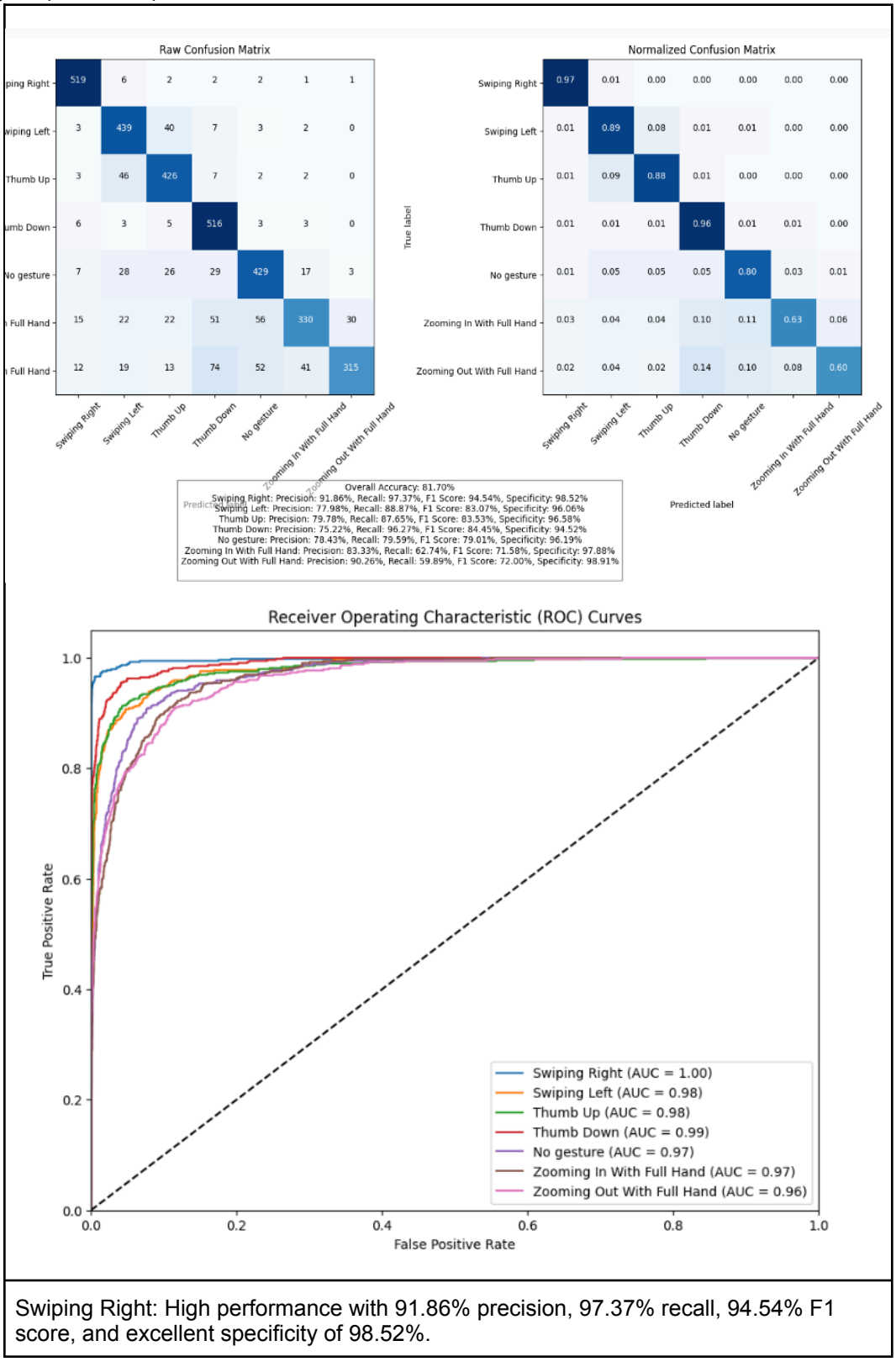
Refer: https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html

3. real-time gesture recognition



4. Results:

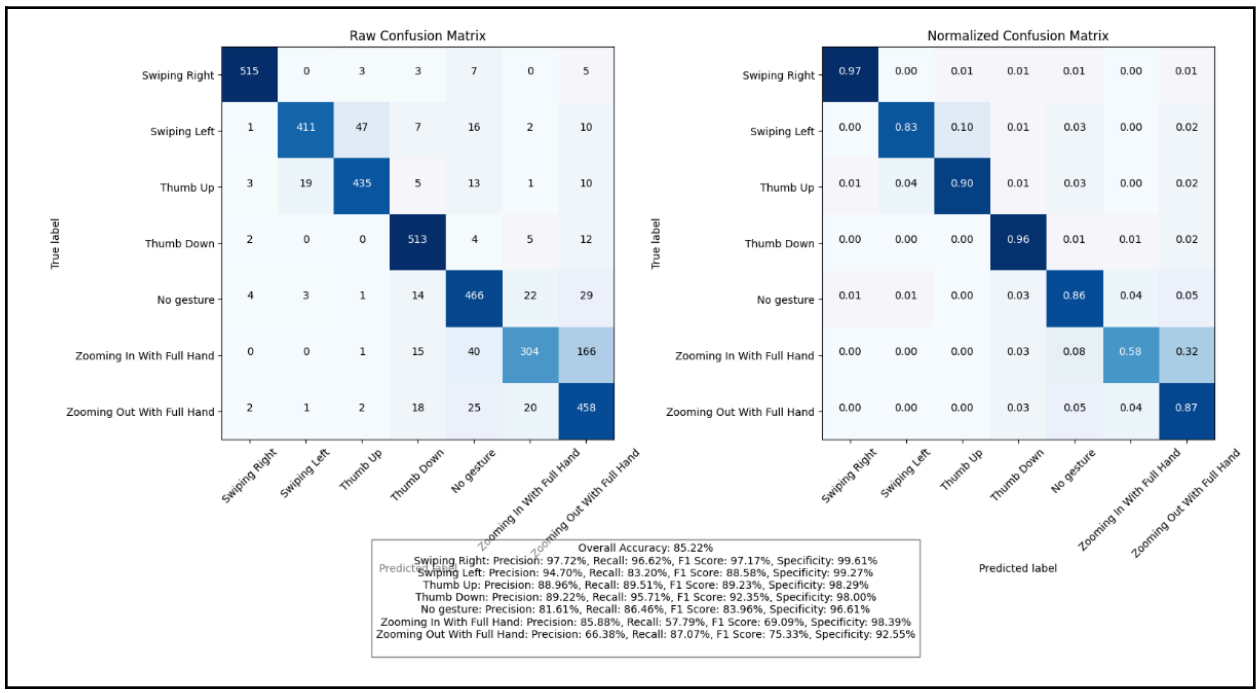
a)3DCNN-4 layers(elu, SGD)

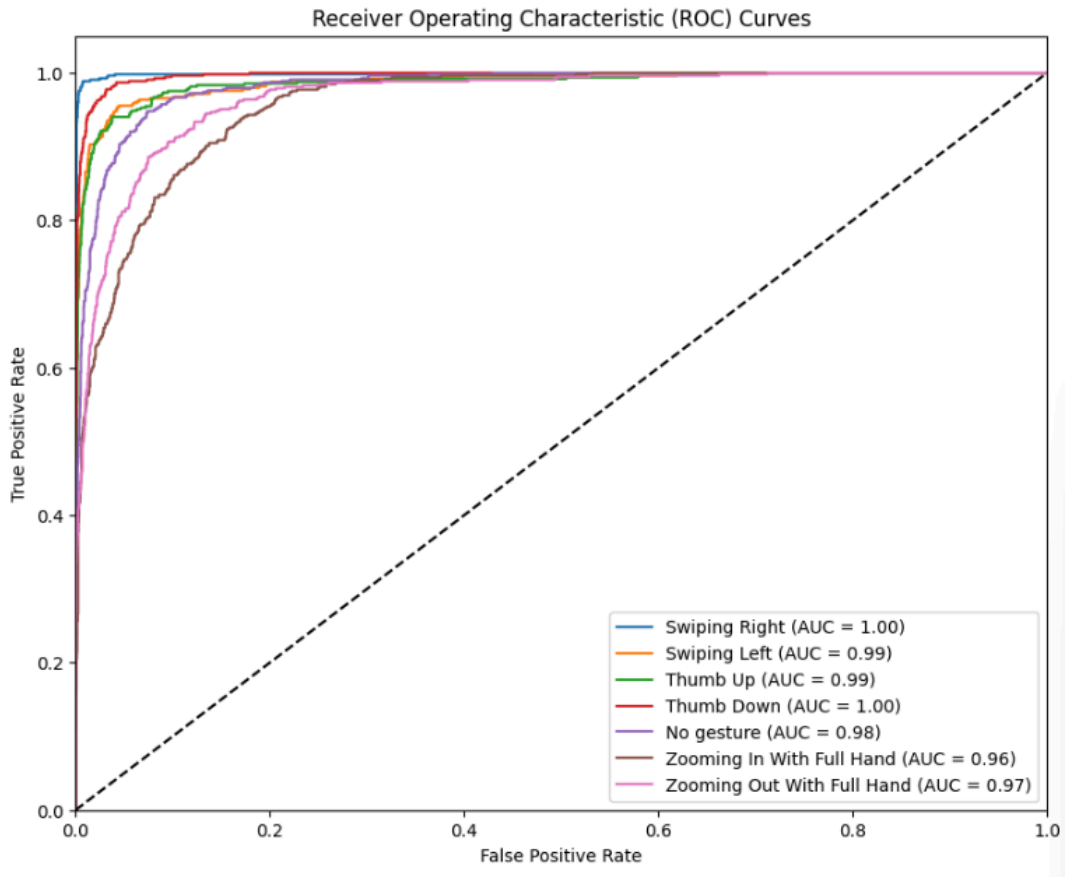


Swiping Right: High performance with 91.86% precision, 97.37% recall, 94.54% F1 score, and excellent specificity of 98.52%.

Swiping Left: Lower precision (77.98%) and recall (88.87%) compared to Swiping Right, suggesting the model struggles more with left swipes.
Thumb Up: Shows the largest misclassification with a lower precision (79.78%) and recall (82.65%).
Zooming Gestures: These show the lowest performance, especially for Zooming Out, which has a 59.89% recall and 72.00% F1 score. There's significant confusion between Zooming In and Out, which might suggest the model struggles to differentiate them clearly.
The model performs well for gestures like **Swiping Right**, **Swiping Left**, and **Thumb Down**.
It struggles with differentiating between **Zooming In** and **Zooming Out**, and there is considerable confusion between **No gesture** and other gestures.

b)3DCNN-2 ConvLSTM-2 layers(elu, Adam)





Better Class Separation: There is a clear improvement in performance across most classes compared to the previous model. The model shows better separation between similar gestures, especially for "Thumb Up," "Thumb Down," and "No Gesture."
Zooming Out With Full Hand still struggles a bit with lower precision, meaning the model often confuses it with other gestures, but its recall is quite good.

The changes in the model architecture, such as the increased number of filters, and the use of **ConvLSTM2D** layers, allow it to better capture temporal dependencies in the data. This likely contributes to the improvement in accuracy and performance for most gestures.

Also, the usage of the **Adam optimizer** with a small learning rate seems to be helping the model converge better during training.

c)3DCNN-3 layers + ConvLSTM 3 layers (elu, rmsprop)
(Evaluation metrics in notebook)

d)3DCNN-4 layers + ConvLSTM 1 layers (relu, Adam)
(Evaluation metrics in notebook)

Model	Optimizer	Accuracy	Notable Observations
3DCNN-4 layers (elu, SGD)	SGD(0.001)	81.7 %	Struggles with "Zooming In/Out" distinction and "Thumb Up" misclassification
3DCNN-2 ConvLSTM-2 layers (elu)	Adam(0.0001)	85.21 %	Improved class separation, better performance in "Thumb Down" and "No Gesture"
3DCNN-3 layers + ConvLSTM 3 layers (elu)	RMSprop(0.0001)	83.95 %	No significant improvement
3DCNN-4 layers + ConvLSTM 1 layer (relu)	Adam(0.0001)	87.85 %	Best results, least misclassifications(Zoom in, zoom out not good, misclassifies)

(Other evaluation metrics is noted in above table)

Final Notes:

- **Model 1 (3DCNN-4 layers, SGD)**: This model struggles with distinguishing between "Zooming In" and "Zooming Out," with low recall for "Zooming Out" gestures.
- **Model 2 (3DCNN-2 ConvLSTM-2 layers, Adam)**: The incorporation of ConvLSTM layers helped capture temporal dependencies better, leading to higher accuracy and improved performance for many gesture classes.
- **Model 3 (3DCNN-3 layers + ConvLSTM 3 layers, RMSprop)**:The model showed a decent performance, though slightly lower than some of the other models. The reduced dropout rate was used to maintain better generalization, and ConvLSTM layers helped capture temporal dependencies effectively.
- **Model 4 (3DCNN-4 layers + ConvLSTM 1 layer, Adam)**: .This model performed much better, with a higher accuracy compared to the 3DCNN-3 layers + ConvLSTM 3 layers model. The higher dropout (0.5) helped with regularization, preventing overfitting, and the Adam optimizer contributed to better convergence.

Thank You