Gesture Recognition Project Summary

Contributors:

- Vignesh Kumar
- Tushar Agarwal
- Chaitanya

Architectures used:

1. 3DCNN
2. CNN + RNN

Models Overview:

Model Type	Train Accuracy	Test Accuracy	Model Details
CNN+RNN	**0.93**	**0.88**	Used Transfer Learning, GRU with 64 units and Activation function as 'elu', set both dropouts as 0.5, with 32 units for densely connected layers,retrained last 15 layers in the pre-trained model, results in a good accuracy close to 90% with very low parameters
CNN+RNN	0.73	0.73	Used Transfer Learning, Set ReduceLRonPlateau as 0.1, GRU with 512 units and dense layer with 512 units , set both dropouts as 0.5, results in a moderate accuracy but more parameters
CNN+RNN	0.74	0.82	Used Transfer Learning, GRU with 64 units and Activation function as 'elu', set both dropouts as 0.5, with 32 units for densely connected layers, retrained last 12 layers in the pre-trained model , results in a good accuracy close to 90% with very low parameters
CNN+RNN	0.57	0.57	Used Transfer Learning, GRU with 64 units and Activation function as 'selu' ,'tanh' and 'elu', set both dropouts as 0.5, with 32 units for densely connected layers,and the accuracy did not improve

Model Type	Train Accuracy	Test Accuracy	Model Details
CNN+RNN	0.96	0.86	Used Transfer Learning, GRU with32 units and Activation function as 'elu', set both dropouts as 0.5, with 32 units for densely connected layers,retrained last 20 layers in the pretrained model, results in a good accuracy with very low parameters
3DCNN	**0.80**	**0.67**	Used 3 Conv3D layers each with filter sizes (16, 32,64), each conv layer is followed by Batch Normalization and Maxpooling3D layer of size (2,2,2), 2 dropouts, flattened and connected with dense layer of 128 units which resulted in relatively average accuracy of 67%
3DCNN	0.78	0.66	Used 3 Conv3D layers each with filter sizes (16,32,64), each conv layer is followed by Batch Normalization layers, the last 2 conv 3d layers size is reduced by Maxpooling3D layer of size (2,2,2) flattened and connected with dense layer of 256 units which resulted in relatively average accuracy of 66%
3DCNN	0.40	0.23	Used custom crop function which crops image based on contours, that has max area. This resulted in a very poor accuracy

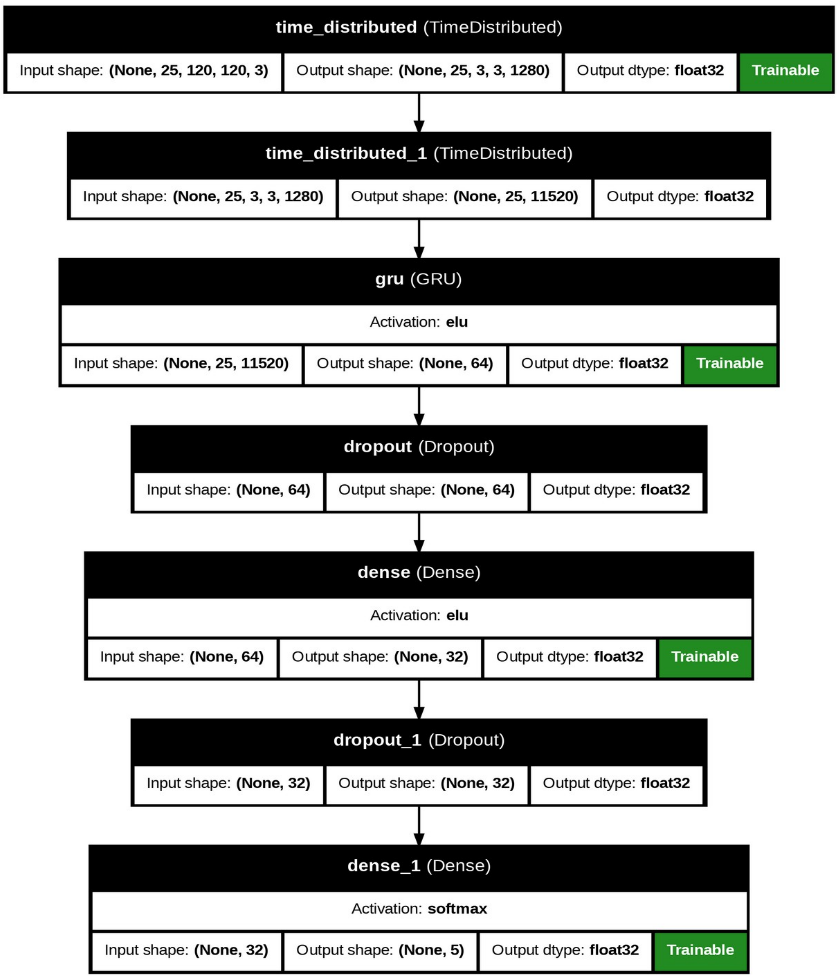
Details of Top Performing Model:

- **CNN+RNN:**
    - Total params: 11,018,257 (42.03 MB)
    - Trainable params: 3,266,757 (12.46 MB)
    - Non-trainable params: 1,217,984 (4.65 MB)
    - Optimizer params: 6,533,516 (24.92 MB)

We used pre-trained model **MobileNetV2**, initially didn't retrain any layers of the base model. After many epochs, it did not result in a considerable accuracy, hence we retrained last 6,15,20 layers respectively of the base model, the most promising result was when retraining the **last 15 layers.**

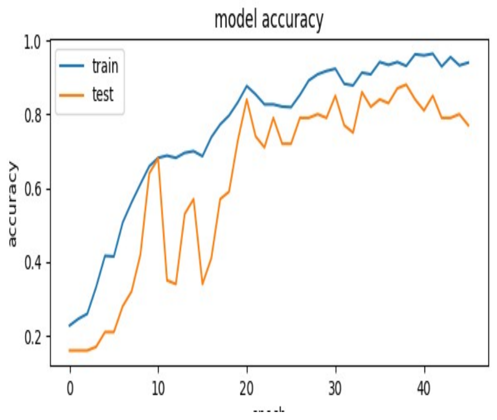
Initially we started with 10 epochs and 64 as batch size but model training and validation accuracy was only around 50%. We then increased it to 30 epochs and 64 as batch size, but model training and validation accuracy was around 90% and 60% respectively. clearly training accuracy was starting to over-fit.
Therefore, we increased no of **Epochs** to **50**, added additional **dropouts**, reduced **Batch size** to **50** and set the **'reduceLRonPlateau' to 0.8** which resulted in good **validation accuracy** of **88%** whereas the **training accuracy** was **93%.** which also resulted in **very less parameters**

- We tried to keep the model layers such that it mostly results in less no of parameters.
- As part transfer learning approach, we used MobileNetV2 as a pre-trained model for CNN+RNN Model because it was **computationally efficient**.



Our focus was more on creating a template model structure, based on which we add or remove layers to fine tune and optimize for better results, which took quite a lot of time. Due to resource constraints, we mostly trained on freely available cloud infrastructures. Many of our experiments were mostly on the Epochs, model layers and parameters perspective. Most of which are listed in **Model evaluation sheet** shared with this **ZIP Archive**.

After overcoming the initial bottlenecks, our priority shifted from building just a simple model to something more specific to the given requirements, wherein we tried to spend more time on pre processing the images, some of the steps we performed for pre processing, resulted in sub par accuracy.



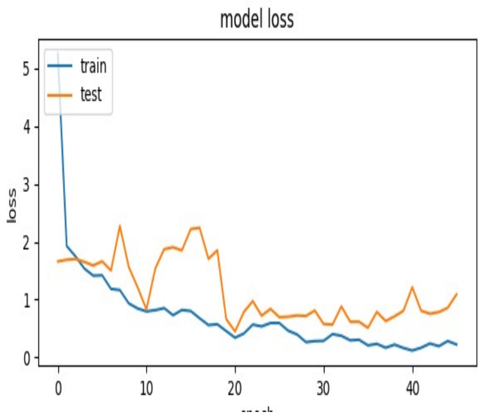
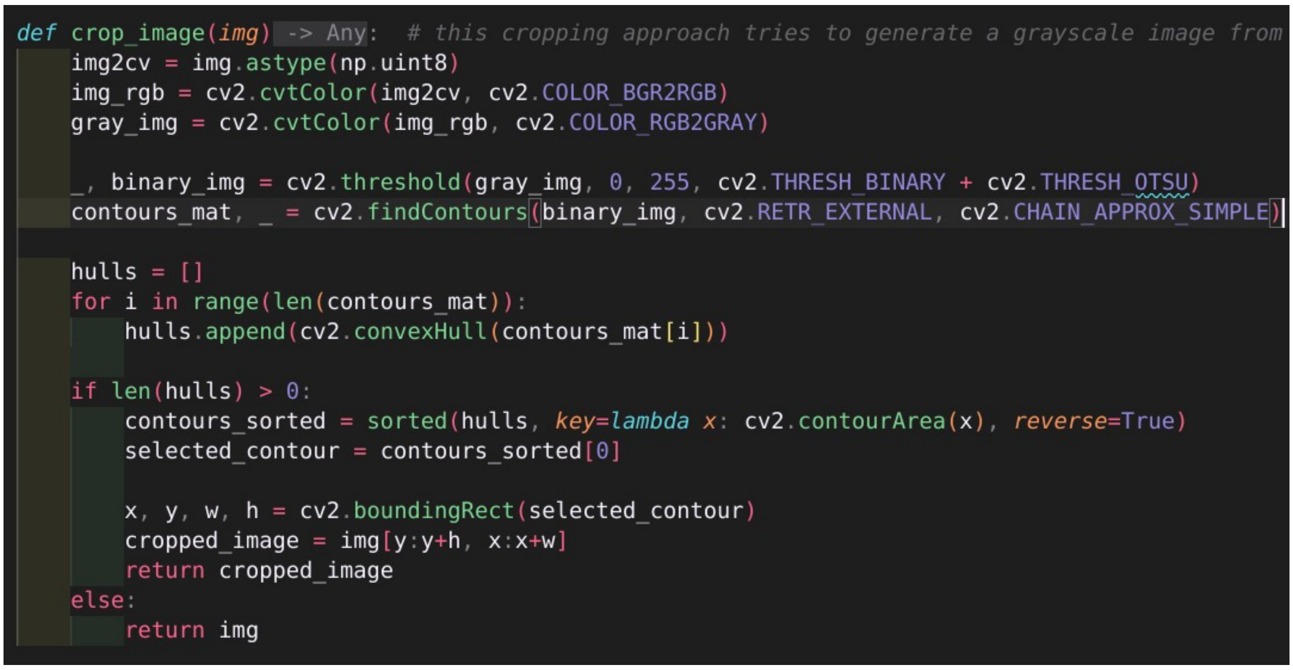
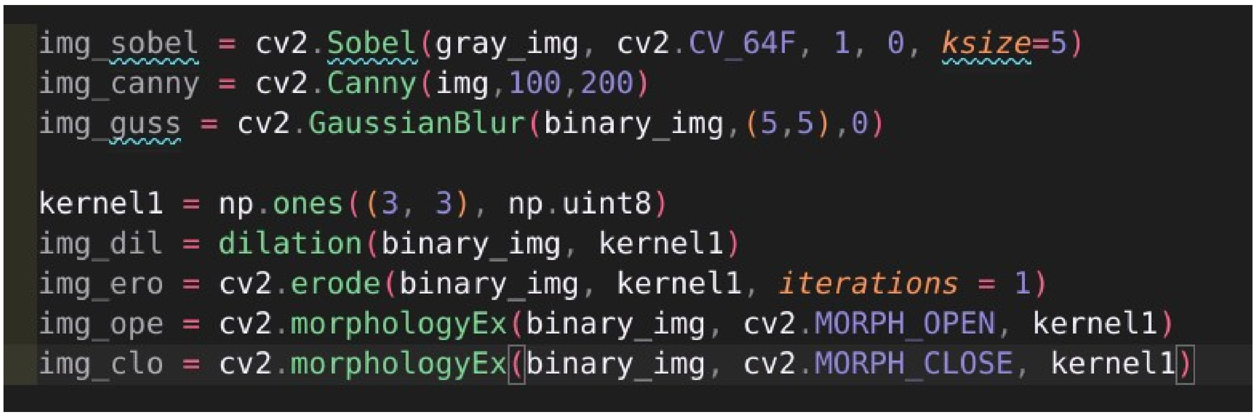


Image Preprocessing:

Cropping:
This cropping approach tries to generate a grayscale image from original image and then tries to find contours, using the contours, it is then passed to Hulls which are contours with the "convexHull" function from cv2, it tries to select the contour that has maximum area, and tries to crop it using the **boundingRect**, this approach works for certain types shapes but for some images it still returns the entire image


```python
def crop_image(img) -> Any:  # this cropping approach tries to generate a grayscale image from
    img2cv = img.astype(np.uint8)
    img_rgb = cv2.cvtColor(img2cv, cv2.COLOR_BGR2RGB)
    gray_img = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)

    _, binary_img = cv2.threshold(gray_img, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    contours_mat, _ = cv2.findContours(binary_img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    hulls = []
    for i in range(len(contours_mat)):
        hulls.append(cv2.convexHull(contours_mat[i]))

    if len(hulls) > 0:
        contours_sorted = sorted(hulls, key=lambda x: cv2.contourArea(x), reverse=True)
        selected_contour = contours_sorted[0]

        x, y, w, h = cv2.boundingRect(selected_contour)
        cropped_image = img[y:y+h, x:x+w]
        return cropped_image
    else:
        return img
```

Additionally we also certain tried data augmentation approaches, dilation, erode, opening and closing, more experiment is necessary for this to work as expected.

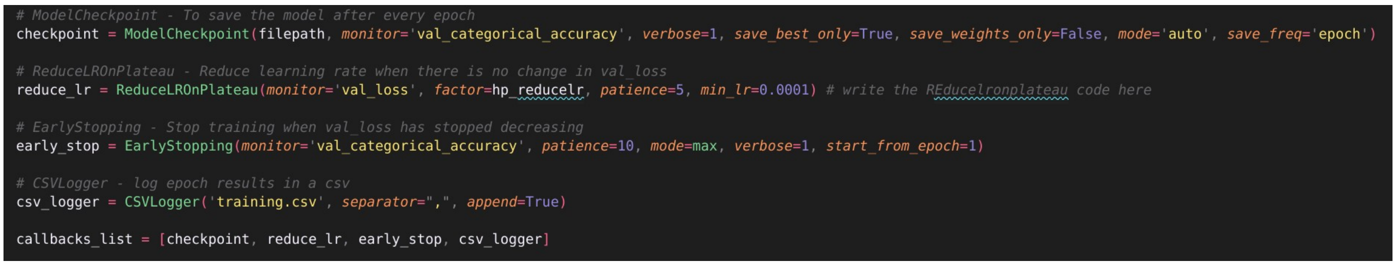

```python
img_sobel = cv2.Sobel(gray_img, cv2.CV_64F, 1, 0, ksize=5)
img_canny = cv2.Canny(img,100,200)
img_guss = cv2.GaussianBlur(binary_img,(5,5),0)

kernel1 = np.ones((3, 3), np.uint8)
img_dil = dilation(binary_img, kernel1)
img_ero = cv2.erode(binary_img, kernel1, iterations = 1)
img_ope = cv2.morphologyEx(binary_img, cv2.MORPH_OPEN, kernel1)
img_clo = cv2.morphologyEx(binary_img, cv2.MORPH_CLOSE, kernel1)
```

Callbacks:

ModelCheckpoint :
We set **Val Accuracy** as metric to monitor for creating the Weights after every epochs, only if there is aupward change in the accuracy the callback function saves the weights file in the output folder. We noticed that sometimes the model does not perform well , and if we have set an higher epoch it keeps running without any improvement in accuracy or decrease in loss , in order to overcome such scenarios we used Early Stopping after a patience level of 10 epochs.

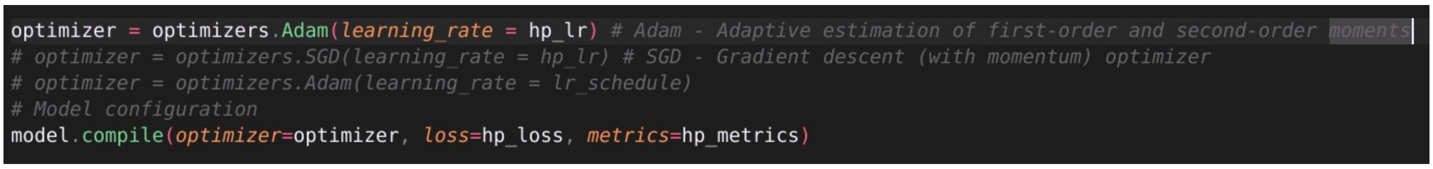
<u>ReduceLROnPlateau</u> -
since Reduce learning rate when there is no change in val_loss

<u>CSV logger</u>:
All our epoch level status have been logged to a seperate **CSV logger** for further analysis



```
# ModelCheckpoint - To save the model after every epoch
checkpoint = ModelCheckpoint(filepath, monitor='val_categorical_accuracy', verbose=1, save_best_only=True, save_weights_only=False, mode='auto', save_freq='epoch')

# ReduceLROnPlateau - Reduce learning rate when there is no change in val_loss
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=hp_reducelr, patience=5, min_lr=0.0001) # write the REducelronplateau code here

# EarlyStopping - Stop training when val_loss has stopped decreasing
early_stop = EarlyStopping(monitor='val_categorical_accuracy', patience=10, mode=max, verbose=1, start_from_epoch=1)

# CSVLogger - log epoch results in a csv
csv_logger = CSVLogger('training.csv', separator=",", append=True)

callbacks_list = [checkpoint, reduce_lr, early_stop, csv_logger]
```

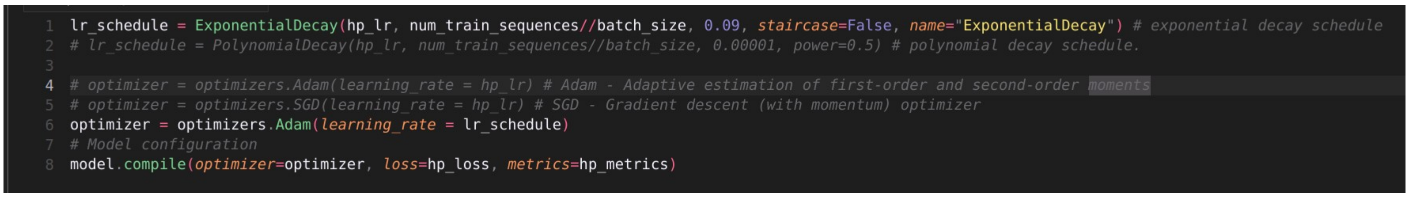
<u>Optimizers</u>:
We tried to experiment with **various optimizers** and **Adam** stands out among the optimizers,



```
optimizer = optimizers.Adam(learning_rate = hp_lr) # Adam - Adaptive estimation of first-order and second-order moments
# optimizer = optimizers.SGD(learning_rate = hp_lr) # SGD - Gradient descent (with momentum) optimizer
# optimizer = optimizers.Adam(learning_rate = lr_schedule)
# Model configuration
model.compile(optimizer=optimizer, loss=hp_loss, metrics=hp_metrics)
```

<u>Exponential decay schedule:</u>
Exponential decay in deep learning refers to the gradual reduction of a parameter, such as the learning rate or weight decay, following an exponential function. It is commonly used in learning rate scheduling, where the learning rate decreases over time as te= t0 − $\eta_t = \eta_0 e^{-\lambda t}$, ensuring stable convergence. This helps prevent overshooting and improves generalization. Exponential decay is also used in moving averages (e.g., Adam optimizer's moment estimates) and weight regularization to reduce overfitting

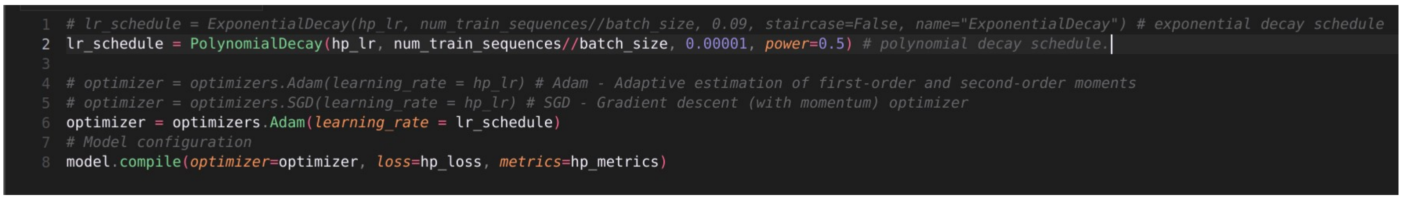
We tried the Exponential decay though it avoids the overfitting and helps in gradual reduction therefore it does not enter into a plateau , it didn't result in better accuracy. We kept the decay step as train_seq//batch size and decay_rate as 0.09 and staircase=False.



```
lr_schedule = ExponentialDecay(hp_lr, num_train_sequences//batch_size, 0.09, staircase=False, name="ExponentialDecay") # exponential decay schedule
# lr_schedule = PolynomialDecay(hp_lr, num_train_sequences//batch_size, 0.00001, power=0.5) # polynomial decay schedule.

# optimizer = optimizers.Adam(learning_rate = hp_lr) # Adam - Adaptive estimation of first-order and second-order moments
# optimizer = optimizers.SGD(learning_rate = hp_lr) # SGD - Gradient descent (with momentum) optimizer
optimizer = optimizers.Adam(learning_rate = lr_schedule)
# Model configuration
model.compile(optimizer=optimizer, loss=hp_loss, metrics=hp_metrics)
```

<u>Polynomial decay schedule:</u>
Unlike exponential decay, polynomial decay decreases more gradually, allowing finer control over learning rate reduction. It helps maintain stability during training, prevents premature convergence, and can improve model generalization, especially in long training cycles.

We kept the decay step as train_seq//batch size



```
# lr_schedule = ExponentialDecay(hp_lr, num_train_sequences//batch_size, 0.09, staircase=False, name="ExponentialDecay") # exponential decay schedule
lr_schedule = PolynomialDecay(hp_lr, num_train_sequences//batch_size, 0.00001, power=0.5) # polynomial decay schedule.

# optimizer = optimizers.Adam(learning_rate = hp_lr) # Adam - Adaptive estimation of first-order and second-order moments
# optimizer = optimizers.SGD(learning_rate = hp_lr) # SGD - Gradient descent (with momentum) optimizer
optimizer = optimizers.Adam(learning_rate = lr_schedule)
# Model configuration
model.compile(optimizer=optimizer, loss=hp_loss, metrics=hp_metrics)
```