ML PROJECT REPORT

GROUP MEMBERS	
ROLL NO	**NAME**
12140070	Aditya Sharma
12140200	Animesh Singh
12140490	Busa Viraj

PROJECT DESCRIPTION

GitHub Link: https://github.com/AnimeshSingh0/ML_PROJECT

Sign Language Recognition and Translation

Our project centers on creating a machine-learning solution for American Sign Language (ASL) interpretation and communication. By employing neural networks and deep learning techniques, our main objective was to build a practical model that connects visual ASL gestures to textual representation. This effort seeks to foster smooth communication between deaf or hearing-impaired individuals and the broader hearing community.

Our ASL Interpretation project holds importance in promoting inclusivity and breaking down communication barriers between the deaf or hearing-impaired community and the wider society.

Dataset

The dataset being used is the American Sign Language Dataset.
We proceeded with this dataset instead of the Indian Sign Language dataset because the Indian Sign Language Dataset had finger-spelling data which is finger signs for all letters of the alphabet and numbers(0-9), whereas the dataset we are utilizing has video data, processed in parquet files. Since our aim is to translate sign language in real time, it makes sense if we work with video data.

MODELS

Model - 1: Transformer

Data Preprocessing

a. Truncate long videos:
   The number of frames in the input data (data0) is reduced to a maximum of INPUT_SIZE * INPUT_SIZE frames.

b. Find the dominant hand:
   The code compares the summed absolute coordinates of the left and right hands to determine which one is the dominant hand.

c. Filter out frames without coordinates of the dominant hand:
   Frames where the dominant hand has no coordinates are removed.

d. Normalize frame indices:
   The frame indices are cast to float32 and normalized to start with 0.

e. Gather relevant landmark columns:
   Depending on the dominant hand, specific landmark columns are selected.

f. Handle cases where the number of frames is less than INPUT_SIZE:
   If the number of frames is less than INPUT_SIZE, the frames are padded with zeros, and the frame indices are padded with -1.

g. Handle cases where the number of frames needs to be downsampled to INPUT_SIZE:
   If the number of frames is greater than INPUT_SIZE, the frames are either repeated or padded to achieve the desired size.

h. Reshape and mean pool:
   The frames are reshaped to a size of INPUT_SIZE x N_COLS x N_DIMS, and a mean pooling operation is applied along the frame dimension.

i. Fill NaN values with 0:

Any remaining NaN values in the data are replaced with 0.

These steps collectively preprocess the input data to make it suitable for feeding into a neural network model.

Features

Out of the 543 landmark points (468: face, 33: pose, 21 left and right hand each), a selected few (66) were used for training the model. It was also determined whether the user was right-handed or left-handed.

Why transformer?

Transformers are well-suited for sign language translation due to their proficiency in handling sequential data and capturing long-range dependencies. Their parallelization capabilities, attention mechanisms, and adaptability to varying input sizes make them effective for interpreting the temporal and dynamic nature of sign language expressions.

Model details, training time, and validation

Following are the details regarding the model configuration

a. Scaled Dot-Product Attention (scaled_dot_product):
   Calculates attention scores between input elements.

b. MultiHeadAttention (MultiHeadAttention):
   Divides input into heads, applies attention, and concatenates results.

c. Transformer Block (Transformer):
   Combines MultiHeadAttention and a MLP sublayer.

d. Landmark Embedding (LandmarkEmbedding):
   Embeds landmark data supporting missing values.

e. Embedding (Embedding):
   Combines positional embeddings and landmark embeddings.

f. Positional Embedding (positional_embedding):
   Encodes position information for frames.

g. Loss Function (scce_with_ls):
   Custom loss combining categorical cross entropy with label smoothing.

h. Final Model (get_model):
   Assembles the complete model architecture.

The validation accuracy achieved after this is around 77%.
Training time for this model is approximately 1.23 hours (74 minutes).

Model - 2: LSTM

Data Preprocessing

- Normalized frame indices
- Clubbed together relevant landmark columns.
- Replaced Nan Values with 0 for the absent landmark features.

Feature Engineering

- Extracted features related to the face, left hand, pose, and right hand.
- Engineered new features by taking the mean and standard deviation of the x,y, and z values of the nodes.
- Concatenated the extracted features into a comprehensive feature vector (xfeat), providing a consolidated representation of the input data.

Why LSTM?

Well, our decision is backed by LSTM's knack for understanding the time-related nuances stitched into gesture sequences. Imagine it as a tech-savvy interpreter—able to smoothly adapt to different gesture lengths and complexities, like a seasoned dance partner. Not just that, it's an expert at filtering out noise and grasping the bigger picture, much like understanding the vibe of a conversation. LSTM doesn't need a massive dataset to get the hang of things; it's like a quick learner, making the best out of limited labeled data. So, in a nutshell, LSTM is our go-to because it's like having a perceptive dance partner who not only follows your moves but also gets the rhythm and context of the entire dance floor.

Model Details:

- LSTM Model Architecture:
    - Number of Layers: 2 by default, adjustable based on model complexity.
    - Dropout: 0.2 dropout applied to mitigate overfitting.
    - Architecture: Utilizes an LSTM layer followed by two fully connected (FC) layers (`fc1` and `fc2`), each with ReLU activation.
    - Loss Function: CrossEntropyLoss employed for classification tasks.

- Forward Pass:
- LSTM Layer: Processes input sequences, capturing temporal dependencies.
- Squeeze Operation: Checks and squeezes the output tensor if it retains sequence information.

- Training Steps:
    - Training Step Function: `training_step` calculates loss during the training phase.
    - Target Squeezing: Targets are squeezed to handle dimensionality.
    - Gradient Clipping: Optional gradient clipping for stability during training.

- Evaluation and Validation:
- Validation Step Function: `validation_step` computes loss and accuracy during validation.
- Epoch End Function: `epoch_end` prints and logs training and validation results.

- Additional Functions:
- Evaluation Function
- Learning Rate Handling

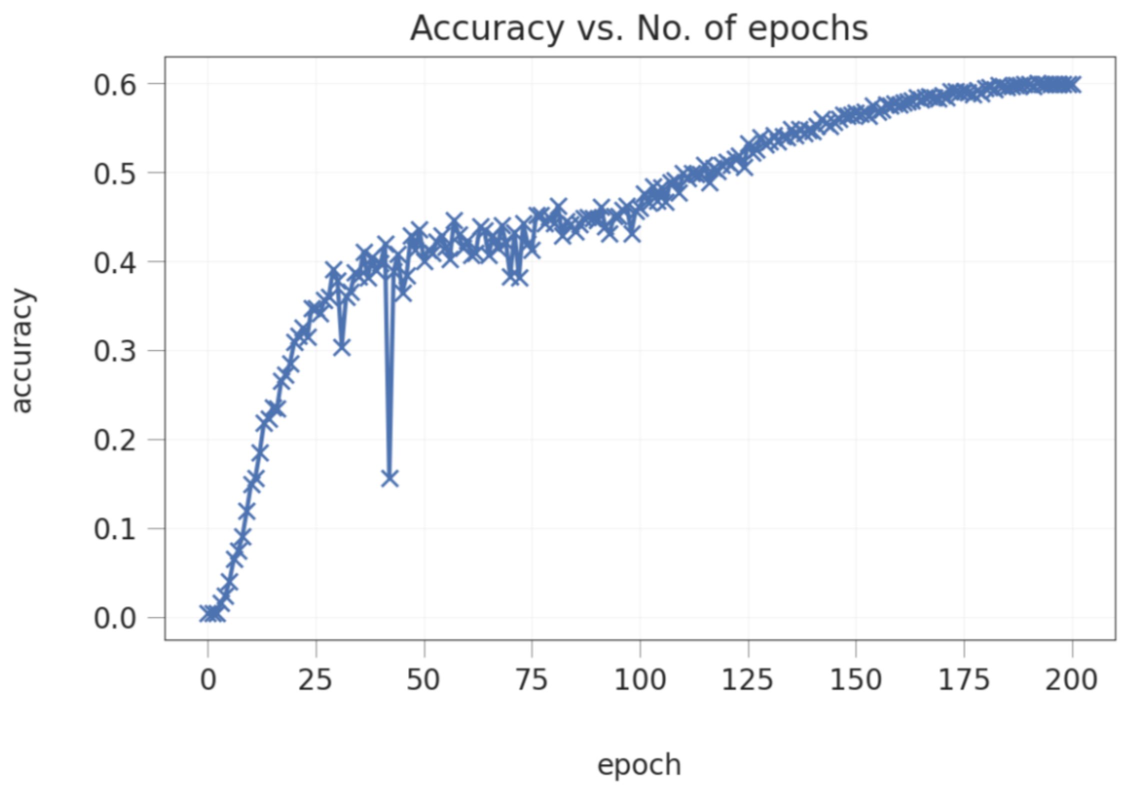
- One-Cycle Training

The Validation accuracy achieved after this is 60%.

The training time of this model is 23 minutes.

- **Hyperparameter Tuning**
  a. Optimizer = Adam
  b. Max Learning rate = 0.006
  c. Weight Decay= 1e-6
  d. epochs=200
  e. Dropout (or grad_clip)=0.1
  f. activation_function=Relu
  g. Batch_Size = 512

  These are the final hyperparameters after trying a lot of combinations. Reducing the batch size reduces the accuracy, the accuracy converges around 60 so 200 epochs are enough. Increasing the weight decay or Max Learning Rate makes the graph not converge properly.



Model - 3: Ensemble Model

Data Preprocessing:

A. **Handling Landmarks and Input Shape**
    a. Landmark Data Setup
        i. The script initializes configurations for handling landmark data, defining essential parameters such as the number of frames (NUM_FRAMES) and landmark points for different body parts like lips, left hand, and right hand.
        ii. Landmark points are organized into specific sets, including averaging sets, with the total number of landmarks (LANDMARKS) calculated as the sum of individual landmarks and sets to be used for subsequent data processing.
    b. Input Shape Determination
        i. The code establishes the input shape (INPUT_SHAPE) for the data, considering whether to include Z-axis information (DROP_Z). The shape is derived based on the number of frames and the total count of landmarks, with an altered structure if Z-axis information is excluded (DROP_Z set to True).
        ii. Additionally, a flattened input shape (FLAT_INPUT_SHAPE) is computed, to fit it as an input in the neural network.

B. **Helper Functions**
    a. Calculates 2 parameters namely **mean** and **Std_deviation.**
    b. Another function is written which concates the mean and std_devitation . Reshaping the concatenated tensor '**x_out**' into a single row, resulting in a shape of (1, INPUT_SHAPE[1]*2).
    c. We have also replaced all the NaN values with zeros to ensure all the values are finite.

C. **Feature Gen**
    a. The FeatureGen class is defined to create a custom layer for a TensorFlow-based neural network. It inherits functionalities from **tf.keras.layers.Layer** and initializes the object using the constructor method, ensuring proper initialization of the inherited attributes and methods from the superclass. This class can then be used to define custom behaviors for feature generation within a neural network architecture.

b. Input Data Processing
    i. Averaging operations are performed on specific sets of landmarks defined in averaging_sets, followed by extraction of individual landmark data points from x_in.

c. Data Padding and segmentation
    i. Data is padded symmetrically to ensure uniformity in the number of frames (NUM_FRAMES) by using the **tf.pad** function. This padding process is performed in segments (SEGMENTS) to manage data consistency.
    ii. After segmentation, mean and standard deviation computations are applied to each segment using the flatten_means_and_stds function, producing standardized feature sets.

d. NaN handling and resizing
    i. Handling NaN values is crucial. tf.image.resize is utilized to resize the data to fit a specific shape (NUM_FRAMES and LANDMARKS). Prior to resizing, NaN values are replaced with the average values along the respective dimension to mitigate their impact.
    ii. Reshaping operations are performed to generate a unified output shape (1, INPUT_SHAPE[0]*INPUT_SHAPE[1]), ensuring consistency for further processing.

e. The function finally outputs the processed and concatenated tensor, encapsulating diverse feature representations for subsequent use in modeling or analysis.

Model Training and Hyperparameter tuning

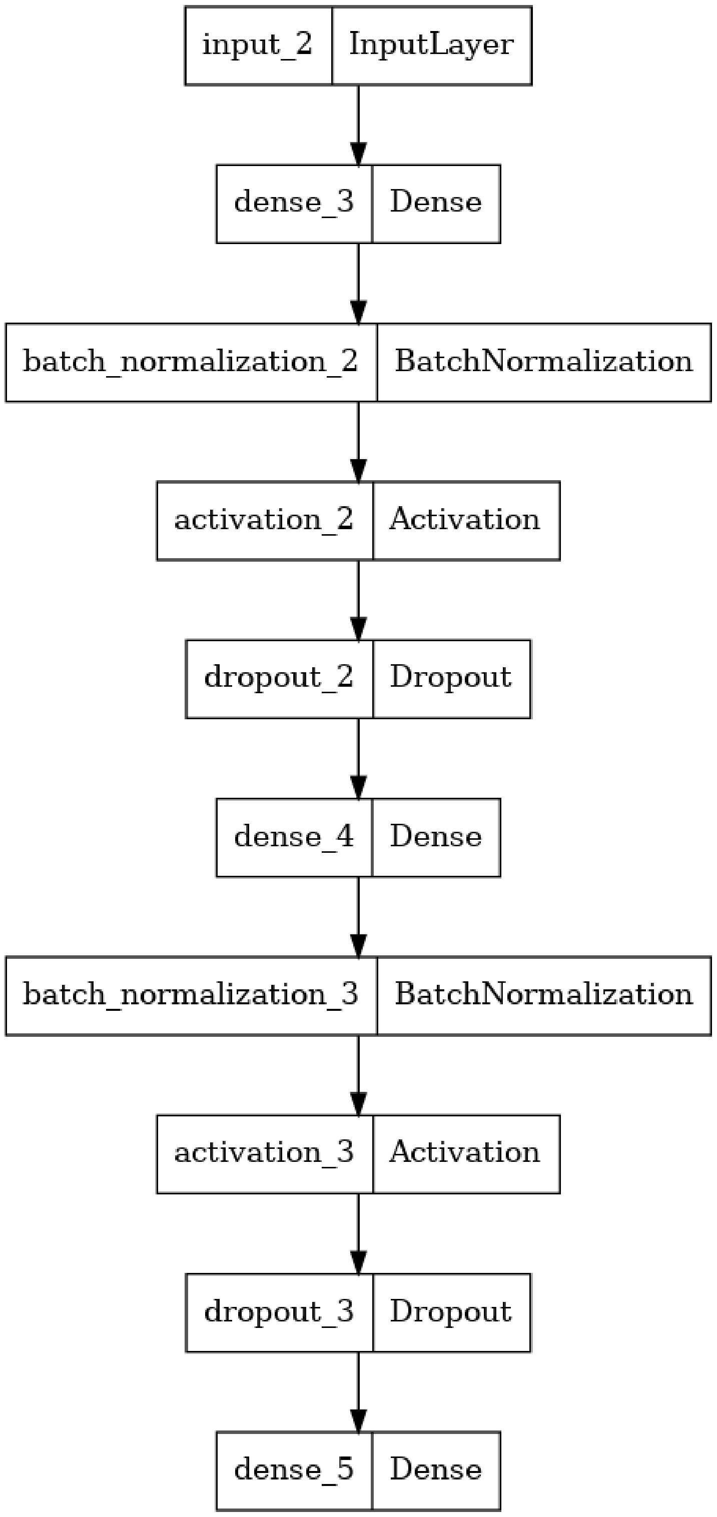
1. **Hyperparameter Tuning**: We have used Keras_tuner library for hyperparameter tuning. It takes the model and returns the optimal hyperparameters for the particular model. We are using BayesianOptimization for tuning the parameters. The parameters involved are a follows
    a. Build-Model:This is a user defined function that constructs and returns a neural network model. The hyperparameter tuning tool, in

this case, the BayesianOptimization, will optimize the hyperparameters of this model function.

b. Objective: It specifies the metric that the hyperparameter tuning aims to optimize. In this case, 'val_acc', which stands for validation accuracy, indicating that the tuning process aims to maximize the validation accuracy of the model.
c. Max_trials: It determines the maximum number of hyperparameter combinations or trials that the tuner will explore in its search for the best-performing model. For instance, setting it to 20 means that 20 different sets of hyperparameters will be evaluated.
d. Executions_per_trial: This parameter defines the number of executions or runs for each trial. Each trial involves training and evaluating the model with a specific set of hyperparameters. Having multiple executions per trial helps reduce randomness and provides a more reliable evaluation.

2. **Hyperparameters:**
    a. Optimizer = Adam
    b. Learning_rate = 0.000333
    c. Learning_rate_patience= 0.2
    d. starting_layer_size=1024
    e. dropout=0.3
    f. activation_function=gelu (Gaussian Error Linear Unit)

3. Model preparation


input_2 | InputLayer
dense_3 | Dense
batch_normalization_2 | BatchNormalization
activation_2 | Activation
dropout_2 | Dropout
dense_4 | Dense
batch_normalization_3 | BatchNormalization
activation_3 | Activation
dropout_3 | Dropout
dense_5 | Dense

a. Hyperparameter Definition: The build_model function starts by defining hyperparameters using the hp object. These hyperparameters control the structure and properties of the model.

b. Hyperparameter Search: The hyperparameters are set using the hp.Int, hp.Float, and hp.Choice methods. These methods specify the range of values, step size, and choices for each hyperparameter.

c. Hyperparameter Configuration: The DROPOUTS_len variable defines the number of dropout layers in the model. The default value is 2.

d. Model Input Layer: The model input layer is defined using the tf.keras.layers.Input function. The input shape is specified as (flat_frame_len,), where flat_frame_len is the length of the flattened audio frame.

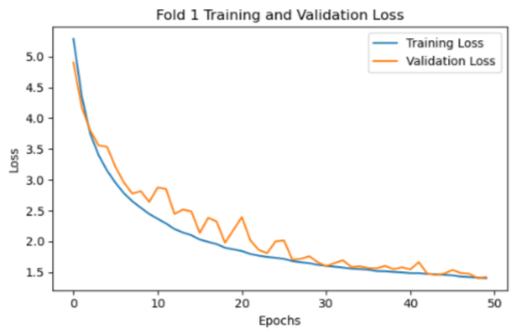
e. FC Block Definition: The fc_block function defines a single fully connected layer block. The block takes an input tensor, output channel count, dropout rate, and activation function as parameters.

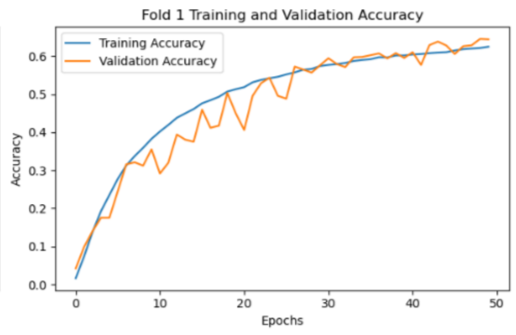
f. Model Layers: The main part of the model consists of a series of fc_block layers. Each layer reduces the output channel count by half compared to the previous layer.

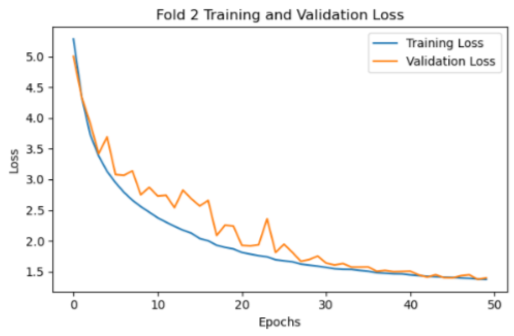
g. Model Output Layer: The model output layer is a densely connected layer with n_labels neurons, where n_labels is the number of output classes. The activation function of the output layer is softmax, which produces a probability distribution over the output classes.

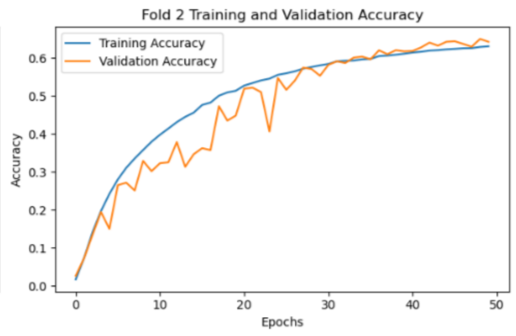
4. Training

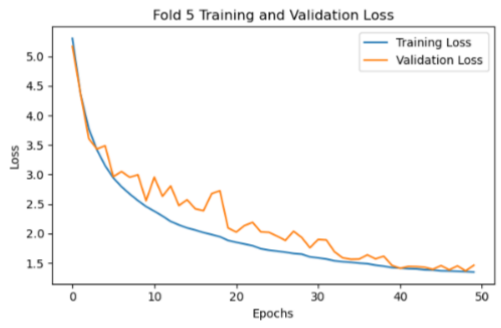
a. Training is done by dividing datasets in the 6 folds. Out of these 6 folds, we have taken an ensemble with the best 3 validation accuracy. The validation accuracies were in the range of 62-65%.
b. The number of epochs for each fold is 50 and learning rate was scheduled to decrease whenever it hits a plateau.
c. Below are the loss and validation accuracy curves.
d. The final training time is 15 minutes.

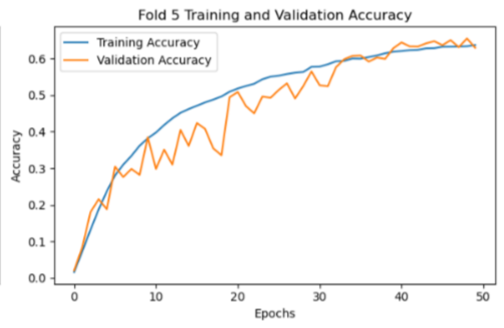


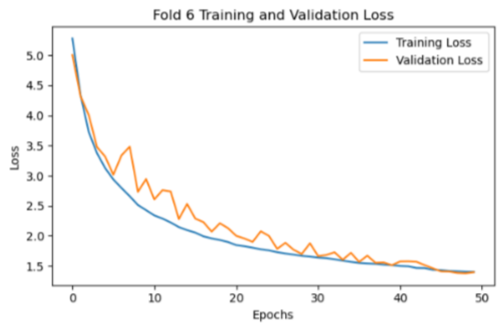


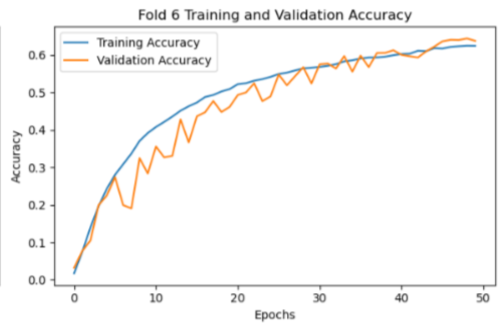












C. Comparison

Model Name	Complexity	Ease of Interpretation	Accuracy	Training
Feed Forward	Basic	Easy	55%	9 Minutes
Ensemble	Intermediate	Moderate	64%	17 Minutes
LSTM	Intermediate	Moderate	60%	23 Minutes
Transformer	High	Difficult	77%	73 Minutes

- Feed Forward: It is a basic model with only a simple neural network implementation. Basically created with the intention of understanding how to deal with the parquet files and this many points.

- Ensemble: A model which optimizes hyperparameters to get its optimum values with the help of **Keras_tuner** library and **bayesian_optimization.** Different combinations of hyperparameters were used on different folds and the best 3 were used in ensembles.

- LSTM: LSTM (Long Short-Term Memory) models effectively capture long-term dependencies in sequential data but are inherently complex, with intricate architectures and numerous parameters. Interpreting LSTMs is challenging due to their black-box nature, making it difficult to understand how specific features influence predictions, limiting their transparency compared to simpler models.

- Transformer: A transformer model utilizing multi-head attention, trained on neatly preprocessed and augmented data. Added regularization techniques like dropout and weight decay callback (reduce weights when they get too high). The model uses landmark embeddings to capture spatial and positional embedding to capture temporal components in the data.

Of course, our final model would be the Transformer Model.

Conversion of Real Time video to Parquet files (containing tabular data of video)

- Objective:
  The code aims to perform real-time holistic pose estimation using a webcam feed.
- Libraries Used:
  OpenCV: For capturing video frames and displaying output.
  MediaPipe: Specifically, the Holistic model with 543 landmarks for accurate pose estimation.
  Pandas: For creating and managing a DataFrame to store landmark data.
  NumPy: Used for numerical operations and handling NaN values.

- Processing Steps:
  Video frames are continuously captured from the default camera.
  Each frame is converted to RGB format, as required by the MediaPipe Holistic model.

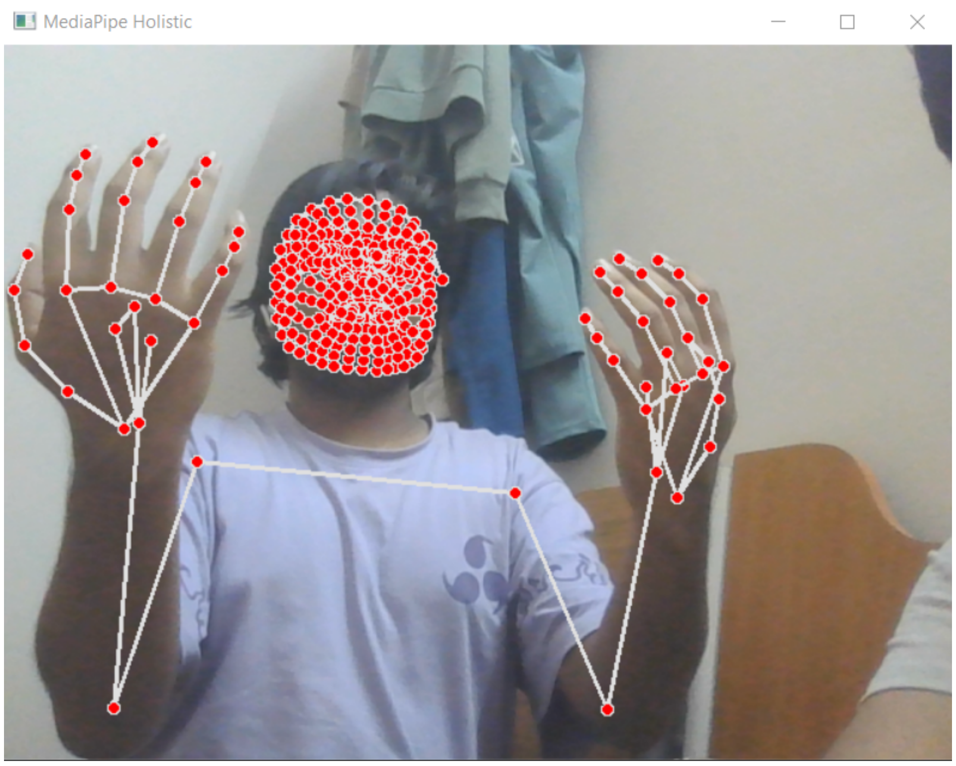
- Holistic Model:
  The MediaPipe Holistic model is employed to detect and track 543 landmarks.
  Landmarks include facial features, left and right hand points, and overall body posture.

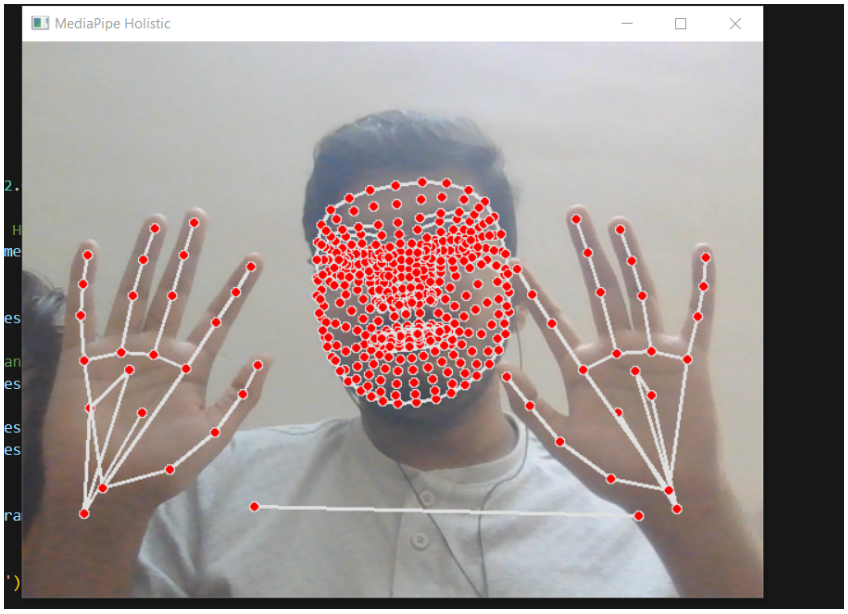
- Real-time Display:
  The original video feed is displayed in real-time.
  Landmarks are annotated on the video feed, providing a visual representation of the pose estimation.

- User Interaction:
  The program runs until the user presses the 'q' key.

- Result:

The 'landmarks.parquet' file contains detailed information about 543 (468 facial, 33 pose, 21 left and right hand each) landmarks per frame of the video.





D. Comparison to Research Paper

The architecture employed in the referenced research papers is notably more intricate and complex than the models we utilize. While our models may not attain the same level of complexity, resulting in a potential shortfall in accuracy compared to those sophisticated architectures, they still perform admirably in their designated tasks.

It's crucial to acknowledge that while our models may not reach the pinnacle of accuracy achieved by the complex architectures in the research papers, they strike a balance by offering a more straightforward and interpretable solution that may be advantageous in certain contexts. In essence, our models successfully fulfill their objectives, while maintaining interpretability.

E. Contribution of Each Member

- **Aditya Sharma (12140070)**
    - Data Augmentation for the transformer model
    - Full Preprocessing for LSTM Model.
    - Feature Engineering for the model
    - Full Architecture of the LSTM Model
    - LR handler and Weighted Decay
    - Completed the first half of the speech-to-gesture pipeline (Not finished yet)

- **Animesh Singh (12140200)**
    - Full Preprocessing for transformer model
    - Configuration of Transformer Model (including dropout layers) and Loss function
    - Learning Rate Scheduler and Weight Decay callback
    - Landmark embeddings to capture spatial and positional embedding to capture temporal components.
    - Visualisation of 543 points of face, hands and pose of each frame whose coordinates are fed into the model.

    - Completed Pipeline from real-time video to generate parquet files having details of 543 landmarks (468 facial, 33 pose, 21 left and right hand each) using OpenCV.
    - Helped in finalizing the pipeline.

- **Busa Viraj (12140490)**
    - Pre-processing for the first draft
    - basic feed-forward nn model training and hyperparameter tuning .
    - Helped in finalizing the pipeline.
    - A better model with an ensemble of the different models (best 3) on an accuracy basis.
    - The hyperparameters were tuned with the help of keras_tuner library where we have used bayesian optimization to get the optimal hyperparameters for our model.
    - The complete model preparation with all the pre-processing, data preparation and model training .Final ensemble model translation into a tflite based model for ease of usage in the further stages..