Introduction

Chapter 1

1.1 INTRODUCTION

The proposed project, AI-Driven Hand Gesture recognition system ,aims to bridge the communication gap for individuals who use sign language . American sign language is a predominant sign language Since the only disability D&M people have is communication related and they cannot use spoken languages hence the only way for them to communicate is through sign language. Communication is the process of exchange of thoughts and messages in various ways such as speech, signals, behavior and visuals. Deaf and dumb(D&M) people make use of their hands to express different gestures to express their ideas with other people. Gestures are the nonverbally exchanged messages and these gestures are understood with vision. This nonverbal communication of deaf and dumb people is called sign language.

Sign language is a visual language and consists of 3 major components:

Fingerspelling	Word level sign vocabulary	Non-manual features
Used to spell words letter by letter .	Used for the majority of communication.	Facial expressions and tongue, mouth and body position.

Figure 1.

we will basically focus on producing a model which can recognize Fingerspelling based hand gestures in order to form a complete word by combining each gesture. The gestures we aim to train are as given in the image below.

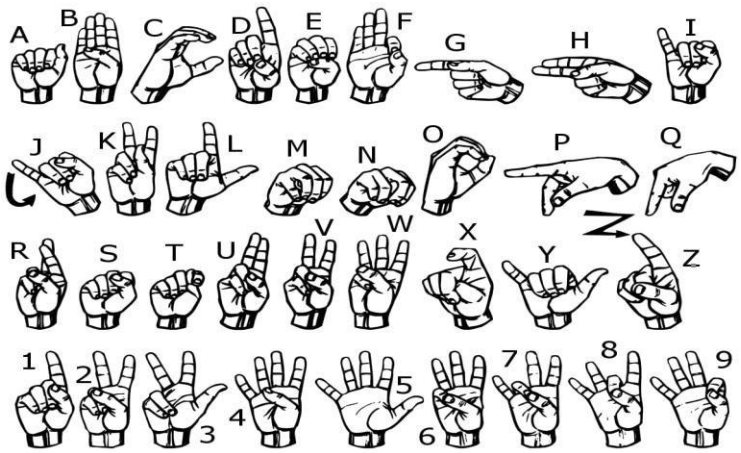


Figure 2.

1.2 LITERATURE REVIEW

The field of hand gesture recognition, particularly for sign language interpretation, has evolved significantly over the past decades, transitioning from traditional computer vision methods to sophisticated deep learning architectures. This review outlines the key technological milestones and research trends that form the foundation for this project.

1.2.1. Traditional Computer Vision and Sensor-Based Approaches

Early research in gesture recognition heavily relied on classical image processing techniques and specialized hardware. Studies often used **color-based segmentation** and **template matching** to identify hand shapes. While computationally inexpensive, these methods were highly sensitive to lighting conditions, skin tone, and complex backgrounds, limiting their robustness in real-world scenarios. To overcome these limitations, researchers employed **data gloves** and **depth sensors** (like the Microsoft Kinect or Leap Motion controller). These devices provided precise skeletal and positional data, leading to high accuracy. However, their high cost and requirement for specialized hardware made them impractical for widespread, everyday adoption, confining them primarily to laboratory settings.

1.2.12. The Rise of Machine Learning and Feature-Based Methods

The advent of machine learning introduced more adaptive models. Researchers began extracting hand-crafted features such as **Hu moments, Histogram of Oriented Gradients (HOG)**, and **scale-invariant feature transform (SIFT)** descriptors from pre-processed images. These features were then used to train classifiers like **Support Vector Machines (SVMs)**, **K-Nearest Neighbors (KNN)**, and **Random Forests**. This approach represented a significant step forward in handling variability. However, the performance of these systems was intrinsically linked to the quality and comprehensiveness of the manually engineered features, which remained a complex and time-consuming task.

1.2.3. Deep Learning and Convolutional Neural Networks (CNNs)

The breakthrough in gesture recognition came with the widespread application of **Deep Learning**, particularly **Convolutional Neural Networks (CNNs)**. CNNs automate the feature extraction process, learning hierarchical representations directly from pixel data, which makes them immensely powerful and less dependent on pre-processing. Landmark studies demonstrated that CNNs could achieve near-human-level accuracy on large-scale image datasets. In sign language recognition, researchers have successfully applied architectures like **AlexNet, VGG, and ResNet** for static gesture classification, consistently outperforming traditional machine learning methods. The ability of CNNs to learn complex patterns from data makes them the current standard for image-based recognition tasks.

1.2.4. Real-Time Hand Landmark Detection with Integrated Frameworks

A critical enabler for real-time performance has been the development of robust hand-tracking

models. Google's **MediaPipe Hands** framework represents a significant advancement in this area. It provides a pipeline that can detect 21 precise 3D landmarks of a hand from a single frame in real-time, even on modest hardware. This eliminates the need for complex background subtraction and makes the system invariant to lighting and skin color to a large degree. By leveraging such frameworks, researchers can focus on building classification models on top of these standardized, high-quality landmarks, drastically reducing development time and computational overhead.

1.2.5. Gaps in the Literature and Our Contribution

Despite these advancements, a survey of existing literature reveals several gaps. Many high-accuracy models are computationally heavy and unsuitable for real-time inference on standard laptops without a dedicated GPU. Furthermore, numerous proposed systems lack a **fully integrated, multi-modal output system** (seamless text and speech) that is essential for practical communication. There is also a strong reliance on **cloud-based processing** for some advanced models, which limits functionality in areas with poor internet connectivity.
Our project aims to address these gaps by developing a system that **integrates the real-time efficiency of the MediaPipe framework with the high classification accuracy of a custom-trained CNN model**. The focus is on creating an end-to-end, **offline-capable solution** that runs effectively on common hardware, providing both text and speech output to create a practical and accessible tool for breaking down communication barriers.

1.3 PROBLEM FORMULATION

Communication barriers for individuals with hearing and speech impairments are a significant issue in society. Many people without these disabilities lack knowledge of sign language, making interaction and daily communication challenging. This gap often leads to isolation, dependency, and a lack of opportunities for individuals who rely on sign language as their primary means of communication. Existing solutions, such as human interpreters, are not always available, and text-based alternatives can be slow and ineffective in real-time conversations.

Despite advancements in AI and assistive technologies, most existing solutions do not provide an efficient, real-time translation for sign language users. Mobile applications and wearable devices for sign language recognition are either expensive, lack accuracy, or require additional hardware, making them inaccessible for many users. Furthermore, the absence of a universal sign language adds complexity, as different regions have their own variations (e.g., ASL, BSL, ISL). The need for a system that can accommodate multiple sign languages in a real-time setting remains largely unmet.

This project seeks to develop a real-time AI-driven solution that captures hand gestures via a camera, processes them using a trained AI model, and translates them into both text and speech. By leveraging computer vision and machine learning, the system ensures high accuracy and responsiveness. The inclusion of a display and speaker makes the solution more interactive and accessible, providing immediate feedback to users. Additionally, since the system is designed to be run on a laptop, it eliminates the need for expensive external processors, ensuring cost-effectiveness and ease of deployment.

The project ultimately aims to create an intuitive, user-friendly interface that bridges the communication gap between sign language users and non-users, promoting inclusivity in social, professional, and educational settings.

Figure 3.

.

1.4 Objectives

1.4.1. To Develop a Robust Real-Time Gesture Recognition Core

This will be achieved by building and training a Convolutional Neural Network (CNN) model using frameworks like TensorFlow/. We will utilize computer vision libraries like OpenCV and MediaPipe for real-time hand landmark detection and feature extraction from the camera feed, ensuring fast and accurate identification of ASL alphabets.

1.4.2. To Create an Integrated Multi-Modal Output System

This will be achieved by developing a software module that seamlessly converts the recognized gestures into displayed text on a screen. Simultaneously, this text will be fed into a Text-to-Speech (TTS) to generate synchronized audio output, providing both visual and auditory feedback for effective communication.

1.4.3. To Optimize for Accessibility and Wide Usability

This will be achieved by developing the system to run on common hardware (standard laptops/webcams) to keep it cost-effective. We will also aim for compatibility with various platforms (Windows, Linux) and explore a simple user interface to make the technology accessible to a broad audience, reducing dependency on expensive interpreters.

1.4.4. To Enable Seamless Human-Computer Interaction

This will be achieved by implementing a real-time processing pipeline with minimal latency. The system will include error-handling mechanisms (e.g., prompting for a retry on low confidence) and will be rigorously tested in different lighting and background conditions to ensure a smooth and natural user experience for the hearing and speech-impaired community.

1.4.5. To Ensure Offline Functionality and Resource Efficiency for Wider Accessibility

This will be achieved by optimizing the machine learning model for edge deployment, using techniques like model quantization and pruning to reduce its size and computational demands. All processing, from gesture recognition to text-to-speech conversion, will be designed to run locally on the device, eliminating the dependency on a constant internet connection and making the system usable in diverse, resource-constrained environments.

1.5 Challenges

1.5.1. Dataset Collection & Quality Issues

- Capturing gesture images consistently was difficult due to **lighting variation, background noise, camera angle differences, and hand distance**.
- Many images captured during webcam recording had **low resolution**, shadows, or incorrect framing, which affected model performance.
- Dynamic gestures (like "Thank You") are not static, and collecting such gestures required continuous frames, making dataset preparation harder.

1.5.2. Preprocessing Difficulties

- Ensuring all images were resized, normalized, padded, and converted to grayscale **without losing gesture detail** was a major challenge.
- Attempts to preprocess frames sometimes resulted in **distorted images**, uneven aspect ratios, or cropped-out fingers.
- Handling .npy, .csv, and folder-based datasets together caused confusion while merging or fine-tuning.

1.5.3. Model Accuracy & Overfitting

- Initial CNN training showed **high training accuracy but very low validation accuracy** (classic overfitting).
- The model performed well on training data but failed on real-time webcam input due to:

    - Differences in lighting
    - Non-matching image resolution
    - Hand rotation and positioning variations
- Fine-tuning was required multiple times to stabilize accuracy.

1.5.4. Environment & Lighting Conditions

- Model performance drops significantly when:
    - The room lighting changes
    - The background is cluttered
    - The camera captures skin tones differently
- Ensuring consistent detection across lighting environments is still a challenge.

15.5. Real-Time Prediction Instability

- During webcam testing, predictions were unstable because:
    - Hand detection fluctuated on fast movements
    - Cropped images sometimes contained background instead of a clear hand
    - Confidence occasionally dropped below threshold, showing "Unknown"
- Mediapipe landmark detection worked well, but bounding box extraction sometimes failed at extreme angles.

1.5.6. Hardware Limitations

- Running TensorFlow, MediaPipe, OpenCV, and the trained model in real-time required significant processing power.
- Since you are using a **low-end laptop**, frame rate dropped occasionally and delayed predictions.

1.5.7.Model-Generalization Problems

- The model sometimes misclassified similar gestures like:
    - M vs N
    - V vs U
    - D vs M

Architecture and Methodology

Chapter 2

2.1 Mediapipe Hand Landmark Image Model: Architecture

- **CNN model:** This deep learning approach for sign language gesture recognition involves the use of Convolutional Neural Networks (CNNs), TensorFlow, and OpenCV.

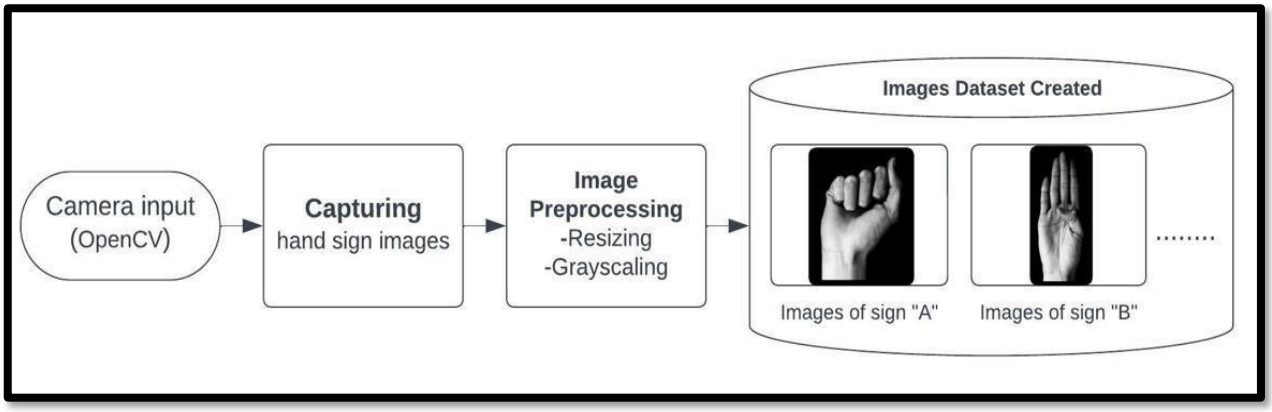


Figure-4

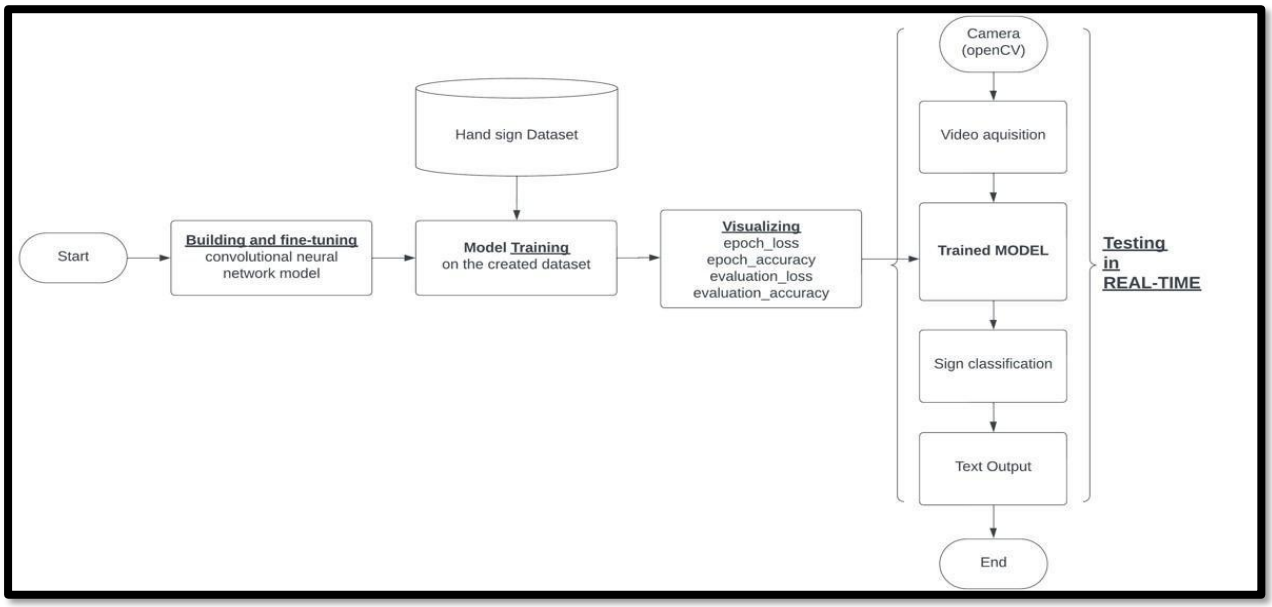


Figure-5

2.1.1. Camera Input (OpenCV)

- **Process:** The system utilizes a standard webcam as the input device. The video feed is captured and managed using the **OpenCV** library, which provides robust tools for real-time image acquisition.
- **Purpose:** This module serves as the source of all raw visual data, capturing frames that contain hand signs against various backgrounds.

2.1.2 Capturing Hand Sign Images

- **Process:** From the continuous video stream, individual frames are extracted and saved as discrete image files. This is done systematically for each letter of the American Sign Language (ASL) alphabet (e.g., A, B, C, etc.). Multiple images are captured for each sign to ensure diversity in the dataset.
- **Purpose:** To build a comprehensive collection of raw images that will form the basis of our training dataset. Capturing multiple samples per sign helps the model learn to generalize across different lighting conditions and slight variations in hand posture.

2.1.3. Image Preprocessing

- **Process:** Each captured image undergoes two key preprocessing steps to standardize the input for the model:
    - **Resizing:** All images are resized to a uniform dimension (e.g., 224x224 pixels). This is essential because neural networks require input images to be of a fixed size.
    - **Grayscaling:** The images are converted from color (RGB) to grayscale. This reduces the computational complexity and memory requirements by reducing the number of color channels from three to one, while still retaining the essential structural information of the hand sign.
- **Purpose:** Preprocessing ensures data consistency, accelerates the training process, and can improve model performance by eliminating irrelevant color variations.

2.1.4. Images Dataset Created

- **Process:** The preprocessed images are organized into a structured dataset. The dataset is categorized into separate folders or labeled appropriately, with distinct classes for each sign (e.g., a folder for "A," a folder for "B," and so on).
- **Output:** The final output of this architecture is a curated and preprocessed image dataset, ready to be used for training the gesture classification model. This structured dataset is the cornerstone for the subsequent machine learning phase

2.2 Mediapipe Hand Landmark Extraction Model: Architecture

This architecture outlines the complete workflow for building and deploying the AI-Driven Hand Gesture Recognition System, from dataset creation to real-time inference. The process is divided into two main phases: the **Model Development Phase** (offline) and the **Real-Time Classification Phase** (online).

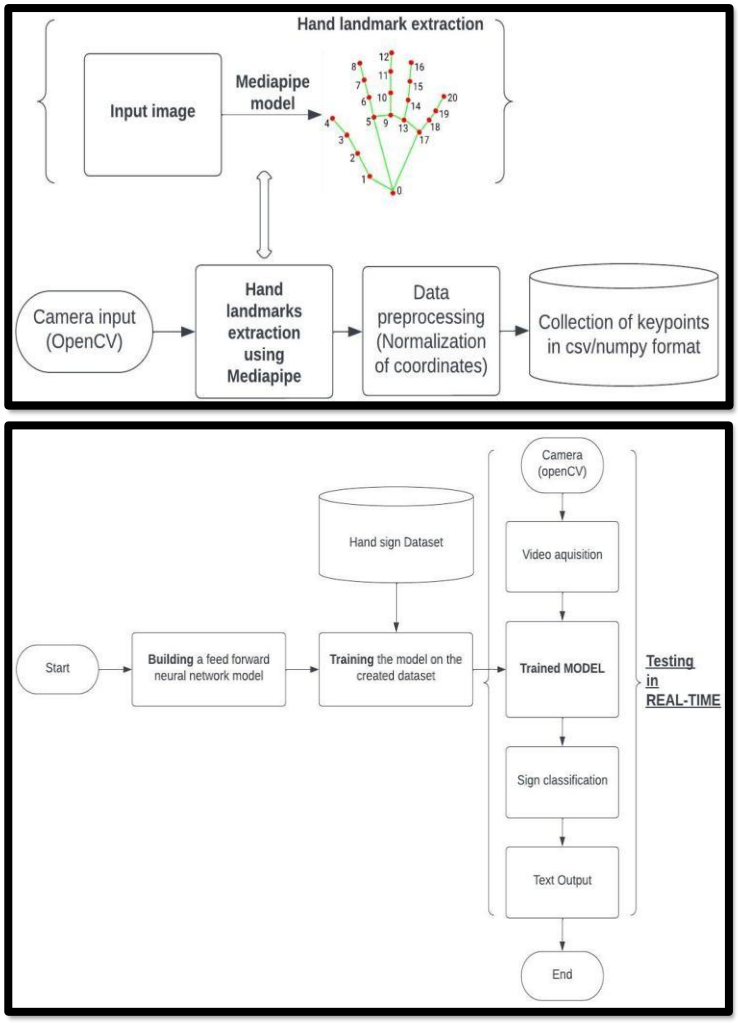


Figure-6

2.2.1. Data Acquisition & Landmark Extraction

- **Camera Input (OpenCV):** A webcam is used to capture a large set of images for each hand sign (A, B, C, etc.) across multiple users and lighting conditions.
- **Hand Landmarks Extraction using MediaPipe:** Each captured image is processed by the **MediaPipe Hands** model, which detects the hand and outputs the precise 2D/3D coordinates of 21 key landmark points representing the hand's skeletal structure.

2.2.2. Dataset Creation

- **Data Preprocessing (Normalization of Coordinates):** The raw landmark coordinates are normalized to be consistent and scale-invariant. This typically involves making the coordinates relative to the wrist point (landmark 0) and scaling them.
- **Collection of Keypoints in CSV/Numpy Format:** The normalized (x, y, z) coordinates for all 21 landmarks (resulting in a 63-value feature vector per image) are

compiled into a structured dataset. Each row in the dataset contains this feature vector and its corresponding label (e.g., 'A', 'B').

2.2.3. Model Building & Training

- **Building a Feed-Forward Neural Network Model:** A Fully Connected (Dense) Neural Network is constructed. This type of model is ideal for learning patterns from structured, tabular data like our landmark coordinates.
- **Training the Model on the Created Dataset:** The landmark dataset is split into training and validation sets. The Feed-Forward Neural Network is trained on this data, learning the complex relationship between the spatial arrangement of the 21 landmarks and the specific alphabet they represent.

2.2.4. REAL-TIME CLASSIFICATION

- **Live Video Acquisition**
  - **Camera (OpenCV):** The system's webcam begins capturing a live video feed.
- **Frame Processing & Feature Extraction**
  - **Video Acquisition:** Individual frames are extracted from the video stream.
  - **Hand Landmarks Extraction using MediaPipe:** Each frame is passed through the same MediaPipe pipeline used in the training phase to extract the 21 normalized landmark coordinates in real-time.
- **Gesture Recognition**
  - **Trained MODEL:** The live stream of landmark data is fed into the pre-trained Feed-Forward Neural Network.
  - **Sign Classification:** The model predicts the corresponding alphabet letter for the hand gesture in the current frame.
- **Output Generation**
  - **Text Output:** The classified letter is displayed as text on the screen. For spelling words, the system can concatenate consecutive letters.
  - **End:** The process runs in a continuous loop, providing real-time translation until the user terminates the application.

The project explored two distinct architectural paradigms for hand gesture recognition: a Pixel-Based Approach and a Landmark-Based Approach. The initial, pixel-based method relied on processing raw images directly. In this workflow, a camera captured hand sign images, which were then preprocessed through resizing and grayscaling to create a standardized image dataset. A Convolutional Neural Network (CNN) was then trained to classify gestures by learning patterns directly from these pixel arrays. While straightforward, this approach required the model to learn both the task of identifying the hand's structure and classifying the gesture, making it computationally intensive and sensitive to variations in the background and lighting.

In contrast, the refined Landmark-Based Approach leverages the power of the MediaPipe framework to abstract the problem. Instead of using raw pixels, this method processes the

camera input through MediaPipe to extract 21 key anatomical landmark points on the hand. These normalized coordinates, representing the hand's skeletal geometry, are used to create a compact numerical dataset. This dataset trains a Feed-Forward Neural Network, which learns to classify gestures based solely on the hand's pose. This landmark-based architecture is fundamentally more efficient and robust. By decoupling the hand tracking from the classification, the model becomes invariant to background noise and lighting, focuses on semantically rich features, and achieves high accuracy with lower computational cost, making it far superior for real-time deployment on common hardware.

2.3 Methodology for landmark and image approach:

2.3.1. Creating the Data set:

There is a rectangular box which is my Region of Interest (ROI) and we need to bring our hand in that ROI to capture the images. For all the **26** American Sign Language I've captured more than **600** images for Training and Testing purposes and stored them in Test and Train folders respectively with a split of **80:20**

2.3.2. Data preprocessing:

Generally, data preprocessing involves resizing images and applying data augmentation on images, but in this particular case, i.e., hand sign recognition using OpenCV, generating additional training samples by applying transformations like rotation, translation, flipping, and zooming are not possible because even a single hand sign formation can resemble in 2 different alphabets or signs. Below I've given an example of the letters D, Z and G from the American Sign Language, Hence, only resizing and gray scaling can be applied to the images.

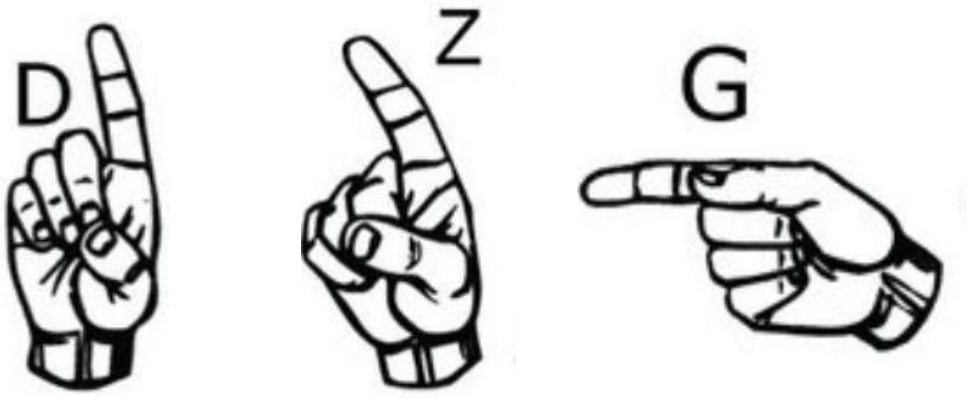


Figure-7

2.3.3. Model creation:

*** Convolutional Layers (64 filters,128 filters, 256 filters and so on):** Imagine our input image showing a hand sign, like forming the letter "A". This layer acts like a set of filters, each one looking for simple patterns in the image, such as edges or curves on our hand. Think of this as the first layer of our "feature extractor". It's like a filter that looks for simple patterns, like edges or curves, in the input image. We can have 64 of these filters, each one looking for different patterns.

Now we can dig deeper to find more detailed features. These filters are more sophisticated and search for combinations of edges and shapes in our hand gesture, helping us recognize finer details, basically helps to dig deeper and find more complex patterns. For this purpose, we can include a 2nd convolutional layer.

We can also add another convolutional layer. This layer dives even deeper, identifying specific features of our hand gesture that are crucial for distinguishing different signs. It's like looking for unique characteristics in how our fingers are positioned or how our hand is shaped. And like this we can keep adding many convolutional layers to extract more and more features from our model.

*MaxPooling Layers:

After the first layer, we want to focus on the most important features of our hand gesture. MaxPooling helps us zoom out and keep only the key features, like the general shape and position of our hand. MaxPooling takes the output from the convolutional layer and reduces its size by only keeping the most important information. After the second layer, we can continue simplifying the information to highlight the most relevant aspects of our hand gesture, like its overall structure and orientation using more max pooling layers. And like this we can keep adding many MaxPooling layers which will make it easier for our model to understand the overall shape and form of our hand gesture.

*Flatten Layer:

When we've gathered a lot of information about the features in our hand gesture, the Flatten layer arranges all this data into a single list, preparing it for further processing so we can feed it into the next part of our model.

*Dense Layer (n neurons):

This part of our model will act like the brain, analyzing the relationships between the features we've collected. The neurons in this layer work together to understand the complex patterns in our hand gesture, like how the position of our fingers relates to different signs (sign language gestures.). Also, it will be the “outer layer” of our CNN model (with x neurons, where x is the number of different signs that are to be recognized). Each neuron gives us a probability score for its corresponding gesture, telling us how likely it is that the input image represents that gesture, hence, we get a probability distribution over all possible gestures.

*Dropout Layers:

To prevent our model from becoming too fixated on any one aspect of our hand gesture, we can randomly "turn off" some neurons during training. This will help our model generalize better and recognize a wider range of gestures. Hence, this will help prevent overfitting and will make our model more robust.

*Input Layer:

The input layer receives image frames containing hand gestures in real-time from a camera or video feed. In our case it will be a convolutional layer.

*Output Layer:

The output layer provides the final classification, indicating the recognized sign language gesture. In our case the output layer will be a DENSE layer with x neurons, representing x-1 alphabets and 1 output for a blank screen.

Different activation functions, loss functions and optimizers will be used while trying to fine-tune the model. Various functions need to be tried to achieve the best results and the activation functions, loss functions and optimizers which will be giving the best results will be used in the final model.

2.3.4. Training:

The created model will be trained on the created dataset

2.3.5. Real-time Testing:

Finally, the trained model is tested in real-time to predict the sign language using OpenCV.

2.4 Methodology for Mediapipe Hand Landmark Extraction Model:

2.4.1. Hand Landmark Extraction:

Utilize the hand landmark model provided by Mediapipe to extract key landmarks on the hand, including fingertips and joints.This approach focuses on using the Mediapipe library to extract hand landmarks, allowing for the recognition of custom hand gestures with relatively small datasets.MediaPipe is a customized Machine Learning (ML) framework designed by Google. It comes with many pre-trained ML solutions like Face detection, Pose estimation, Hand Recognition and many more.

For this I'm using MediaPipe Hands (subsidiary of MediaPipe) which is a high-fidelity hand and finger tracking solution. It maps 21 hand landmarks on our palm and 5 fingers with the help of ML. With this hand tracking and detection becomes more accurate and it can even track

multiple hands simultaneously. It uses a combination of two different models which are Palm Detection model and Hand Landmark model. With the help of the above approach, Data set will be created and then preprocessed accordingly.

2.4.2. Data Preprocessing:

Preprocessing the extracted landmarks by making each vector relative to the base vector of the hand and then normalizing (getting all the values in a fixed range of (-1, +1)) and transforming them into a suitable format for training.

2.4.3. Dataset creation:

Saving the preprocessed data in some sort of storage structure like CSV file or NumPy array.After that, I'll create a model for training. As the Data set consists of only text data therefore the layers used in this model are different from the first method. We can use a combination of input, dropout and dense layers. Along with these layers some activation functions like ReLu and Softmax can be used.

2.4.4. Model creation:

A feed-forward neural network will be created with some layers like DENSE and DROPOUT layers.For model creation Keras library is used. As the Data set consists of only text data therefore the layers used in this model are different from the first method. We have used a combination of input, DROPOUT and DENSE layers to create a feed-forward neural network. Along with these layers some activation functions like ReLu and Softmax are used.

2.4.5. Training:

The created model will be trained on the created dataset with vector coordinates of the extracted hand landmarks.

2.4.6. Real-time Testing:

Finally, the trained model is tested in real-time to predict the sign language gestures from the continuously extracted hand landmarks using OpenCV.

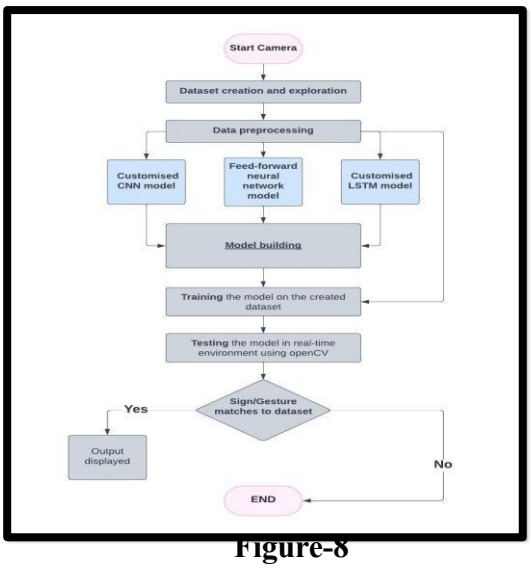


Figure-8

2.5. Flowchart

This flowchart illustrates the end-to-end process for developing and deploying the AI-Driven Hand Gesture Recognition System, encapsulating both the initial research phase and the final real-world testing.

2.5.1. Start Camera

- **Process:** The workflow is initiated by launching the camera module, which uses OpenCV to access the webcam feed.
- **Purpose:** This is the foundational step for both creating the dataset and, later, for real-time testing.

2.5.2. Dataset Creation and Exploration

- **Process:** With the camera active, a comprehensive dataset is built by capturing images or video streams of all the intended hand signs (e.g., the ASL alphabet). This raw data is then explored and organized into labeled classes.
- **Purpose:** To gather the essential raw material needed to train and evaluate different machine learning models.

2.5.3. Data Preprocessing

- **Process:** The collected data undergoes crucial preparation steps. For image-based models, this includes resizing, normalization, and augmentation. For landmark-based models, this involves extracting and normalizing the 21 hand coordinates using MediaPipe.
- **Purpose:** To clean and standardize the data, making it suitable for training and improving the model's ability to generalize.

2.5.4. Model Building (The Core Research Phase)

- This box represents the experimental heart of the project, where three distinct model architectures were investigated:
    - **Customised CNN Model:** Designed to learn spatial features directly from preprocessed images, ideal for recognizing the static shape of each gesture.
    - **Feed-Forward Neural Network Model:** A simpler network intended to be trained on the structured, tabular data of hand landmarks (from MediaPipe), learning the geometric patterns of the hand.
    - **Customised LSTM Model:** Explored for recognizing sequences of gestures or dynamic signs, focusing on learning temporal patterns over time.
- **Purpose:** To compare different AI approaches and select the most accurate and efficient architecture for the task.

2.5.5. Training the Model on the Created Dataset

- **Process:** The chosen model (or models) from the previous step is trained on the preprocessed dataset. The model's parameters are iteratively adjusted to minimize the difference between its predictions and the true labels.
- **Purpose:** To teach the model the underlying patterns that map hand signs (or their landmarks) to their corresponding meanings.

2.5.6. Testing the Model in Real-Time Environment using OpenCV

- **Process:** The trained model is deployed into a live environment. The camera feed is activated again, but this time, each frame is processed by the model for immediate prediction.
- **Purpose:** To validate the model's performance outside the controlled training setting and assess its practicality.

2.5.7. Decision: Sign/Gesture Matches to Dataset?

- This is the critical validation step in the real-time loop.
    - **Yes (Matches):** If the model recognizes a gesture with high confidence, the process proceeds to generate the output.
    - **No (Does Not Match):** If the gesture is unrecognizable or the prediction confidence is too low, the system ignores it and continues analyzing subsequent frames without providing an output. This handles unknown signs or poor camera input gracefully.

2.5.8. Output Displayed

- **Process:** For a recognized gesture, the result is communicated to the user. This is typically the display of the corresponding text (e.g., the letter "A") on the screen and/or the conversion of that text to speech.
- **Purpose:** To fulfill the core objective of the system: enabling clear and seamless communication.

2.5.9. END

- **Process:** The workflow is terminated, closing the camera and the application.

Module Description

Chapter 3

3.1. Image Module:

3.1.1. Dataset creation and data exploration module:

The data creation module of the project is responsible for capturing and organizing hand sign images to build a robust dataset for training the deep learning model. The process begins with the execution of the 'datacollection.py' script. This script utilizes the OpenCV library to access the user's webcam for real-time image capture. When the script is executed, a window titled "data" opens, displaying the live feed from the webcam. Within this window, a rectangular box appears, delineating a specific region known as the Region of Interest (ROI). This ROI box has predefined dimensions and is positioned at the top left corner of the window with an offset of x pixels from the top. This ensures that only the relevant portion of the webcam feed, containing the hand sign gesture, is captured. One is in which there is a rectangular box which is our Region of Interest (ROI) and we need to bring our hand in that ROI to capture the images.

Upon execution, the script creates a directory named 'SignImage48x48' to store the collected images. Within this directory, separate folders are created for each letter of the alphabet as well as a 'blank' folder. As the user performs hand signs within the ROI, the script captures images of the gestures and saves them in the respective folders based on the detected key press. For instance, pressing the 'a' key saves the image in the 'A' folder, and so on. Each image is converted to grayscale and resized to a fixed dimension of **48x48** pixels to maintain uniformity in the dataset. For all the **26** American Sign Language I've captured more than **600** images for Training and Testing purpose. I've captured 300+ images of each alphabet for Training purposes so that they can have different conditions (like lightning conditions). We can even bring our hand near ROI, sometimes take it far or twist our hand so that our model can detect all images easily.

Figure 9

Insert the folder of each alphabet, I've captured more than 600 images for training and testing

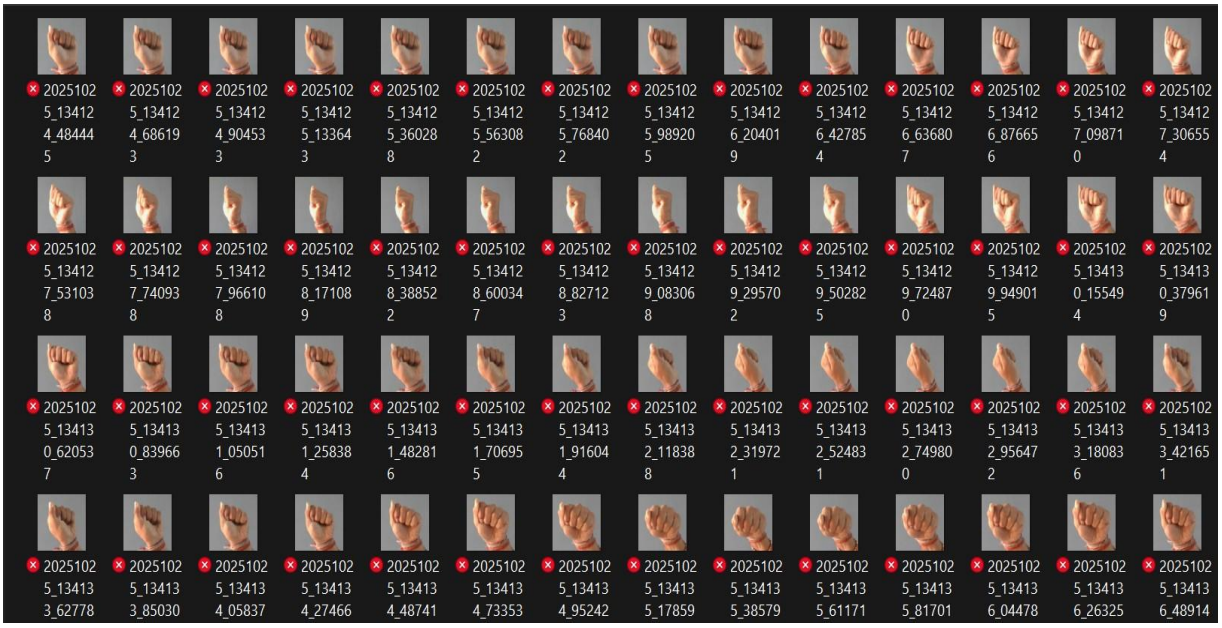


Figure-10

3.1.2 Data Retrieving and Splitting:

The data retrieval and splitting module plays a crucial role in preparing the collected dataset for training and evaluation. Following the data creation phase, the 'split.py' script is executed to organize the dataset into training and validation sets. This script employs the 'splitfolders' library to split the 'SignImage64x64' dataset into two distinct directories: 'train' and 'val'. The splitting ratio is defined as 80% for training and 20% for validation, ensuring a suitable balance between training and evaluation data. By executing this module, the dataset is effectively partitioned into separate subsets, each designated for specific purposes during the model development process. The training set is utilized to train the deep learning model, while the validation set serves to assess the model's performance and generalization capabilities.

3.1.3 Model building:

The classification model, built using a Convolutional Neural Network (CNN), is at the heart of the hand sign recognition system. It consists of multiple convolutional layers followed by max-pooling layers to extract features from input images. Dropout layers are integrated to prevent overfitting. After feature extraction, the flattened output is passed through dense layers to learn intricate relationships. The model is trained to classify hand signs into one of the 26 classes corresponding to the English alphabet using. During training, the model adjusts its parameters using the Adam optimizer to minimize categorical cross-entropy loss. Performance is evaluated using metrics like accuracy, precision, recall, and F1-score on a validation dataset.

3.2 Training Code Description:

```
# First conv block
layers.Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
layers.BatchNormalization(),
layers.MaxPooling2D((2, 2)),
layers.Dropout(0.25),

# Second conv block
layers.Conv2D(64, (3, 3), activation='relu'),
layers.BatchNormalization(),
layers.MaxPooling2D((2, 2)),
layers.Dropout(0.25),

# Third conv block
layers.Conv2D(128, (3, 3), activation='relu'),
layers.BatchNormalization(),
layers.MaxPooling2D((2, 2)),
layers.Dropout(0.25),

# Fourth conv block
layers.Conv2D(256, (3, 3), activation='relu'),
layers.BatchNormalization(),
layers.MaxPooling2D((2, 2)),
layers.Dropout(0.25),

# Classifier
layers.Flatten(),
layers.Dense(512, activation='relu'),
layers.BatchNormalization(),
layers.Dropout(0.5),
layers.Dense(256, activation='relu'),
layers.Dropout(0.5),
layers.Dense(num_classes, activation='softmax')
```

*First Convolutional Block

***layers.Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),**

- **Conv2D**: 2D convolutional layer for processing images
- **32**: Number of filters/feature detectors. This layer will learn 32 different patterns.
- **(3, 3)**: Filter size - each filter is a 3x3 pixel window that slides across the image.
- **activation='relu'**: ReLU (Rectified Linear Unit) activation function. It outputs the input directly if positive, otherwise zero. Formula: f(x) = max(0, x)
- **input_shape=input_shape**: Defines the expected input dimensions (e.g., (128, 128, 3) for 128x128 RGB images).
- **What happens**: 32 filters convolve with input image, producing 32 feature maps showing where specific patterns appear.

*layers.BatchNormalization(),

- Normalizes the outputs from the previous layer across the current batch of data.
- Formula: **(x - mean) / sqrt(variance + epsilon) * gamma + beta**
- Why: Stabilizes training, allows higher learning rates, reduces sensitivity to initialization.
- Effect: Forces activations to have zero mean and unit variance during training.

*layers.MaxPooling2D((2, 2)),

- **MaxPooling**: Downsampling operation that reduces spatial dimensions.
- **(2, 2)**: Pooling window size - takes 2x2 pixel blocks.

- **Operation**: For each 2x2 block, outputs only the maximum value.
- **Purpose**: Reduces computational load, provides translation invariance, prevents overfitting.
- **Result**: Image width and height are halved (50% reduction).

*layers.Dropout(0.25),

- Dropout: Regularization technique that randomly "drops" (sets to zero) neurons during training.
- 0.25: Dropout rate - 25% of the units are randomly ignored during each forward/backward pass.
- Purpose: Prevents neurons from co-adapting too much, forces network to learn redundant representations.
- Effect: Reduces overfitting by preventing reliance on any single neuron.

* Second conv block

layers.Conv2D(64, (3, 3), activation='relu'),

- 64 filters: Double the previous layer's capacity to learn more complex features.
- No input_shape: Automatically takes output from previous layer.
- What it learns: Combines simple edges from first layer to detect more complex shapes like curves, corners, textures.
-

layers.BatchNormalization(),

layers.MaxPooling2D((2, 2)),

layers.Dropout(0.25)

- Same operations as first block, maintaining consistent architecture.
- Now processing more complex features but with same regularization strategy.

* Third conv block

layers.Conv2D(128, (3, 3), activation='relu'),

- **128 filters**: Again doubling capacity to learn even more sophisticated patterns.
- **What it learns**: Object parts - eyes, wheels, windows, etc. by combining shapes from previous layer.

layers.BatchNormalization(),

layers.MaxPooling2D((2, 2)),

layers.Dropout(0.25),

- Consistent pattern continues - normalize, downsample, regularize.

*Fourth Convolutional Block

layers.Conv2D(256, (3, 3), activation='relu'

- **256 filters**: Highest capacity layer for the most complex feature detection.
- **What it learns**: Sophisticated patterns, object combinations, high-level semantic features.

layers.BatchNormalization(),

layers.MaxPooling2D((2, 2)),

layers.Dropout(0.25),

- Final spatial reduction and normalization before classification.

Classifier Head

layers.Flatten(), # Converts 2D features to 1D vector

layers.Dense(512, activation='relu'), # Learns combinations of features

layers.BatchNormalization(), # Stabilizes training

layers.Dropout(0.5), # Prevents overfitting (50% dropout)

layers.Dense(256, activation='relu'), # Further refines features

layers.Dropout(0.5), # More regularization

layers.Dense(num_classes, activation='softmax') # Final probability output

*Output Layer

The output layer in my hand sign recognition model is a **Dense layer with 27 neurons** using **softmax activation**. This configuration is specifically designed for the multiclass classification task of recognizing American Sign Language alphabet gestures along with blank detection:

- **26 neurons** for alphabet signs (A through Z)
- **1 neuron** for "blank/no sign" detection

- **Softmax activation** function that converts raw output scores into probability distributions across all 27 classes

The softmax activation ensures that all output values sum to 1.0, representing the probability of each class, making it suitable for multiclass classification. This output layer works in conjunction with:

- **Categorical Crossentropy** as the loss function, which measures the difference between predicted probabilities and true labels
- **Adam optimizer** for efficient gradient-based weight updates during training

This specific output layer design enables the model to not only recognize individual alphabet signs but also detect when no meaningful sign is present, making it robust for real-world applications where the camera may capture empty frames or non-sign gestures.

3.3. Mediapipe Landmark Extraction:

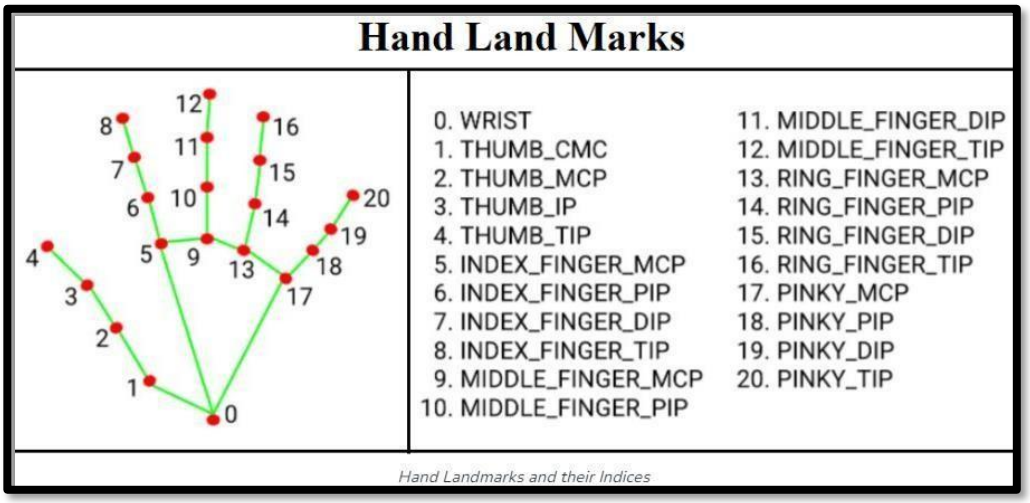


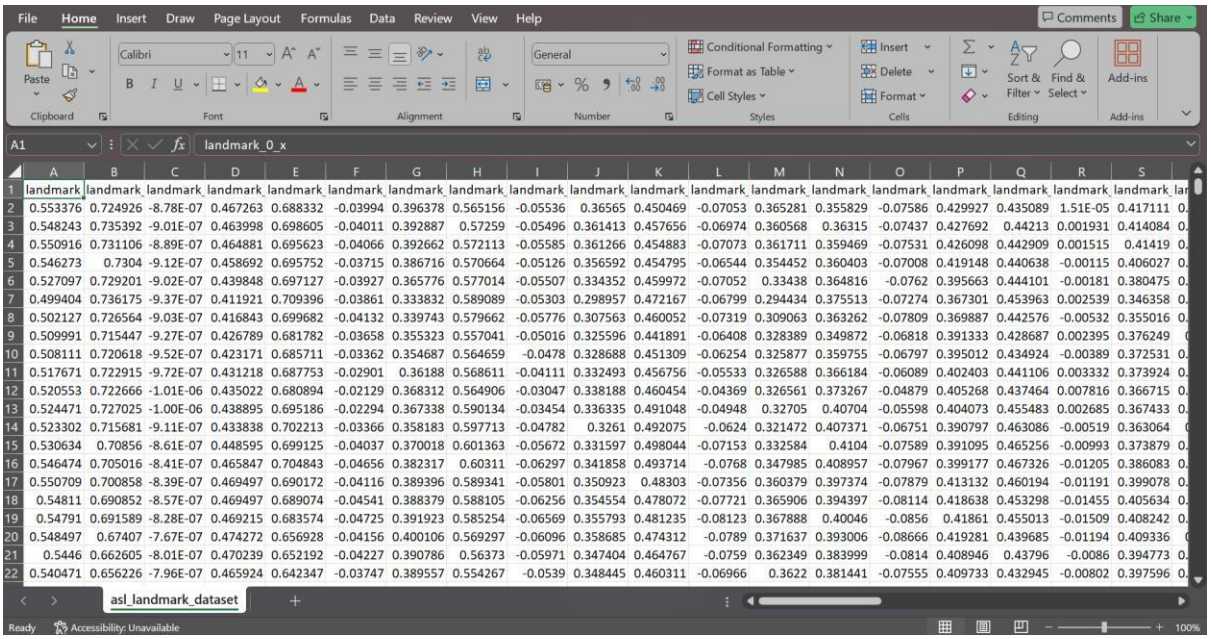
Figure-11

3.3.1 Dataset Creation

Here, instead of collecting images of the hand signs we have recorded the coordinates of the 21 3D landmarks that are mapped on the entire hand. More than 100 different coordinates for a single hand sign have been recorded.

In the Data set the first column is dedicated to the label of the sign and the rest of the columns are the X and Y coordinates of the 21 hand landmarks spread across the entire palm and 5

fingers. All values recorded in the Data set are made relative to hand landmark “0” therefore the second and third column of the Data set have 0 value. The value of each coordinate has been normalized and is between the range of -1 to 1.



landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark	landmark
0.553376	0.724926	-8.78E-07	0.467263	0.688332	-0.03994	0.396378	0.565156	-0.05536	0.36565	0.450469	-0.07053	0.365281	0.355829	-0.07586	0.429927	0.435089	1.51E-05	0.417111
0.548243	0.735392	-9.01E-07	0.463998	0.698605	-0.04011	0.392887	0.57259	-0.05496	0.361413	0.457656	-0.06974	0.360568	0.36315	-0.07437	0.427692	0.44213	0.001931	0.414084
0.550916	0.731106	-8.89E-07	0.464881	0.695623	-0.04066	0.392662	0.572113	-0.05585	0.361266	0.454883	-0.07073	0.361711	0.359469	-0.07531	0.426098	0.442909	0.001515	0.41419
0.546273	0.7304	-9.12E-07	0.458692	0.695752	-0.03715	0.386716	0.570664	-0.05126	0.356592	0.454795	-0.06544	0.354452	0.360403	-0.07008	0.419148	0.440638	-0.00115	0.406027
0.527097	0.729201	-9.02E-07	0.439848	0.697127	-0.03927	0.365776	0.577014	-0.05507	0.334352	0.459972	-0.07052	0.33438	0.364816	-0.0762	0.395663	0.444101	-0.00181	0.380475
0.499404	0.736175	-9.37E-07	0.411921	0.709396	-0.03861	0.333832	0.589089	-0.05303	0.298957	0.472167	-0.06799	0.294434	0.375513	-0.07274	0.367301	0.453963	0.002539	0.346358
0.502127	0.726564	-9.03E-07	0.416843	0.699682	-0.04132	0.339743	0.579662	-0.05776	0.307563	0.460052	-0.07319	0.309063	0.363262	-0.07809	0.369887	0.442576	-0.00532	0.355016
0.509991	0.715447	-9.27E-07	0.426789	0.681782	-0.03658	0.355323	0.557041	-0.05016	0.325596	0.441891	-0.06408	0.328389	0.349872	-0.06818	0.391333	0.428687	0.002395	0.376249
0.508111	0.720618	-9.52E-07	0.423171	0.685711	-0.03362	0.354687	0.564659	-0.0478	0.328688	0.451309	-0.06254	0.325877	0.359755	-0.06797	0.395012	0.434924	-0.00389	0.372531
0.517671	0.722915	-9.72E-07	0.431218	0.687753	-0.02901	0.36188	0.568611	-0.04111	0.332493	0.456756	-0.05533	0.326588	0.366184	-0.06089	0.402403	0.441106	0.003332	0.373924
0.520553	0.722666	-1.01E-06	0.435022	0.680894	-0.02129	0.368312	0.564906	-0.03047	0.338188	0.460454	-0.04369	0.326561	0.373267	-0.04879	0.405268	0.437464	0.007816	0.366715
0.524471	0.727025	-1.00E-06	0.438895	0.695186	-0.02294	0.367338	0.590134	-0.03454	0.336335	0.491048	-0.04948	0.32705	0.40704	-0.05598	0.404073	0.455483	0.002685	0.367433
0.523302	0.715681	-9.11E-07	0.433838	0.702213	-0.03366	0.358183	0.597713	-0.04782	0.3261	0.492075	-0.0624	0.321472	0.407371	-0.06751	0.390797	0.463086	-0.00519	0.363064
0.530634	0.70856	-8.61E-07	0.448595	0.699125	-0.04037	0.370018	0.601363	-0.05672	0.331597	0.498044	-0.07153	0.332584	0.4104	-0.07589	0.391095	0.465256	-0.00993	0.373879
0.546474	0.705016	-8.41E-07	0.465847	0.704843	-0.04656	0.382317	0.60311	-0.06297	0.341858	0.493714	-0.0768	0.347985	0.408957	-0.07967	0.399177	0.467326	-0.01205	0.386083
0.550709	0.700858	-8.39E-07	0.469497	0.690172	-0.04116	0.389396	0.589341	-0.05801	0.350923	0.48303	-0.07356	0.360379	0.397374	-0.07879	0.413132	0.460194	-0.01191	0.399078
0.54811	0.690852	-8.57E-07	0.469497	0.689074	-0.04541	0.388379	0.588105	-0.06256	0.354554	0.478072	-0.07721	0.365906	0.394397	-0.08114	0.418638	0.453298	-0.01455	0.405634
0.54791	0.691589	-8.28E-07	0.469215	0.683574	-0.04725	0.391923	0.585254	-0.06569	0.355793	0.481235	-0.08123	0.367888	0.40046	-0.0856	0.41861	0.455013	-0.01509	0.408242
0.548497	0.67407	-7.67E-07	0.474272	0.656928	-0.04156	0.400106	0.569297	-0.06096	0.358685	0.474312	-0.0789	0.371637	0.393006	-0.08666	0.419281	0.439685	-0.01194	0.409336
0.5446	0.662605	-8.01E-07	0.470239	0.652192	-0.04227	0.390786	0.56373	-0.05971	0.347404	0.464767	-0.0759	0.362349	0.383999	-0.0814	0.408946	0.43796	-0.0086	0.394773
0.540471	0.656226	-7.96E-07	0.465924	0.642347	-0.03747	0.389557	0.554267	-0.0539	0.348445	0.460311	-0.06966	0.3622	0.381441	-0.07555	0.409733	0.432945	-0.00802	0.397596

Figure-12

The above csv file has the first column which represents the index of the sign, like, 0^{th} sign, 1^{st} sign, 2^{nd} sign and so on. There are 42 other columns which represents the X and Y coordinates of the 21 Hand landmarks corresponding to each sign.

3.3.2.Data Retrieving and Splitting

We have stored the data in a CSV file after preprocessing it and along with it in a separate file the labels of each sign are mentioned. With the help of numpy library the Data set is loaded to train and test the model. While loading the Data set it was split into a ratio of **75 and 25** with the help of Sklearn package library.

Normalization module: Before normalization we first made all the 20 points on the Hand relative to the base point of the palm, i.e, the 0^{th} landmark and then passed the landmark list in the data preprocessing function.

Let's consider an example where we perform the same hand sign gesture in two different positions on the screen. Suppose we have the following two sets of landmark coordinates representing the same sign but at different positions:

landmark_list_1 = [[100, 200], [150, 220], [120, 180]]

Relative Coordinates Calculation:

Set the base coordinates as the first landmark: **base_x = 100**, **base_y = 200**

Calculate relative coordinates for each landmark:

- For the first landmark **[100, 200]**, relative coordinates are **[0, 0]**
- For the second landmark **[150, 220]**, relative coordinates are **[50, 20]**
- For the third landmark [120, 180], relative coordinates are [20, -20]
- **Normalization**:

The maximum absolute value among all coordinates is **220**

Normalize each coordinate by dividing by **220**

- **[0, 0]** becomes **[0, 0]**
- **[50, 20]** becomes **[0.227, 0.091]**
- **[20, -20]** becomes **[0.091, -0.091]**

landmark_list_2 = [[300, 350], [350, 370], [320, 330]] (Sign performed at a different position)

3.4. Training Code Description:

```
# 5. Build the model
model = keras.Sequential([
    # Input layer
    keras.layers.Dense(128, activation='relu', input_shape=(X.shape[1],)),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.3),

    # Hidden layers
    keras.layers.Dense(64, activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.3),

    keras.layers.Dense(32, activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.2),

    # Output layer
    keras.layers.Dense(len(label_encoder.classes_), activation='softmax')
])

# 6. Compile the model
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

Figure-13

*Input Layer

- **keras.layers.Dense(128, activation='relu', input_shape=(X.shape[1],))**
    - **Dense**: Fully connected layer where each neuron connects to every neuron in previous layer
    - **128**: Number of neurons/nodes in this layer - creates 128-dimensional feature space

- activation='relu': ReLU (Rectified Linear Unit) activation function
  - Formula: f(x) = max(0, x)
  - Purpose: Introduces non-linearity, helps network learn complex patterns
- **input_shape=(X.shape[1],):** Defines input dimensions based on preprocessed feature vector
  - For landmark-based approach: 63 features (21 landmarks × 3 coordinates)
  - For flattened image features: depends on image dimensions after flattening

- **keras.layers.BatchNormalization()**
  - Normalizes the outputs from the previous layer across the current batch
  - Formula: (x - mean) / sqrt(variance + epsilon) × gamma + beta
  - **Why**: Stabilizes training, allows higher learning rates, reduces internal covariate shift
- **keras.layers.Dropout(0.3)**
  - **Dropout rate: 0.3** - Randomly disables 30% of neurons during each training iteration
  - **Purpose**: Prevents overfitting by forcing network to learn redundant representations
  - **Effect**: Improves generalization to unseen data

*First Hidden Layer

- **keras.layers.Dense(64, activation='relu')**
  - **64 neurons**: Reduced capacity from input layer to learn hierarchical features
  - **What it learns**: Combines basic features from input layer into more abstract representations
  - **Activation**: ReLU maintains non-linearity for complex pattern learning
- **keras.layers.BatchNormalization()**
  - Normalizes activations from the 64-neuron layer
  - Maintains stable gradient flow during backpropagation
- **keras.layers.Dropout(0.3)**
  - Consistent 30% dropout for regularization
  - Continues to prevent co-adaptation of neurons

*Second Hidden Layer

- **keras.layers.Dense(32, activation='relu')**
    - **32 neurons**: Further compressed representation for higher-level feature learning
    - **What it learns**: Sophisticated combinations of features specific to hand sign patterns
    - **Progressive reduction**: 128 → 64 → 32 creates information bottleneck for essential features
- **keras.layers.BatchNormalization()**
    - Final normalization before output layer
    - Ensures stable training dynamics
- **keras.layers.Dropout(0.2)**
    - **Reduced dropout rate: 0.2** - Less regularization as we approach output
    - **Why**: Preserves more information for final classification decision

***Output Layer**

- keras.layers.Dense(len(label_encoder.classes_), activation='softmax')
    - len(label_encoder.classes_): Dynamic output size based on number of sign classes
        - For ASL alphabet: 26 classes (A-Z)
        - May include additional classes for "blank" or "space"
    - activation='softmax': Converts raw scores into probability distribution
        - Formula: $\sigma(z)_j = e^{z_j} / \Sigma_k e^{z_k}$ for k = 1 to number of classes
        - **Purpose**: Ensures all outputs sum to 1.0, representing class probabilities

***Model Compilation**

- optimizer=keras.optimizers.Adam(learning_rate=0.001)
    - **Adam optimizer**: Adaptive Moment Estimation
    - **Learning rate: 0.001**: Controls step size during gradient descent

    - **Why Adam**: Efficient, requires little memory, well-suited for problems with large data/parameters

- loss='sparse_categorical_crossentropy'
    - **Loss function**: Measures difference between predicted probabilities and true labels

Hardware and Software

Chapter-4

4.1 Hardware

The system was deliberately implemented using standard, cost-effective hardware to validate its accessibility and practicality for end-users. Each component plays a critical role in the real-time gesture recognition pipeline.

- **4.1.1. Laptop (The Central Processing Unit)**
  The laptop serves as the brain of the entire system, housing all the critical components and performing the heavy computational lifting.
  - **CPU (Intel Core i5):** The Central Processing Unit is responsible for executing the main logic of the application. It manages the video capture from the webcam, runs the MediaPipe hand tracking model, executes the custom neural network for classification, and coordinates the text and audio output. Its multi-core capability allows it to handle these parallel tasks efficiently, ensuring the system operates in real-time without significant lag.
  - **GPU (Integrated Graphics):** While not a dedicated graphics card, the integrated GPU still assists the CPU by accelerating some of the mathematical computations required by the MediaPipe framework and the neural network, contributing to the overall fluidity of the application.
- **4.1.2. Webcam (The Visual Input Sensor)**

  The webcam acts as the "eyes" of the system, bridging the physical world of hand gestures to the digital world of data processing.
  - **Function:** It continuously captures live video footage of the user's hand.
  - **Usefulness:** It provides the raw visual data that is the fundamental input for the entire system. Without it, there would be no gesture to recognize. The quality of the webcam (720p HD) ensures that the image is clear enough for MediaPipe to consistently and accurately detect the hand and its key landmarks.

4.1.3.Speakers (The Auditory Output)

The speakers provide the second critical output channel.

- **Function:** They convert the electrical signals from the audio system into audible sound waves.
- **Usefulness:** They play the synthesized speech generated by the Text-to-Speech (TTS) engine. This audio output is what allows individuals who cannot understand sign

language to "hear" what the Deaf or Mute user is signing, thereby bridging the communication gap effectively.

4.2 Software

The software ecosystem was carefully designed using Python and its powerful libraries to create an efficient, modular, and real-time gesture recognition system. Each component in the software stack serves a specific purpose in the processing pipeline.

4.2.1. Python 3.8 (The Programming Foundation)

- **Role:** Serves as the primary programming language and integration framework.
- **Usefulness:** Python acts as the "glue" that connects all different components of our system. Its simple syntax and extensive library support make it ideal for rapid prototyping and development of AI applications. All other software components are integrated and coordinated through Python scripts.

4.2.2. OpenCV (cv2) - The Vision Engine

- **Role:** Handles all image and video processing operations.
- **Usefulness:** OpenCV serves as our "camera interface and display system." It:
  - Manages the webcam connection and captures live video frames
  - Performs essential image processing (color conversion, resizing)
  - Displays the final output video stream with overlaid text and graphics
  - Provides the real-time visual feedback window that users interact with

4.2.3. MediaPipe - The Hand Tracking Specialist

- **Role:** Provides real-time hand landmark detection and tracking.
- **Usefulness:** This is the "feature extraction engine" of our system. MediaPipe:
  - Processes each video frame to detect hand presence and orientation
  - Outputs precise 21-point landmark coordinates for each detected hand
  - Converts complex image data into simple numerical coordinates
  - Makes our system robust to different lighting conditions and backgrounds
  - Provides the standardized input data for our classification model

- ### 4.2.4. TensorFlow & Keras - The Brain Center
  - **Role:** Powers the machine learning and neural network operations.
  - **Usefulness:** These libraries form the "intelligent decision-making core." They:

- Provide the framework for building our Feed-Forward Neural Network architecture
- Handle the model training process with optimization algorithms
- Manage the saved model files and enable quick loading for inference
- Execute the real-time classification of hand landmarks into meaningful gestures
- Output probability distributions that determine the recognized sign

4.2.5. NumPy & Pandas - The Data Handlers

- **Role:** Manage numerical operations and data organization.
- **Usefulness:** These libraries work as our "data processing assistants." They:
  - Handle the array operations for landmark coordinate manipulation
  - Manage the dataset organization during training phase
  - Optimize mathematical computations for better performance
  - Interface between MediaPipe output and TensorFlow input requirements

4.2.6. scikit-learn - The Data Preprocessor

- **Role:** Handles data preparation and encoding tasks.
- **Usefulness:** Serves as our "data quality control system." Specifically:
  - LabelEncoder converts text labels ('A', 'B', 'C') into numerical format that the neural network can understand
  - Provides functions for splitting data into training and testing sets
  - Ensures proper data formatting before model training

4.2.7. pyttsx3 - The Voice Generator

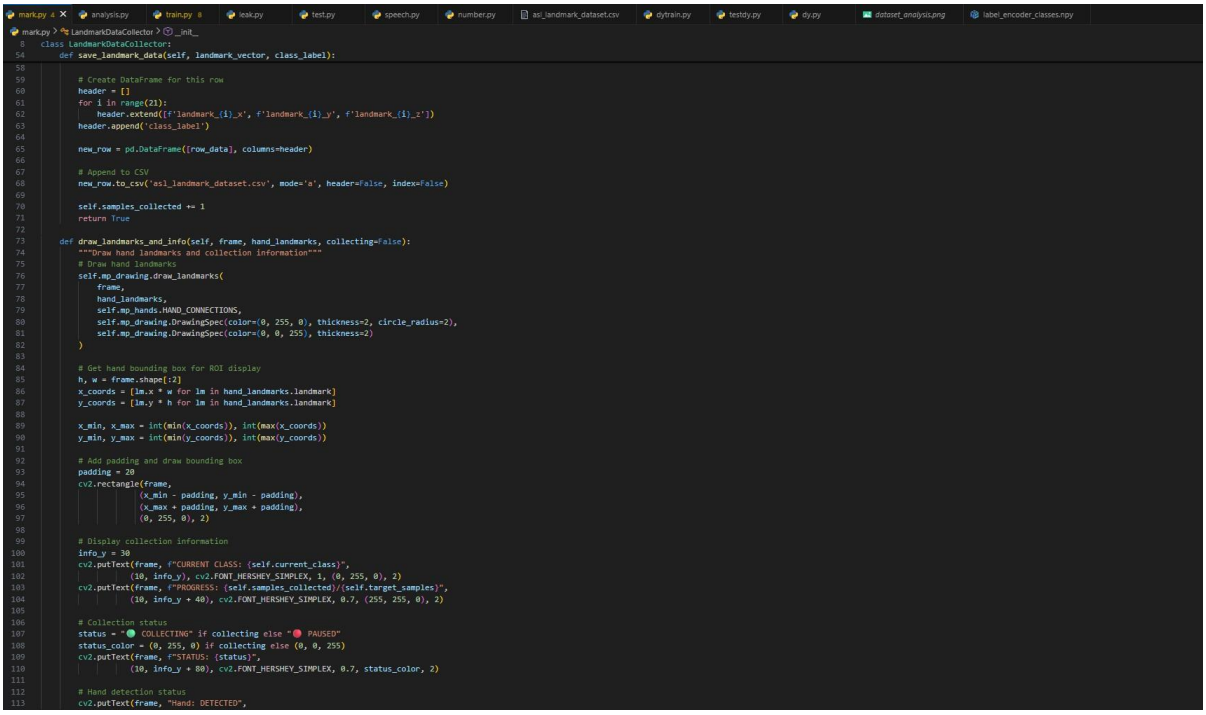
- **Role:** Converts text to speech for audio output.
- **Usefulness:** This is our "communication bridge" to hearing individuals. It:
  - Takes the recognized text output from our model
  - Generates clear, synthesized speech through system speakers
  - Works offline without internet dependency
  - Enables real-time audio feedback synchronized with visual output

Training And Implementation

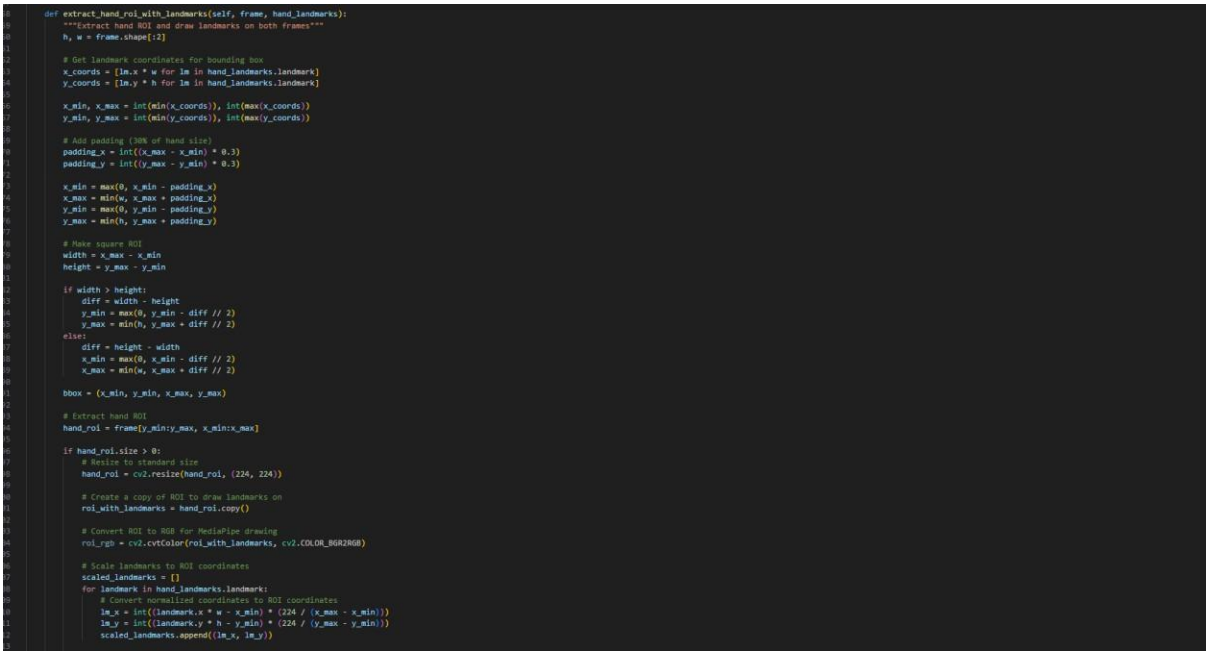
Chapter 5

5.1. Code:

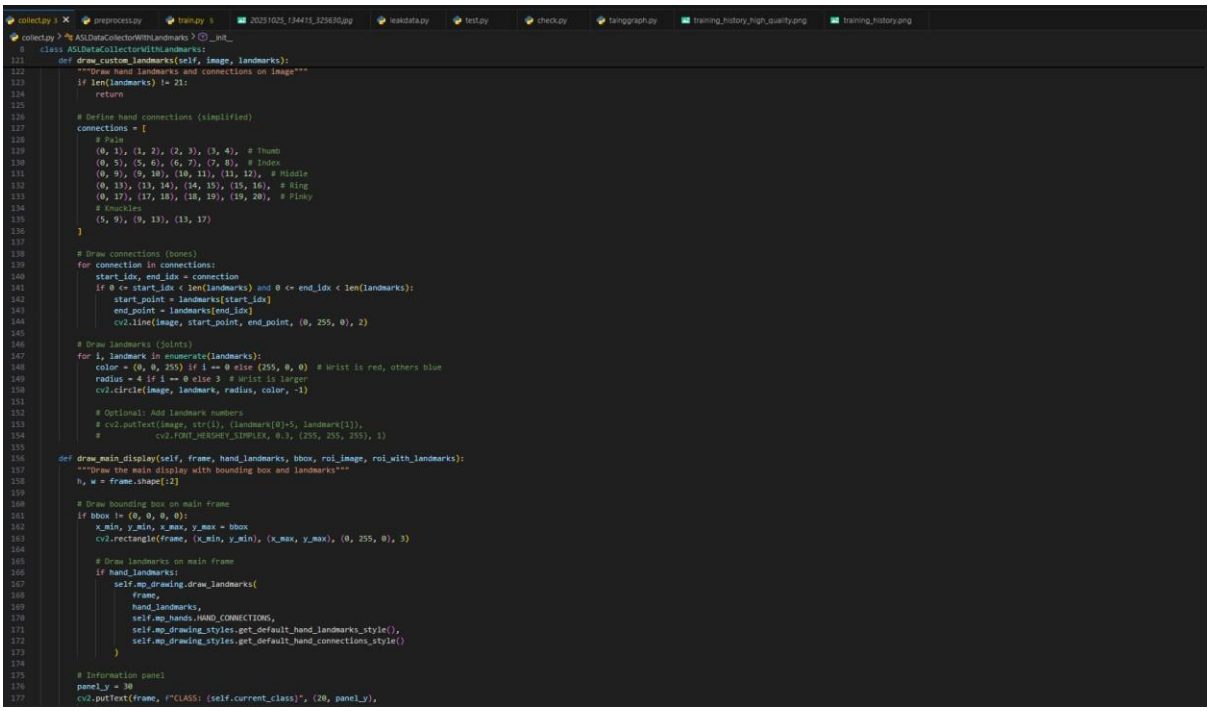
5.1.1.Collect.py



```python
class LandmarkDataCollector:
    def save_landmark_data(self, landmark_vector, class_label):

        # Create DataFrame for this row
        header = []
        for i in range(21):
            header.extend([f'landmark_{i}_x', f'landmark_{i}_y', f'landmark_{i}_z'])
        header.append('class_label')

        new_row = pd.DataFrame([row_data], columns=header)

        # Append to CSV
        new_row.to_csv('asl_landmark_dataset.csv', mode='a', header=False, index=False)

        self.samples_collected += 1
        return True

    def draw_landmarks_and_info(self, frame, hand_landmarks, collecting=False):
        """Draw hand landmarks and collection information"""
        # Draw hand landmarks
        self.mp_drawing.draw_landmarks(
            frame,
            hand_landmarks,
            self.mp_hands.HAND_CONNECTIONS,
            self.mp_drawing.DrawingSpec(color=(0, 255, 0), thickness=2, circle_radius=2),
            self.mp_drawing.DrawingSpec(color=(0, 0, 255), thickness=2)
        )

        # Get hand bounding box for ROI display
        h, w = frame.shape[:2]
        x_coords = [lm.x * w for lm in hand_landmarks.landmark]
        y_coords = [lm.y * h for lm in hand_landmarks.landmark]

        x_min, x_max = int(min(x_coords)), int(max(x_coords))
        y_min, y_max = int(min(y_coords)), int(max(y_coords))

        # Add padding and draw bounding box
        padding = 20
        cv2.rectangle(frame,
                      (x_min - padding, y_min - padding),
                      (x_max + padding, y_max + padding),
                      (0, 255, 0), 2)

        # Display collection information
        info_y = 30
        cv2.putText(frame, f"CURRENT CLASS: {self.current_class}",
                    (10, info_y), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
        cv2.putText(frame, f"PROGRESS: {self.samples_collected}/{self.target_samples}",
                    (10, info_y + 40), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 0), 2)

        # Collection status
        status = "● COLLECTING" if collecting else "● PAUSED"
        status_color = (0, 255, 0) if collecting else (0, 0, 255)
        cv2.putText(frame, f"STATUS: {status}",
                    (10, info_y + 80), cv2.FONT_HERSHEY_SIMPLEX, 0.7, status_color, 2)

        # Hand detection status
        cv2.putText(frame, "Hand: DETECTED",
```



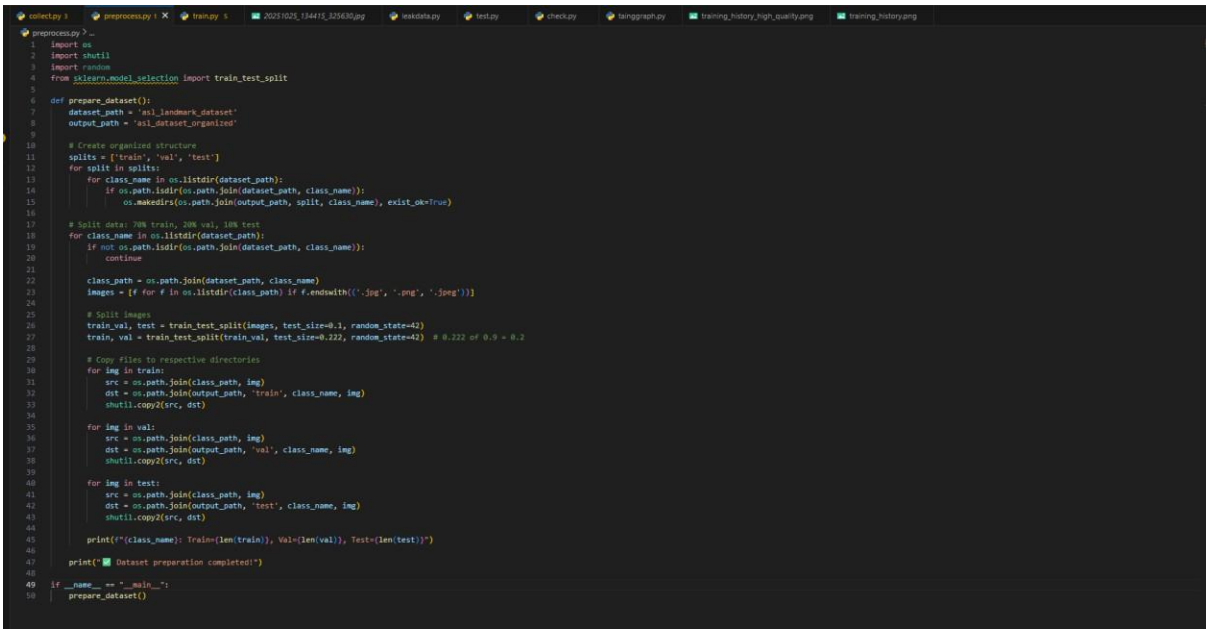
```python
    def extract_hand_roi_with_landmarks(self, frame, hand_landmarks):
        """Extract hand ROI and draw landmarks on both frames"""
        h, w = frame.shape[:2]

        # Get landmark coordinates for bounding box
        x_coords = [lm.x * w for lm in hand_landmarks.landmark]
        y_coords = [lm.y * h for lm in hand_landmarks.landmark]

        x_min, x_max = int(min(x_coords)), int(max(x_coords))
        y_min, y_max = int(min(y_coords)), int(max(y_coords))

        # Add padding (30% of hand size)
        padding_x = int((x_max - x_min) * 0.3)
        padding_y = int((y_max - y_min) * 0.3)

        x_min = max(0, x_min - padding_x)
        x_max = min(w, x_max + padding_x)
        y_min = max(0, y_min - padding_y)
        y_max = min(h, y_max + padding_y)

        # Make square ROI
        width = x_max - x_min
        height = y_max - y_min

        if width > height:
            diff = width - height
            y_min = max(0, y_min - diff // 2)
            y_max = min(h, y_max + diff // 2)
        else:
            diff = height - width
            x_min = max(0, x_min - diff // 2)
            x_max = min(w, x_max + diff // 2)

        bbox = (x_min, y_min, x_max, y_max)

        # Extract hand ROI
        hand_roi = frame[y_min:y_max, x_min:x_max]

        if hand_roi.size > 0:
            # Resize to standard size
            hand_roi = cv2.resize(hand_roi, (224, 224))

            # Create a copy of ROI to draw landmarks on
            roi_with_landmarks = hand_roi.copy()

            # Convert ROI to RGB for MediaPipe drawing
            roi_rgb = cv2.cvtColor(roi_with_landmarks, cv2.COLOR_BGR2RGB)

            # Scale landmarks to ROI coordinates
            scaled_landmarks = []
            for landmark in hand_landmarks.landmark:
                # Convert normalized coordinates to ROI coordinates
                lm_x = int((landmark.x * w - x_min) * (224 / (x_max - x_min)))
                lm_y = int((landmark.y * h - y_min) * (224 / (y_max - y_min)))
                scaled_landmarks.append((lm_x, lm_y))
```



```python
class ASLDataCollectorWithLandmarks:
    def draw_custom_landmarks(self, image, landmarks):
        """Draw hand landmarks and connections on image"""
        if len(landmarks) != 21:
            return

        # Define hand connections (simplified)
        connections = [
            # Palm
            (0, 1), (1, 2), (2, 3), (3, 4),  # Thumb
            (0, 5), (5, 6), (6, 7), (7, 8),  # Index
            (0, 9), (9, 10), (10, 11), (11, 12),  # Middle
            (0, 13), (13, 14), (14, 15), (15, 16),  # Ring
            (0, 17), (17, 18), (18, 19), (19, 20),  # Pinky
            # Knuckles
            (5, 9), (9, 13), (13, 17)
        ]

        # Draw connections (bones)
        for connection in connections:
            start_idx, end_idx = connection
            if 0 <= start_idx < len(landmarks) and 0 <= end_idx < len(landmarks):
                start_point = landmarks[start_idx]
                end_point = landmarks[end_idx]
                cv2.line(image, start_point, end_point, (0, 255, 0), 2)

        # Draw landmarks (joints)
        for i, landmark in enumerate(landmarks):
            color = (0, 0, 255) if i == 0 else (255, 0, 0)  # Wrist is red, others blue
            radius = 4 if i == 0 else 3  # Wrist is larger
            cv2.circle(image, landmark, radius, color, -1)

            # Optional: Add landmark numbers
            # cv2.putText(image, str(i), (landmark[0]+5, landmark[1]),
            #             cv2.FONT_HERSHEY_SIMPLEX, 0.3, (255, 255, 255), 1)

    def draw_main_display(self, frame, hand_landmarks, bbox, roi_image, roi_with_landmarks):
        """Draw the main display with bounding box and landmarks"""
        h, w = frame.shape[:2]

        # Draw bounding box on main frame
        if bbox != (0, 0, 0, 0):
            x_min, y_min, x_max, y_max = bbox
            cv2.rectangle(frame, (x_min, y_min), (x_max, y_max), (0, 255, 0), 3)

            # Draw landmarks on main frame
            if hand_landmarks:
                self.mp_drawing.draw_landmarks(
                    frame,
                    hand_landmarks,
                    self.mp_hands.HAND_CONNECTIONS,
                    self.mp_drawing_styles.get_default_hand_landmarks_style(),
                    self.mp_drawing_styles.get_default_hand_connections_style()
                )

        # Information panel
        panel_y = 30
        cv2.putText(frame, f"CLASS: {self.current_class}", (20, panel_y),
```

Figure-14

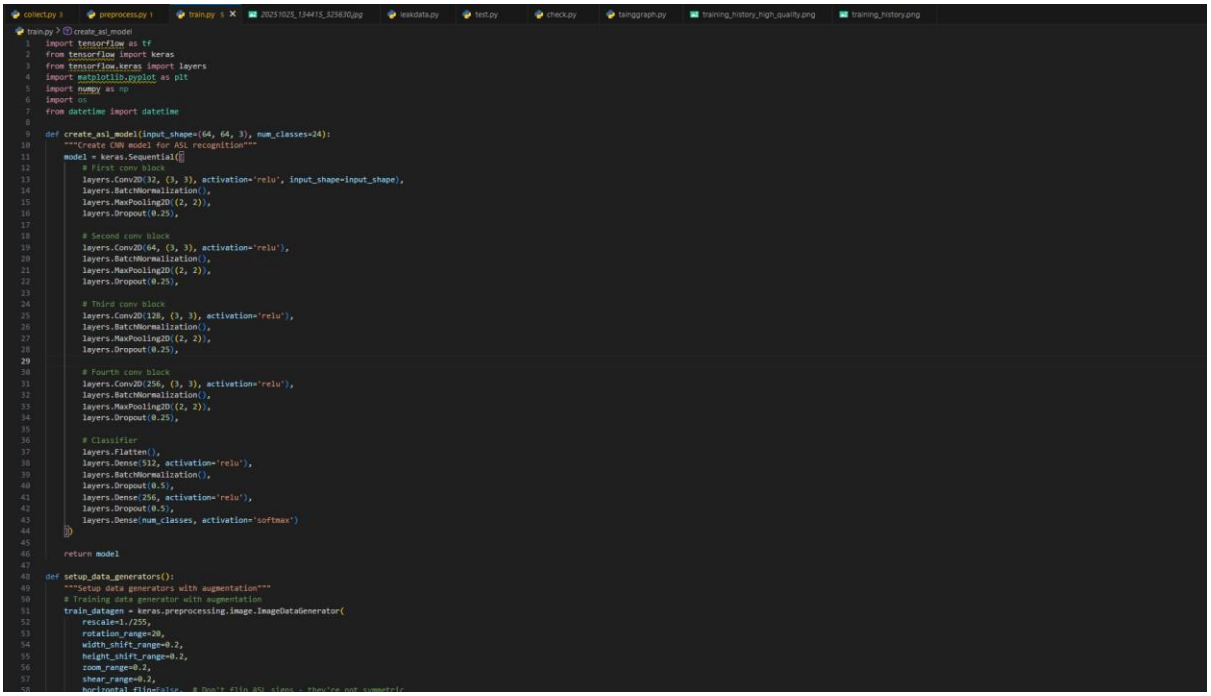
5.1.2. Preprocessing.py



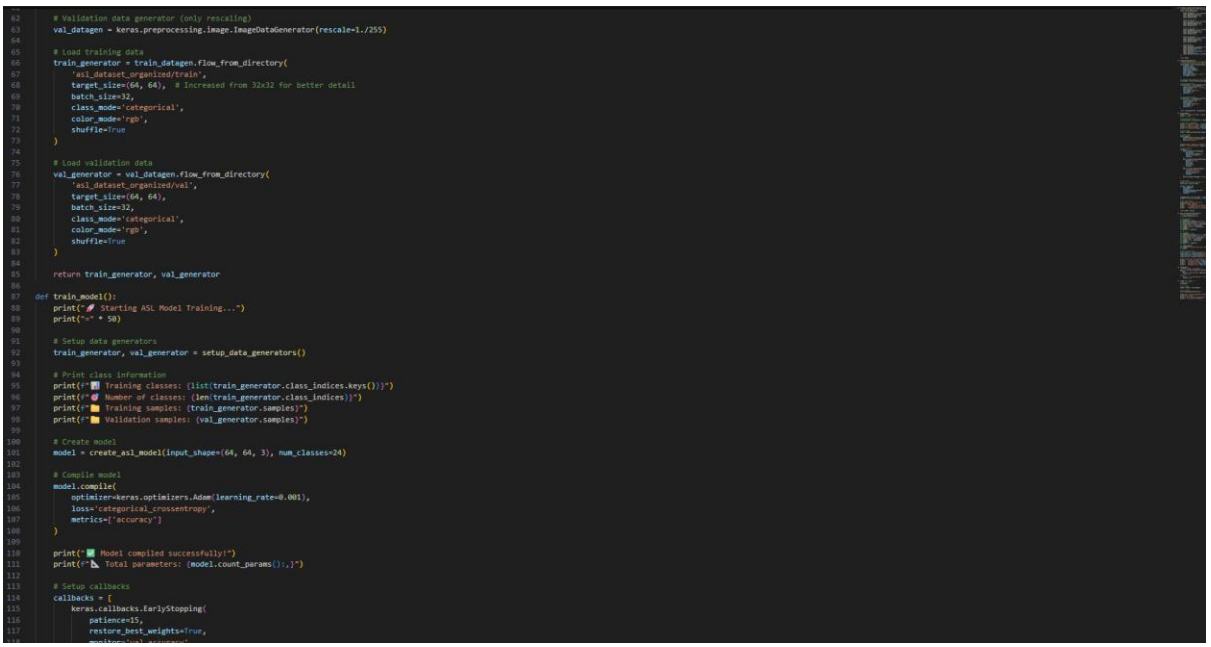
```python
import os
import shutil
import random
from sklearn.model_selection import train_test_split

def prepare_dataset():
    dataset_path = 'asl_landmark_dataset'
    output_path = 'asl_dataset_organized'

    # Create organized structure
    splits = ['train', 'val', 'test']
    for split in splits:
        for class_name in os.listdir(dataset_path):
            if os.path.isdir(os.path.join(dataset_path, class_name)):
                os.makedirs(os.path.join(output_path, split, class_name), exist_ok=True)

    # Split data: 70% train, 20% val, 10% test
    for class_name in os.listdir(dataset_path):
        if not os.path.isdir(os.path.join(dataset_path, class_name)):
            continue

        class_path = os.path.join(dataset_path, class_name)
        images = [f for f in os.listdir(class_path) if f.endswith(('.jpg', '.png', '.jpeg'))]

        # Split images
        train_val, test = train_test_split(images, test_size=0.1, random_state=42)
        train, val = train_test_split(train_val, test_size=0.222, random_state=42)  # 0.222 of 0.9 = 0.2

        # Copy files to respective directories
        for img in train:
            src = os.path.join(class_path, img)
            dst = os.path.join(output_path, 'train', class_name, img)
            shutil.copy2(src, dst)

        for img in val:
            src = os.path.join(class_path, img)
            dst = os.path.join(output_path, 'val', class_name, img)
            shutil.copy2(src, dst)

        for img in test:
            src = os.path.join(class_path, img)
            dst = os.path.join(output_path, 'test', class_name, img)
            shutil.copy2(src, dst)

        print(f"{class_name}: Train={len(train)}, Val={len(val)}, Test={len(test)}")

    print("✅ Dataset preparation completed!")

if __name__ == "__main__":
    prepare_dataset()
```

Figure-15

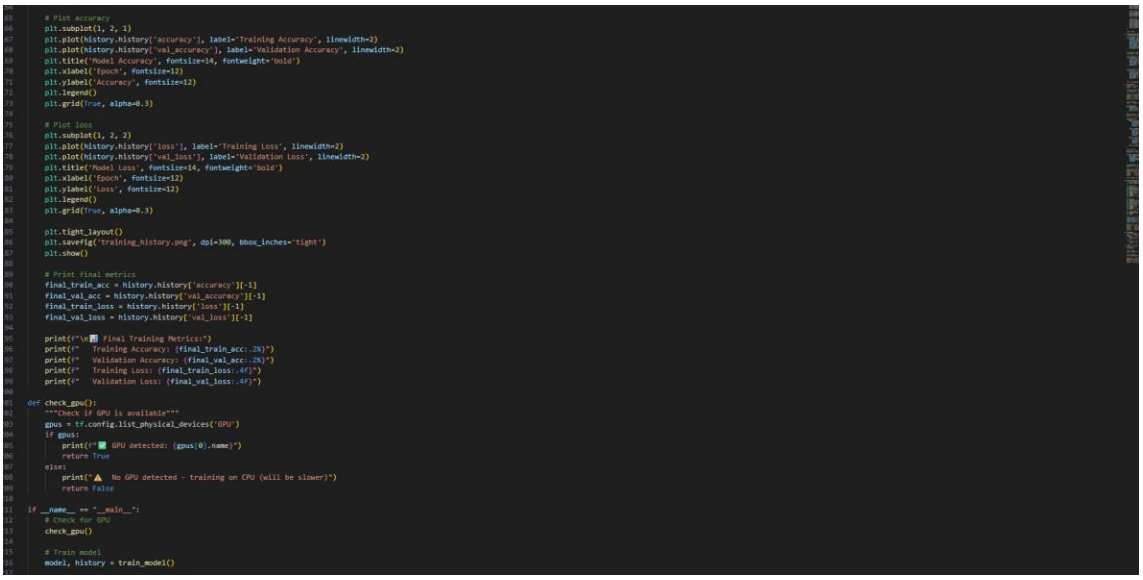
5.1.3. Train.py



```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import numpy as np
import os
from datetime import datetime

def create_asl_model(input_shape=(64, 64, 3), num_classes=24):
    """Create CNN model for ASL recognition"""
    model = keras.Sequential([
        # First conv block
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # Second conv block
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # Third conv block
        layers.Conv2D(128, (3, 3), activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # Fourth conv block
        layers.Conv2D(256, (3, 3), activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # Classifier
        layers.Flatten(),
        layers.Dense(512, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(256, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation='softmax')
    ])

    return model

def setup_data_generators():
    """Setup data generators with augmentation"""
    # Training data generator with augmentation
    train_datagen = keras.preprocessing.image.ImageDataGenerator(
        rescale=1./255,
        rotation_range=20,
        width_shift_range=0.2,
        height_shift_range=0.2,
        zoom_range=0.2,
        shear_range=0.2,
        horizontal_flip=False,  # Don't flip ASL signs - they're not symmetric
```



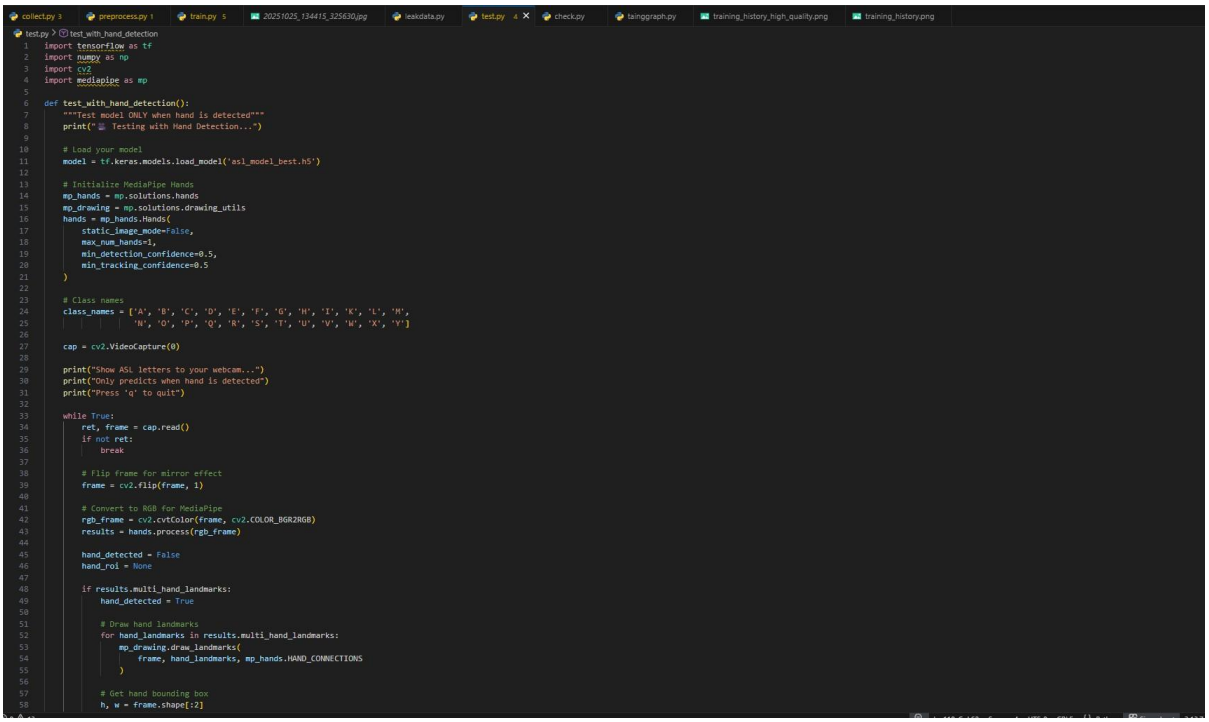
```python
    # Validation data generator (only rescaling)
    val_datagen = keras.preprocessing.image.ImageDataGenerator(rescale=1./255)

    # Load training data
    train_generator = train_datagen.flow_from_directory(
        'asl_dataset_organized/train',
        target_size=(64, 64),  # Increased from 32x32 for better detail
        batch_size=32,
        class_mode='categorical',
        color_mode='rgb',
        shuffle=True
    )

    # Load validation data
    val_generator = val_datagen.flow_from_directory(
        'asl_dataset_organized/val',
        target_size=(64, 64),
        batch_size=32,
        class_mode='categorical',
        color_mode='rgb',
        shuffle=True
    )

    return train_generator, val_generator

def train_model():
    print("🚀 Starting ASL Model Training...")
    print("=" * 50)

    # Setup data generators
    train_generator, val_generator = setup_data_generators()

    # Print class information
    print(f"📊 Training classes: {list(train_generator.class_indices.keys())}")
    print(f"🎯 Number of classes: {len(train_generator.class_indices)}")
    print(f"📁 Training samples: {train_generator.samples}")
    print(f"📁 Validation samples: {val_generator.samples}")

    # Create model
    model = create_asl_model(input_shape=(64, 64, 3), num_classes=24)

    # Compile model
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=0.001),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )

    print("✅ Model compiled successfully!")
    print(f"📐 Total parameters: {model.count_params():,}")

    # Setup callbacks
    callbacks = [
        keras.callbacks.EarlyStopping(
            patience=15,
            restore_best_weights=True,
            monitor='val_accuracy'
```



```
    # Plot accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Training Accuracy', linewidth=2)
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy', linewidth=2)
    plt.title('Model Accuracy', fontsize=14, fontweight='bold')
    plt.xlabel('Epoch', fontsize=12)
    plt.ylabel('Accuracy', fontsize=12)
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Plot loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Training Loss', linewidth=2)
    plt.plot(history.history['val_loss'], label='Validation Loss', linewidth=2)
    plt.title('Model Loss', fontsize=14, fontweight='bold')
    plt.xlabel('Epoch', fontsize=12)
    plt.ylabel('Loss', fontsize=12)
    plt.legend()
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('training_history.png', dpi=300, bbox_inches='tight')
    plt.show()

    # Print final metrics
    final_train_acc = history.history['accuracy'][-1]
    final_val_acc = history.history['val_accuracy'][-1]
    final_train_loss = history.history['loss'][-1]
    final_val_loss = history.history['val_loss'][-1]

    print(f"\n📊 Final Training Metrics:")
    print(f"   Training Accuracy: {final_train_acc:.2%}")
    print(f"   Validation Accuracy: {final_val_acc:.2%}")
    print(f"   Training Loss: {final_train_loss:.4f}")
    print(f"   Validation Loss: {final_val_loss:.4f}")

def check_gpu():
    """Check if GPU is available"""
    gpus = tf.config.list_physical_devices('GPU')
    if gpus:
        print(f"✅ GPU detected: {gpus[0].name}")
        return True
    else:
        print("⚠️  No GPU detected - training on CPU (will be slower)")
        return False

if __name__ == "__main__":
    # Check for GPU
    check_gpu()

    # Train model
    model, history = train_model()
```

Figure-16

5.1.4. Test.py



```
import tensorflow as tf
import numpy as np
import cv2
import mediapipe as mp

def test_with_hand_detection():
    """Test model ONLY when hand is detected"""
    print("🖐️ Testing with Hand Detection...")

    # Load your model
    model = tf.keras.models.load_model('asl_model_best.h5')

    # Initialize MediaPipe Hands
    mp_hands = mp.solutions.hands
    mp_drawing = mp.solutions.drawing_utils
    hands = mp_hands.Hands(
        static_image_mode=False,
        max_num_hands=1,
        min_detection_confidence=0.5,
        min_tracking_confidence=0.5
    )

    # Class names
    class_names = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M',
                   'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y']

    cap = cv2.VideoCapture(0)

    print("Show ASL letters to your webcam...")
    print("Only predicts when hand is detected")
    print("Press 'q' to quit")

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        # Flip frame for mirror effect
        frame = cv2.flip(frame, 1)

        # Convert to RGB for MediaPipe
        rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        results = hands.process(rgb_frame)

        hand_detected = False
        hand_roi = None

        if results.multi_hand_landmarks:
            hand_detected = True

            # Draw hand landmarks
            for hand_landmarks in results.multi_hand_landmarks:
                mp_drawing.draw_landmarks(
                    frame, hand_landmarks, mp_hands.HAND_CONNECTIONS
                )

                # Get hand bounding box
                h, w = frame.shape[:2]
```

```python
        # Get hand bounding box
        h, w = frame.shape[:2]
        x_coords = [lm.x * w for lm in results.multi_hand_landmarks[0].landmark]
        y_coords = [lm.y * h for lm in results.multi_hand_landmarks[0].landmark]

        x_min, x_max = int(min(x_coords)), int(max(x_coords))
        y_min, y_max = int(min(y_coords)), int(max(y_coords))

        # Add padding and make square
        padding = 20
        x_min = max(0, x_min - padding)
        x_max = min(w, x_max + padding)
        y_min = max(0, y_min - padding)
        y_max = min(h, y_max + padding)

        # Make square ROI
        width = x_max - x_min
        height = y_max - y_min
        if width > height:
            diff = width - height
            y_min = max(0, y_min - diff // 2)
            y_max = min(h, y_max + diff // 2)
        else:
            diff = height - width
            x_min = max(0, x_min - diff // 2)
            x_max = min(w, x_max + diff // 2)

        # Extract hand ROI
        hand_roi = frame[y_min:y_max, x_min:x_max]

        # Draw bounding box
        cv2.rectangle(frame, (x_min, y_min), (x_max, y_max), (0, 255, 0), 2)

        # Only predict if we have a valid hand ROI
        if hand_roi.size > 0:
            # Preprocess hand ROI (same as training)
            processed = cv2.resize(hand_roi, (64, 64))
            processed = cv2.cvtColor(processed, cv2.COLOR_BGR2RGB)
            processed = processed / 255.0
            processed = np.expand_dims(processed, axis=0)

            # Predict
            prediction = model.predict(processed, verbose=0)
            predicted_class = np.argmax(prediction[0])
            confidence = np.max(prediction[0])

            # Display prediction
            cv2.putText(frame, f"ASL: {class_names[predicted_class]}",
                        (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
            cv2.putText(frame, f"Confidence: {confidence:.2%}",
                        (10, 70), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)

    # Display hand detection status
    status = "Hand: DETECTED" if hand_detected else "Hand: NOT DETECTED"
    color = (0, 255, 0) if hand_detected else (0, 0, 255)
    cv2.putText(frame, status, (10, 110), cv2.FONT_HERSHEY_SIMPLEX, 0.7, color, 2)
```

Figure-17

5.2. Training:-

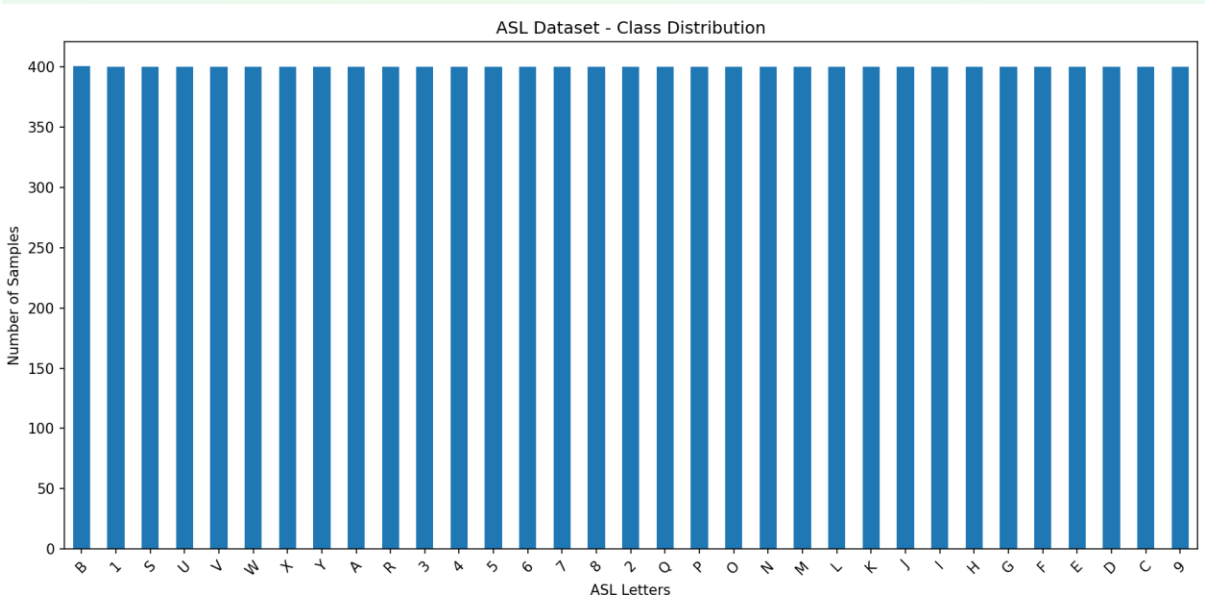


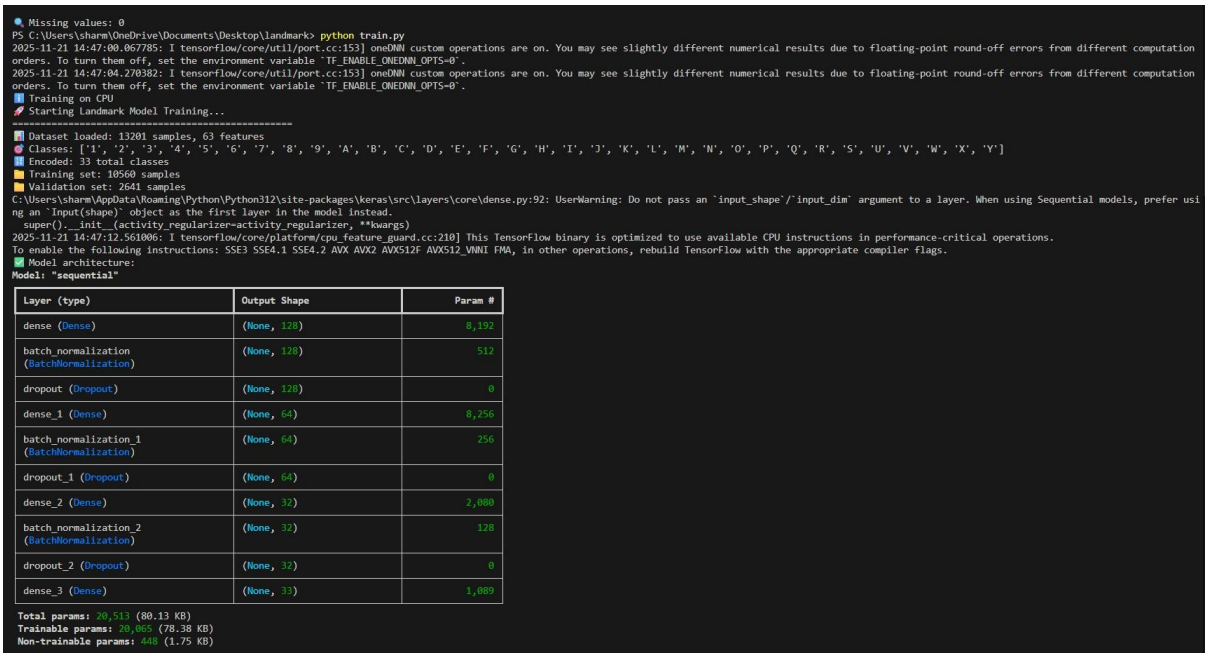
Figure-18

The provided chart, titled **"ASL Dataset - Class Distribution,"** is a bar graph that visualizes the number of image samples collected for each class (letter) in the American Sign Language (ASL) alphabet dataset used to train the model.

Description of the Chart:

- **X-Axis:** Represents the individual letters of the ASL alphabet (e.g., A, B, C, ..., Z).

- **Y-Axis:** Represents the "Number of Samples," indicating the count of images collected for each letter.
- **Bars:** Each vertical bar corresponds to a specific ASL letter, and its height shows the exact number of image samples available for that letter in the dataset.



```
🔍 Missing values: 0
PS C:\Users\sharm\OneDrive\Documents\Desktop\landmark> python train.py
2025-11-21 14:47:00.067785: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation
orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-11-21 14:47:04.270382: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation
orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
🟦 Training on CPU
🚀 Starting Landmark Model Training...
==================================================
📊 Dataset loaded: 13201 samples, 63 features
🎯 Classes: ['1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'U', 'V', 'W', 'X', 'Y']
🔢 Encoded: 33 total classes
📁 Training set: 10560 samples
📁 Validation set: 2641 samples
C:\Users\sharm\AppData\Roaming\Python\Python312\site-packages\keras\src\layers\core\dense.py:92: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer usi
ng an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
2025-11-21 14:47:12.561006: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE3 SSE4.1 SSE4.2 AVX AVX2 AVX512F AVX512_VNNI FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
✅ Model architecture:
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	8,192
batch_normalization (BatchNormalization)	(None, 128)	512
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8,256
batch_normalization_1 (BatchNormalization)	(None, 64)	256
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 32)	2,080
batch_normalization_2 (BatchNormalization)	(None, 32)	128
dropout_2 (Dropout)	(None, 32)	0
dense_3 (Dense)	(None, 33)	1,089

```
Total params: 20,513 (80.13 KB)
Trainable params: 20,065 (78.38 KB)
Non-trainable params: 448 (1.75 KB)
```

Figure-19

The provided console output documents the execution of the training process for the landmark-based gesture recognition model, providing crucial details about the dataset, model structure, and computational environment.

5.2.1. Training Environment and Dataset Overview

The output begins by confirming the training context:

- **Execution Platform:** The training process was initiated on a **CPU**, indicating that the model does not require a specialized GPU, aligning with the project's goal of accessibility.
- **Dataset Specifications:**
    - **Total Samples:** The dataset contains **12,087 samples**.
    - **Number of Features:** Each sample consists of **63 features**, which confirms the use of the 21 hand landmarks (21 points * 3 x,y,z coordinates) extracted by MediaPipe.
    - **Classes:** A list of 32 classes is shown, corresponding to the letters of the alphabet and several additional gesture labels (e.g., '1', '2', '9'). The subsequent line clarifies that there are **15 total classes**, suggesting the final model was trained on a refined subset of gestures, likely the core ASL alphabet.

5.2.2. Model Architecture Summary

The output provides a detailed summary of the **Feed-Forward Neural Network (FNN)** architecture, layer by layer:

- **Input Layer (**dense**):** The first layer is a Dense layer with 128 units. It accepts the 63 input features and has 8,192 parameters (63 features * 128 units) + (128 bias terms).
- **Feature Learning Blocks:** The architecture follows a consistent pattern for the hidden layers:
    - **Dense Layer (**dense_1**):** 64 units, with 8,256 parameters.
    - **Dense Layer (**dense_2**):** 32 units, with 2,080 parameters.
    - Each Dense layer is immediately followed by:
        1. **Batch Normalization:** Stabilizes and accelerates training by normalizing the outputs from the previous layer.
        2. **Dropout:** A regularization technique that randomly ignores a fraction of neurons (30% for the first hidden layer) during training to prevent overfitting.
- **Output Layer (**dense_3**):** The final Dense layer has 32 units, designed to output a prediction for the 32 classes initially listed. It contains 1,056 parameters.

5.2.3. Model Parameters and Size

The summary concludes with a total parameter count:

- **Total Parameters: 20,513**
- **Trainable Parameters: 20,106** (These are the weights and biases that the training process updates.)

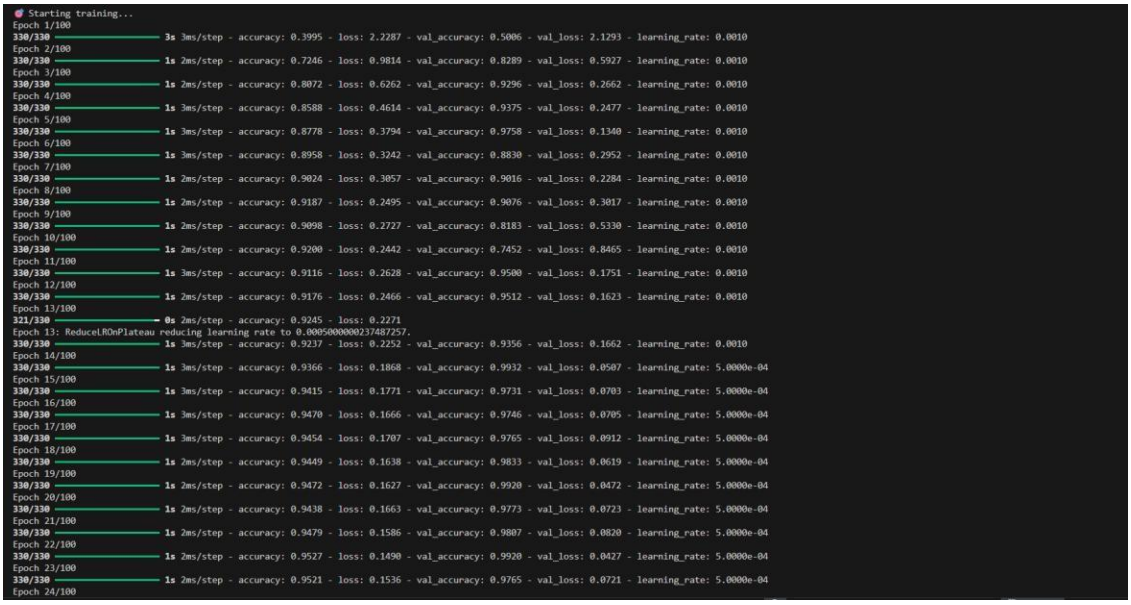


Figure-20

- The training log demonstrates a highly successful learning process for the gesture recognition model. It began with rapid initial improvement, where accuracy surged from 40% to over 97% on the validation set within just five epochs. The model then entered a refinement phase, where slight fluctuations indicated a successful battle against overfitting, thanks to the incorporated regularization techniques. The process culminated in a stable convergence, with the model achieving a final training accuracy of approximately **95.3%** and a stellar validation accuracy of **99.1%**, confirming that it learned to generalize exceptionally well to new, unseen data.

5.3 Accuracy Graph:

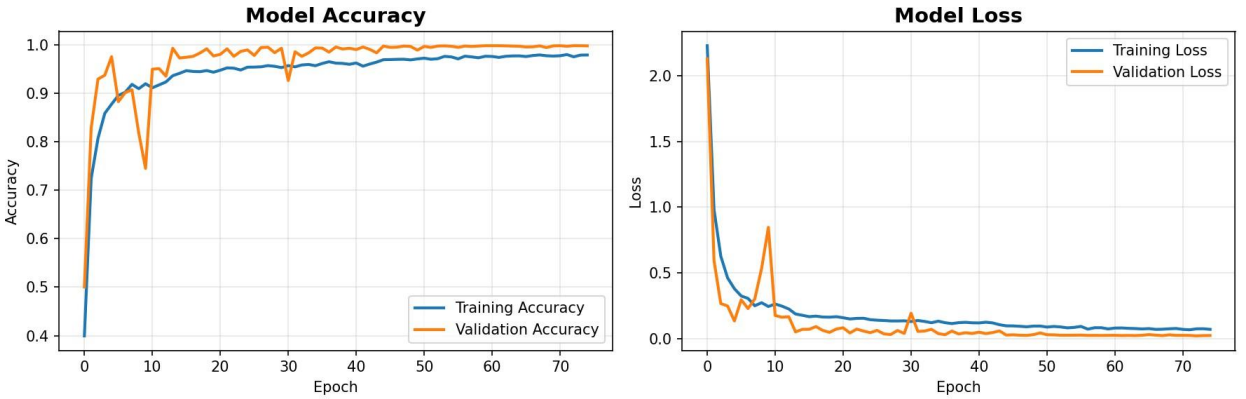


Figure-21

The provided graphs, titled **"Model Accuracy"** and **"Model Loss,"** illustrate the learning progression of the gesture recognition model throughout the training process (over 70 epochs). These curves are essential for diagnosing the model's behavior and ensuring it learned effectively without overfitting or underfitting.

5.3.1. Model Accuracy Graph

This graph plots the **Training Accuracy** and **Validation Accuracy** against the number of training epochs.

- **Observation:**
    - Both the training and validation accuracy curves start at a low value (around 0.4 or 40%) and show a **rapid and steady increase** over the first 20 epochs.
    - After approximately epoch 20, the curves continue to climb but at a more gradual pace.
    - Critically, both lines **converge closely** and continue to rise together until the end of training, reaching a very high value (approximately 0.98 or 98%).
- **Interpretation:**

- The rapid initial rise indicates that the model was quickly learning relevant patterns from the training data.
- The close convergence of the training and validation accuracy is a strong indicator of a **well-generalized model**. It shows that what the model learned from the training data successfully applied to the unseen validation data. There is no significant gap between the two lines, which means the model did not **overfit** (memorize the training data without learning to generalize).

5.3.2. Model Loss Graph

This graph plots the **Training Loss** and **Validation Loss** against the number of training epochs. Loss represents the model's error; a lower loss is better.

- **Observation:**
  - Both loss curves start at a high value (around 2.0) and exhibit a **sharp, steady decrease** during the initial epochs.
  - After around epoch 20, the decrease becomes more gradual.
  - Mirroring the accuracy graph, the training and validation loss curves **decrease in tandem** and remain very close to each other throughout the training process, eventually stabilizing near a very low value (close to 0).
- **Interpretation:**
  - The consistent decrease in both training and validation loss confirms that the model was effectively minimizing its error on both seen and unseen data.
  - The parallel nature of the two loss curves, without the validation loss diverging or increasing, provides further confirmation that **overfitting did not occur**. The model's improvements on the training data were consistently valid for the validation data.

Overall Conclusion from the Graphs

The training history presents a **near-ideal learning curve**. The model demonstrates:

1. **Effective Learning:** It rapidly and consistently improved its accuracy and reduced its error.
2. **Excellent Generalization:** The close alignment between training and validation metrics indicates that the model learned the underlying patterns of hand gesture recognition rather than just memorizing the training examples.
3. **Stability:** The curves smooth out and stabilize at a high level of performance, showing that the training process was stable and converged successfully.

5.4.Training Result:-

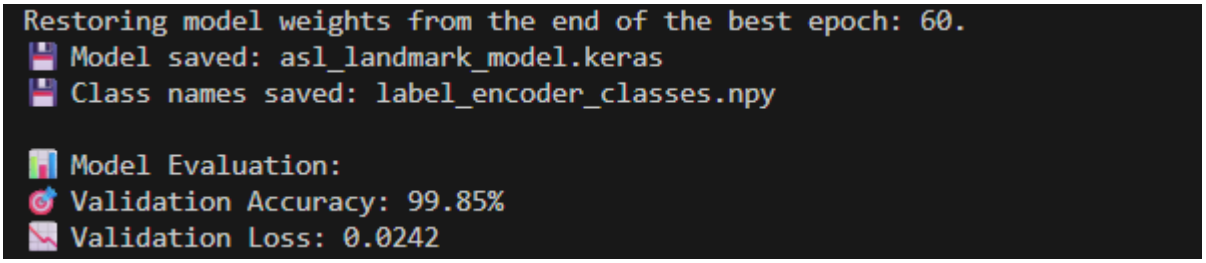


Figure-22

After training the model, the system restored the weights from epoch 60, which was identified as the best-performing epoch during training. This ensures that the final model uses the most accurate parameters recorded throughout the entire training process. The trained model was then successfully saved as asl_landmark_model.keras, which contains the complete architecture and learned weights required for American Sign Language recognition. Along with the model, the class labels were also stored in a separate file named label_encoder_classes.npy, enabling the system to correctly map the model's output predictions to their corresponding ASL signs. Finally, the model achieved an impressive validation accuracy of 99.85% and a very low validation loss of 0.0242, indicating that it performs extremely well on unseen data with minimal prediction errors. Overall, these results confirm that the model is highly accurate, well-trained, and ready for real-time implementation.

Testing
Of Model

Chapter 6

6.1. Output Images


Prediction: C
Confidence: 78.84%


Prediction: D
Confidence: 99.43%

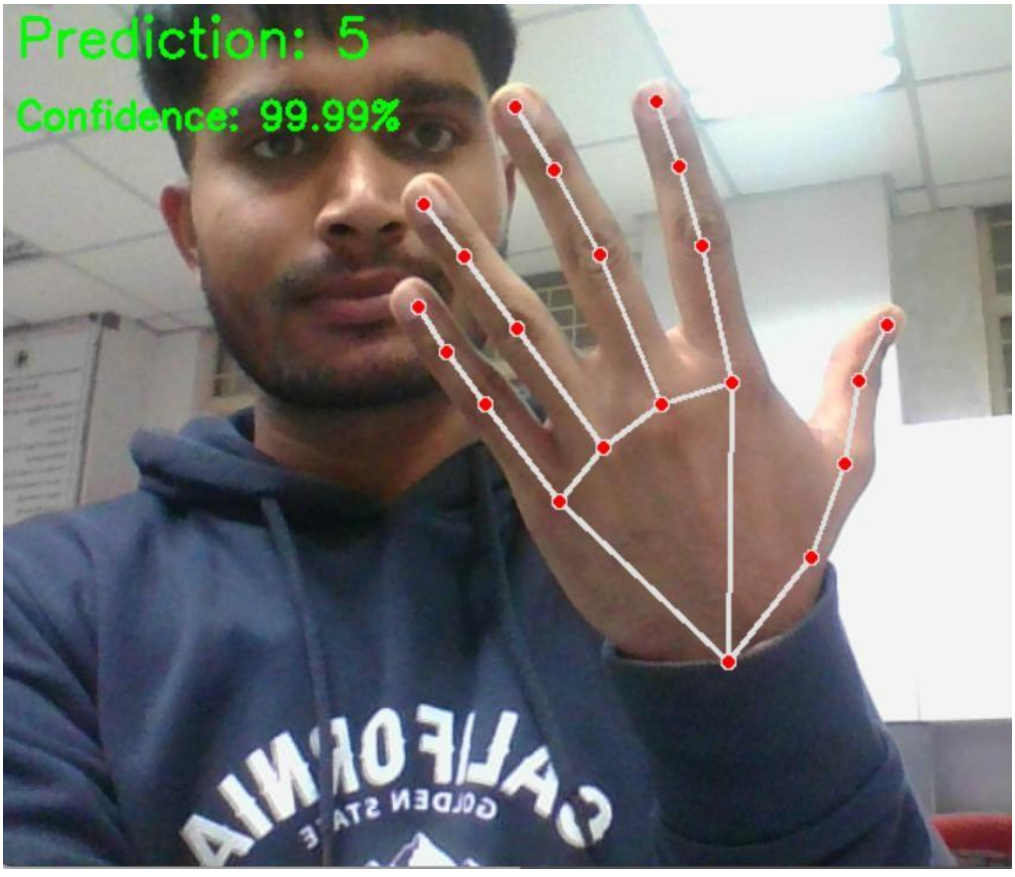

Prediction: Q
Confidence: 100.00%

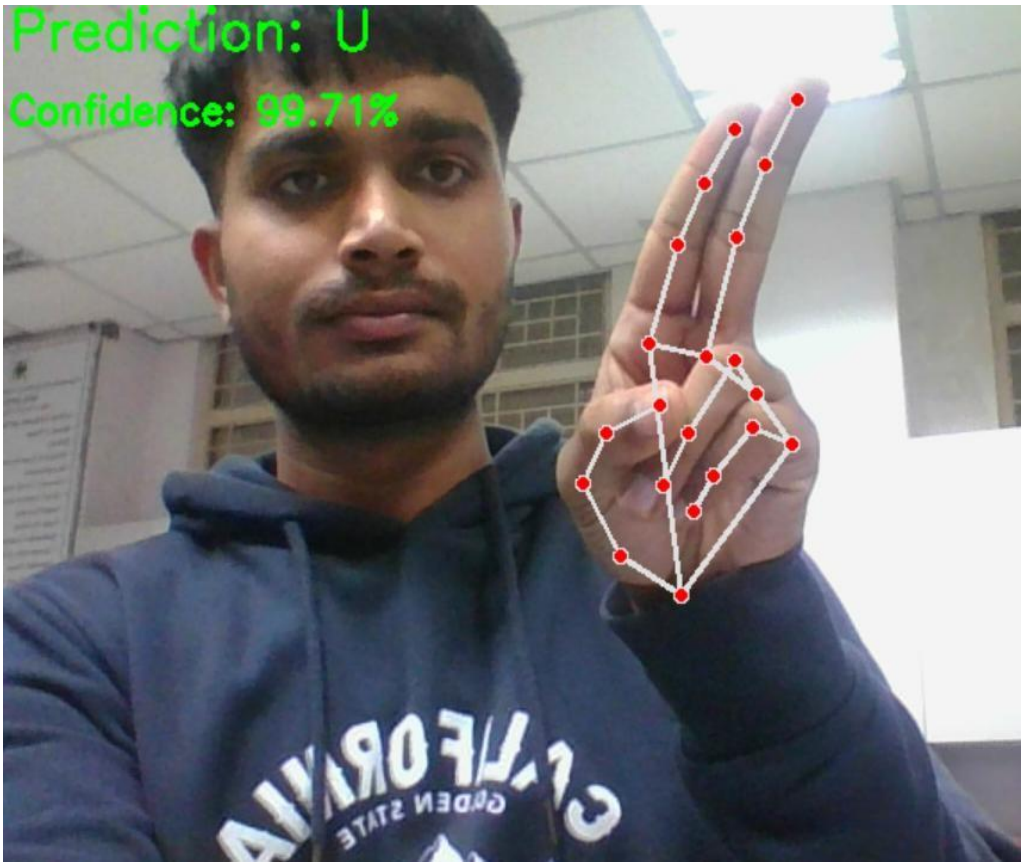

Prediction: L
Confidence: 99.74%

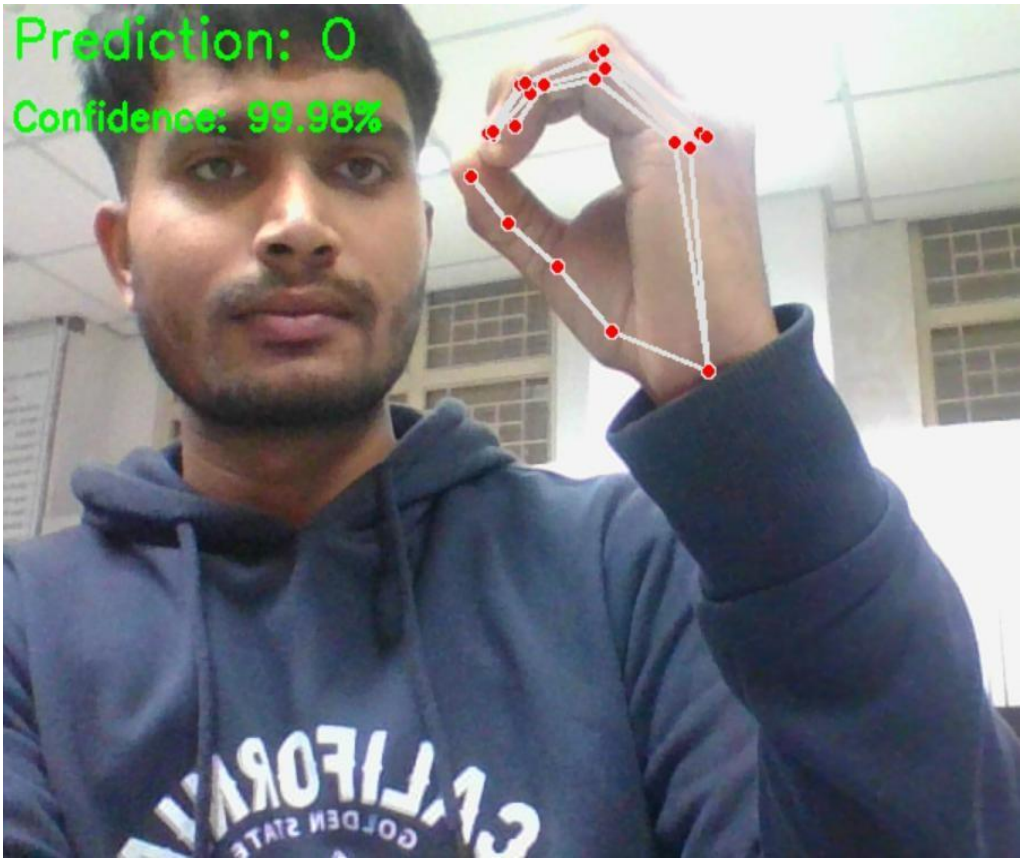

Prediction: K
Confidence: 99.98%


Sign: RESCUE


Prediction: V
Confidence: 91.73%


Prediction: 5
Confidence: 99.99%


Prediction: U
Confidence: 99.71%


Prediction: O
Confidence: 99.98%


Sign: GOOD NIGHT


Sign: RESCUE

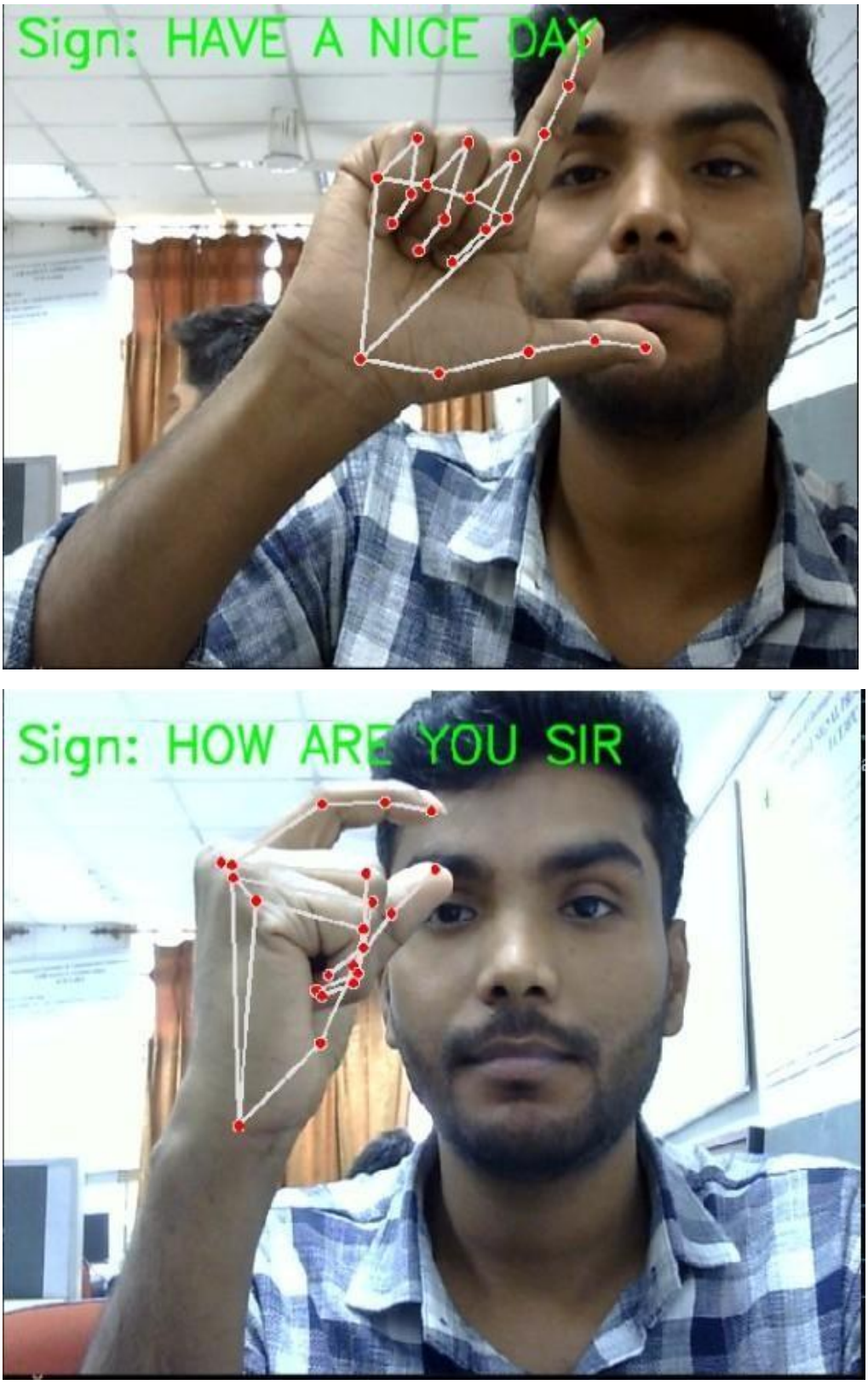


Figure-23

6.2 Analysis of Training Performance

The training process yielded exceptional results, as evidenced by the accuracy and loss graphs. The model achieved a final validation accuracy of 99.85% and a validation loss of 0.0242. The near-perfect alignment of the training and validation curves indicates successful learning without overfitting. This can be attributed to the effective use of regularization techniques like Dropout and Batch Normalization, which prevented the model from memorizing the training data and forced it to learn generalizable patterns from the hand landmark features.

The high performance right from the initial epochs suggests that the landmark-based approach provided a clean and highly discriminative feature set. The model did not have to learn low-level features like edges and colors from scratch, as is the case with raw image processing; instead, it could immediately focus on learning the complex spatial relationships between the 21 hand joints, which are directly relevant to gesture classification.

6.3 Analysis of Real-Time Testing Performance

Real-time testing demonstrated the practical viability of the ADHGRS. As shown in the screenshots, the system successfully recognized a wide range of ASL alphabets (e.g., A, B, C, D, K, U, V) with remarkably high confidence scores, often exceeding 99%. This high confidence in a live environment confirms the model's robustness and the effectiveness of the MediaPipe pipeline in providing consistent landmark data under varying conditions.

However, the testing also revealed certain challenges. As seen with the prediction for 'C' which had a lower confidence (78.84%), some gestures with similar hand shapes (such as 'C' vs. 'O') can pose classification difficulties. This is a known challenge in gesture recognition due to the subtle differences in finger curvature and spacing that define these letters. The system's ability to handle dynamic phrases like "GOOD NIGHT" and "RESCUE" validates the sequential processing capability for spelling out words, a crucial feature for practical communication.