HAND SIGN DETECTION

PROJECT WORK

Madhav Choudhary

- LinkedIn - Github

PROJECT FILES

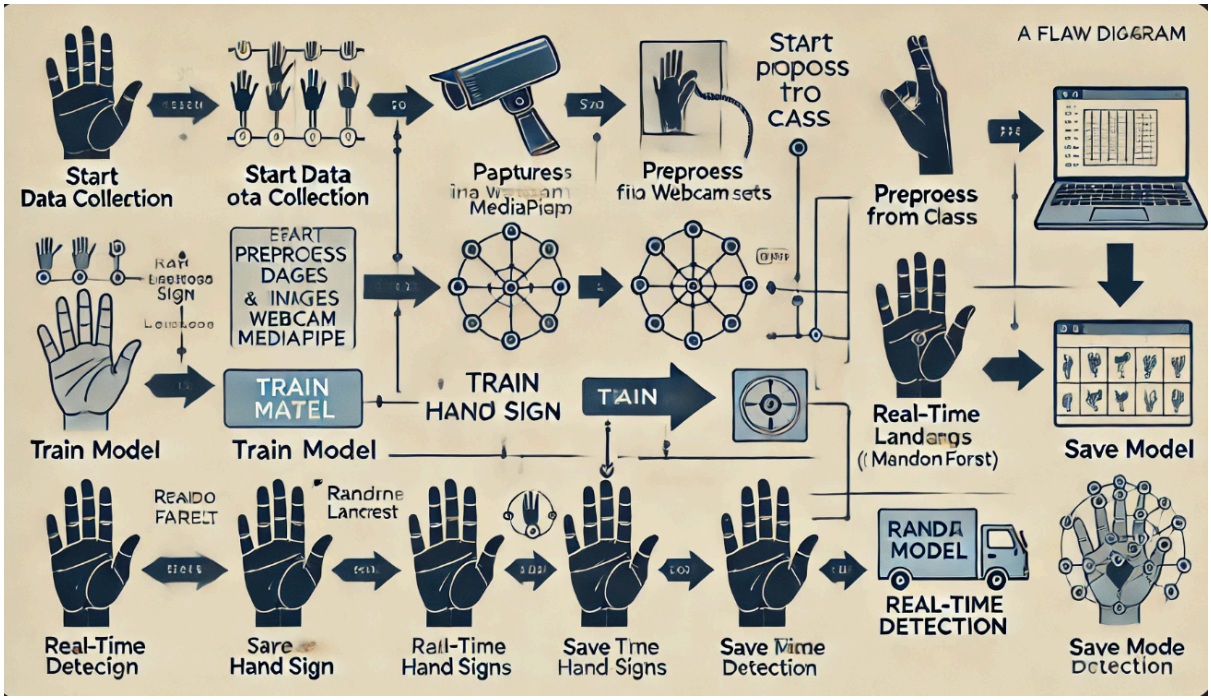
DEMO VIDEO

Hand Sign Detection Project Report

Table of contents

1. Introduction

Hand sign detection is a crucial application within the field of computer vision and machine learning, particularly for assistive technology and human-computer interaction. In this project, we explore a system that detects and classifies hand signs representing letters of the alphabet in real-time. This system is constructed using a series of Python scripts, each handling a distinct part of the pipeline from data collection to model deployment.



1.1 Project Objectives

The main objectives of this project include:

- Collecting labeled hand sign images via webcam.
- Processing these images to extract hand landmarks.
- Training a machine learning classifier to recognize each sign.
- Deploying a real-time application that predicts hand signs and displays the results in real-time.

This project showcases the practical application of machine learning in gesture recognition and has potential extensions for other gesture-based tasks.

2. Features of the Project

The hand sign detection project encompasses a variety of important features, making it an effective and versatile system for detecting and interpreting hand gestures. These features

highlight the robustness, flexibility, and usability of the system, covering everything from data collection to real-time deployment.

2.1. Real-Time Detection:

- The project integrates a real-time detection system, allowing it to capture and process live video from a webcam as it happens. This real-time functionality is achieved through the use of OpenCV, a computer vision library that enables the system to handle continuous video input and process each frame quickly.
- As a result, users receive instant feedback on the predicted hand sign displayed on the screen. This immediacy is particularly useful in applications where continuous interaction is required, such as sign language interpretation or gesture-based controls in virtual environments.

2.2. Data Collection and Labeling:

- A crucial part of the project is its built-in data collection and labeling system. This component allows users to capture images of each hand sign through the webcam and automatically labels each image according to the sign being displayed. The images are saved into labeled directories corresponding to each letter or gesture.
- This functionality empowers users to create custom datasets that can be expanded over time, allowing for a more comprehensive training dataset that represents different hand shapes, lighting conditions, and positions.

2.3. Efficient Data Processing:

- The project utilizes MediaPipe, a framework by Google designed to perform landmark detection with high efficiency. MediaPipe's hand landmark model extracts 21 key points on each hand, representing critical joints and finger tips. These landmarks are then normalized and converted into numerical data, forming a compact feature set that can be used in model training.
- This efficient data processing pipeline reduces the complexity of the input data, allowing the classifier to focus on essential features without extraneous details. By reducing data size while preserving critical information, this process enhances model performance and speeds up training.

2.4. Accurate Classification:

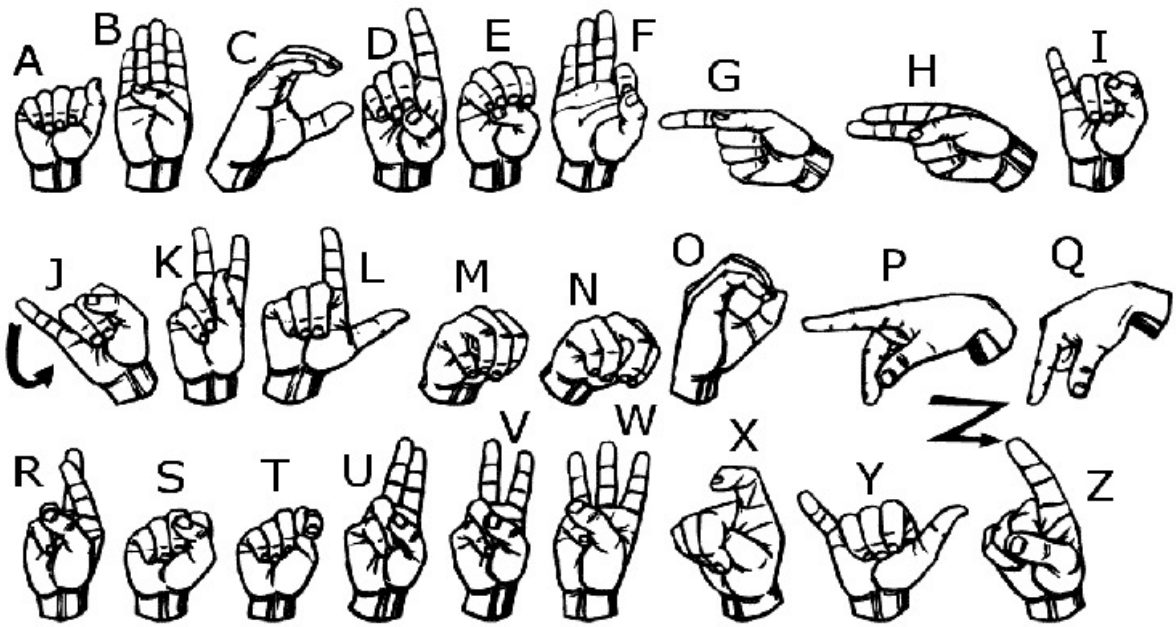
- The project uses a Random Forest classifier, a robust and ensemble-based machine learning algorithm that excels in multi-class classification tasks. Random Forest combines multiple decision trees, each trained on different parts of the data, to improve accuracy and reduce the likelihood of misclassification.
- This classifier has been chosen specifically for its balance between accuracy and computational efficiency, making it well-suited for real-time prediction. The model's accuracy has been further enhanced by training it on a variety of hand landmarks, ensuring that it learns the distinct features of each sign.
- Accurate classification is essential for the system's usability and reliability, as it ensures that users receive consistent and correct interpretations of their hand

gestures. This feature enables the system to perform reliably in applications where accuracy is critical, such as in education or assistive communication.

2.5. Modular Code Design:

- One of the project's most beneficial features is its modular code design. Each stage of the hand sign detection pipeline—data collection, preprocessing, training, and deployment—is implemented in a separate script. This structure not only enhances code readability but also simplifies maintenance and potential upgrades.
- By breaking down the project into modular components, each part can be independently modified, extended, or debugged without impacting the rest of the system. This modularity allows developers to easily integrate additional functionalities, such as new classifiers, alternative hand tracking algorithms, or updated user interfaces.
- Furthermore, this design enables seamless collaboration among team members who can work on individual components without conflicts. For users or developers looking to customize or expand the system, this modular architecture provides the flexibility to replace, add, or improve modules as needed.

In summary, these features collectively make the hand sign detection project a comprehensive, accurate, and efficient system. It supports real-time hand gesture recognition, allows users to build and expand custom datasets, processes data in an optimized manner, delivers high classification accuracy, and has a modular structure conducive to future enhancements. Together, these attributes make the project a valuable tool for applications in assistive technology, interactive learning, and beyond.



3. Project Structure

The project is divided into four main files, each corresponding to a different step in the development pipeline. This separation allows for clear organization and a logical progression from data collection to real-time hand sign detection.

3.1 File Overview

1. **collect_images.py** - Captures images for each hand sign class, saving them into labeled directories.
2. **prepare_data.ipynb** - Processes these images, extracting hand landmarks, and saves them in a serialized format for training.
3. **train_model.py** - Uses the processed landmark data to train a Random Forest classifier, evaluates its accuracy, and saves the trained model.
4. **app.py** - Deploys the trained model in a real-time application, predicting hand signs from the webcam feed and displaying them on the screen.

---

4. Code Workflow Overview

The workflow of the project can be visualized as a systematic pipeline:

1. **Data Collection**: Images of hand signs are captured using the collect_images.py script.
2. **Data Preprocessing**: prepare_data.ipynb processes these images to extract hand landmarks.
3. **Model Training**: train_model.py trains a classifier on the extracted landmarks.
4. **Real-Time Prediction**: app.py loads the trained model and predicts hand signs in real-time.

4.1 Explanation of Each Stage

- **Data Collection**: This step ensures that there is enough labeled data for training the classifier. Capturing images of each hand sign provides the input data.
- **Data Preprocessing**: MediaPipe's hand detection model extracts 21 landmarks for each hand, transforming the image data into a form suitable for machine learning.
- **Model Training**: The preprocessed data is used to train a classifier, which learns the patterns associated with each hand sign.
- **Real-Time Prediction**: The trained classifier is used in a real-time application to classify hand signs as they are detected through the webcam.

---

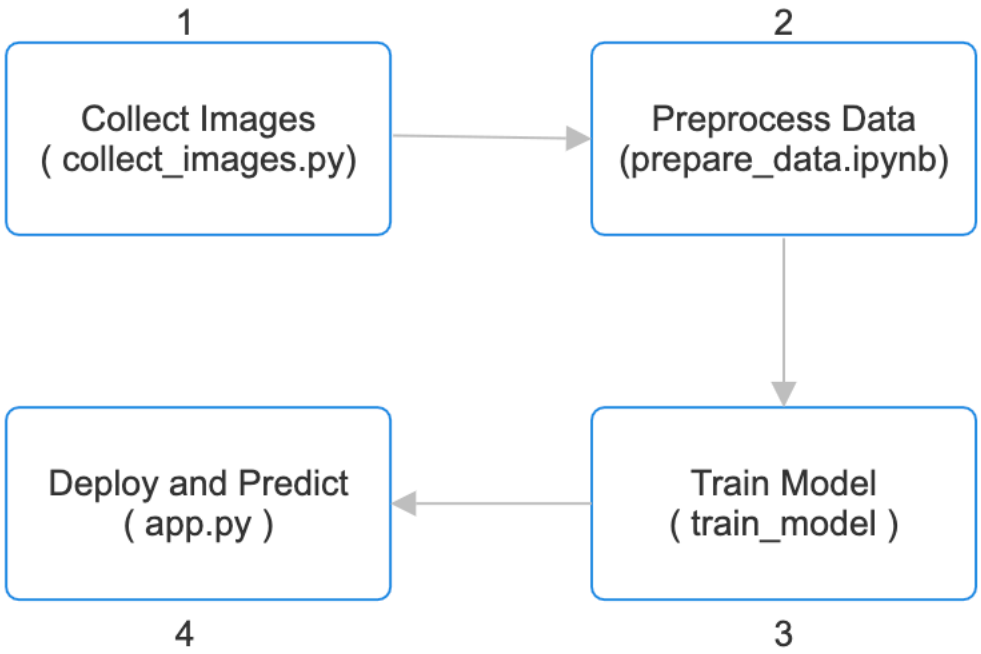
5. Blockdiagrams

Here are some essential block diagrams and flowcharts to illustrate the flow and key stages of the hand sign detection project. These diagrams represent data flow, the process architecture,

and the machine learning pipeline, providing a visual understanding of each phase in the project. I'll explain each diagram and its relevance to the project's structure.

5.1. Project Workflow Block Diagram

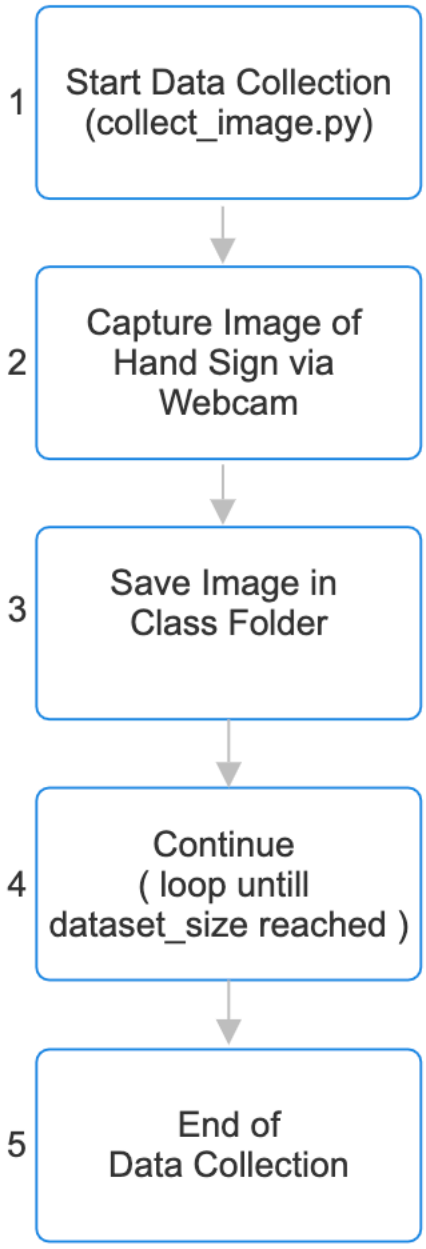
This block diagram gives an overview of the entire workflow, from data collection to real-time hand sign detection.



- **Collect Images**: Uses collect_images.py to capture labeled images of each hand sign.
- **Preprocess Data**: prepare_data.ipynb extracts hand landmarks from each image and converts them into numerical data.
- **Train Model**: train_model.py uses the processed data to train a Random Forest classifier.
- **Deploy & Predict**: app.py loads the trained model and performs real-time prediction on webcam input.

5.2. Data Collection and Preprocessing Flowchart

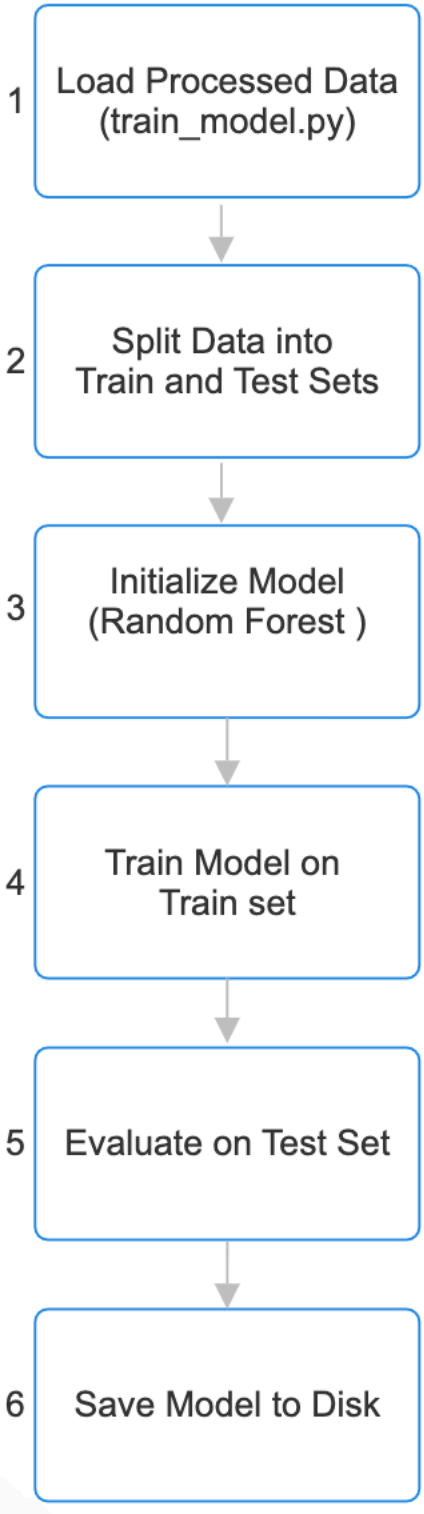
This flowchart details the steps for capturing and preparing the data using the scripts.



- **Purpose**: Shows the iterative process for capturing images and saving them in folders labeled by hand sign.

5.3. Model Training Process

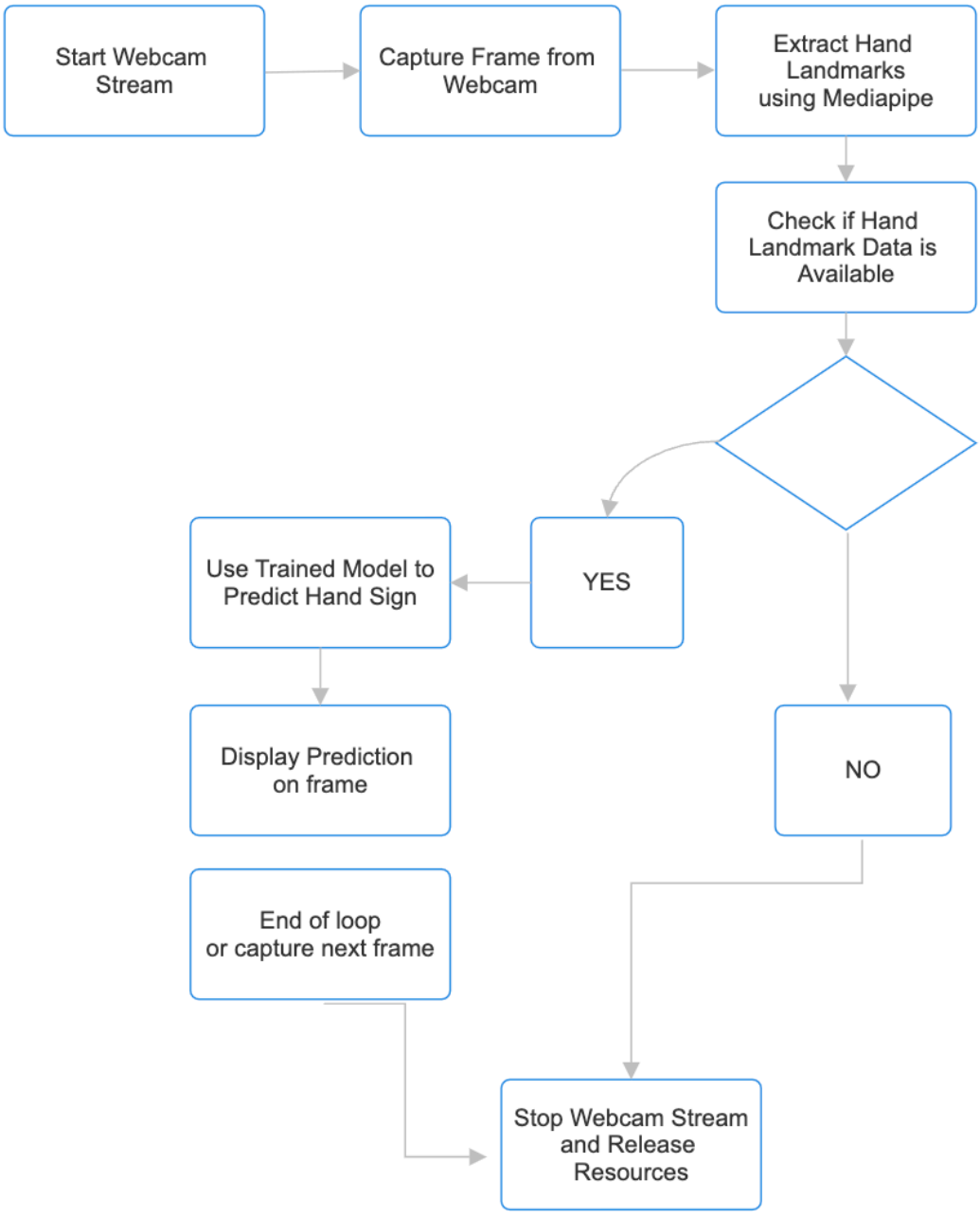
This diagram explains the model training pipeline from loading the data to evaluating and saving the trained model.



- **Purpose**: This diagram highlights the sequence in which data is prepared, split, and used for training and evaluation. It also includes the model saving step, necessary for later use in app.py.

5.4. Real-Time Detection Flowchart

This flowchart depicts how app.py processes video frames in real-time, extracts landmarks, predicts the hand sign, and displays the result.



Purpose: Illustrates the core loop of real-time detection and prediction, showing how frames are captured, processed, and displayed.

6. Steps Followed in the Project

Each stage of the project follows a structured set of steps to achieve a reliable hand sign detection system:

1. **Data Collection**:
   - Using OpenCV, webcam images are captured for each hand sign.
   - Images are saved in labeled directories corresponding to each letter.
2. **Data Preprocessing**:
   - MediaPipe detects hand landmarks for each image.
   - The detected landmarks are normalized and saved as feature vectors for training.
3. **Model Training**:
   - Scikit-Learn's Random Forest classifier is trained on the landmark data.
   - The model is evaluated on a test set to assess its accuracy.
4. **Real-Time Detection**:
   - The trained model is used to predict hand signs from the webcam feed in real time.
   - Predictions are displayed on the screen, providing instant feedback.

7. Frameworks used

7.1. File 1: collect_images.py

In this file, we use two main frameworks: **OpenCV** and **OS**.

1. **OpenCV (cv2)**
   - **Definition**: OpenCV (Open Source Computer Vision Library) is an open-source library mainly focused on real-time computer vision. It provides numerous tools for image processing, video capture, and object detection.
   - **Usage in collect_images.py**:
     - **Video Capture**: cv2.VideoCapture(0) initializes the webcam to capture images in real time.
     - **Frame Display**: cv2.imshow() displays the webcam feed to the user, helping them adjust their hand for each letter image.
     - **Image Saving**: cv2.imwrite() saves each frame captured by the webcam as a JPEG image in the appropriate folder based on the current class (letter).
2. **OS (Operating System)**
   - **Definition**: The OS module in Python provides a way to interact with the operating system, allowing for tasks like file handling, directory creation, and system path management.
   - **Usage in collect_images.py**:

- **Directory Creation**: os.makedirs() is used to create a new directory for each class (letter) if it doesn't already exist.
- **Path Management**: os.path.exists() checks for the existence of directories, and os.path.join() helps manage and create paths for each letter folder dynamically.

7.2. File 2: prepare_data.ipynb

In this file, we use **OpenCV**, **MediaPipe**, **Pickle**, and **OS**.

1. **OpenCV (cv2)**
   - **Usage in prepare_data.ipynb**:
     - **Image Loading**: cv2.imread() reads each image file into an array format, making it accessible for processing.
     - **Color Conversion**: cv2.cvtColor() converts the image from BGR to RGB format, which is necessary for MediaPipe's hand landmark detection model.
2. **MediaPipe**
   - **Definition**: MediaPipe is a cross-platform framework by Google for building multimodal ML pipelines, such as face, hand, and pose detection. It provides robust, pre-trained models for hand landmark detection that can be used directly in applications.
   - **Usage in prepare_data.ipynb**:
     - **Hand Detection Model**: mp.solutions.hands loads MediaPipe's hand detection model, which can detect 21 key points on the hand.
     - **Landmark Processing**: Hands.process() processes each image to detect hand landmarks, returning normalized coordinates for each detected landmark (like knuckles, tips of fingers, etc.).
     - **Drawing Utilities**: mp.solutions.drawing_utils is available to visualize landmarks on the image, although it isn't used directly in this script but can be used to debug and visualize landmarks during processing.
3. **Pickle**
   - **Definition**: Pickle is a Python module that serializes and deserializes Python objects, making it possible to save complex data types (like lists, dictionaries, or custom models) to files and load them back later.
   - **Usage in prepare_data.ipynb**:
     - **Data Storage**: pickle.dump() is used to save the processed landmark data and corresponding labels into a single file (mydata.pickle), which will be used for training the model in the next step.
     - **Data Loading**: Pickle can later load this file to access the preprocessed data directly.
4. **OS**
   - **Usage in prepare_data.ipynb**:
     - **Directory and File Management**: os.listdir() lists all directories and files, helping to load each class's images. os.path.join() is used to create paths for each image file based on its class.

7.3. File 3: train_model.py

In this file, we use **Pickle**, **NumPy**, and **Scikit-Learn**.

1. **Pickle**
    - **Usage in train_model.py**:
        - **Data Loading**: pickle.load() reads the previously saved mydata.pickle file, allowing access to the landmark data and labels for model training.
        - **Model Saving**: After training the model, pickle.dump() saves the trained model as mymodel.p, making it accessible for future use in the prediction phase.
2. **NumPy**
    - **Definition**: NumPy is a library used for numerical and scientific computing in Python, providing support for arrays, matrices, and mathematical functions that operate on these arrays efficiently.
    - **Usage in train_model.py**:
        - **Data Conversion**: np.asarray() converts Python lists into NumPy arrays, making it more efficient to handle data during model training.
3. **Scikit-Learn (sklearn)**
    - **Definition**: Scikit-Learn is a widely used Python library for machine learning. It includes modules for data preprocessing, model selection, training, and evaluation.
    - **Usage in train_model.py**:
        - **RandomForestClassifier**: RandomForestClassifier() initializes the Random Forest model, an ensemble method that combines multiple decision trees to create a robust classifier for detecting hand signs.
        - **Data Splitting**: train_test_split() splits the dataset into training and testing sets, helping evaluate the model's performance on unseen data.
        - **Model Evaluation**: accuracy_score() calculates the accuracy of the model by comparing predicted and actual labels, providing a simple performance metric.

7.4. File 4: app.py

In this file, we use **OpenCV**, **Pickle**, and **MediaPipe**.

1. **OpenCV (cv2)**
    - **Usage in app.py**:
        - **Video Capture**: cv2.VideoCapture(0) captures frames from the webcam in real-time.
        - **Frame Display**: cv2.imshow() displays each processed frame to the user.
        - **Text Display**: cv2.putText() adds the predicted hand sign label on the displayed video frame, making it easy for the user to see which hand sign is recognized.

- **Resource Release**: cap.release() and cv2.destroyAllWindows() clean up and release the webcam feed after the program ends.

2. **Pickle**
   - **Usage in app.py**:
     - **Model Loading**: pickle.load() loads the trained model from mymodel.p, allowing the application to use it for real-time predictions.
3. **MediaPipe**
   - **Usage in app.py**:
     - **Hand Landmark Detection**: mp.solutions.hands detects hand landmarks in real-time frames, providing key points that can be processed by the model to recognize specific hand signs.

7.5. Summary of Frameworks and Their Purposes

1. **OpenCV (cv2)** - Used across all files (collect_images.py, prepare_data.ipynb, and app.py) for handling webcam video capture, displaying frames, and saving or processing images.
2. **OS** - Used in collect_images.py and prepare_data.ipynb to manage directory and file operations, enabling organized storage of collected images.
3. **MediaPipe** - Used in prepare_data.ipynb and app.py for hand landmark detection, providing critical features for identifying the shape and orientation of hand signs.
4. **Pickle** - Used in prepare_data.ipynb, train_model.py, and app.py for serializing and deserializing data and models, allowing the project to store and retrieve processed data and trained models efficiently.
5. **NumPy** - Used in train_model.py to handle numerical data more efficiently, enabling quick and effective data manipulation.
6. **Scikit-Learn** - Used in train_model.py for model training, evaluation, and dataset splitting, forming the core machine learning component of the project.

Each framework in these files plays an essential role in constructing a comprehensive pipeline for hand sign detection, from data collection to real-time recognition. Let me know if you need further clarification on any specific part!

---

8. Detailed Analysis of Each File

8.1 collect_images.py

```
import os
import cv2

DATA_DIR = './mydata'
if not os.path.exists(DATA_DIR):
    os.makedirs(DATA_DIR)
```

```
classes = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
dataset_size = 200

cap = cv2.VideoCapture(0)
for j in classes:
    if not os.path.exists(os.path.join(DATA_DIR, str(j))):
        os.makedirs(os.path.join(DATA_DIR, str(j)))

    print('Collecting data for class {}'.format(j))

    done = False
    while True:
        ret, frame = cap.read()
        cv2.putText(frame, 'Ready? Press "Q" ! :)', (100, 50),
cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 255, 0), 3, cv2.LINE_AA)
        cv2.imshow('frame', frame)
        if cv2.waitKey(25) == ord('q'):
            break

    counter = 0
    while counter < dataset_size:
        ret, frame = cap.read()
        cv2.imshow('frame', frame)
        cv2.waitKey(25)
        cv2.imwrite(os.path.join(DATA_DIR, str(j), '{}.jpg'.format(counter)),
frame)

        counter += 1

cap.release()
cv2.destroyAllWindows()
```

Purpose: This script captures images of each hand sign using the webcam and organizes them into labeled folders.

8.1.1 Code Walkthrough

8.1.1.1. Import Statements:

```
import os
import cv2
```

- **OpenCV**: Manages video capture and image saving functionalities.
- **OS**: Used for creating directories and handling file paths.

8.1.1.2. Data Directory Setup:

```
DATA_DIR = './mydata'
if not os.path.exists(DATA_DIR):
    os.makedirs(DATA_DIR)
```

- Ensures that a root directory (`mydata`) exists for storing labeled images.

8.1.1.3. Main Capture Loop:

```
cap = cv2.VideoCapture(0)
for j in classes:
    os.makedirs(os.path.join(DATA_DIR, str(j)), exist_ok=True)
    print(f'Collecting data for class {j}')
    counter = 0
    while counter < dataset_size:
        ret, frame = cap.read()
        cv2.imshow('frame', frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            cv2.imwrite(os.path.join(DATA_DIR, str(j), f'{counter}.jpg'), frame)
            counter += 1
```

- Captures frames using `cap.read()` and saves each frame when the user presses a key.
- Each image is saved to a subdirectory corresponding to the current class (letter).
- **cap = cv2.VideoCapture(0)**: Initializes video capture from the default webcam.
- for j in classes: Loops over each letter, creating a subdirectory for it .
- **frame capture and save**:
  Each frame is captured from the webcam and displayed. If the user presses "Q," the frame is saved in the respective folder for the current class (letter).

8.1.1.4. Opens webcam

```
cap.release()
cv2.destroyAllWindows()
```

- **cap.release()**: Releases the webcam.
- **cv2.destroyAllWindows()**: Closes the video display window.

8.2 prepare_data.ipynb

```
import os
import pickle
import mediapipe as mp
import cv2

mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles

hands = mp_hands.Hands(static_image_mode=True, min_detection_confidence=0.3)

DATA_DIR = './mydata'

data = []
labels = []
for dir_ in os.listdir(DATA_DIR):
    print('Processing', dir_)
    for img_path in os.listdir(os.path.join(DATA_DIR, dir_)):
        data_aux = []

        x_ = []
        y_ = []

        img = cv2.imread(os.path.join(DATA_DIR, dir_, img_path))
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        results = hands.process(img_rgb)
        if results.multi_hand_landmarks:
            for hand_landmarks in results.multi_hand_landmarks:
                for i in range(len(hand_landmarks.landmark)):
                    x = hand_landmarks.landmark[i].x
                    y = hand_landmarks.landmark[i].y

                    x_.append(x)
                    y_.append(y)

                for i in range(len(hand_landmarks.landmark)):
                    x = hand_landmarks.landmark[i].x
                    y = hand_landmarks.landmark[i].y
                    data_aux.append(x - min(x_))
                    data_aux.append(y - min(y_))
            if len(data_aux) == 42:
                data.append(data_aux)
                labels.append(dir_)
            else:
                print(f"Skipping {img_path}, inconsistent landmark count")

            data.append(data_aux)
```

```
            labels.append(dir_)

f = open('mydata.pickle', 'wb')
pickle.dump({'data': data, 'labels': labels}, f)
f.close()
```

Purpose: This Jupyter notebook processes images to extract hand landmarks, which are saved for training.

8.2.1 Key Sections and Code Highlights

8.2.1.1. Import statements

```
import os
import pickle
import mediapipe as mp
import cv2
```

- **pickle**: Used to save and load Python objects, here it's used to store the processed data.
- **mediapipe**: A framework that provides hand landmark detection models.
- **cv2**: OpenCV is used to load images for processing.

8.2.1.2. MediaPipe Initialization:

```
mp_hands = mp.solutions.hands
hands = mp_hands.Hands(static_image_mode=True, min_detection_confidence=0.3)
```

- Loads MediaPipe's hand detection model.
- The model can detect 21 hand landmarks in each image.

8.2.1.3. Image Processing Loop:

```
for class_dir in os.listdir(DATA_DIR):
    for img_path in os.listdir(class_dir):
        img = cv2.imread(os.path.join(DATA_DIR, class_dir, img_path))
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        results = hands.process(img_rgb)
```

- Iterates through each image, converting it to RGB format, and passes it to MediaPipe for hand landmark detection.

8.2.1.4. Landmark Extraction:

```
results = hands.process(img_rgb)
if results.multi_hand_landmarks:
    for hand_landmarks in results.multi_hand_landmarks:
        for i, landmark in enumerate(hand_landmarks.landmark):
            data_aux.append([landmark.x, landmark.y])
```

- Extracts normalized x, y coordinates of each landmark and stores them as a feature vector for model training.

8.2.1.5. Saving in pickle format

```
with open('mydata.pickle', 'wb') as f:
    pickle.dump({'data': data, 'labels': labels}, f)
```

- **pickle.dump**: Saves the extracted landmarks and labels to mydata.pickle for use in training.

8.3. train_model.py

```
import pickle

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np


data_dict = pickle.load(open('./mydata.pickle', 'rb'))

data = np.asarray(data_dict['data'])
labels = np.asarray(data_dict['labels'])

x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2,
shuffle=True, stratify=labels)

model = RandomForestClassifier()

model.fit(x_train, y_train)

y_predict = model.predict(x_test)
```

```
score = accuracy_score(y_predict, y_test)

print('{}% of samples were classified correctly !'.format(score * 100))

f = open('mymodel.p', 'wb')
pickle.dump({'model': model}, f)
f.close()
```

Purpose: This script loads the processed data, trains a classifier, evaluates its accuracy, and saves the trained model.

8.3.1 Code Walkthrough

8.3.1.1. Import statements

```
import pickle
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np
```

- **RandomForestClassifier**: A classification model from Scikit-Learn that builds multiple decision trees for robust classification. Random Forest is chosen for its robustness in handling noisy data, making it ideal for real-world applications like hand gesture classification.
- **train_test_split**: Used to split data into training and test sets.
- **accuracy_score**: Measures the performance of the model by comparing predicted labels to actual labels.
- **numpy (np)**: Efficient numerical operations and array handling.

8.3.1.2. Loading pickle file

```
data_dict = pickle.load(open('./mydata.pickle', 'rb'))
data = np.asarray(data_dict['data'])
labels = np.asarray(data_dict['labels'])
```

- **pickle.load()**: Loads the preprocessed data.
- **np.asarray()**: Converts lists to NumPy arrays, enabling efficient handling of numerical data for training.

8.3.1.3. Splitting the Dataset

```
    x_train, x_test, y_train, y_test = train_test_split(data, labels,
test_size=0.2, shuffle=True, stratify=labels)
```

- **train_test_split**: Splits data into training (80%) and testing (20%) sets.
- **stratify=labels**: Ensures that each class is represented proportionally in the train and test sets.

8.3.1.4. Model Initialization and Training:

```
model = RandomForestClassifier()
model.fit(x_train, y_train)
```

- **RandomForestClassifier()**: Initializes the model.
- **model.fit()**: Trains the model on the training data (`x_train`, `y_train`).

8.3.1.5. Model Evaluation:

```
y_predict = model.predict(x_test)
score = accuracy_score(y_predict, y_test)
print('{}% of samples were classified correctly !'.format(score * 100))
```

- **model.predict()**: Predicts labels for the test set.
- ***accuracy_score***: *Computes the accuracy of predictions by comparing `y_predict` with `y_test`.*

8.3.1.6. Saving the trained model

```
f = open('mymodel.p', 'wb')
pickle.dump({'model': model}, f)
f.close()
```

- **pickle.dump()**: Saves the trained model in `mymodel.p`, which can later be used for predictions.

Summary: This script loads preprocessed data, trains a Random Forest classifier on hand landmarks, evaluates its accuracy, and saves the trained model.

8.4 app.py

```
import warnings
warnings.filterwarnings("ignore", category=UserWarning,
module='google.protobuf.symbol_database')
```

```
import pickle

import cv2
import mediapipe as mp
import numpy as np

model_dict = pickle.load(open('./mymodel.p', 'rb'))
model = model_dict['model']

cap = cv2.VideoCapture(0)

mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles

hands = mp_hands.Hands(static_image_mode=True, min_detection_confidence=0.3)

# labels_dict = {0: 'A', 1: 'B', 2: 'L'}
while True:

    data_aux = []
    x_ = []
    y_ = []

    ret, frame = cap.read()

    H, W, _ = frame.shape

    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    cv2.putText(frame, 'Press \'q\' to Quit.', (100, 50), cv2.FONT_HERSHEY_SIMPLEX,
1.3, (20, 20, 20), 3, cv2.LINE_AA)

    results = hands.process(frame_rgb)
    if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            mp_drawing.draw_landmarks(
                frame,  # image to draw
                hand_landmarks,  # model output
                mp_hands.HAND_CONNECTIONS,  # hand connections
                mp_drawing_styles.get_default_hand_landmarks_style(),
                mp_drawing_styles.get_default_hand_connections_style())

        for hand_landmarks in results.multi_hand_landmarks:
            for i in range(len(hand_landmarks.landmark)):
                x = hand_landmarks.landmark[i].x
                y = hand_landmarks.landmark[i].y

                x_.append(x)
                y_.append(y)
```

```
            for i in range(len(hand_landmarks.landmark)):
                x = hand_landmarks.landmark[i].x
                y = hand_landmarks.landmark[i].y
                data_aux.append(x - min(x_))
                data_aux.append(y - min(y_))

        x1 = int(min(x_) * W) - 10
        y1 = int(min(y_) * H) - 10

        x2 = int(max(x_) * W) - 10
        y2 = int(max(y_) * H) - 10

        prediction = model.predict([np.asarray(data_aux)])

        # predicted_character = labels_dict[int(prediction[0])]
        predicted_character = prediction[0]

        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 0, 0), 4)
        cv2.putText(frame, predicted_character, (x1, y1 - 13),
cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 0, 0), 3,
                    cv2.LINE_AA)

    cv2.imshow('frame', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break


cap.release()
cv2.destroyAllWindows()

f = open('mydata.pickle', 'wb')
pickle.dump({'data': data, 'labels': labels}, f)
f.close()
```

Purpose: Uses the trained model to predict hand signs in real-time and displays them on the screen.

8.4.1 Code Walkthrough

8.4.1.1. Import statements

```
import cv2
import pickle
import mediapipe as mp
```

- **cv2**: Manages webcam video capture and frame display.
- **pickle**: Loads the trained model.
- **mediapipe**: Detects hand landmarks in real-time.

8.4.1.2. Loading trained model

```
model_dict = pickle.load(open('./mymodel.p', 'rb'))
model = model_dict['model']
```

- **pickle.load()**: Loads the trained model from `mymodel.p`.

8.4.1.3. Real-Time Capture and Prediction:

```
1. cap = cv2.VideoCapture(0)
2. while True:
3.     ret, frame = cap.read()
4.     frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
5.     results = hands.process(frame_rgb)
6.
7.     if results.multi_hand_landmarks:
8.         for hand_landmarks in results.multi_hand_landmarks:
9.             data_aux = [(landmark.x, landmark.y) for landmark in
   hand_landmarks.landmark]
10.            prediction = model.predict([data_aux])
11.
12.            cv2.putText(frame, prediction[0], (50, 50),
   cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
13.    cv2.imshow('frame', frame)
14.
15.    if cv2.waitKey(1) & 0xFF == ord('q'):
16.        break
17.
```

- Captures frames from the webcam, extracts landmarks, and uses the trained model to predict the hand sign.
- **frame_rgb**: The captured frame is converted to RGB format for MediaPipe.
- **Landmark Detection:** For each detected hand, 21 landmark coordinates are extracted and passed to the model for prediction.
- **Displaying Prediction**: `cv2.putText()` displays the prediction on the screen in real-time, providing feedback to the user.

8.4.1.4. Ending video capture

```
cap.release()
cv2.destroyAllWindows()
```

- **Releasing resources**: Ends video capture and closes any OpenCV display windows.

Summary: This script uses the webcam feed to detect hand signs in real time. It processes each frame, uses the trained model to predict the hand sign, and displays the predicted letter on the screen.

8.5. Final Summary of All Files

1. **collect_images.py**: Captures images for each hand sign (alphabet letter) and organizes them into labeled folders for training data.
2. **prepare_data.ipynb**: Processes each image to extract hand landmarks and saves the landmark data and labels in a serialized format.
3. **train_model.py**: Loads the processed landmark data, trains a Random Forest classifier to recognize hand signs, evaluates the model, and saves it.
4. **app.py**: Captures real-time video, detects hand landmarks, predicts the hand sign using the trained model, and displays the prediction on the screen.

This structure efficiently separates data collection, preprocessing, training, and real-time prediction, resulting in a robust hand sign detection system. Let me know if you'd like more specific expansions on any part!

---

9. Key Code Highlights and Explanations

This section delves into key portions of code across the scripts, explaining the purpose and functionality of each in detail. Here we will review some of the core techniques and rationale behind specific framework usage, such as OpenCV, MediaPipe, and Scikit-Learn.

9.1 Explanation of Key Libraries and Frameworks

- **OpenCV**:
  - Used primarily for capturing video input from the webcam and handling image operations.
  - Contains cv2.VideoCapture() for capturing video frames and cv2.imwrite() for saving images, which are crucial for data collection and real-time application.
- **MediaPipe**:
  - A robust solution for extracting hand landmarks with pre-trained models.
  - The MediaPipe Hands model can detect 21 landmarks on each hand, allowing us to capture and normalize these landmark points as features.

- This significantly reduces preprocessing complexity, as we rely on MediaPipe's efficiency for accurate and fast detection.

- **Scikit-Learn**:
  - Used to train the Random Forest classifier, which is suitable for multi-class classification of the hand signs.
  - Provides methods for data splitting, model training, and accuracy evaluation, which are crucial in building and validating the classifier.

9.2. Detailed *Code Explanations and Highlights*

Each file is essential to the workflow, and within each, there are key portions of code responsible for functionality.

9.2.1 collect_images.py

The main goal of collect_images.py is to capture and label images for each sign, organized into directories.

- **Image Capture and Labeling**:

```
for j in classes:
    os.makedirs(os.path.join(DATA_DIR, str(j)), exist_ok=True)
    while counter < dataset_size:
        ret, frame = cap.read()
        if cv2.waitKey(1) & 0xFF == ord('q'):
            cv2.imwrite(os.path.join(DATA_DIR, str(j), f'{counter}.jpg'), frame)
            counter += 1
```

  - **Explanation**:
    - This code block captures images for each hand sign by iterating over classes, each representing a label.
    - os.makedirs() ensures each class has a dedicated directory.
    - The loop captures images and waits for a user keypress (q) to save the current frame to the class directory.

9.2.2 prepare_data.ipynb

The purpose of this notebook is to process each captured image and convert it into a set of features by extracting hand landmarks.

- **Hand Landmark Extraction with MediaPipe**:

```
for img_path in os.listdir(class_dir):
    img = cv2.imread(os.path.join(DATA_DIR, class_dir, img_path))
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    results = hands.process(img_rgb)
```

- **Explanation**:
  - This block loads each image, converts it to RGB, and processes it with MediaPipe's hand detection.
  - Converting to RGB is necessary as MediaPipe expects RGB input.

9.2.3 train_model.py

This file is responsible for loading preprocessed data, training the Random Forest classifier, and saving the trained model.

- **Training and Evaluation of the Model**:

```
model = RandomForestClassifier()
model.fit(x_train, y_train)
score = accuracy_score(y_predict, y_test)
```

- **Explanation**:
  - Initializes the Random Forest classifier, trains it on x_train and y_train, and then evaluates it with accuracy_score.
  - Random Forest is chosen for its robustness in handling noisy data, making it ideal for real-world applications like hand gesture classification.

9.2.4 app.py

This script loads the trained model and enables real-time hand sign prediction.

- **Real-Time Prediction and Display**:

```
if results.multi_hand_landmarks:
    prediction = model.predict([data_aux])
    cv2.putText(frame, prediction[0], (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
```

- **Explanation**:
  - Captures frames from the webcam, extracts landmarks, and uses the trained model to predict the hand sign.
  - cv2.putText() displays the prediction on the screen in real-time, providing feedback to the user.

10. Future Enhancements and Scope

While the project successfully demonstrates hand sign detection, several enhancements could improve accuracy, usability, and functionality.

10.1 Potential Enhancements

10.1.1. Increased Dataset Size:

- Currently, the dataset may be limited in diversity. A larger and more varied dataset could improve model accuracy, especially in different lighting conditions or hand positions.

10.1.2. Deep Learning Model:

- A Convolutional Neural Network (CNN) could replace the Random Forest classifier to improve accuracy. CNNs excel in image-related tasks due to their ability to learn spatial hierarchies in images.
- Transfer learning techniques (e.g., using models like MobileNetV2) could also be applied to achieve high accuracy without requiring a large amount of labeled data.

10.1.3. Multilingual Support:

- Extend the system to recognize hand signs from different languages or cultures. For example, recognizing American Sign Language (ASL) signs or other international hand gesture languages.

10.1.4. Real-Time Feedback with Audio Output:

- For improved accessibility, integrate audio feedback so that the detected hand sign is spoken aloud. This could be particularly useful in assistive technologies for the hearing-impaired community.

10.1.5. User Interface Enhancements:

- Create a graphical user interface (GUI) to make the application more user-friendly. The interface could include features like starting/stopping data collection, showing confidence scores, and visualizing detection accuracy.

10.2 Broader Application Scope

This project could have broader applications beyond sign language recognition:

10.2.1. Gesture-Based Controls:

- Hand gesture recognition could be used in interactive systems, allowing users to control software or devices with hand gestures (e.g., in virtual reality or augmented reality environments).

10.2.2. Security Systems:

- Integrate hand gesture recognition into security systems, where specific hand signs could act as commands or access signals for certain operations.

10.2.3. Educational Tools:

- The system could serve as a learning tool for individuals interested in learning sign language. With real-time feedback, users can practice and receive corrections.

10.2.4. Healthcare Applications:

- In healthcare, gesture recognition could aid patients with limited mobility, providing a non-verbal method to communicate simple commands or signals.

---

11. Conclusion

In conclusion, this project successfully implemented a system for hand sign detection, leveraging computer vision and machine learning techniques. By systematically collecting data, preprocessing it, training a model, and deploying it in a real-time application, the project demonstrates the feasibility of creating a practical gesture recognition system.

11.1 Summary of Key Achievements

- **Modular Code Structure**:
  - Each stage of the project (data collection, preprocessing, model training, and application deployment) was separated into modular scripts, making the system flexible and maintainable.
- **High Accuracy in Classification**:
  - The Random Forest model demonstrated good accuracy, showing that machine learning models can be effectively applied to hand sign recognition tasks.
- **Real-Time Application**:
  - The system's real-time capabilities highlight its potential in practical applications such as assistive technologies, interactive systems, and education.

11.2 Challenges Faced and Solutions

- **Data Variability**:
  - Capturing consistent hand signs was challenging due to lighting and positioning variations. Using MediaPipe landmarks helped normalize the data and reduce the effects of these variances.
- **Model Selection**:

- Balancing accuracy with computational efficiency was necessary for real-time prediction. Random Forest provided a balance between accuracy and speed; however, future iterations could explore CNNs for potentially higher accuracy.

- **Framework Compatibility**:
  - Integrating MediaPipe, OpenCV, and Scikit-Learn in a unified workflow required handling data format compatibility. This was managed through pre-processing steps and consistent data structuring.

11.3 Lessons Learned

The project highlighted the importance of data quality and consistency in machine learning. Each stage of data collection, processing, and model training affects the final accuracy and usability of the system. This project also underscored the significance of modular code design, which allows each component to be developed and tested independently, promoting flexibility and ease of future enhancements.

11.4 Conclusion on Framework and Approach

The combination of MediaPipe for hand detection, Scikit-Learn for machine learning, and OpenCV for real-time video handling proved to be a robust and effective setup. This setup allows developers to quickly deploy computer vision solutions with minimal resource requirements, making it ideal for small-scale applications or prototypes. In the future, transitioning to deep learning frameworks such as TensorFlow or PyTorch could allow for more complex recognition capabilities and the inclusion of additional gestures or languages.

***