Real-Time Hand Gesture Recognition Using OpenCV and Machine Learning Techniques

Project Report submitted for the partial fulfilment of requirement for the degree of **Bachelor of Technology (B. Tech)** in **Computer Science and Engineering** of the

University of Calcutta

Submitted by

Abhishek Raj
B.Tech. 8th Semester
Roll No: T91/CSE/204030
Registration No: D01-1111-170-17

Md. Shalique
B.Tech. 8th Semester
Roll No: T91/CSE/204039
Registration No: D01-1111-0059-20

Raushan Kumar
B.Tech. 8th Semester
Roll No: T91/EE/204093
Registration No: D01-1111-0112-20

Under the guidance of

Prof. Banani Saha

Department of Computer Science and Engineering
University of Calcutta

JD-2, Sector-III, Salt Lake City, Kolkata 700106

June, 2024

Department of Computer Science and Engineering

University of Calcutta

JD-2, Sector-III, Salt Lake, Kolkata-700106

CERTIFICATE

This is to certify that **Abhishek Raj, Md. Shalique,** and **Raushan Kumar**, students of the *8th* semester of Bachelor of Technology (B. Tech) in Computer Science and Engineering in the Department of *Computer Science and Engineering*, University of Calcutta, have completed their project report entitled "**Real-Time Hand Gesture Recognition Using OpenCV and Machine Learning Techniques**" under my supervision and guidance, for the partial fulfilment of their degree of Bachelor of Technology in Computer Science and Engineering at the University of Calcutta.

Chairman
Board of Studies (B.Tech.)
Computer Science and
Engineering University of Calcutta
Place: Kolkata, West Bengal, India
Date:

External Examiner

Prof. Banani Saha
Department of Computer Science and Engineering
University of Calcutta
Place: Kolkata, West Bengal, India
Date:

Acknowledgement

We are pleased to express our gratitude to all those who have extended their support and encouragement to help us carry out our project work. We express our reverence and gratitude to our project guide Prof. Banani Saha, Professor of the Department of Computer Science and Engineering, University of Calcutta, for her meticulous guidance, congenial discussion, constructive criticism, and constant encouragement throughout the project. We would also like to acknowledge the researchers and scholars who have significantly contributed to the “Computer Vision” and “Sign Language” fields. Their pioneering work and published papers served as a valuable foundation for our project, and we are grateful for their dedication to advancing knowledge in this domain.

Abhishek Raj	**Md. Shalique**	**Raushan Kumar**
B.Tech. 8th Semester	B.Tech. 8th Semester	B.Tech. 8th Semester
Roll No: T91/CSE/204030	Roll No: T91/CSE/204039	Roll No: T91/EE/204093
Registration No: D01-1111-170-17	Registration No: D01-1111-0059-20	Registration No: D01-1111-0112-20

Abstract

This project addresses the communication challenges faced by the deaf and mute community by developing a real-time hand gesture recognition system utilizing computer vision and machine learning techniques. Leveraging OpenCV and the MediaPipe Hands solution, the system captures and processes real-time video frames to detect and translate sign language gestures into text. The project encompasses several phases: data collection, where real-time video frames of hand gestures are captured; data extraction, involving palm detection and hand landmark modeling to identify specific 2D coordinates of the hand; model training, where various machine learning models are evaluated for their accuracy and efficiency in gesture recognition; and real-time detection, deploying the trained model to translate gestures instantaneously. This innovative approach not only enhances communication for the deaf and mute but also contributes to the advancement of assistive technologies by employing state-of-the-art machine learning models. The project demonstrates significant social impact by promoting inclusivity and accessibility.

Introduction

This project addresses a significant communication barrier faced by the deaf and mute community by developing a real-time hand gesture recognition system using computer vision and machine learning techniques. The World Health Organization estimates that 360 million people worldwide have some form of hearing disability, highlighting the urgent need for effective communication tools.

Objective: To create a system that accurately detects and translates sign language gestures into text, thereby facilitating seamless communication for the deaf and mute population.

Phases of the Project:

- Raw Data Collection: Utilizing the OpenCV library, we capture real-time video frames to gather data on hand gestures. This phase ensures the accumulation of diverse and extensive datasets necessary for model training.
- Data Extraction: Implementing a two-step process involving a palm detector and a hand landmark model. The palm detector identifies the hand's position, while the hand landmark model pinpoints specific 2D coordinates of the hand. This phase is crucial for extracting high-fidelity hand landmarks required for precise gesture recognition.
- Model Training: The extracted 2D landmarks are labeled and used to train various machine learning classification models. The performance of these models is evaluated to identify the most accurate and efficient one. This comparative analysis ensures that the best model is selected for real-time application.
- Real-Time Detection: The chosen model from the training phase is deployed for real-time detection. The system processes live video frames, transforming them through the same data extraction methods before feeding them into the trained model for immediate gesture translation.

Significance:

Innovation: This project employs advanced computer vision and machine learning techniques to develop a robust and real-time gesture recognition system.

Social Impact: By bridging the communication gap for the deaf and mute community, this system promotes inclusivity and accessibility.

Technological Advancement: The project leverages OpenCV and state-of-the-art machine learning models, contributing to the growing field of assistive technologies.

Review

Sign Language Recognition is a breakthrough for communication among deaf-mute society and has been a critical research topic for years. Although some of the previous studies have successfully recognized sign language, it requires many costly instruments including sensors, devices, and high-end processing power. However, such drawbacks can be easily overcome by employing artificial intelligence-based techniques. (Refat et al. [2])

The type of data that we have followed for this Literature Review are only camera mode and static signs (both single and double handed signs). Techniques used and average accuracy are shown in the tables. Based on these parameters, the review for two important sign languages: American Sign Language (ASL) and Indian Sign Language (ISL) have been analysed and documented respectively. Most of the data used here are taken from Ankita and Prateek [3].

There has been tracking of skin color blobs corresponding to the hands into a body–facial space centered on the user's face to extract gestures from a succession of video images. Differentiating between deictic and symbolic movements serves as the goal. The image is filtered using an indexing table with quick lookups. with similar skin tones are clustered into blobs. Based on the position (x, y) and colorimetry (Y, U, V) of skin color pixels, blobs are statistical objects that help identify homogeneous areas. Various sign languages such as Arabic, and Indian also exist. Researchers have tried to build models to recognize them and correctly classify them. Saleh, Yaser, et al. [4] have tried to recognize Arabic sign language by fine-tuning deep neural networks.

Survey of the Existing Models/Work:

Author	Single/double handed	Technique Used	Recognition Rate (Accuracy)	Limitations
American Sign Language:				
Ragab et al. [5] (2013)	Single	SVM, random forest	94%	Limited characters used
Kumar et al. [6] (2016)	Single	SVM	93%	Skin-color may be misleading
Saha et al. [7] (2016)	Single	Madaline neural network	90%	Suitable for linear data only
Karayilan and Kiliç [8] (2017)	Single	NN	85% (histogram features)	Matrix-based approach leads to overfitting
Oyedotun and Khashman [9] (2017)	Single	CNN	91.33%	Limited symbols used
Refat et al. [1] (2023)	Single	Multi-headed CNN	92%	Large number of images required

Indian Sign Language:				
Agrawal et al. [10] (2012)	Double	Multi-class SVM	93%	Only 2 hand signs are used
Yasir et al. [11] (2015)	Double	SVM	86%	Dataset is small
Rao and Kishore [12] (2017)	Single	ANN	90.00%	Coloured gloves are required

Table 1: ASL and ISL works with their achievements

Summary/Gaps identified in the Survey

1. Most of the work have been done on single language and are only focused on data-based accuracy. Very few focus on real time accuracy or language portability.
2. Existing datasets often focus on single sign languages, are small, and lack diversity in signer demographics and environmental conditions. This biases models towards specific scenarios and limits generalizability. [4]
3. Biases in datasets and models can perpetuate inequalities. Research on fair and inclusive sign language technologies is crucial.
4. Collecting large, diverse datasets is challenging due to accessibility and privacy concerns of the deaf community

References:

1. Razieh Rastgoo, Kourosh Kiani, Sergio Escalera, (2021) Sign Language Recognition: A Deep Survey, ScienceDirect
2. Refat Khan Pathan, Munmun Biswas, Suraiya Yasmin, Mayeen Uddin Khandaker, Mohammad Salman & Ahmed A. F. Youssef (2023) Sign language recognition using the fusion of image and hand landmarks through multi-headed convolutional neural network, Scientific Reports
3. Ankita Wadhawan, Prateek Kumar (2019) Sign Language Recognition Systems: A Decade Systematic Literature Review, Archives of Computational Methods in Engineering
4. Saleh, Yaser and Ghassan F. Issa. (2020) "Arabic Sign Language Recognition through Deep Neural Networks Fine-Tuning." Int. J. Online Biomed. Eng. 16: 71-83 rg, pp 143–151
5. Ragab A, Ahmed M, Chau SC (2013) Sign language recognition using Hilbert curve features. In: International conference image analysis and recognition. Springer, Berlin, Heidelbe
6. Kumar A, Thankachan K, Dominic MM (2016) Sign language recognition. In: 3rd IEEE international conference on recent advances in information technology (RAIT), pp 422–428
7. Saha S, Lahiri R, Konar A, Nagar AK (2016) A novel approach to American Sign Language recognition using Madaline neural network. In: IEEE symposium series on computational intelligence (SSCI), pp 1–6
8. Karayılan T, Kılıç Ö (2017) Sign language recognition. In: IEEE international conference on computer science and engineering (UBMK), pp 1122–1126
9. Oyedotun OK, Khashman A (2017) Deep learning in vision based static hand gesture recognition. Neural Comput Appl 28(12):3941–3951
10. Agrawal SC, Jalal AS, Bhatnagar C (2012) Recognition of Indian Sign Language using feature fusion. In: 4th IEEE international conference on intelligent human computer interaction (IHCI), pp 1–5
11. Yasir F, Prasad PC, Alsadoon A, Elchouemi A (2015) Sift based approach on Bangla Sign Language recognition. In: IEEE 8th international workshop on computational intelligence and applications (IWCIA), pp 35–39
12. Rao GA, Kishore PVV (2017) Selfe video based continuous Indian Sign Language recognition system. Ain Shams Eng J 9(4):1929–1939.

https://doi.org/10.1016/j.asej.2016.10.013

13. Vagdevi Kommineni, How to use transfer learning for sign language recognition (freecodecamp.org)

Chapter 1: Data Collection

The data collection phase is crucial as it forms the foundation for training accurate and robust machine learning models. We utilize the python library OpenCV for capturing and processing real-time video frames to gather the necessary data. We utilised the VideoCapture() class for our purpose.

1.1 Introduction to OpenCV

OpenCV (Open-Source Computer Vision Library) is an open-source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.

The library has more than 2500 optimized algorithms, which include a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, and much more.

1.2 Architecture of OpenCV

OpenCV is written in C++ and has a templated interface that works seamlessly with STL containers. The primary components of OpenCV architecture are:

- **Core Functionality**: Basic structures such as the multidimensional array Mat and basic operations like arithmetic, logic, and linear algebra operations.
- **Image Processing**: Image transformation techniques like filtering, geometrical transformations, and color space conversions.
- **Video Analysis**: Techniques to capture, record, and analyze video frames.
- **Camera Calibration and 3D Reconstruction**: Functions to calibrate camera systems and reconstruct 3D structures.
- **Machine Learning**: Algorithms for training models such as support vector machines (SVM), neural networks, and boosting.

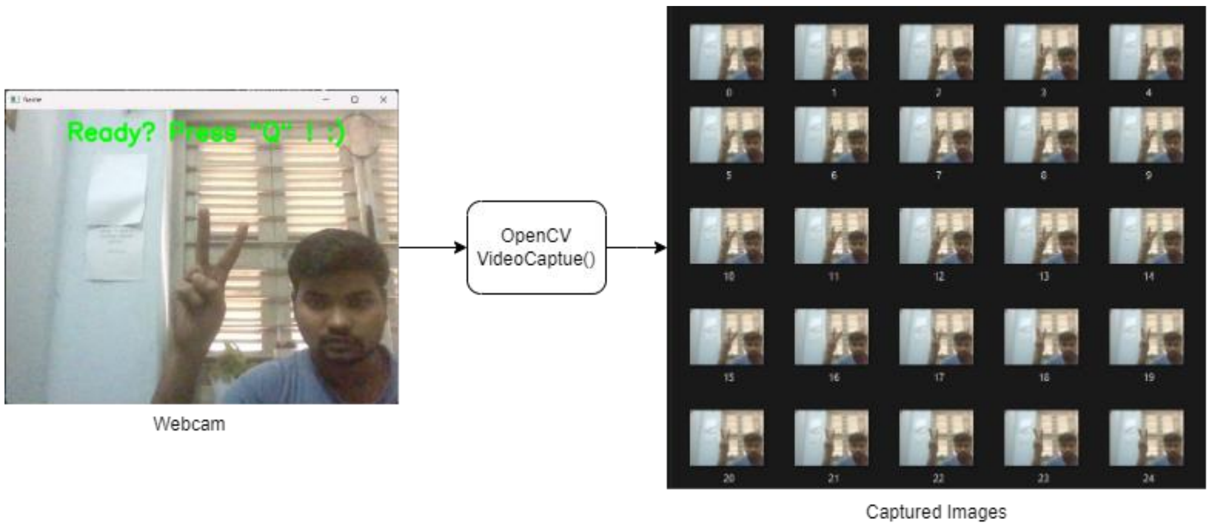


Figure 1.1: Capturing images through webcam

1.3 Implementation

The data collection process involves capturing real-time video frames using a webcam and storing them as image files. The collected images are categorized into different classes representing various hand gestures.

The cv2.VideoCapture is a class in the OpenCV library that allows us to capture video from different sources and perform various tasks on the video such as reading, processing, and writing on video files.[1]

The read() method was used to capture image with wait key set to 25 FPS. And the imwrite() method was used to save the captured images to the class folder.

References:

1. OpenCV: cv Namespace Reference: https://docs.opencv.org/3.4/d2/d75/namespacecv.html

Chapter 2: Data Preprocessing

In this project data preprocessing is a crucial step that transforms raw image data into a structured format suitable for machine learning models. This chapter details the preprocessing steps, utilizing the MediaPipe Hands solution, to extract hand landmarks from images and store them in a pickle file for further use.

2.1 Introduction

Data preprocessing involves several steps to ensure that the data is clean, consistent, and ready for model training. In this project, we use the MediaPipe Hands solution, a real-time hand tracking library developed by Google Research, to detect and process hand landmarks from images. This library provides robust hand tracking and landmark detection capabilities, which are essential for accurately capturing hand gestures.

2.2 Architecture of MediaPipe Hands

The MediaPipe Hands solution comprises two main components:

1. **Palm Detector**: This model detects the palm and provides a bounding box around it.
2. **Hand Landmark Model**: This model takes the cropped image of the palm and predicts 21 hand landmarks with 2.5D coordinates.

This architecture allows for efficient and accurate hand tracking by reducing the need for extensive data augmentation and enabling real-time performance on various devices, including mobile phones.

2.2.1. Palm Detector:

To detect where hands are in an image, we use a special model designed for fast, real-time applications on mobile devices, like the BlazeFace model, which is also available in MediaPipe. Detecting hands is challenging because hands come in many sizes and can be in different positions, often overlapping with each other. Unlike faces, which have distinct features like eyes and mouths, hands don't have such obvious patterns, making them harder to detect.

Our approach focuses on detecting palms instead of entire hands because it's easier to identify the boundaries of rigid parts like palms and fists compared to hands with spread fingers. Palms are also smaller, so our method of suppressing duplicate detections works well, even when hands overlap, like during a handshake. By using only square boxes to detect palms, we reduce the complexity of our model.

We use a feature extraction method that helps the model understand the context of the entire scene, even for small objects like hands. Additionally, during training, we minimize errors using a specific loss function, which helps the model handle a wide range of hand sizes effectively. The overall design of our palm detector is shown in Figure 2, and we analyze the effectiveness of our design choices in Table 1.

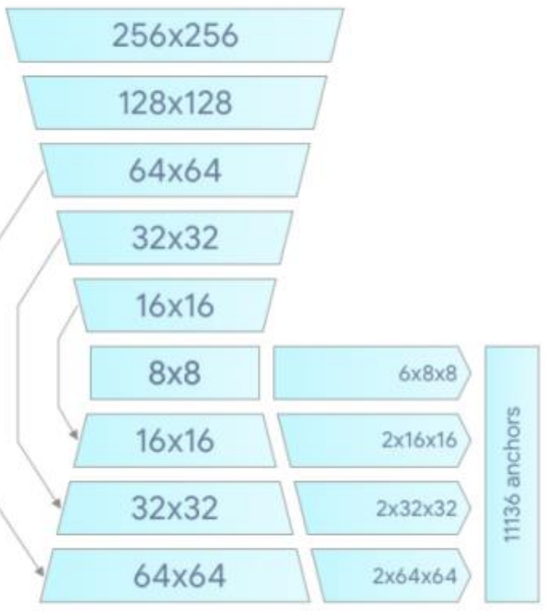


Figure 2.2: Palm detector model architecture.

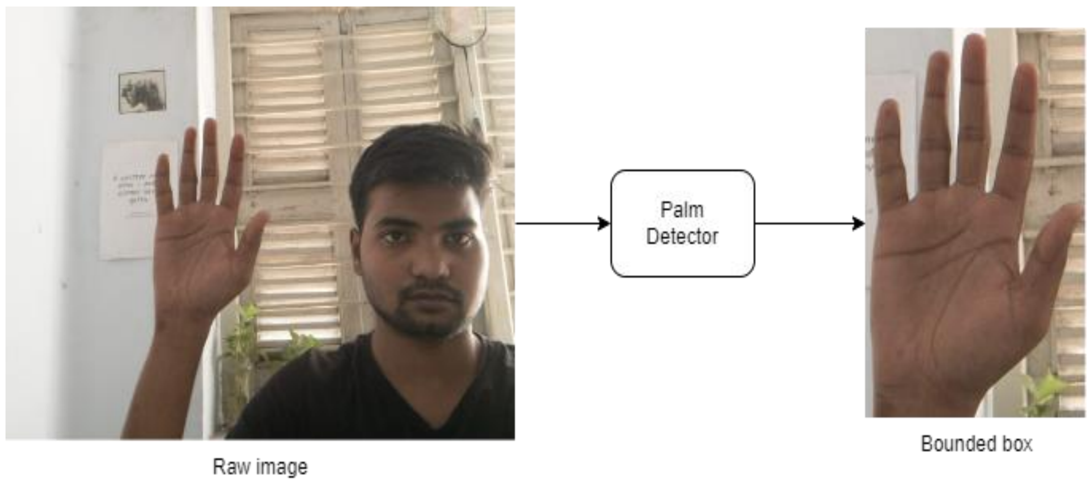


Figure 2.3: Capturing images through webcam

2.2.2. Hand Landmark Model:

After detecting palms in the entire image, our hand landmark model accurately locates 21 2.5D coordinates within the identified hand areas using regression. This model understands hand poses consistently, even when hands are partially visible or overlapping. It provides three main outputs:

1. 21 hand landmarks with x, y, and depth information.
2. A hand presence indicator.
3. A classification of left or right hand.

The model is trained on real-world and synthetic data for 2D coordinates and depth information. To handle tracking issues, it predicts the likelihood of a well-aligned hand in the image, triggering a reset if confidence is low. Determining handedness is crucial for AR/VR interactions, enabling unique functionalities for each hand. The model is optimized for real-time mobile GPU processing, with variations for CPU-based mobile devices and desktops with higher accuracy requirements.

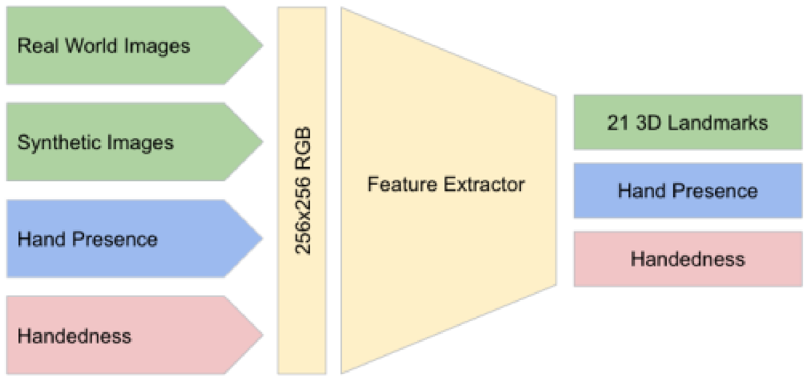
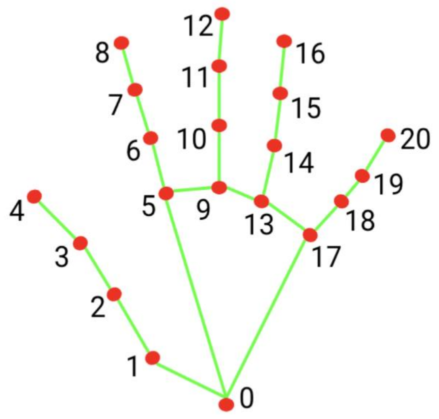


Figure 2.4: Architecture of our hand landmark model.



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

Figure 2.5: Classification of the 21 points

We only use the 21 hand landmarks with x and y coordinates in our classification model to predict the signs. These landmarks are stored in a pickle file for each of the images for further training of the classification models.

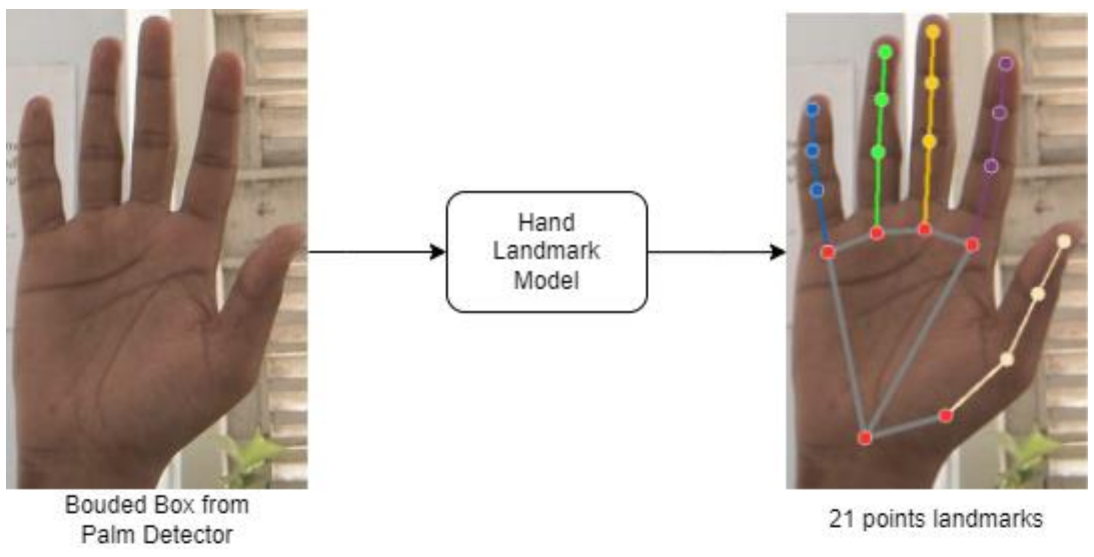


Figure 2.6: 21 hand landmarks from detected palm

2.3. The pickle module:

In Python, we sometimes need to save the object on the disk for later use. This can be done by using Python pickle.[2]

Python pickle module is used for serializing and de-serializing a Python object structure. The serialization process is a way to convert a data structure into a linear form that can be stored or transmitted over a network. In Python, serialization allows you to take a complex object structure and transform it into a stream of bytes that can be saved to a disk or sent over a network. [3]

In context of pickle module, serialization and deserialization is refered to as pickling and unpickling respectively.

The Python pickle module includes four main methods:

1) pickle.dump(obj, file): Saves the object obj to a file.
2) pickle.dumps(obj): Returns a string containing the object obj.
3) pickle.load(file): Reads an object from a file.
4) pickle.loads(string): Returns an object from a string.

The first two methods are used during the pickling process, and the other two are used during unpickling. The only difference between dump() and dumps() is that the first creates a file containing the serialization result, whereas the second returns a string.

2.4. Steps in Data Preprocessing

1. Loading and Reading Images

   The preprocessing starts with loading images from the dataset directory. Each image is read and converted from BGR to RGB format, as required by the MediaPipe library.

2. Hand Landmark Detection

   Using the MediaPipe Hands solution, each image is processed to detect hand landmarks. The key steps are:

   Initialization: The `mp_hands.Hands` object is initialized in static image mode with a minimum detection confidence of 0.3.

   Processing: Each image is converted to RGB and passed through the `hands.process` method to obtain hand landmarks.

3. Normalizing Landmarks

   The detected hand landmarks are normalized to make the model invariant to the scale and position of the hand. The normalization involves:

   - Extracting the x and y coordinates of each landmark.

   - Normalizing these coordinates relative to the minimum x and y values within the detected hand region.

4. Storing Data

The processed data, comprising normalized hand landmarks and corresponding labels, is stored in a dictionary. This dictionary is then serialized and saved as a pickle file for later use in model training. This approach ensures that the data is efficiently stored and easily accessible.

2.5. Conclusion

Data preprocessing is a vital step in building an effective hand gesture recognition model. By leveraging MediaPipe Hands, we efficiently detect and process hand landmarks from images, ensuring that the data is well-structured and ready for model training. This preprocessing pipeline enables us to build robust and accurate machine learning models for real-time hand gesture recognition.

References:

1. Zhang, F., Bazarevsky, V., Vakunov, A., Tkachenka, A., Sung, G., Chang, C.-L., & Grundmann, M. (2020). MediaPipe Hands: On-device Real-time Hand Tracking. Google Research. Available at: [MediaPipe Hands](https://mediapipe.dev)
2. Understanding Python Pickling with example - GeeksforGeeks: https://www.geeksforgeeks.org/understanding-python-pickling-example/
3. The Python pickle Module: How to Persist Objects in Python – Real Python: https://realpython.com/python-pickle-module/

Chapter 3: Model Training

In this chapter, we describe the process of training various machine learning models for hand gesture recognition. The goal is to identify the most effective model for accurate and real-time hand gesture classification. We implemented and compared three different classifiers: Decision Tree, Random Forest, and K-Nearest Neighbors (KNN).

3.1 Introduction to Machine Learning Classifiers

Machine learning classifiers are algorithms that categorize input data into different classes based on learned patterns. These models are trained on labelled datasets and then used to predict the class labels of new, unseen data. In this project, we focus on three widely used classifiers, each with unique strengths and applications.

3.1.1 Decision Tree Classifier

A Decision Tree classifier is a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.[1] Key characteristics include:

- Structure: The model is structured as a tree, where each internal node represents a "test" on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label.
- Training: The model recursively splits the training data into subsets based on the value of a feature, maximizing the information gain at each split.
- Criterion: The criterion determines the function to measure the quality of a split. Common criteria include: [2]
    - Gini: Measures the impurity of a node. Lower values indicate a better split.
    - Entropy: Based on information gain. It measures the unpredictability in the data.
    - Log Loss: Measures the performance of a classification model whose output is a probability value between 0 and 1.
- Advantages: Decision Trees are easy to interpret, require little data preprocessing, and can handle both numerical and categorical data.

3.1.2 Random Forest Classifier

Random Forest is an ensemble learning method that operates by constructing multiple decision trees during training and outputting the mode of the classes (classification) of the individual trees.[3] Key characteristics include:

- Structure: Comprises a multitude of decision trees, each trained on a random subset of the data.

- Training: Each tree is built from a random sample of the data with replacement (bootstrap sampling), and a random subset of features is considered for splitting nodes.
- Criterion: Like Decision Trees, Random Forest uses criteria like Gini, Entropy, and Log Loss for splitting nodes.[4]
- Advantages: Random Forest improves predictive accuracy and controls overfitting. It is robust to noise and can handle large datasets with higher dimensionality.

3.1.3 K-Nearest Neighbors (KNN) Classifier

K-Nearest Neighbors is a simple, instance-based learning algorithm that classifies new instances based on a majority vote of its neighbors, with the instance being assigned to the class most common among its k-nearest neighbors.[5] Key characteristics include:

- Structure: The model stores all available cases and classifies new cases based on a similarity measure (e.g., distance functions).
- Training: No explicit training phase. The algorithm stores the training instances and performs classification directly on new instances.
- n_neighbors: This parameter determines the number of neighbors to consider for determining the class of a given sample. In our experiments, we use 50 and 100 neighbors.[6]
- Advantages: KNN is easy to understand and implement, requires no assumptions about data distribution, and adapts naturally to multi-class classification problems.

3.2 Implementation and Model Training

3.2.1 Data Loading and Preparation

We begin by loading the pre-processed data from a pickle file. The data consists of hand landmarks captured in previous steps, labelled according to the gesture class. The data is transformed into numpy array which is appropriate format for machine learning operations.

We then divide the data into train and test sets. We first shuffle the data and select 80% for training and 20% for testing.

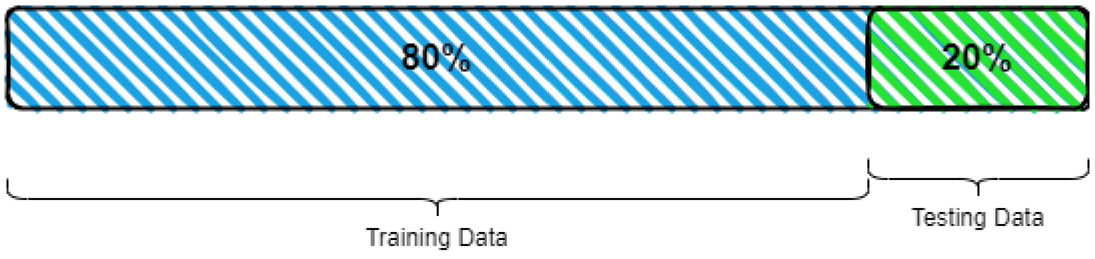


Figure 3.1: Splitting the data for training and testing

3.2.2. Model Training and Evaluation

We train several models using different classifiers and evaluate their performance using accuracy scores. We train three different models: Decision Tree, Random Forest, and K-Nearest Neighbors, and evaluate their performance on the test dataset.

The Decision Tree and random is further trained for three different splitting criteria: gini, entropy and log-loss. The K-Nearest Neighbors is trained for two different n_neighnbors: n = 50 and n = 100 as we have around 100 data points for each class. So, there are total of eight different models.

We evaluate the models on the three most used parameters: Accuracy, Precision, and Recall [7]

3.2.2.1 Accuracy

Accuracy is the ratio of correctly predicted instances to the total instances in the dataset. It measures the overall effectiveness of the model in correctly classifying both positive and negative instances.

$$Accuracy = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Instances}}\}$$

True Positives (TP): Instances correctly predicted as positive.

True Negatives (TN): Instances correctly predicted as negative.

Accuracy is a useful metric when the classes are balanced. However, it can be misleading in imbalanced datasets where one class may dominate.

3.2.2.2. Precision

Precision is the ratio of correctly predicted positive instances to the total instances predicted as positive. It measures the accuracy of the positive predictions.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives}+\text{False Positives}}\}\}$$

False Positives (FP): Instances incorrectly predicted as positive.

High precision indicates that the model has a low false positive rate. Precision is particularly important in scenarios where the cost of false positives is high.

3.2.2.3. Recall

Recall, also known as sensitivity or true positive rate, is the ratio of correctly predicted positive instances to the total actual positive instances. It measures the model's ability to identify positive instances.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives}+\text{False Negatives}}\}\}$$

False Negatives (FN): Instances incorrectly predicted as negative.

High recall indicates that the model has a low false negative rate. Recall is crucial in situations where missing positive instances (false negatives) is costly.

3.2.3 Selecting and Saving the Best Model

Based on the metrics scores, we select the best-performing model and save it using pickle for later use in real-time detection. Here, we select the best model according to the accuracy, we break the ties by precision and recall. From the following table we can see that the 'random_forest_gini' and 'random_forest_entropy', i.e., Random Forest Classifiers with criterion as Gini and Entropy respectively, has the highest accuracy. Among them the Random Forest Classifier with Entropy as the splitting criteria has the better precision. So, we would save the 'random_forest_entropy' model for our real-time use.

'Classifier'	'accuracy'	'precision'	'recall'
'decision_tree_gini'	98.321	98.405	98.321
'decision_tree_entropy'	97.948	98.013	97.948
'decision_tree_log_loss'	97.761	97.845	97.761
'random_forest_gini'	99.254	99.328	99.254
'random_forest_entropy'	99.254	99.347	99.254
'random_forest_log_loss'	99.067	99.188	99.067
'k_neighbors_n50'	93.657	95.238	93.657
'k_neighbors_n100'	87.5	90.252	87.5

Table 3.1: Comparision of the Model to select the best

3.3 Conclusion

This chapter covered the training and evaluation of three machine learning models for hand gesture recognition. We compared the performance of Decision Tree, Random Forest, and K-Nearest Neighbors classifiers, ultimately selecting the best model based on accuracy. This trained model will be used in the next phase for real-time hand gesture detection.

References

1. Quinlan, J. R. (1986). Induction of decision trees. Machine learning, 1(1), 81-106.
2. DecisionTreeClassifier — scikit-learn 1.5.0 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

3. Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32.
4. RandomForestClassifier — scikit-learn 1.5.0 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
5. Cover, T., & Hart, P. (1967). Nearest neighbor pattern classification. IEEE transactions on information theory, 13(1), 21-27.
6. KNeighborsClassifier — scikit-learn 1.5.0 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
7. Iftikhar Alam, Abdul Hameed, Riaz Ahmad Ziar, (2024). "Exploring Sign Language Detection on Smartphones: A Systematic Review of Machine and Deep Learning Approaches", Advances in Human-Computer Interaction, vol. 2024, Article ID 1487500, 36 pages. https://doi.org/10.1155/2024/1487500

Chapter 4: Real Time Detection

4.1 Introduction to Real-Time Detection

In this chapter, we detail the implementation of the real-time hand gesture recognition system using the trained machine learning model. The system captures live video feed, processes each frame to detect hand landmarks, and uses the model to predict the hand gesture. The recognized gestures are displayed on the video feed in real time.

Real-time detection is a crucial phase in the hand gesture recognition system, as it involves processing live video streams to identify and classify gestures dynamically. This requires efficient and fast image processing and machine learning techniques to ensure minimal latency and high accuracy.

4.2 Tools and Libraries

4.2.1 OpenCV (cv2)

OpenCV is an open-source computer vision and machine learning software library. It contains over 2500 optimized algorithms, which are used for various tasks, including object detection, facial recognition, and gesture recognition. For our project, OpenCV is used to capture video frames from the webcam and display the processed frames with annotated gestures.

Key Functions Used

- *cv2.VideoCapture*: This function is used to capture video from the webcam. It initializes the video capture object.
- *cv2.cvtColor*: This function converts images from one color space to another. In our case, it converts BGR (default color space for images read by OpenCV) to RGB, which is required by the Mediapipe hands solution.
- *cv2.imshow*: This function displays an image in a window. We use it to show the processed video frames with the detected hand gestures.
- *cv2.rectangle*: This function draws a rectangle on the image. We use it to highlight the detected hand in the video frame.
- *cv2.putText*: This function overlays text on the image. It is used to display the predicted gesture above the detected hand.

4.2.2 Mediapipe

Mediapipe is a cross-platform framework for building multimodal applied machine learning pipelines. It is developed by Google and provides state-of-the-art machine learning solutions for various tasks, including hand tracking.

Key Functions Used

- *mp.solutions.hands:* This module contains the hand tracking solution that

detects and tracks hands in images or video streams.
- *mp_drawing*: This module provides utilities for drawing the detected landmarks on the image.

4.3 Real-Time Detection Implementation

The implementation of real-time hand gesture detection involves capturing video frames, processing each frame to detect hand landmarks, and using the trained machine learning model to predict the gesture. The predicted gesture is then displayed on the video feed.

Steps of the Implementation:

1. Load the Trained Model: The trained model is loaded from a pickle file.

2. Initialize Video Capture: The webcam is accessed using `cv2.VideoCapture(0) `.

3. Initialize Mediapipe Hands Solution: The Mediapipe hands solution is initialized to detect hand landmarks.

4. Define Label Dictionary: A dictionary mapping model predictions to hand gestures is defined.

5. Real-Time Processing Loop: The loop captures video frames, processes each frame to detect hand landmarks, and predicts the hand gesture using the trained model. The detected hand and the predicted gesture are displayed on the video feed.

4.4 Results and Discussion

The real-time detection system successfully captures and processes video frames to recognize hand gestures. The use of OpenCV and Mediapipe ensures efficient and accurate detection of hand landmarks. The trained machine learning model effectively classifies the gestures, and the system provides real-time feedback by displaying the predicted gestures on the video feed.

References

1. OpenCV Documentation: https://opencv.org/

2. Mediapipe Documentation: https://mediapipe.dev/

Appendix 1: Python Code

Appendix 1.1: Raw Data Collection:

```
import os
import cv2

# Define the directory where the data will be stored
DATA_DIR = './data'

# Create the data directory if it doesn't exist
if not os.path.exists(DATA_DIR):
    os.makedirs(DATA_DIR)

# Define the number of classes and the size of the dataset for each class
number_of_classes = 3
dataset_size = 100

# Initialize the video capture from the default webcam
cap = cv2.VideoCapture(0)

# Loop over each class to collect data
for j in range(number_of_classes):
    # Define the starting class label (0 in this case)
    Ndone = 0

    # Create a directory for the current class if it doesn't exist
    if not os.path.exists(os.path.join(DATA_DIR, str(j + Ndone))):
        os.makedirs(os.path.join(DATA_DIR, str(j + Ndone)))

    # Print a message indicating which class data is being collected
    print('Collecting data for class {}'.format(j + Ndone))

    # Flag to indicate if data collection is done (not used in the code)
    done = False

    # Wait for the user to be ready and press 'Q' to start data collection
    while True:
        ret, frame = cap.read()  # Capture a frame from the webcam
        # Display a message on the frame
        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)  # Show the frame with the message
        if cv2.waitKey(25) == ord('q'):  # Wait for the user to press 'Q'
            break  # Exit the loop and start data collection

    # Counter to keep track of the number of images collected
    counter = 0
    while counter < dataset_size:
        ret, frame = cap.read()  # Capture a frame from the webcam
        cv2.imshow('frame', frame)  # Show the frame
        cv2.waitKey(25)  # Wait for 25 milliseconds
        # Save the captured frame to the appropriate directory
        cv2.imwrite(os.path.join(DATA_DIR, str(j + Ndone),
'{}.jpg'.format(counter)), frame)
        counter += 1  # Increment the counter

# Release the video capture object and close all OpenCV windows
cap.release()
cv2.destroyAllWindows()
```

Appendix 1.2: Data Transformation:

```
import os
import pickle
import mediapipe as mp
import cv2
import matplotlib.pyplot as plt

# Initialize the Mediapipe hands solution
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles

# Create a hands object with static image mode and a minimum detection
confidence of 0.3
hands = mp_hands.Hands(static_image_mode=True,
min_detection_confidence=0.3)

# Define the directory where the data is stored
DATA_DIR = './data'

# Initialize empty lists to hold the data and labels
data = []
labels = []

# Loop through each directory (representing a class) in the data directory
for dir_ in os.listdir(DATA_DIR):
    # Loop through each image file in the current directory
    for img_path in os.listdir(os.path.join(DATA_DIR, dir_)):
        data_aux = []

        x_ = []
        y_ = []

        # Read the image file
        img = cv2.imread(os.path.join(DATA_DIR, dir_, img_path))
        # Convert the image from BGR to RGB
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        # Process the image to detect hand landmarks
        results = hands.process(img_rgb)
        if results.multi_hand_landmarks:
            # Loop through each detected hand landmark
            for hand_landmarks in results.multi_hand_landmarks:
                for i in range(len(hand_landmarks.landmark)):
                    # Extract the x and y coordinates of each landmark
                    x = hand_landmarks.landmark[i].x
                    y = hand_landmarks.landmark[i].y
                    x_.append(x)
                    y_.append(y)

                # Normalize the coordinates by subtracting the minimum
values
                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_))

            # Append the processed data and corresponding label to the
lists
            data.append(data_aux[:42])
            labels.append(dir_)

# Save the collected data and labels into a pickle file
f = open('data.pickle', 'wb')
pickle.dump({'data': data, 'labels': labels}, f)
f.close()
```

Appendix 1.3: Model Training:

```
import pickle
import numpy as np
from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy_score, precision_score, recall_score

# Load the data from the pickle file
data_dict = pickle.load(open('./data.pickle', 'rb'))

# Convert the data and labels to numpy arrays
data = np.asarray(data_dict['data'])
labels = np.asarray(data_dict['labels'])
```

```
# Split the data into training and testing sets (80% train, 20% test), with
stratification to maintain label proportions
x_train, x_test, y_train, y_test = train_test_split(data, labels,
test_size=0.2, shuffle=True, stratify=labels)

# Define the models to be trained and evaluated
models = {
    "decision_tree_gini" : DecisionTreeClassifier(criterion="gini"),
    "decision_tree_entropy" : DecisionTreeClassifier(criterion="entropy"),
    "decision_tree_log_loss" :
DecisionTreeClassifier(criterion="log_loss"),
    "random_forest_gini" : RandomForestClassifier(criterion="gini"),
    "random_forest_entropy" : RandomForestClassifier(criterion="entropy"),
    "random_forest_log_loss" :
RandomForestClassifier(criterion="log_loss"),
    "k_neighbors_n50" : KNeighborsClassifier(n_neighbors=50),
    "k_neighbors_n100" : KNeighborsClassifier(n_neighbors=100)
}

# Initialize dictionaries to store predictions and scores for each model
y_predict = {}
score = {}

# Set the initial best model to 'decision_tree_gini'
best = "decision_tree_gini"

# Train and evaluate each model
for model in models:
    # Fit the model to the training data
    models[model].fit(x_train, y_train)
    # Predict the labels for the test data
    y_predict[model] = models[model].predict(x_test)
    # Calculate accuracy, precision, and recall scores for the model
    score[model] = [
        round(100*accuracy_score(y_predict[model], y_test), 3),
        round(100*precision_score(y_predict[model], y_test,
average='weighted'), 3),
        round(100*recall_score(y_predict[model], y_test,
average='weighted'), 3)
    ]
    # Update the best model if the current model has a higher score
(comparison is based on the first metric)
    if score[model] > score[best]:
        best = model

# Print the scores of all models
print(score)

# Save the best model to a pickle file
f = open('model.p', 'wb')
pickle.dump({'model': models[best]}, f)
f.close()
```

Appendix 1.4: Real Time Detection:

```
import pickle
import cv2
import mediapipe as mp
import numpy as np

# Load the saved model from the pickle file
model_dict = pickle.load(open('./model.p', 'rb'))
model = model_dict['model']

# Initialize the video capture from the webcam
cap = cv2.VideoCapture(0)

# Initialize MediaPipe Hands solution
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles

# Set up the MediaPipe Hands
hands = mp_hands.Hands(static_image_mode=True,
min_detection_confidence=0.3)

# Define a dictionary to map numerical labels to their corresponding
characters
labels_dict = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7:
'H', 8: 'I', 9: 'J', 10: 'K', 11: 'L', 12: 'M', 13: 'N', 14: 'O', 15: 'P',
16: 'Q', 17: 'R', 18: 'S', 19: 'T', 20: 'U', 21: 'V', 22: 'W', 23: 'X', 24:
'Y', 25: 'Z', 26: '1', 27: '2', 28: '3', 29: '4', 30: '5', 31: '6', 32:
'7', 33: '8', 34: '9'}

# Start an infinite loop to process the video frames
while True:
    data_aux = []  # List to store the normalized coordinates
    x_ = []  # List to store the x-coordinates of the hand landmarks
    y_ = []  # List to store the y-coordinates of the hand landmarks

    ret, frame = cap.read()  # Capture a frame from the webcam

    H, W, _ = frame.shape  # Get the dimensions of the frame

    # Convert the frame from BGR to RGB color space
    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    # Process the frame using MediaPipe Hands
    results = hands.process(frame_rgb)
    if results.multi_hand_landmarks:  # Check if any hand landmarks are
detected
        for hand_landmarks in results.multi_hand_landmarks:
            # Draw the hand landmarks and connections on the frame
```

```
                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())

            # Extract and normalize the hand landmark coordinates
            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_))  # Normalize x-coordinates
                    data_aux.append(y - min(y_))  # Normalize y-coordinates

            # Calculate the bounding box of the hand
            x1 = int(min(x_) * W) - 10
            y1 = int(min(y_) * H) - 10

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

            # Predict the hand gesture using the trained model
            prediction = model.predict([np.asarray(data_aux[:42])])

            # Get the predicted character from the labels dictionary
            predicted_character = labels_dict[int(prediction[0])]

            # Draw the bounding box and the predicted character on the frame
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 0, 0), 4)
            cv2.putText(frame, predicted_character, (x1, y1 - 10),
cv2.FONT_HERSHEY_SIMPLEX, 1.3, (0, 0, 0), 3, cv2.LINE_AA)

        # Display the frame
        cv2.imshow('frame', frame)
        cv2.waitKey(1)  # Wait for 1 millisecond before processing the next
frame

# Release the video capture object and close all OpenCV windows
cap.release()
cv2.destroyAllWindows()
```

Appendices 2: System Information

Device information:

Processor: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz

Installed RAM: 8.00 GB (7.78 GB usable)

System type: 64-bit operating system, x64-based processor

OS: Windows 11 Home Single Language, Version: 23H2

Software Information:

Code Editor: Visual Studio Code(user) Version: 1.89.1

Language: Python, Version: 3.12.3 64-Bit (Microsoft Store)

The data used in this project can be found in our github repository along with the required code for the whole process. https://github.com/rakupcode/real-time-hand-gesture-recognition