PROJECT REPORT ON

Hand Sign recognition using deep learning

Submitted to

Department of Computer Applications

in partial fulfillment for the award of the degree of

MASTER OF COMPUTER APPLICTIONS

Batch (2023-2025)

Submitted by

Name of the student: Rahul Chaudhry

Enrollment Number: GE-20211900

Under the Guidance of

Dr.Kamlesh Chandra Purohit

GRAPHIC ERA DEEMED TO BE UNIVERSITY DEHRADUN

May -2024

Graphic Era
Deemed to be University

CANDIDATE'S DECLARATION

I hereby certify that the work presented in this project report entitled "**Hand Sign recognition using deep learning**" in partial fulfilment of the requirements for the award of the Degree of Master of Computer Applications (M.C.A.) in the Department of Computer Applications of the Graphic Era (Deemed to be University), Dehradun shall be carried out by the undersigned under the supervision of **Dr. Kamlesh Chandra Purohit**, Department of Computer Application, Graphic Era Deemed to be University, Dehradun, India.

This work has not been submitted elsewhere for the award of a degree/diploma/certificate.

Name	**University Roll no**	**Signature**
Rahul Chaudhry	**1102830**	

This is to certify that the above-mentioned statement in the candidate's declaration is correct to the best of my knowledge.

Date: **Name and Signature**

Dr.Kamlesh Chandra Purohit

Signature of Supervisor **Signature of External Examiner**

HOD

ACKNOWLEDGEMENT

I would like to express my special thanks of gratitude to my **President Sir ”Dr. Kamal Ghansala**” for their able guidance and support in completing my work.

I would also like to extend my gratitude to the “**Mr. Upendra Aswal** ” for providing me with all the facility that was required.

I feel it is a pleasure to be indebted to, **Dr.Kamlesh Chandra Purohit**, Department of Computer Application for their invaluable support, advice and encouragement.

I would also like to thank all my friends who have helped me during the project work

Date:
01-06-2024

Rahul Chaudhry
MCA IInd Semester ‘A’
Roll No.: 57

CERTIFICATE OF ORIGINALITY

This is to certify that the project report entitled "**Hand Sign Recognition using Deep Learning**" submitted to **Graphic Era University, Dehradun** in partial fulfilment of the requirement for the award of the degree of **MASTER OF COMPUTER APPLICATIONS (MCA)**, is an authentic and original work carried out by Mr./Ms. **Rahul Chaudhry** with enrolment number **GE-20211900** under my supervision and guidance.

The matter embodied in this project is genuine work done by the student and has not been submitted whether to this University or to any other University / Institute for the fulfilment of the requirements of any course of study.

Signature of the Student:	**Signature of the Guide**
Date:	**Date:**
Enrolment No.: GE-20211900	
Name and Address	**Name and Address**
of the Student:	**of the Guide:**
Rahul Chaudhry	**Dr.Kamlesh Chandra Purohit**
Dehradun, Uttarakhand	

Special Note:

CERTIFICATE OF INTERNAL GUIDE

This is to certify that Mr./Ms. **Rahul Chaudhry** bearing university Roll no **1102830** of 2023-25 batch, has completed his/her project on "**Hand Sign Recognition using Deep Learning**" under the supervision of **Dr.Kamlesh Chandra Purohit** and has submitted this project report under my guidance in partial fulfilment of their requirements for the degree of Master of Computer Application. To the best of my knowledge and belief, this project report has been prepared by the student and has not been submitted to any other institute or university for the award of any degree or diploma.

Date: **Signature of Internal Guide**

Name: Dr. Kamlesh Chandra Purohit

Table of Contents

ABSTRACT

Hand sign detection is a prominent field in computer vision that focuses on identifying and classifying hand gestures, which can be used for various applications including human-computer interaction, sign language recognition, and gesture-based control systems. Each individual's hand signs serve as a means to convey information or commands, and the detection and recognition of these signs involve complex processes of image analysis and pattern recognition. This report outlines the steps involved in detecting hand signs from video or image frames using keypoint classification techniques.

The primary goal of this project is to detect hand signs in a given video or image frame using a pre-trained keypoint classifier model. The process begins with detecting keypoints of the hand, which involves identifying the positions of significant points on the hand such as knuckles and fingertips. Subsequently, the detected keypoints are classified into predefined categories of hand signs using a neural network model.

Several methods exist for hand sign recognition, including keypoint-based classification, which utilizes the spatial configuration of hand keypoints to identify gestures. In this project, we employ a neural network model trained on a dataset of hand keypoints, utilizing TensorFlow and Keras libraries in Python for model building and training. The model is further optimized and converted into a TensorFlow Lite model for efficient deployment on edge devices. The use of keypoint classification for hand sign detection not only enhances the accuracy of recognition but also ensures real-time performance.

Keywords: hand sign detection, keypoint classification, TensorFlow, OpenCV, Python

CHAPTER 1 INTRODUCTION

1.1 OVERVIEW

Hand sign recognition is a significant area of research within computer vision that focuses on identifying and interpreting hand gestures from digital images or video frames. This technology has vast applications, including human-computer interaction, sign language recognition, and gesture-based control systems. Researchers are developing multiple methodologies to enhance the accuracy and efficiency of hand sign recognition systems.

Hand sign recognition systems work by detecting and analyzing key points on the hand to classify various gestures. The most advanced methods use deep learning algorithms to pinpoint and measure the positions of keypoints, such as knuckles and fingertips, from a given image. By analyzing these key points, the system can identify specific hand signs and map them against a pre-defined set of gestures.

While initially developed as a type of computer application, hand sign recognition systems have seen broader applications in recent times, including integration with smartphones and other technological devices. As an essential aspect of automated interaction systems, hand sign recognition involves analyzing physiological characteristics, classifying it as a form of biometric technology. Despite being less precise than some other biometric technologies, such as iris or fingerprint recognition, hand sign recognition is widely adopted due to its contactless and non-invasive nature.

Hand sign recognition systems are deployed in various domains, including advanced human-computer interaction, video surveillance, and automated classification of gestures. This technology continues to evolve, driven by advancements in machine learning and deep learning algorithms, aiming to improve real-time performance and recognition accuracy.

1.2PURPOSE

1.2.1HUMAN PSYCHOLOGY

Understanding human psychology is crucial in the development and implementation of hand sign recognition systems. Our project recognizes the significance of human cognition and behavior in the utilization of this technology.

Perception and Recognition

Human perception plays a vital role in recognizing hand signs. Our project considers how individuals perceive hand movements, gestures, and configurations, and how these perceptual processes influence the effectiveness of hand sign recognition algorithms.

Attention and Focus

Human attention and focus affect the reliability of hand sign recognition systems. We explore factors such as attentional biases, distractions, and cognitive load, and their impact on users' ability to accurately recognize hand signs in real-time scenarios.

w

Memory and Familiarity

Human memory and familiarity with hand signs influence recognition accuracy. Our project investigates how memory processes, including encoding, storage, and retrieval, influence the performance of hand sign recognition algorithms, particularly in identifying familiar hand signs versus unfamiliar ones.

Emotion and Context

Emotions and contextual factors shape human interactions with hand sign recognition technology. We examine how emotional states, cultural norms, and situational contexts impact users' trust, acceptance, and willingness to engage with hand sign recognition systems.

Ethical and Social Considerations

Human values, beliefs, and ethical considerations guide the development and deployment of hand sign recognition systems. Our project addresses ethical dilemmas, privacy concerns, and societal implications associated with the use of hand sign recognition technology, aiming to promote responsible and inclusive design practices.

Understanding these psychological aspects is essential for creating hand sign recognition systems that are not only accurate and efficient but also user-friendly and ethically sound. By integrating insights from human psychology, our project aims to develop a robust and reliable hand sign recognition system that meets the needs and expectations of users.

1.2.2FACTS & STATISTICS

In the realm of hand sign recognition technology, facts and statistics underscore the importance of robust security measures and the prevalence of security breaches:

Our current data indicates that in 2015, in India alone, there were 148,708 reported incidents of unauthorized access and security breaches. These breaches often occur due to the lack of robust authentication methods, leaving systems vulnerable to exploitation. At least 22 percent of these breaches were attributed to unauthorized access through stolen or compromised credentials. This highlights the critical need for more secure authentication mechanisms to prevent unauthorized access to sensitive information.

In 2019, the number of reported incidents of identity theft and impersonation reached alarming levels, with 1.65 lakh cases reported in India alone. The lack of proper security measures, such as multi-factor authentication and biometric recognition, has contributed to the rise in identity theft incidents. Hand sign recognition technology offers a promising solution to this problem by providing a passive and non-intrusive means of authentication.

One of the main advantages of hand sign recognition technology is its ability to provide seamless and secure authentication without the need for additional hardware or cumbersome authentication processes. By leveraging hand sign biometrics, organizations can enhance security and prevent unauthorized access to sensitive data and resources.

Furthermore, the implementation of hand sign recognition technology can help mitigate the risks associated with traditional authentication methods, such as password theft and phishing attacks. With hand sign recognition, users are authenticated based on unique hand configurations and gestures, reducing the likelihood of unauthorized access and identity theft.

Overall, the adoption of hand sign recognition technology represents a significant step towards enhancing security and protecting sensitive information from unauthorized access. By embracing innovative authentication methods, organizations can safeguard their data and resources against evolving security threats and ensure the integrity and confidentiality of their systems.

1.3)PROBLEM STATEMENT

The project focuses on implementing hand sign detection using Intel's open-source Computer Vision Library (OpenCV) and Python dependency. It comprises various scripts designed to detect hand signs in images and live video feeds, capture hand sign images for dataset creation, train classifiers for recognition, and recognize trained hand signs.

All scripts are written in Python 3.6.5 and are accompanied by documented code, providing essential tools and insights for hand sign detection with OpenCV. The project demonstrates the implementation of various algorithms and detection approaches, which will be elaborated upon in the project report.

Hand sign detection holds significance in various fields such as sign language translation, gesture-based interfaces, human-computer interaction, and robotics. It can greatly improve communication accessibility for individuals with hearing impairments and facilitate intuitive interaction in technological systems. Additionally, accurate hand sign detection has potential applications in surveillance, healthcare, and virtual reality environments

1.4)OBJECTIVES

The objectives of this project focusing on hand sign detection using TensorFlow and MediaPipe are outlined as follows:

- Optimized Hand Sign Detection: Develop and implement algorithms to accurately detect hand signs in images and video frames using TensorFlow and MediaPipe, ensuring high precision and efficiency in the detection process.
- Minimization of False Positives: Minimize false positives by refining the hand sign detection model to accurately distinguish between hand signs and background elements, thereby enhancing the reliability of the detection system.
- Real-Time Detection Capability: Enable real-time detection of hand signs in video streams by optimizing the computational efficiency of the detection model, allowing for seamless integration into applications requiring immediate feedback.
- Multi-Hand Sign Recognition: Extend the detection capabilities to recognize and classify multiple hand signs simultaneously within the same frame, facilitating more comprehensive analysis and interpretation of gestures in complex scenarios.
- Cross-Platform Compatibility: Ensure cross-platform compatibility by developing the detection model and associated software components to be easily deployable on various devices and operating systems, enhancing accessibility and usability.
- User-Friendly Interface: Integrate a user-friendly interface for interacting with the hand sign detection system, providing intuitive controls and visual feedback to users, thereby enhancing the overall user experience.
- Performance Evaluation and Optimization: Conduct thorough performance evaluation of the hand sign detection model, identifying bottlenecks and areas for optimization to enhance speed, accuracy, and resource efficiency, thereby ensuring robust performance across diverse environments.
- Documentation and Knowledge Sharing: Document the development process, including model architecture, training methodologies, and implementation details, to facilitate knowledge sharing and replication of the project by other researchers and developers in the field.

CHAPTER 2 LITERATURE REVIEW

In the field of computer vision and human-computer interaction, hand gesture recognition has garnered significant attention due to its potential applications in various domains such as sign language translation, virtual reality interaction, and human-computer interface design. Researchers have explored different approaches to address the challenges associated with accurate and efficient hand gesture recognition.

One prominent approach involves the use of deep learning techniques, particularly convolutional neural networks (CNNs) and recurrent neural networks (RNNs), for feature extraction and classification. These models have demonstrated impressive performance in recognizing hand gestures from images or video streams. Moreover, transfer learning and data augmentation techniques have been employed to enhance the robustness and generalization capabilities of these models.

In addition to deep learning, hand-crafted feature extraction methods have also been extensively studied. These methods often involve extracting relevant features such as hand shape, finger positions, and motion trajectories, followed by applying machine learning algorithms for classification. While these approaches may lack the scalability and flexibility of deep learning models, they are often more interpretable and computationally efficient, making them suitable for real-time applications.

Furthermore, the availability of large-scale annotated datasets, such as the American Sign Language (ASL) dataset and the ChaLearn Looking at People (CLAP) dataset, has played a crucial role in advancing the state-of-the-art in hand gesture recognition. These datasets facilitate the training and evaluation of complex models and enable researchers to benchmark their algorithms against existing methods.

Despite significant progress, several challenges remain in the field of hand gesture recognition. These include robustness to variations in lighting conditions, occlusions, and background clutter, as well as the need for real-time performance on resource-constrained devices. Addressing these challenges requires interdisciplinary research efforts combining expertise in computer vision, machine learning, signal processing, and human-computer interaction.

Overall, the literature highlights the potential of hand gesture recognition technology to revolutionize human-computer interaction and enable new applications in areas such as healthcare, education, gaming, and assistive technology. Continued research and development in this field are essential to overcome existing limitations and unlock the full potential of hand gesture recognition system

CHAPTER 3 AIM AND SCOPE

3.1Aim

The aim of our project is to develop a real-time hand gesture recognition system capable of accurately interpreting and classifying hand movements and gestures in various contexts. This system will leverage computer vision techniques and machine learning algorithms to analyze input from a camera feed and translate hand gestures into actionable commands or inputs for human-computer interaction.

3.2Objectives

- Algorithm Development: Develop robust algorithms for hand gesture detection, tracking, and classification using computer vision techniques and machine learning models.

- Feature Engineering: Investigate and implement effective methods for extracting relevant features from hand images, including hand shape, finger positions, and motion trajectories, to improve gesture recognition accuracy.

- Model Training and Optimization: Train and optimize deep learning models, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), to accurately classify hand gestures in real-time with high efficiency.

- Real-time Performance: Ensure that the developed system operates in real-time with low latency, enabling seamless interaction between users and computing devices without perceptible delays.

- User Interface Integration: Integrate the gesture recognition system with existing user interfaces or applications to enable users to control software and devices using hand gestures intuitively.

- Testing and Evaluation: Conduct comprehensive testing and evaluation of the system's performance under various conditions, including different lighting conditions, backgrounds, and hand orientations, to assess its robustness and reliability.

- Usability and Accessibility: Design the system with usability and accessibility in mind, making it intuitive and easy to use for individuals with diverse backgrounds and abilities, including those with disabilities.

- Documentation and Deployment: Provide thorough documentation and instructions for deploying and using the hand gesture recognition system in different settings, ensuring ease of implementation and maintenance.

3.3Scope

- Hand Gesture Recognition
- Single Hand Gestures
- Indoor Environment
- Personalized Experiences
- Healthcare
- Real-time Performance

3.4Limitation

- **Accuracy:** The accuracy of hand gesture recognition may vary based on factors such as lighting conditions, background clutter, occlusions, and variations in hand shape and orientation. While efforts will be made to improve accuracy, the system may not achieve perfect recognition in all scenarios.

- **Gesture Complexity:** The system may struggle to recognize complex or subtle hand gestures that involve fine-grained movements or variations in finger positions. Gestures that closely resemble each other or lack distinct features may pose challenges for classification.

- **Single-Hand Recognition:** Initially, the system will focus on recognizing gestures made by a single hand. Gestures involving multiple hands or interactions between hands may not be fully supported in the current implementation.

- **Limited Gesture Set:** The predefined set of recognized gestures may be limited in scope, potentially restricting the range of interactions possible with the system. Adding new gestures may require updates to the system's training data and classification models.

- **Hardware Constraints:** The performance of the system may be influenced by the hardware capabilities of the computing device, including processing power, memory, and camera quality. Low-end devices may experience slower response times or reduced accuracy.

- **Environmental Factors:** The system's performance may be affected by environmental factors such as changes in lighting, background clutter, or occlusions. While efforts will be made to mitigate these effects, the system may not perform optimally in all environments.

- **Real-time Constraints:** Achieving real-time performance may be challenging, especially on resource-constrained devices or when processing large volumes of image data. Delays in gesture recognition may occur, impacting the user experience in time-sensitive applications.

- **User Adaptation:** Users may require some time to adapt to the system's gesture recognition capabilities and learn the supported gestures. Training and providing feedback to users may be necessary to improve interaction efficiency and reduce user frustration.

- **Privacy and Security:** The use of cameras for gesture recognition raises privacy and security concerns, particularly regarding the collection and storage of user data. Measures must be implemented to ensure user consent, data protection, and secure transmission/storage of sensitive information.

- **Maintenance and Updates:** The system may require periodic updates and maintenance to address bugs, improve performance, and add new features. Ensuring seamless deployment of updates and backward compatibility with existing applications is essential for user satisfaction.

3.5Motivation

Our project is motivated by the increasing demand for intuitive and efficient human-computer interaction (HCI) systems, especially in contexts where traditional input methods are impractical. By leveraging hand gesture recognition technology, we aim to provide users with a natural and seamless way to interact with digital devices and applications, enhancing user experience across various domains such as gaming, healthcare, education, and automotive sectors. Moreover, our system aims to improve accessibility for individuals with disabilities or mobility impairments, fostering inclusion and participation in digital spaces. Through user-centric design and compatibility across different platforms, we aspire to create a more inclusive and accessible computing environment, empowering users to express themselves and engage with digital content in more meaningful ways.

CHAPTER 4 PROPOSED METHODOLOGY

4.1 EXISTING SYSTEM

For our project, which focuses on real-time hand gesture recognition and analysis, the utilization of several key libraries is indispensable. Firstly, OpenCV (cv2) serves as a cornerstone for computer vision tasks, enabling us to process and analyze video streams efficiently. Its extensive functionality facilitates tasks such as image manipulation and object detection, crucial for identifying and tracking hand gestures. NumPy complements OpenCV by providing essential numerical computing capabilities, allowing for efficient handling and manipulation of large arrays and matrices, fundamental for processing visual data. Additionally, Mediapipe plays a pivotal role in our project by offering specialized solutions for hand landmark detection. Leveraging Mediapipe's pre-trained models and APIs enables us to accurately detect and track hand landmarks in real-time, laying the foundation for robust hand gesture recognition and analysis algorithms. Together, these libraries form the backbone of our project, empowering us to develop a comprehensive system for real-time hand gesture analysis with high accuracy and efficiency.

DRAWBACKS:

- OpenCV, while powerful, may have a steep learning curve for beginners.
- TensorFlow, being a deep learning framework, introduces complexity in terms of model design, training, and inference.
- NumPy, while efficient, may not be optimized for specific hardware architectures.
- Integrating and managing dependencies between OpenCV, NumPy, and Mediapipe can be challenging.
- Mediapipe's accuracy for hand landmark detection may vary depending on environmental factors such as lighting conditions and occlusions.
- TensorFlow's integration with other libraries like OpenCV and Mediapipe may involve additional configuration and compatibility issues.

4.2 REQUIREMENT ANALYSIS

4.2.1HARDWARE

Processor: AMD Ryzen 5 3550H.

RAM: 16 GB RAM

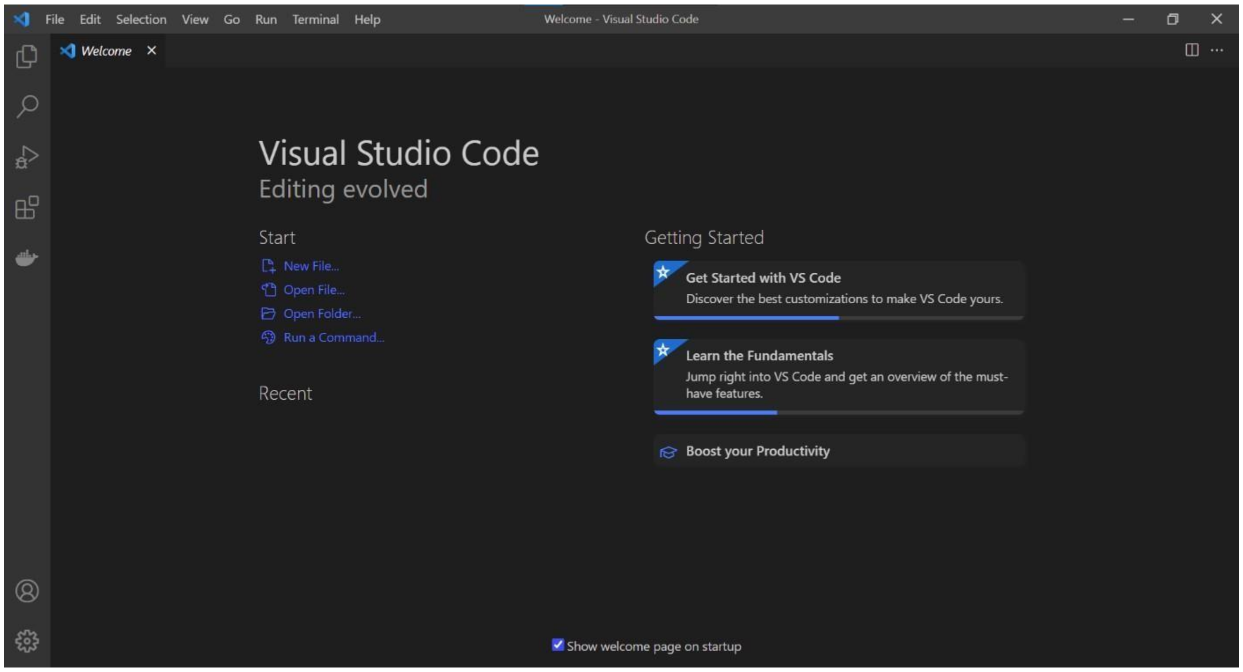
Hard Disk: 1 TB

Web cam: In-built laptop camera

4.2.2 SOFTWARE

VS Code: Visual Studio Code (Fig 1)**,** also commonly referred too as VS Code, is a source code editor made by Microsoft with the Electron framework, for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. User can change the theme, keyboard shortcuts, preference, and install extension that add functionality.

In the Stack Overflow 2022 Developer Survey, Visual Studio Code was ranked the most popular developer environment tool among 71,010 respondents, with 74.48% reporting that they use it.


File Edit Selection View Go Run Terminal Help
Welcome - Visual Studio Code
Welcome
Visual Studio Code
Editing evolved
Start
New File...
Open File...
Open Folder...
Run a Command...
Recent
Getting Started
Get Started with VS Code
Discover the best customizations to make VS Code yours.
Learn the Fundamentals
Jump right into VS Code and get an overview of the must-have features.
Boost your Productivity
Show welcome page on startup

(Fig 1)

Python 3: Python is the basis of the program that we wrote in the system. It utilizes many of the python libraries.

Windows/ linux/ Mac OS

4.2.3 FUNCTIONAL REQUIREMENTS

Open CV: (Fig 2) OpenCV is a library the usage of which we will broaden actual-time computer vision programs. It in particular used for photo processing, video seize and assessment in conjunction with capabilities like face detection and object detection. OpenCV is the big open-supply library Used for laptop imaginative and prescient and also system gaining knowledge of and image processing and now it plays a prime function in actual-time operation which can be very critical in current systems. Through the use of it, you could process photographs, videos, real- time motion pictures to understand gadgets, faces, or even handwriting of a human which can be used for a big selection of functions.

Operating System: An operating system, or "os" is software program that interacts with the hardware and permits one-of-a-kind applications to run. Every laptop pc, pill, and cellular telephone consists of an operating system that gives simple functionality for the device. Common desktop jogging structures consist of home windows, os x, and linux.

Numpy: NumPy, short for Numerical Python, is a fundamental library for numerical computing in Python. It provides support for large multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. Within this project, NumPy is utilized extensively for data manipulation, numerical operations, and array handling. It facilitates tasks such as storing image pixel values, performing calculations on hand landmark coordinates, and manipulating data arrays during model training.

Mediapipe:

Mediapipe is a versatile framework developed by Google that facilitates the development of machine learning pipelines for various tasks involving real-time perception. Specifically, in this project, Mediapipe's hand tracking module is employed for hand landmark detection and tracking. By leveraging its pre-trained models and efficient inference mechanisms, Mediapipe enables accurate and real-time detection of hand landmarks, which is essential for subsequent analysis and interaction in the application.

- Mediapipe is a framework for building multimodal machine learning pipelines.
- Used for hand landmark detection and tracking.

TensorFlow (tf):

TensorFlow is an open-source deep learning framework developed by Google for building and training machine learning models. It offers a comprehensive ecosystem of tools and libraries for various aspects of machine learning, including neural networks, natural language processing, and computer vision. In this project, TensorFlow is employed for constructing, training, and evaluating deep learning models. Its high-level API simplifies the process of defining neural network architectures and optimizing model parameters.

- TensorFlow is an open-source machine learning framework.
- Used for building, training, and deploying deep learning models

Scikit-learn:

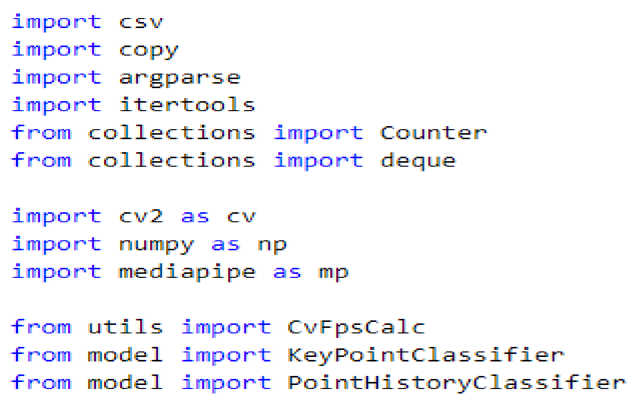
Scikit-learn is a versatile machine learning library built on top of NumPy, SciPy, and Matplotlib. It provides simple and efficient tools for data mining and data analysis, including classification, regression, clustering, and dimensionality reduction. Within this project, Scikit-learn is utilized for tasks such as data preprocessing, model evaluation, and performance metrics computation. Its easy-to-use interface and extensive documentation make it a popular choice for machine learning practitioners.

- Scikit-learn is a machine learning library for Python.
- Used for data preprocessing and evaluation.

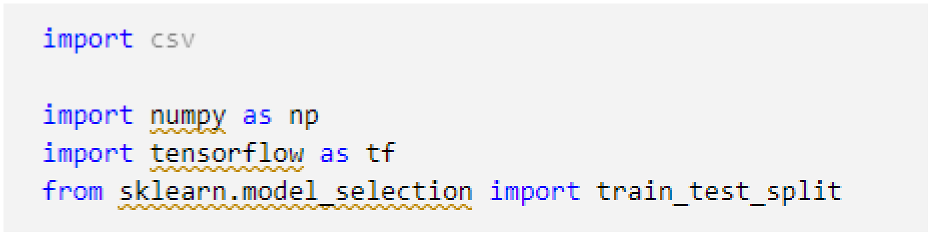
Pandas:

Pandas is a powerful data manipulation and analysis library built on top of NumPy. It offers data structures and functions for efficiently handling structured data and performing various data analysis tasks, such as data cleaning, transformation, and exploration. In this project, Pandas is employed for reading and processing CSV data containing hand keypoints and corresponding labels. Its DataFrame abstraction simplifies the manipulation and analysis of tabular data, enhancing the efficiency of data preprocessing tasks.

- Pandas is a data manipulation and analysis library.
- Used for reading and manipulating CSV data.



```
import csv
import copy
import argparse
import itertools
from collections import Counter
from collections import deque

import cv2 as cv
import numpy as np
import mediapipe as mp

from utils import CvFpsCalc
from model import KeyPointClassifier
from model import PointHistoryClassifier
```



```
import csv

import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
```

(Fig 2)

CHAPTER 5 METHODOLOGY

5.1. Data Collection

Data Collection involves capturing data from the source, processing it, and preparing it for further analysis or usage. In the provided code, this stage primarily focuses on collecting hand landmark data from a video feed using a camera, processing the data, and logging it for analysis. Here's a breakdown of the key components involved in this stage:

Camera Setup:

- The code initializes a camera capture device using OpenCV (cv.VideoCapture), setting parameters such as width and height for capturing the video feed.

Hand Landmark Detection:

- The code utilizes the Mediapipe library (mediapipe) for detecting hand landmarks in each frame of the video feed. It configures the Hands module from Mediapipe to perform hand landmark detection.

Data Logging:

- Detected hand landmarks are processed and logged into CSV files for further analysis. The code contains functions (logging_csv) to log the landmark data into CSV files, separated by different modes (logging key points or point history) and numeric identifiers.

Frame Processing:

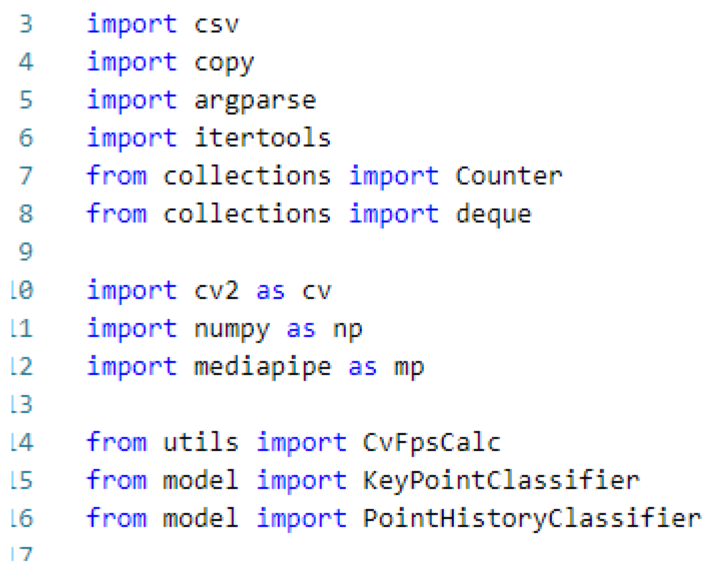
- Each frame from the video feed undergoes processing to detect hand landmarks. Landmark detection is performed on each hand detected in the frame.

User Interaction:

- The user can interact with the program by pressing keys to select different modes and numeric identifiers for data logging.

Overall, Stage I focuses on collecting hand landmark data from the camera feed, processing it, and logging it into CSV files for subsequent stages of analysis or model training.

5.2. Data Collection Code (Fig 3)



```
import csv
import copy
import argparse
import itertools
from collections import Counter
from collections import deque

import cv2 as cv
import numpy as np
import mediapipe as mp

from utils import CvFpsCalc
from model import KeyPointClassifier
from model import PointHistoryClassifier
```

(Fig 3)

1. Importing libraries:

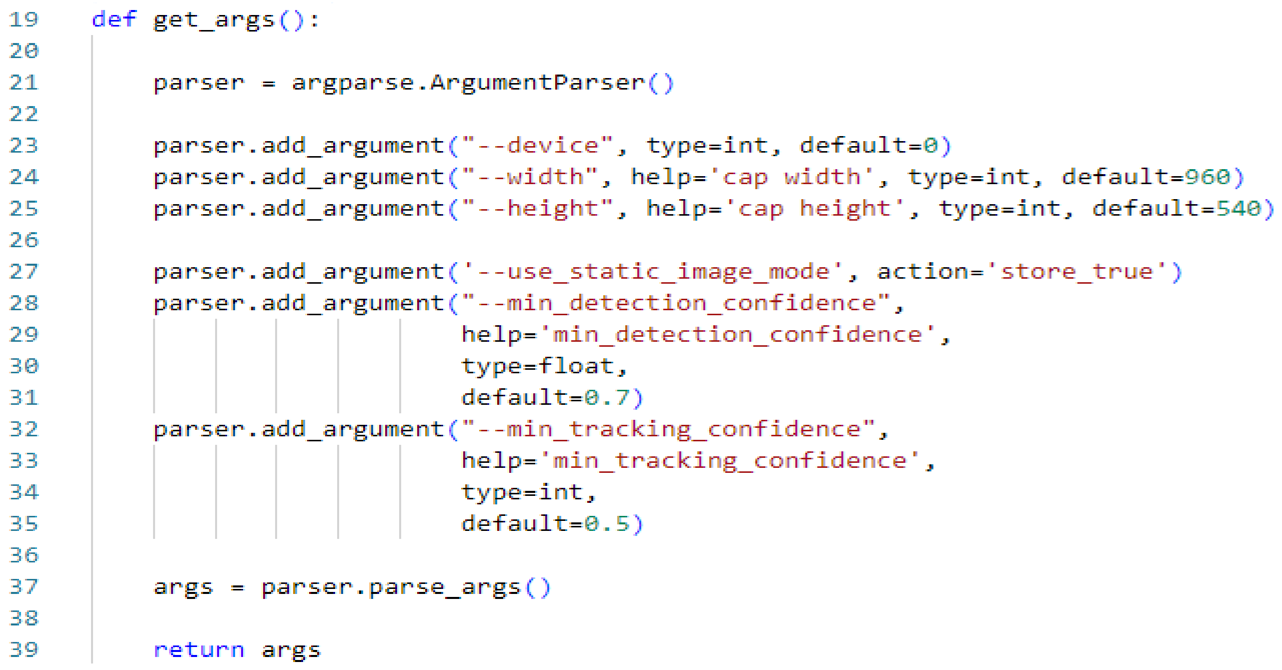
- **import csv**: Used for reading and writing CSV files.
- **import copy**: Used for creating deep copies of objects.
- **import argparse**: Used for parsing command-line arguments.
- **import itertools**: Used for creating iterators for efficient looping.
- **from collections import Counter**: Used for counting occurrences of elements in lists.
- **from collections import deque**: Used for creating a fixed-size queue.

2. Importing external libraries:

- **import cv2 as cv**: OpenCV library for computer vision tasks.
- **import numpy as np**: NumPy library for numerical computations.
- **import mediapipe as mp**: Mediapipe library for hand tracking and gesture recognition.

3. Importing custom modules:

- **from utils import CvFpsCalc**: Custom module for calculating frames per second (FPS) using OpenCV.
- **from model import KeyPointClassifier**: Custom module for hand gesture classification based on key points.
- **from model import PointHistoryClassifier**: Custom module for hand gesture classification based on point history.



```python
def get_args():

    parser = argparse.ArgumentParser()

    parser.add_argument("--device", type=int, default=0)
    parser.add_argument("--width", help='cap width', type=int, default=960)
    parser.add_argument("--height", help='cap height', type=int, default=540)

    parser.add_argument('--use_static_image_mode', action='store_true')
    parser.add_argument("--min_detection_confidence",
                        help='min_detection_confidence',
                        type=float,
                        default=0.7)
    parser.add_argument("--min_tracking_confidence",
                        help='min_tracking_confidence',
                        type=int,
                        default=0.5)

    args = parser.parse_args()

    return args
```

(Fig 4)

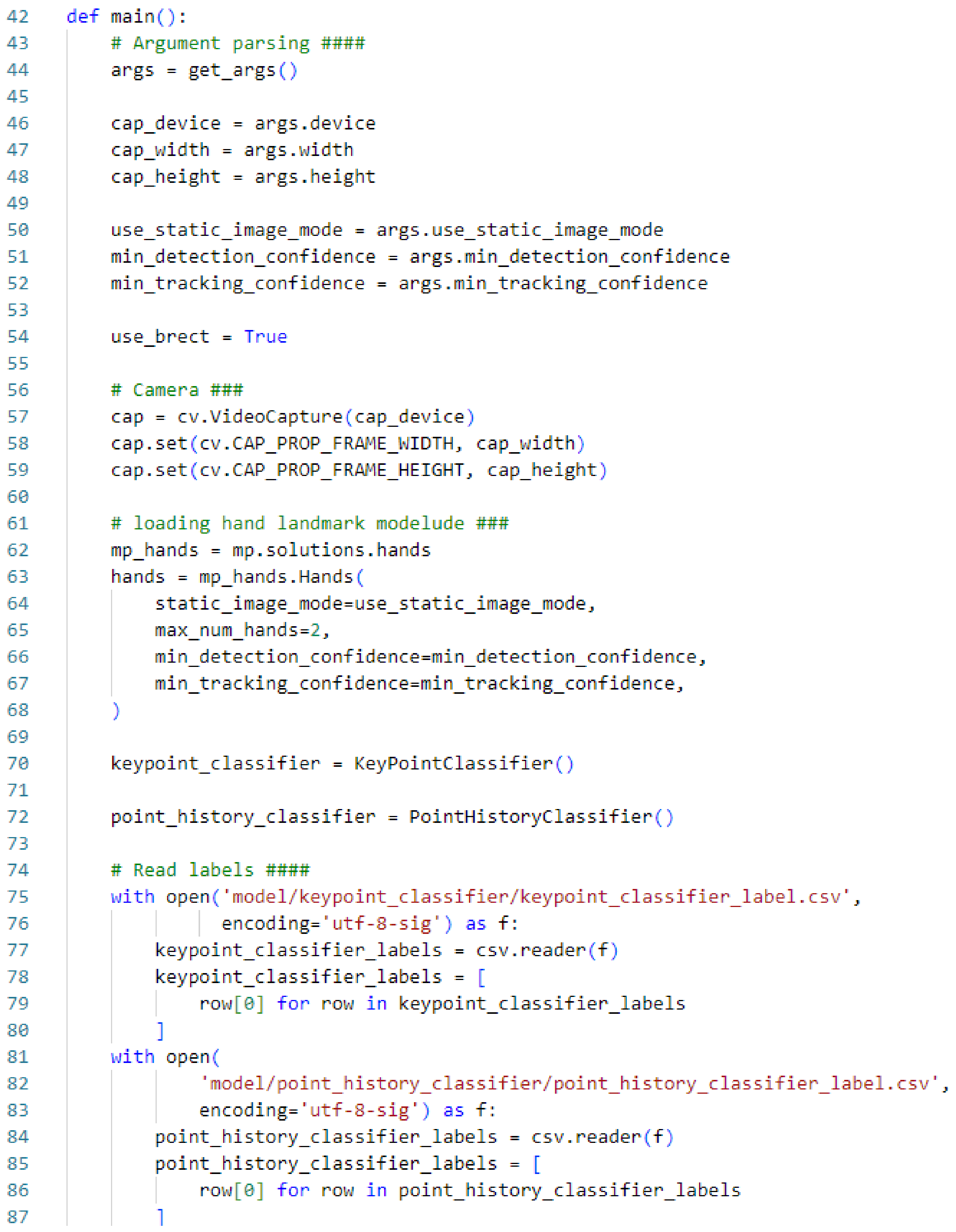
(Fig 4) The **get_args()**function is responsible for parsing command-line arguments using the **argparse** module. Here's a breakdown of the arguments it defines:

- **--device**: Specifies the device index for video capture. Defaults to 0.
- **--width**: Specifies the width of the captured video frame. Defaults to 960 pixels.
- **--height**: Specifies the height of the captured video frame. Defaults to 540 pixels.

- **--use_static_image_mode**: Flag indicating whether to use static image mode for hand detection. If provided, this flag is set to **True**; otherwise, it's **False** by default.
- **--min_detection_confidence**: Specifies the minimum confidence threshold for hand detection. Defaults to 0.7.
- **--min_tracking_confidence**: Specifies the minimum confidence threshold for hand tracking. Defaults to 0.5.

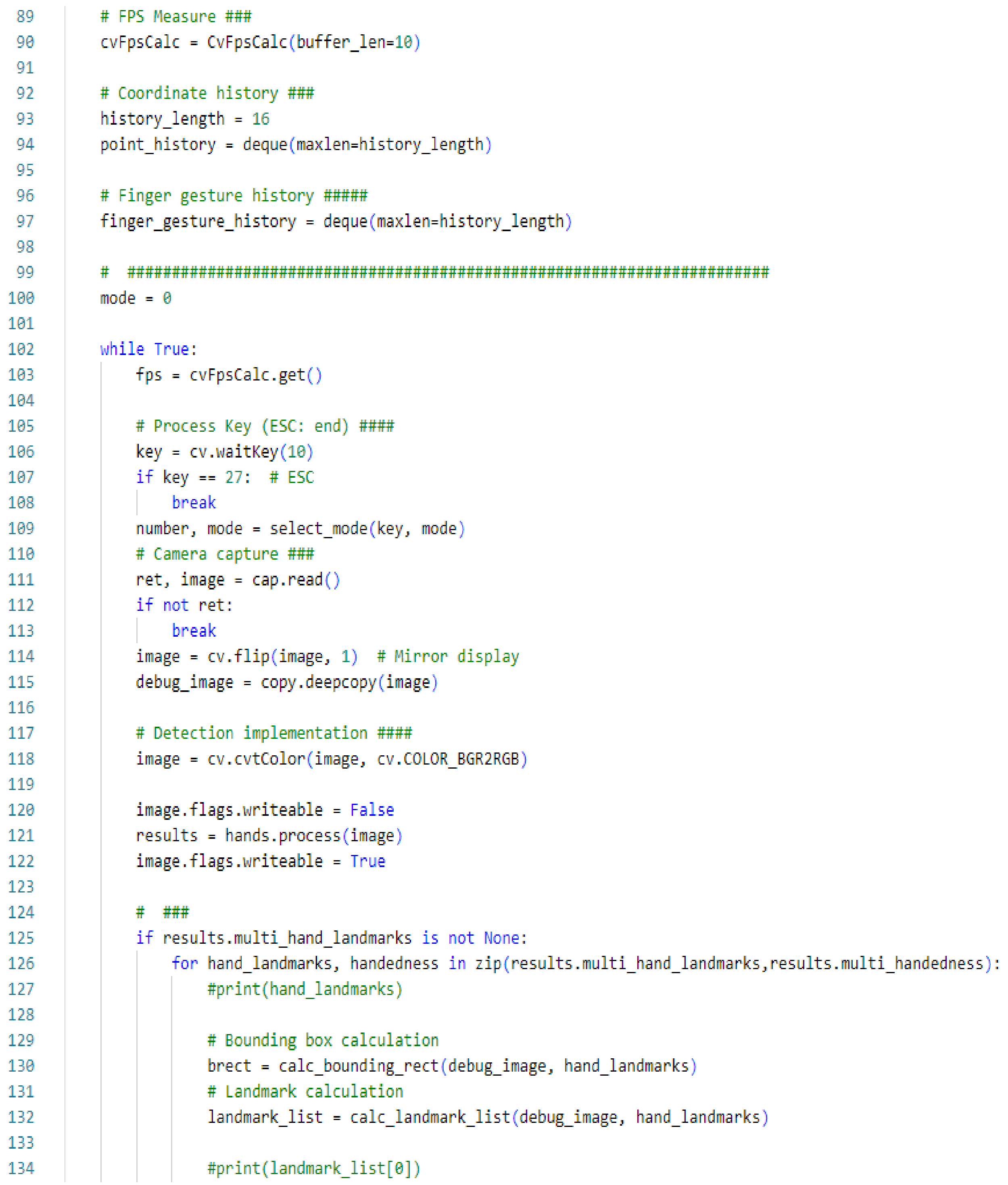
The function returns the parsed arguments as an **argparse.Namespace** object.

This setup allows the user to customize various parameters such as device index, video frame dimensions, and confidence thresholds for hand detection and tracking when running the script from the command line.



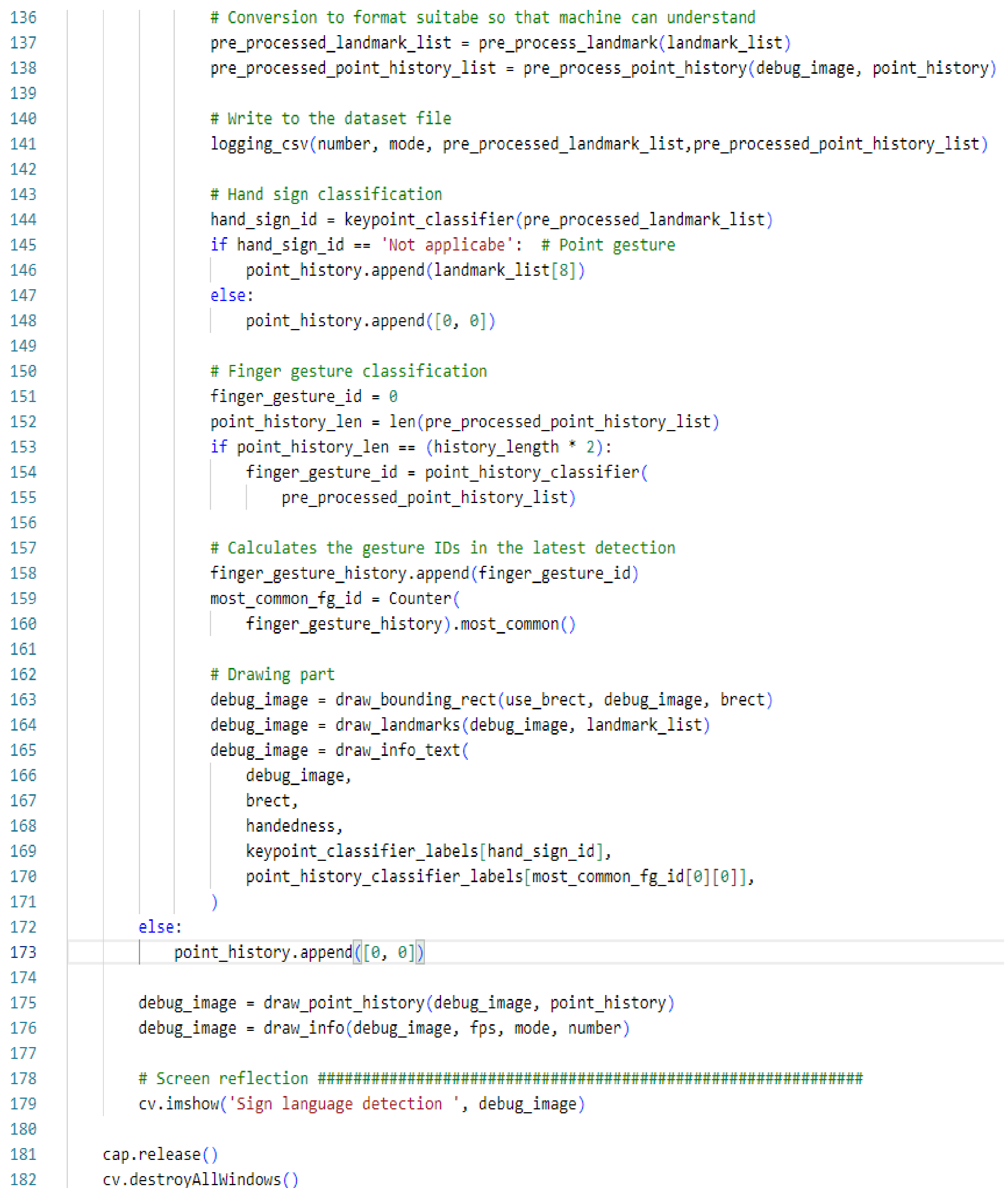
```python
def main():
    # Argument parsing ####
    args = get_args()

    cap_device = args.device
    cap_width = args.width
    cap_height = args.height

    use_static_image_mode = args.use_static_image_mode
    min_detection_confidence = args.min_detection_confidence
    min_tracking_confidence = args.min_tracking_confidence

    use_brect = True

    # Camera ###
    cap = cv.VideoCapture(cap_device)
    cap.set(cv.CAP_PROP_FRAME_WIDTH, cap_width)
    cap.set(cv.CAP_PROP_FRAME_HEIGHT, cap_height)

    # loading hand landmark modelude ###
    mp_hands = mp.solutions.hands
    hands = mp_hands.Hands(
        static_image_mode=use_static_image_mode,
        max_num_hands=2,
        min_detection_confidence=min_detection_confidence,
        min_tracking_confidence=min_tracking_confidence,
    )

    keypoint_classifier = KeyPointClassifier()

    point_history_classifier = PointHistoryClassifier()

    # Read labels ####
    with open('model/keypoint_classifier/keypoint_classifier_label.csv',
              encoding='utf-8-sig') as f:
        keypoint_classifier_labels = csv.reader(f)
        keypoint_classifier_labels = [
            row[0] for row in keypoint_classifier_labels
        ]
    with open(
            'model/point_history_classifier/point_history_classifier_label.csv',
            encoding='utf-8-sig') as f:
        point_history_classifier_labels = csv.reader(f)
        point_history_classifier_labels = [
            row[0] for row in point_history_classifier_labels
        ]
```

(Fig 5)



```python
# FPS Measure ###
cvFpsCalc = CvFpsCalc(buffer_len=10)

# Coordinate history ###
history_length = 16
point_history = deque(maxlen=history_length)

# Finger gesture history #####
finger_gesture_history = deque(maxlen=history_length)

#  ########################################################################
mode = 0

while True:
    fps = cvFpsCalc.get()

    # Process Key (ESC: end) ####
    key = cv.waitKey(10)
    if key == 27:  # ESC
        break
    number, mode = select_mode(key, mode)
    # Camera capture ###
    ret, image = cap.read()
    if not ret:
        break
    image = cv.flip(image, 1)  # Mirror display
    debug_image = copy.deepcopy(image)

    # Detection implementation ####
    image = cv.cvtColor(image, cv.COLOR_BGR2RGB)

    image.flags.writeable = False
    results = hands.process(image)
    image.flags.writeable = True

    #  ###
    if results.multi_hand_landmarks is not None:
        for hand_landmarks, handedness in zip(results.multi_hand_landmarks,results.multi_handedness):
            #print(hand_landmarks)

            # Bounding box calculation
            brect = calc_bounding_rect(debug_image, hand_landmarks)
            # Landmark calculation
            landmark_list = calc_landmark_list(debug_image, hand_landmarks)

            #print(landmark_list[0])
```

(Fig 5)



```
                # Conversion to format suitabe so that machine can understand
                pre_processed_landmark_list = pre_process_landmark(landmark_list)
                pre_processed_point_history_list = pre_process_point_history(debug_image, point_history)

                # Write to the dataset file
                logging_csv(number, mode, pre_processed_landmark_list,pre_processed_point_history_list)

                # Hand sign classification
                hand_sign_id = keypoint_classifier(pre_processed_landmark_list)
                if hand_sign_id == 'Not applicabe':  # Point gesture
                    point_history.append(landmark_list[8])
                else:
                    point_history.append([0, 0])

                # Finger gesture classification
                finger_gesture_id = 0
                point_history_len = len(pre_processed_point_history_list)
                if point_history_len == (history_length * 2):
                    finger_gesture_id = point_history_classifier(
                        pre_processed_point_history_list)

                # Calculates the gesture IDs in the latest detection
                finger_gesture_history.append(finger_gesture_id)
                most_common_fg_id = Counter(
                    finger_gesture_history).most_common()

                # Drawing part
                debug_image = draw_bounding_rect(use_brect, debug_image, brect)
                debug_image = draw_landmarks(debug_image, landmark_list)
                debug_image = draw_info_text(
                    debug_image,
                    brect,
                    handedness,
                    keypoint_classifier_labels[hand_sign_id],
                    point_history_classifier_labels[most_common_fg_id[0][0]],
                )
        else:
            point_history.append([0, 0])

        debug_image = draw_point_history(debug_image, point_history)
        debug_image = draw_info(debug_image, fps, mode, number)

        # Screen reflection #############################################################
        cv.imshow('Sign language detection ', debug_image)

    cap.release()
    cv.destroyAllWindows()
```

(Fig 5)

(Fig 5) The **main()** function is the core of the script. Here's a breakdown of its functionality:

1. **Argument Parsing**: It retrieves command-line arguments using the **get_args()** function.
2. **Camera Initialization**: It sets up the camera capture, specifying the device index (**cap_device**), width (**cap_width**), and height (**cap_height**). The settings are applied to the video capture using OpenCV.
3. **Model Loading**: It loads the hand landmark model using MediaPipe. Two classifiers are also instantiated: **KeyPointClassifier** and **PointHistoryClassifier**.
4. **Label Loading**: It reads the labels for the classifiers from CSV files.
5. **FPS Measurement**: It initializes an object (**cvFpsCalc**) for measuring frames per second.
6. **Coordinate History and Gesture History**: It initializes deque objects for storing coordinate history and finger gesture history.
7. **Main Loop**: The script enters a loop where it continuously captures frames from the camera until the user presses the ESC key.
8. **Key Processing**: Inside the loop, it processes any key inputs. Pressing ESC breaks the loop, ending the program.
9. **Frame Capture**: It captures a frame from the camera and flips it horizontally for mirror display.
10. **Hand Detection**: It processes the captured image to detect hands using the loaded hand landmark model.
11. **Hand Landmarks Processing**: For each detected hand, it calculates the bounding box, extracts landmarks, preprocesses them, and logs them to a dataset file.
12. **Hand Sign Classification**: It classifies hand signs using the **KeyPointClassifier** and updates the point history.

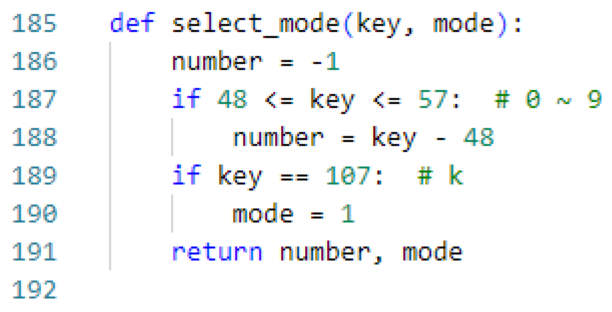
13. **Finger Gesture Classification**: It classifies finger gestures using the **PointHistoryClassifier** based on the point history.

14. **Drawing**: It draws bounding boxes, landmarks, and information text on the frame.

15. **Screen Display**: It displays the annotated frame with hand landmarks, bounding boxes, and information text.

16. **Cleanup**: After the loop ends, it releases the camera and closes all OpenCV windows.

    This script seems to be for sign language detection using hand landmarks and finger gestures, where the hand movements are captured from a camera feed and analyzed in real-time.



```python
def select_mode(key, mode):
    number = -1
    if 48 <= key <= 57:  # 0 ~ 9
        number = key - 48
    if key == 107:  # k
        mode = 1
    return number, mode
```

(Fig 6)

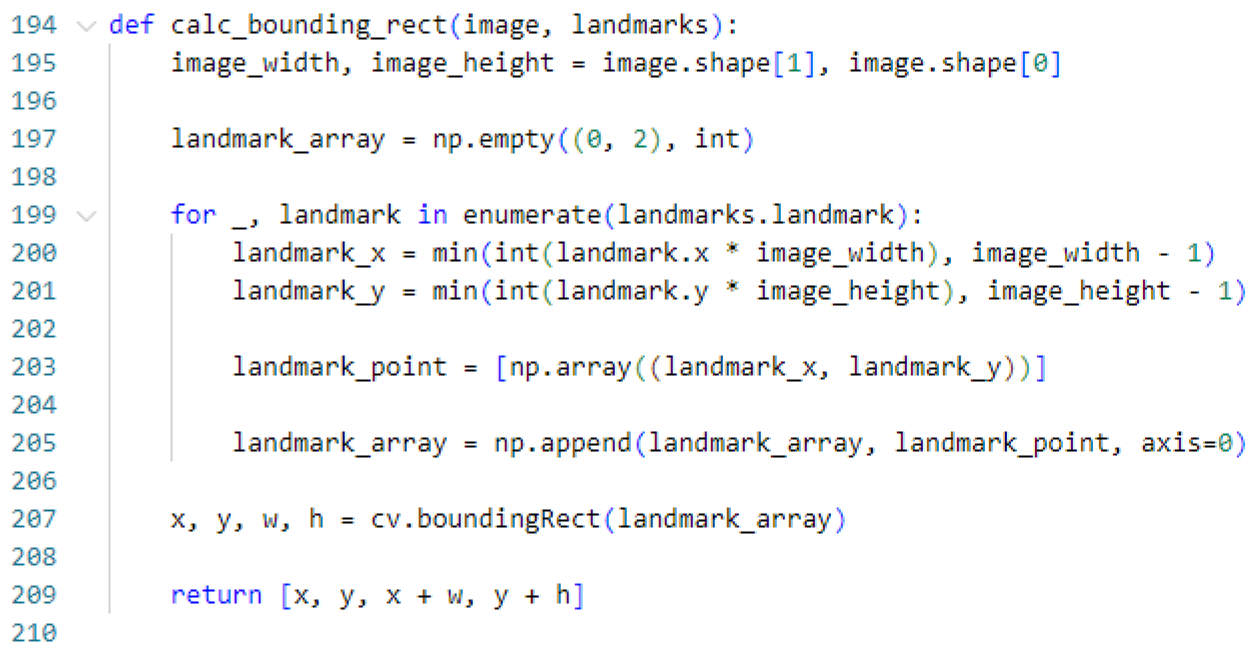
(Fig 6) The **select_mode** function takes a key input and the current mode as parameters and returns the selected number and updated mode. Here's how it works:

1. **Number Selection**: If the key pressed is a number between 0 and 9 (ASCII values 48 to 57), it calculates the corresponding number by subtracting 48 from the key value.

2. **Mode Selection**: If the key pressed is 'k' (ASCII value 107), it switches the mode to 1.

3. **Return Values**: It returns the selected number and the updated mode.

    This function allows users to select a number (0-9) or switch the mode by pressing the 'k' key.



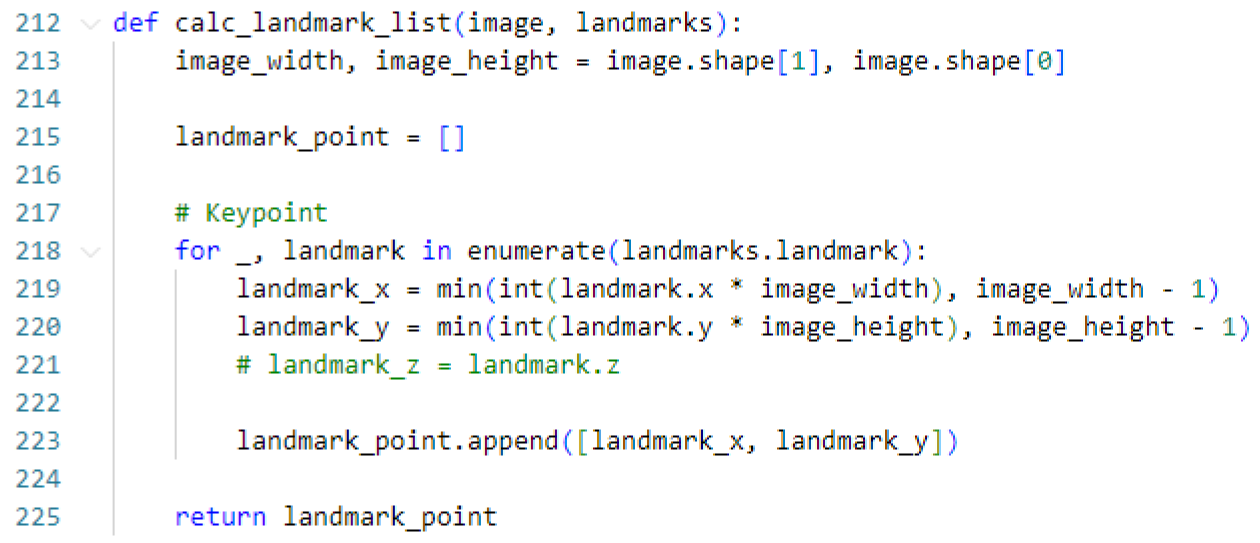
```python
def calc_bounding_rect(image, landmarks):
    image_width, image_height = image.shape[1], image.shape[0]

    landmark_array = np.empty((0, 2), int)

    for _, landmark in enumerate(landmarks.landmark):
        landmark_x = min(int(landmark.x * image_width), image_width - 1)
        landmark_y = min(int(landmark.y * image_height), image_height - 1)

        landmark_point = [np.array((landmark_x, landmark_y))]

        landmark_array = np.append(landmark_array, landmark_point, axis=0)

    x, y, w, h = cv.boundingRect(landmark_array)

    return [x, y, x + w, y + h]

```

(Fig 7)

(Fig 7) The **calc_bounding_rect** function calculates the bounding rectangle around a set of landmarks on an image. Here's how it works:

1. **Input Parameters**: It takes the image and landmarks detected by the hand landmark model as input.
2. **Landmark Processing**: It iterates over each landmark detected and converts its normalized coordinates to pixel coordinates based on the dimensions of the image.
3. **Bounding Rectangle Calculation**: It then computes the bounding rectangle using the **cv.boundingRect** function, which takes a set of points and returns the coordinates of the top-left corner (x, y) and the width and height (w, h) of the rectangle.
4. **Return**: The function returns a list containing the coordinates of the top-left and bottom-right corners of the bounding rectangle: [x1, y1, x2, y2].

This function essentially encapsulates the logic to calculate the bounding rectangle around a set of landmarks on an image.



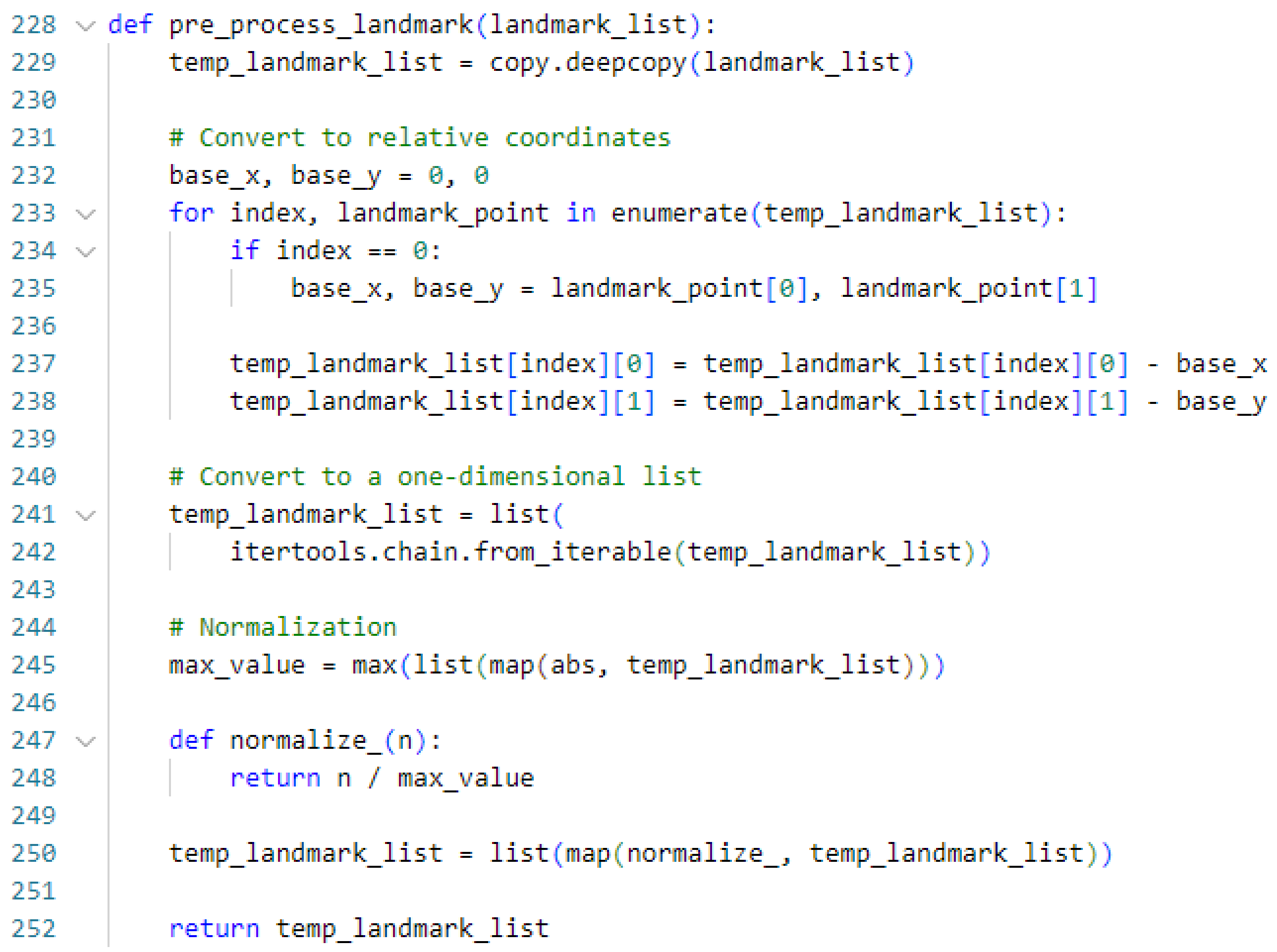
```python
def calc_landmark_list(image, landmarks):
    image_width, image_height = image.shape[1], image.shape[0]

    landmark_point = []

    # Keypoint
    for _, landmark in enumerate(landmarks.landmark):
        landmark_x = min(int(landmark.x * image_width), image_width - 1)
        landmark_y = min(int(landmark.y * image_height), image_height - 1)
        # landmark_z = landmark.z

        landmark_point.append([landmark_x, landmark_y])

    return landmark_point
```

(Fig 8)

(Fig 8) The **calc_landmark_list** function computes a list of landmark points detected in an image. Here's how it works:

1. **Input Parameters**: It takes the image and landmarks detected by the hand landmark model as input.
2. **Image Dimensions**: It retrieves the width and height of the input image.
3. **Landmark Processing**: It iterates over each detected landmark and converts its normalized coordinates to pixel coordinates based on the dimensions of the image. These coordinates represent the location of each landmark point on the image.
4. **List Construction**: It constructs a list of landmark points, where each point is represented as a list **[x, y]** containing the pixel coordinates of the landmark.
5. **Return**: The function returns the list of landmark points.

This function essentially transforms the normalized landmark coordinates into pixel coordinates and organizes them into a list for further processing.



```python
def pre_process_landmark(landmark_list):
    temp_landmark_list = copy.deepcopy(landmark_list)

    # Convert to relative coordinates
    base_x, base_y = 0, 0
    for index, landmark_point in enumerate(temp_landmark_list):
        if index == 0:
            base_x, base_y = landmark_point[0], landmark_point[1]

        temp_landmark_list[index][0] = temp_landmark_list[index][0] - base_x
        temp_landmark_list[index][1] = temp_landmark_list[index][1] - base_y

    # Convert to a one-dimensional list
    temp_landmark_list = list(
        itertools.chain.from_iterable(temp_landmark_list))

    # Normalization
    max_value = max(list(map(abs, temp_landmark_list)))

    def normalize_(n):
        return n / max_value

    temp_landmark_list = list(map(normalize_, temp_landmark_list))

    return temp_landmark_list
```

(Fig 9)

(Fig 9) The **pre_process_landmark** function performs several preprocessing steps on a list of landmark points. Here's a breakdown of what it does:

1. **Deep Copy**: It creates a deep copy of the input **landmark_list** to avoid modifying the original list.

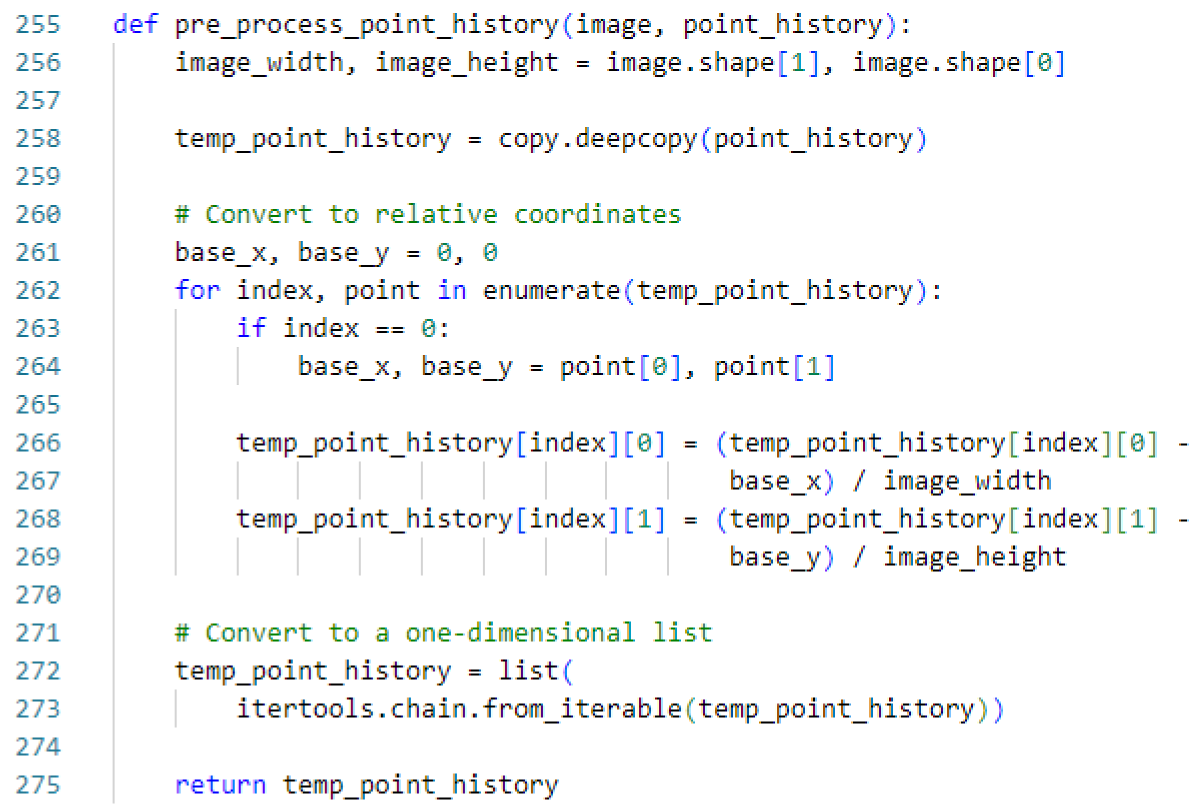
2. **Relative Coordinates**: It calculates the base point (the first landmark point) and converts all landmark points to relative coordinates by subtracting the base point's coordinates from each point's coordinates. This step centers the hand around the origin.

3. **Flattening**: It flattens the list of lists into a one-dimensional list using **itertools.chain.from_iterable**. This step simplifies the data structure for further processing.

4. **Normalization**: It normalizes each coordinate value by dividing it by the maximum absolute value in the list. This step scales the coordinates to a range between -1 and 1.

5. **Return**: The function returns the preprocessed landmark list.

   Overall, this function prepares the landmark data for input into a machine learning model by centering the hand, flattening the list, and normalizing the coordinates.



```python
def pre_process_point_history(image, point_history):
    image_width, image_height = image.shape[1], image.shape[0]

    temp_point_history = copy.deepcopy(point_history)

    # Convert to relative coordinates
    base_x, base_y = 0, 0
    for index, point in enumerate(temp_point_history):
        if index == 0:
            base_x, base_y = point[0], point[1]

        temp_point_history[index][0] = (temp_point_history[index][0] -
                                        base_x) / image_width
        temp_point_history[index][1] = (temp_point_history[index][1] -
                                        base_y) / image_height

    # Convert to a one-dimensional list
    temp_point_history = list(
        itertools.chain.from_iterable(temp_point_history))

    return temp_point_history
```

(Fig 10)

(Fig 10) The **pre_process_point_history** function preprocesses a list of point history by performing the following steps:

1. **Deep Copy**: Creates a deep copy of the input **point_history** to avoid modifying the original list.

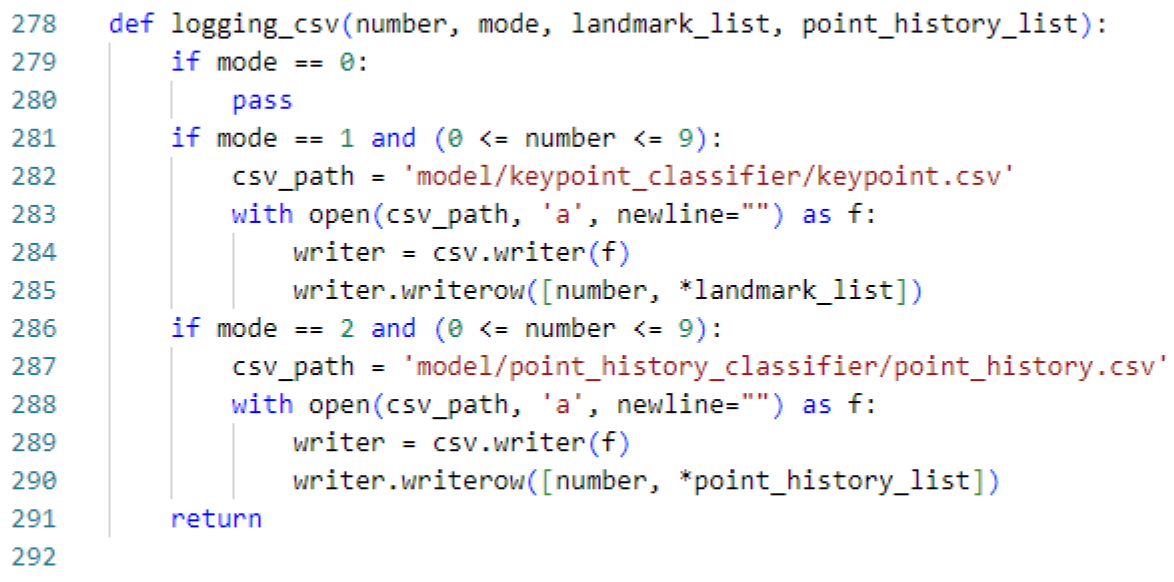
2. **Relative Coordinates**: Calculates the base point (the first point) and converts all points to relative coordinates by subtracting the base point's coordinates from each point's

coordinates. Additionally, it divides the coordinates by the image width and height to normalize them between 0 and 1.

3. **Flattening**: Flattens the list of lists into a one-dimensional list using **itertools.chain.from_iterable**. This step simplifies the data structure for further processing.

4. **Return**: Returns the preprocessed point history as a one-dimensional list.

This function prepares the point history data for input into a machine learning model by centering the points around the origin, normalizing the coordinates, and flattening the list.



```python
def logging_csv(number, mode, landmark_list, point_history_list):
    if mode == 0:
        pass
    if mode == 1 and (0 <= number <= 9):
        csv_path = 'model/keypoint_classifier/keypoint.csv'
        with open(csv_path, 'a', newline="") as f:
            writer = csv.writer(f)
            writer.writerow([number, *landmark_list])
    if mode == 2 and (0 <= number <= 9):
        csv_path = 'model/point_history_classifier/point_history.csv'
        with open(csv_path, 'a', newline="") as f:
            writer = csv.writer(f)
            writer.writerow([number, *point_history_list])
    return

```

(Fig 11)

(Fig 11) The **logging_csv** function is responsible for logging data into CSV files based on the mode and number provided. Here's how it works:

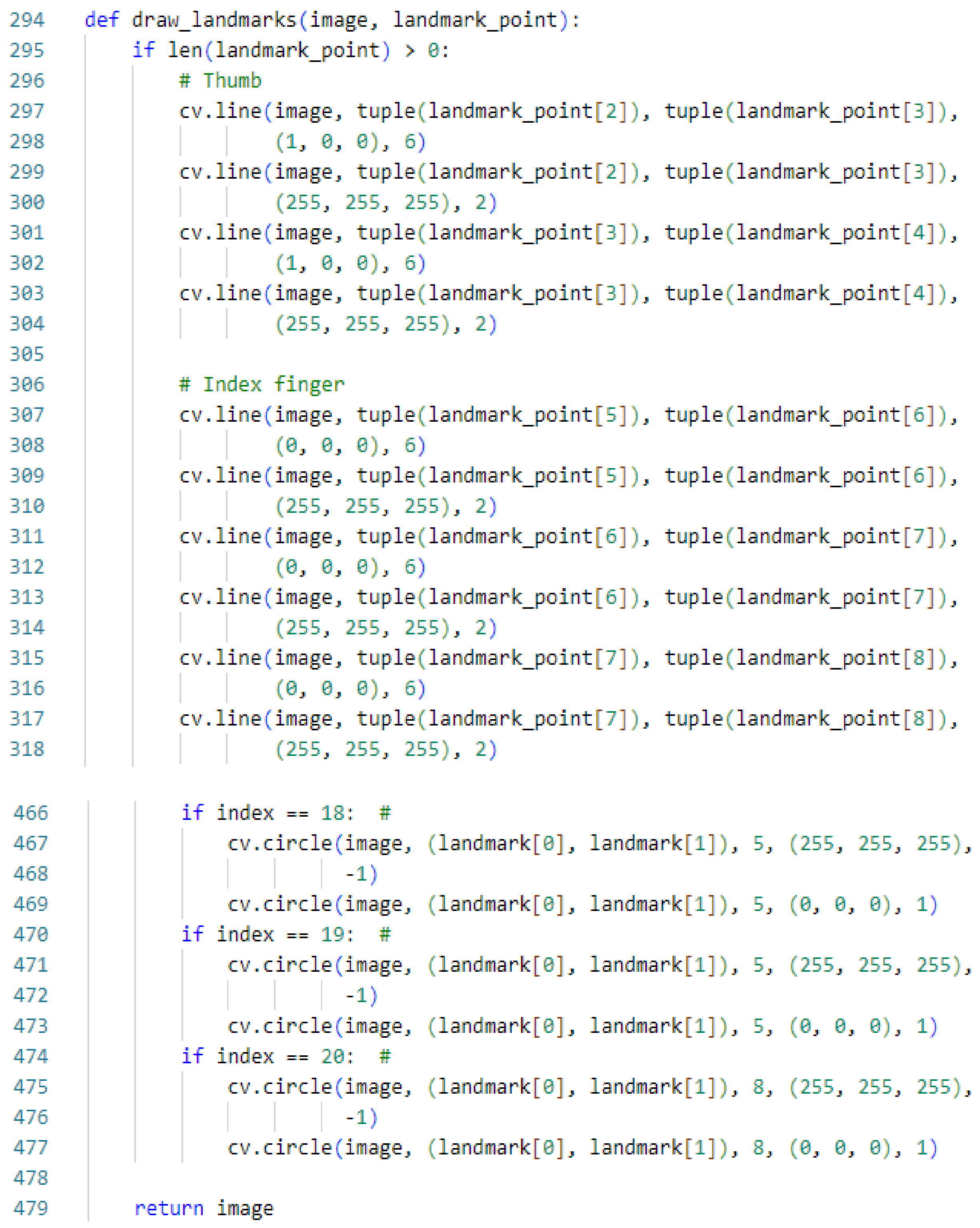
Parameters:

- **number**: An integer representing a numeric label for the data.
- **mode**: An integer representing the mode of operation.
- **landmark_list**: A list containing landmark data.
- **point_history_list**: A list containing point history data.

Logic:

- If the mode is **0**, the function does nothing.
- If the mode is **1** and the number is between 0 and 9 (inclusive), the function appends the **number** and the **landmark_list** to a CSV file located at **'model/keypoint_classifier/keypoint.csv'**.
- If the mode is **2** and the number is between 0 and 9 (inclusive), the function appends the **number** and the **point_history_list** to a CSV file located at **'model/point_history_classifier/point_history.csv'**.
- **Return**: The function doesn't return anything (**None**).

  This function provides a mechanism to log data into separate CSV files based on the mode of operation and the numeric label associated with the data.



```python
def draw_landmarks(image, landmark_point):
    if len(landmark_point) > 0:
        # Thumb
        cv.line(image, tuple(landmark_point[2]), tuple(landmark_point[3]),
                (1, 0, 0), 6)
        cv.line(image, tuple(landmark_point[2]), tuple(landmark_point[3]),
                (255, 255, 255), 2)
        cv.line(image, tuple(landmark_point[3]), tuple(landmark_point[4]),
                (1, 0, 0), 6)
        cv.line(image, tuple(landmark_point[3]), tuple(landmark_point[4]),
                (255, 255, 255), 2)

        # Index finger
        cv.line(image, tuple(landmark_point[5]), tuple(landmark_point[6]),
                (0, 0, 0), 6)
        cv.line(image, tuple(landmark_point[5]), tuple(landmark_point[6]),
                (255, 255, 255), 2)
        cv.line(image, tuple(landmark_point[6]), tuple(landmark_point[7]),
                (0, 0, 0), 6)
        cv.line(image, tuple(landmark_point[6]), tuple(landmark_point[7]),
                (255, 255, 255), 2)
        cv.line(image, tuple(landmark_point[7]), tuple(landmark_point[8]),
                (0, 0, 0), 6)
        cv.line(image, tuple(landmark_point[7]), tuple(landmark_point[8]),
                (255, 255, 255), 2)
```

```python
        if index == 18:  #
            cv.circle(image, (landmark[0], landmark[1]), 5, (255, 255, 255),
                      -1)
            cv.circle(image, (landmark[0], landmark[1]), 5, (0, 0, 0), 1)
        if index == 19:  #
            cv.circle(image, (landmark[0], landmark[1]), 5, (255, 255, 255),
                      -1)
            cv.circle(image, (landmark[0], landmark[1]), 5, (0, 0, 0), 1)
        if index == 20:  #
            cv.circle(image, (landmark[0], landmark[1]), 8, (255, 255, 255),
                      -1)
            cv.circle(image, (landmark[0], landmark[1]), 8, (0, 0, 0), 1)

    return image
```

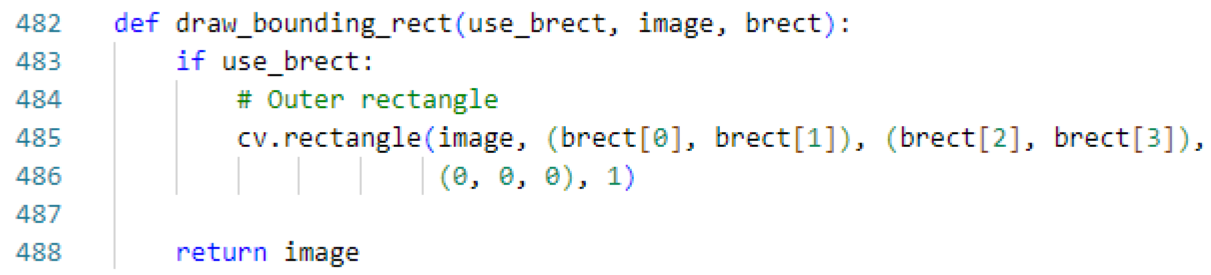
(Fig 12)

(Fig 12) The **draw_landmarks** function is responsible for drawing landmarks on the image. It takes an image and a list of landmark points as input and draws lines and circles to represent hand landmarks.

Here's a breakdown of how it works:

- **Parameters**:
- **image**: The image on which the landmarks will be drawn.
- **landmark_point**: A list containing landmark points.
- **Logic**:
- For each finger:
- It draws lines between consecutive landmark points to represent the fingers.
- It draws circles at each landmark point to highlight them.
- Additionally, it draws lines connecting specific landmarks to represent the palm and fingers.
- **Return**: The function returns the image with the landmarks drawn on it.

This function plays a crucial role in visualizing the hand landmarks detected by the model on the image.



```
def draw_bounding_rect(use_brect, image, brect):
    if use_brect:
        # Outer rectangle
        cv.rectangle(image, (brect[0], brect[1]), (brect[2], brect[3]),
                     (0, 0, 0), 1)

    return image
```

(Fig 13)

(Fig 13) The **draw_bounding_rect** function is responsible for drawing a bounding rectangle around the detected hand region if the **use_brect** flag is set to **True**.

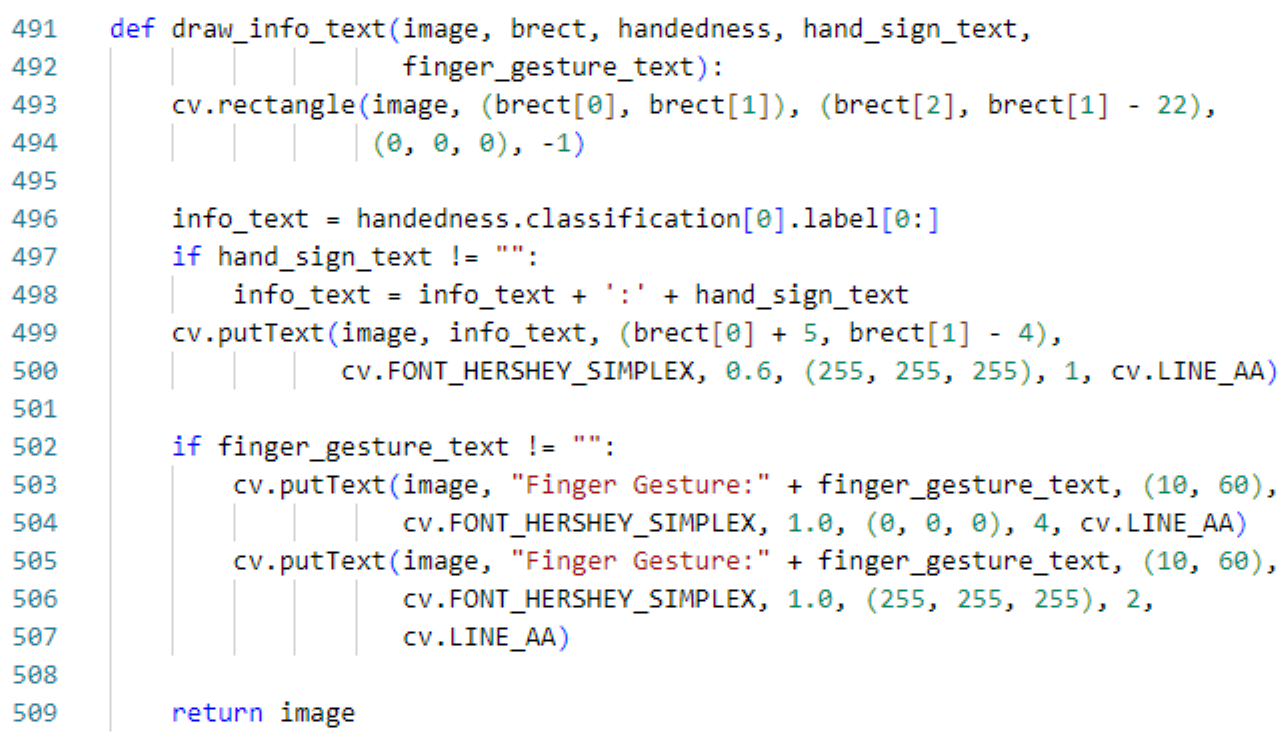
Parameters:

- **use_brect**: A boolean flag indicating whether to draw the bounding rectangle.
- **image**: The image on which the bounding rectangle will be drawn.
- **brect**: A list containing the coordinates of the bounding rectangle in the format **[x1, y1, x2, y2]**, where **(x1, y1)** represents the top-left corner and **(x2, y2)** represents the bottom-right corner.

Logic:

- If **use_brect** is **True**, it draws a rectangle on the image using the coordinates provided in **brect**.
- The color of the rectangle is **(0, 0, 0)** (black), and the thickness is set to **1**.
- **Return**: The function returns the image with or without the bounding rectangle, depending on the value of **use_brect**.

  This function is useful for visually indicating the region of interest (hand) detected by the model on the image.



```python
def draw_info_text(image, brect, handedness, hand_sign_text,
                   finger_gesture_text):
    cv.rectangle(image, (brect[0], brect[1]), (brect[2], brect[1] - 22),
                 (0, 0, 0), -1)

    info_text = handedness.classification[0].label[0:]
    if hand_sign_text != "":
        info_text = info_text + ':' + hand_sign_text
    cv.putText(image, info_text, (brect[0] + 5, brect[1] - 4),
               cv.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 1, cv.LINE_AA)

    if finger_gesture_text != "":
        cv.putText(image, "Finger Gesture:" + finger_gesture_text, (10, 60),
                   cv.FONT_HERSHEY_SIMPLEX, 1.0, (0, 0, 0), 4, cv.LINE_AA)
        cv.putText(image, "Finger Gesture:" + finger_gesture_text, (10, 60),
                   cv.FONT_HERSHEY_SIMPLEX, 1.0, (255, 255, 255), 2,
                   cv.LINE_AA)

    return image
```

(Fig 14)

(Fig 14) The **draw_info_text** function is responsible for drawing text information onto the image. Here's a breakdown of its functionality:

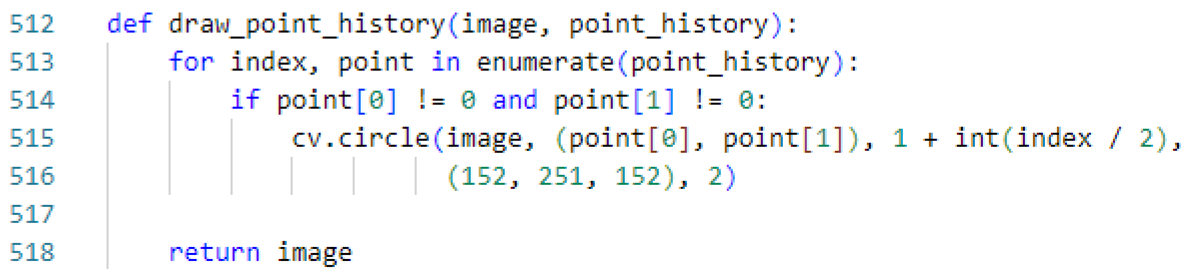
Parameters:

- **image**: The image onto which the text will be drawn.
- **brect**: A list containing the coordinates of the bounding rectangle around the detected hand region.
- **handedness**: The classification result containing information about the handedness (left or right hand).
- **hand_sign_text**: The text representing the detected hand sign.
- **finger_gesture_text**: The text representing the detected finger gesture.

Logic:

- It draws a filled rectangle at the top of the bounding rectangle using the color **(0, 0, 0)** (black).
- It constructs the **info_text** by concatenating the handedness classification label and the **hand_sign_text**.
- It draws the **info_text** at an offset from the top-left corner of the bounding rectangle.
- If **finger_gesture_text** is not empty, it draws additional text indicating the detected finger gesture at coordinates **(10, 60)**.
- The text is drawn twice: first with a thicker black outline to improve visibility (**thickness=4**), and then with the actual content in white (**thickness=2**).
- **Return**: The function returns the image with the drawn text.

  This function is useful for displaying information about the detected hand, including its handedness, detected hand sign, and finger gesture.



```python
def draw_point_history(image, point_history):
    for index, point in enumerate(point_history):
        if point[0] != 0 and point[1] != 0:
            cv.circle(image, (point[0], point[1]), 1 + int(index / 2),
                      (152, 251, 152), 2)

    return image
```

(Fig 15)

(Fig 15) The **draw_point_history** function is responsible for visualizing the history of hand movement by drawing circles on the image corresponding to the points in the **point_history** list:

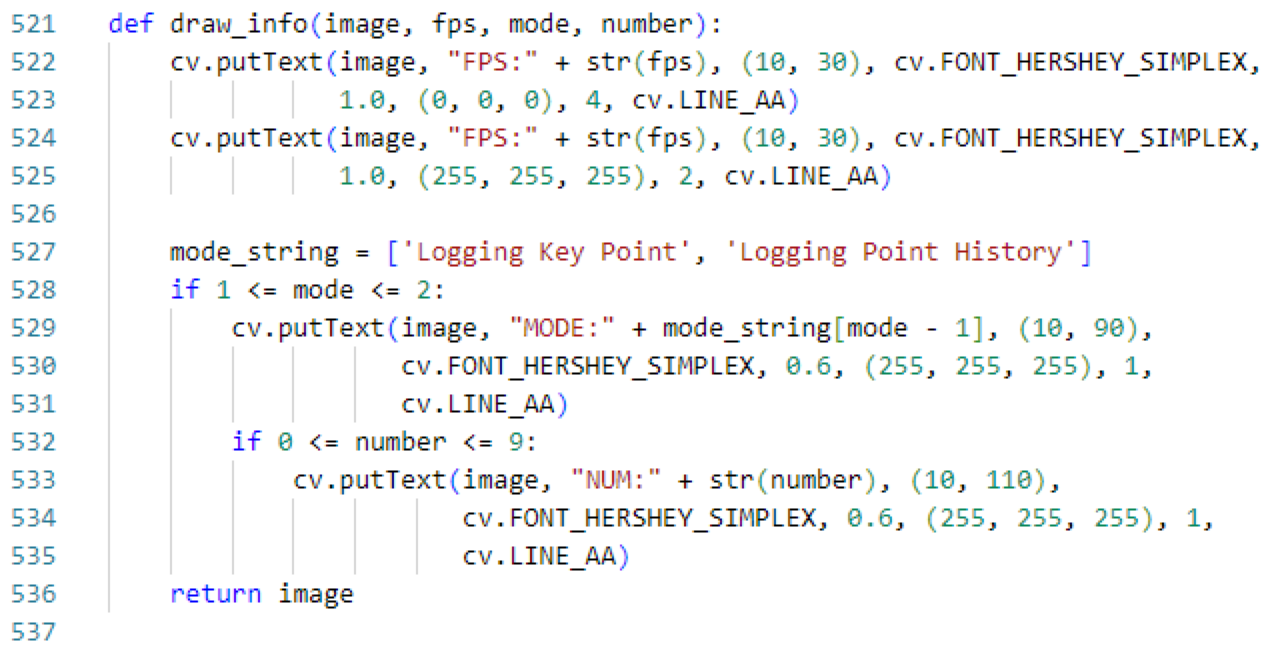
Parameters:

- **image**: The image onto which the points will be drawn.
- **point_history**: A list containing historical points representing hand movement.

Logic:

- It iterates over each point in the **point_history**.
- If the point is not the default value **[0, 0]**, indicating a valid hand position:
- It draws a circle on the image at the coordinates of the point.
- The radius of the circle increases gradually with the index of the point in the history, providing a visual indication of the movement history.
- The circles are drawn with a green color **(152, 251, 152)** and a thickness of **2**.
- **Return**: The function returns the modified image with the drawn point history.

  This function is useful for visualizing the trajectory of hand movement over time, providing feedback on the path followed by the hand during gesture detection.



```python
def draw_info(image, fps, mode, number):
    cv.putText(image, "FPS:" + str(fps), (10, 30), cv.FONT_HERSHEY_SIMPLEX,
               1.0, (0, 0, 0), 4, cv.LINE_AA)
    cv.putText(image, "FPS:" + str(fps), (10, 30), cv.FONT_HERSHEY_SIMPLEX,
               1.0, (255, 255, 255), 2, cv.LINE_AA)

    mode_string = ['Logging Key Point', 'Logging Point History']
    if 1 <= mode <= 2:
        cv.putText(image, "MODE:" + mode_string[mode - 1], (10, 90),
                   cv.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 1,
                   cv.LINE_AA)
        if 0 <= number <= 9:
            cv.putText(image, "NUM:" + str(number), (10, 110),
                       cv.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 1,
                       cv.LINE_AA)
    return image
```

(Fig 16)

(Fig 16) The **draw_info** function adds textual information to the image, including the frames per second (FPS), the current mode, and the selected number (if applicable). Here's a breakdown of its functionality:

Parameters:

- **image**: The image onto which the text will be drawn.
- **fps**: The frames per second value.
- **mode**: The current mode of operation.
- **number**: The selected number (if applicable).

Logic:

- It adds text displaying the FPS value at the top left corner of the image.
- It adds text indicating the current mode of operation, such as "Logging Key Point" or "Logging Point History", below the FPS information.
- If the mode is between 1 and 2 (inclusive) and a number between 0 and 9 has been selected:

- It adds text displaying the selected number below the mode information.
- **Return**: The function returns the modified image with the added textual information.

  This function provides essential information overlaid on the image, aiding in monitoring the application's performance and current state during execution.

1. **Setup and Configuration**:

- Imports necessary libraries like OpenCV, NumPy, and MediaPipe.
- Defines command-line arguments for configuring camera settings and detection thresholds.

2. **Main Functionality**:

- Initializes camera capture and loads the hand landmark model from MediaPipe.
- Sets up custom classifiers for keypoint detection and point history analysis.
- Measures frames per second (FPS) for performance monitoring.

3. **Processing Loop**:

- Enters a continuous loop to capture frames, detect hand landmarks, and analyze gestures.
- Handles key presses to switch between logging modes and specify labels for data logging.

4. **Hand Landmark Detection**:

- Processes each frame to detect hand landmarks using the MediaPipe hand tracking model.
- Calculates bounding rectangles around detected hands for further analysis.

5. **Data Preprocessing**:

- Preprocesses detected landmarks and point history data for input to the classifiers.

6. **Gesture Classification**:

- Classifies hand signs and finger gestures using custom-trained classifiers.
- Logs the preprocessed data to CSV files based on the selected logging mode and labels.

7. **Visualization**:

- Draws bounding rectangles, landmarks, and textual information on the frame for visualization.
- Displays the processed frame with overlays for real-time monitoring.

8. **Cleanup**:

- Releases the camera resources and closes OpenCV windows when the script terminates.

Overall, this script provides a comprehensive pipeline for real-time sign language detection, combining hand landmark detection, gesture analysis, and visualization in a user-friendly interface.

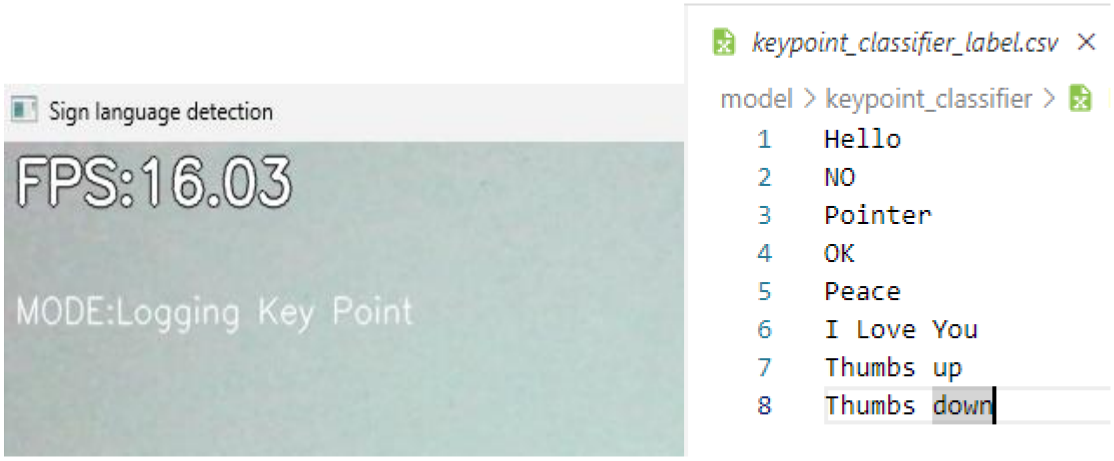
5.3 Creating DataSet .CSV

When you press the 'k' key while the script is running, it triggers a change in the mode of operation.

(Fig 17) Mode Change: The variable **mode** is toggled between two modes: "Logging Key Point" and "Logging Point History". Each time you press 'k', it cycles between these two modes.

Visual Feedback: The current mode is displayed on the screen, indicating whether the script is currently logging key points or point history data.

This functionality allows you to switch between different modes of data logging, depending on your specific requirements or experiments.

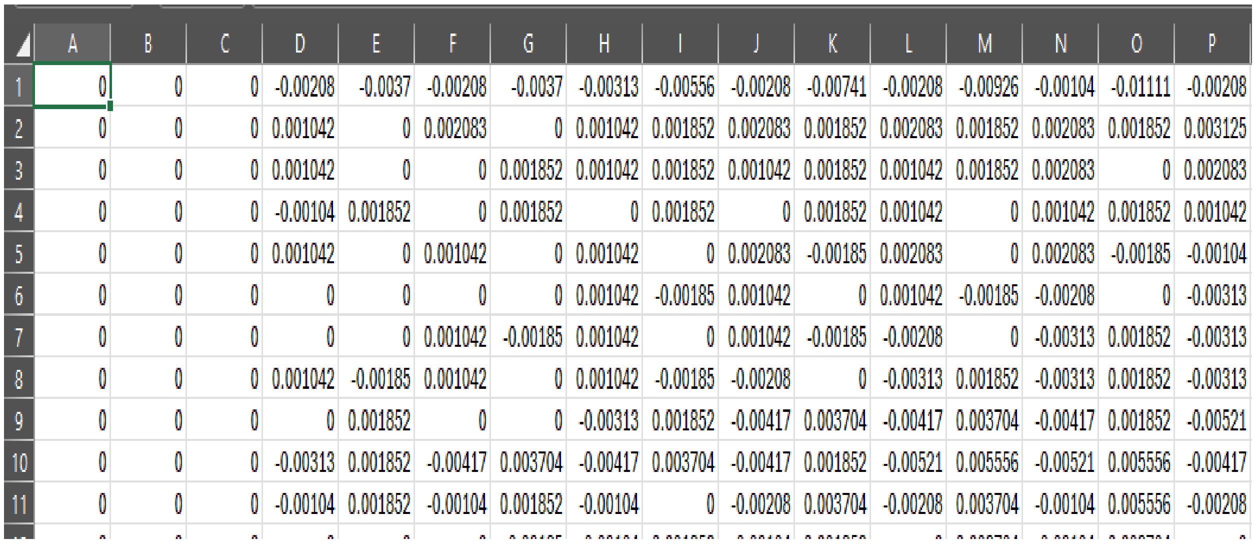


```
FPS:16.03
MODE:Logging Key Point
```

```
Hello
NO
Pointer
OK
Peace
I Love You
Thumbs up
Thumbs down
```

(Fig 17)

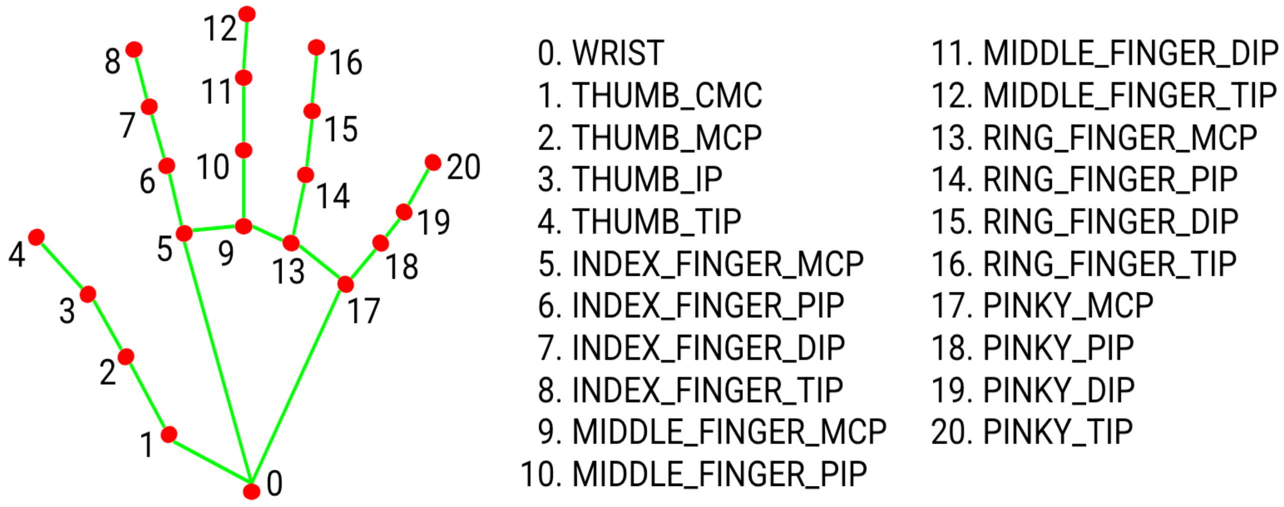
If you press "0" to "9", the key points will be added to "model/keypoint_classifier/keypoint.csv" as shown below. 1st column: Pressed number (used as class ID), 2nd and subsequent columns: Key point coordinates



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	0	0	0	-0.00208	-0.0037	-0.00208	-0.0037	-0.00313	-0.00556	-0.00208	-0.00741	-0.00208	-0.00926	-0.00104	-0.01111	-0.00208
2	0	0	0	0.001042	0	0.002083	0	0.001042	0.001852	0.002083	0.001852	0.002083	0.001852	0.002083	0.001852	0.003125
3	0	0	0	0.001042	0	0	0.001852	0.001042	0.001852	0.001042	0.001852	0.001042	0.001852	0.002083	0	0.002083
4	0	0	0	-0.00104	0.001852	0	0.001852	0	0.001852	0	0.001852	0.001042	0	0.001042	0.001852	0.001042
5	0	0	0	0.001042	0	0.001042	0	0.001042	0	0.002083	-0.00185	0.002083	0	0.002083	-0.00185	-0.00104
6	0	0	0	0	0	0	0	0.001042	-0.00185	0.001042	0	0.001042	-0.00185	-0.00208	0	-0.00313
7	0	0	0	0	0	0.001042	-0.00185	0.001042	0	0.001042	-0.00185	-0.00208	0	-0.00313	0.001852	-0.00313
8	0	0	0	0.001042	-0.00185	0.001042	0	0.001042	-0.00185	-0.00208	0	-0.00313	0.001852	-0.00313	0.001852	-0.00313
9	0	0	0	0	0.001852	0	0	-0.00313	0.001852	-0.00417	0.003704	-0.00417	0.003704	-0.00417	0.001852	-0.00521
10	0	0	0	-0.00313	0.001852	-0.00417	0.003704	-0.00417	0.003704	-0.00417	0.001852	-0.00521	0.005556	-0.00521	0.005556	-0.00417
11	0	0	0	-0.00104	0.001852	-0.00104	0.001852	-0.00104	0	-0.00208	0.003704	-0.00208	0.003704	-0.00104	0.005556	-0.00208

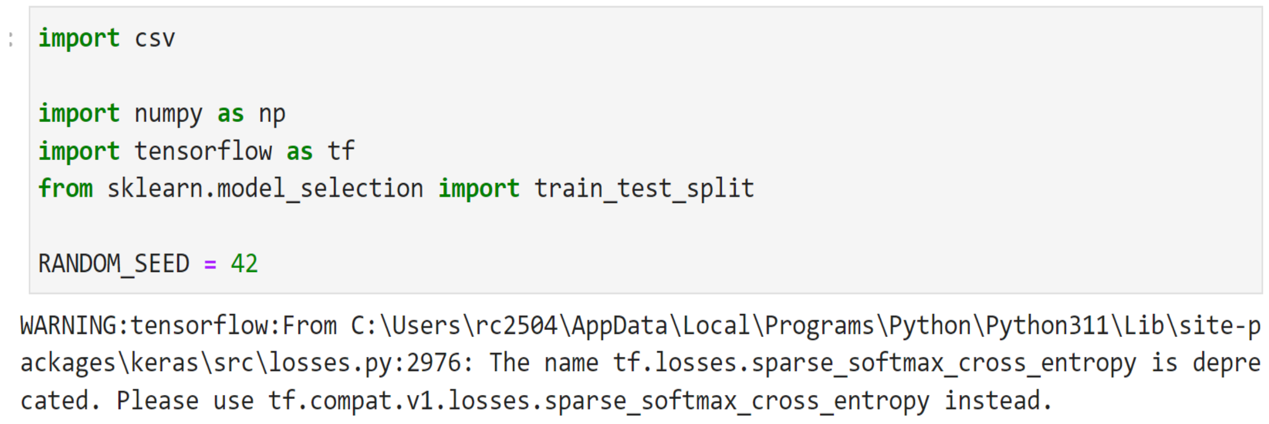
(Fig 18)

The key point coordinates are the ones that have undergone the following preprocessing up to ④.



(Fig 19)

5.4. Model Training

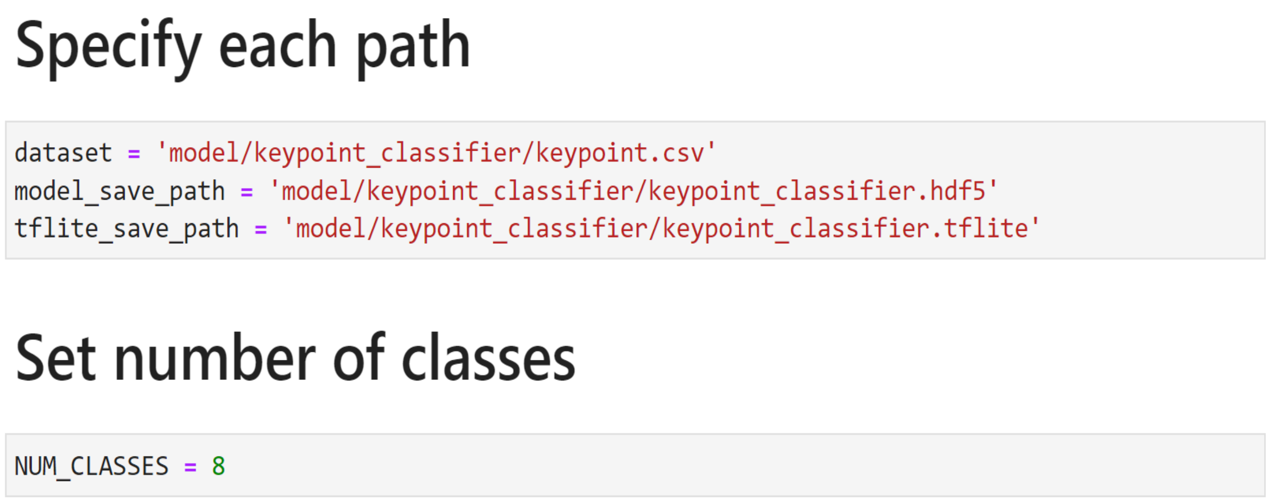


```
import csv

import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split

RANDOM_SEED = 42
```

```
WARNING:tensorflow:From C:\Users\rc2504\AppData\Local\Programs\Python\Python311\Lib\site-p
ackages\keras\src\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is depre
cated. Please use tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
```

(Fig 20)

1. **import csv**: This line imports the CSV module, which provides functions for reading and writing CSV files.
2. **import numpy as np**: This line imports the NumPy library and aliases it as **np**. NumPy is a fundamental package for scientific computing with Python, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
3. **import tensorflow as tf**: This line imports TensorFlow, an open-source machine learning framework developed by Google. TensorFlow provides tools for building and training machine learning models, including neural networks.
4. **from sklearn.model_selection import train_test_split**: This line imports the **train_test_split** function from the scikit-learn library. Scikit-learn is a machine learning library in Python that provides simple and efficient tools for data mining and data analysis. The **train_test_split** function is commonly used to split datasets into training and testing subsets for model evaluation.
5. **RANDOM_SEED = 42**: This line sets a random seed to ensure reproducibility. By setting a random seed, you can get the same random numbers each time you run the code, which is useful for debugging and testing.



Specify each path

```
dataset = 'model/keypoint_classifier/keypoint.csv'
model_save_path = 'model/keypoint_classifier/keypoint_classifier.hdf5'
tflite_save_path = 'model/keypoint_classifier/keypoint_classifier.tflite'
```

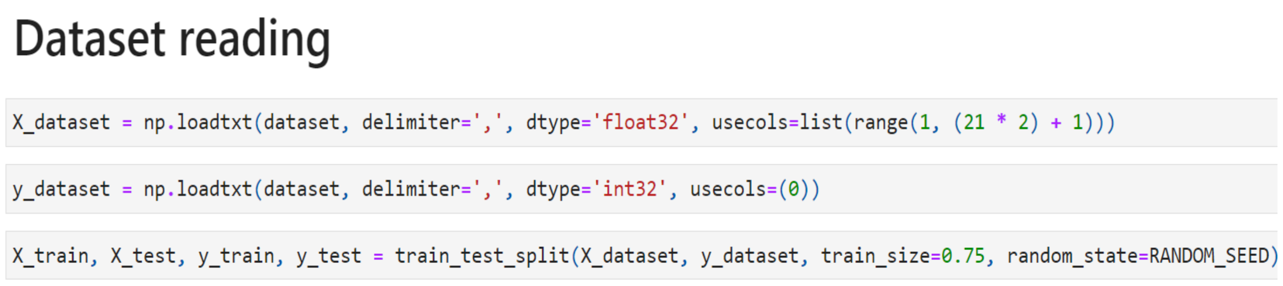
Set number of classes

```
NUM_CLASSES = 8
```

(Fig 21)

1. **dataset**: This path points to the CSV file containing the dataset. In this case, it's **'model/keypoint_classifier/keypoint.csv'**.
2. **model_save_path**: This path indicates where the trained model will be saved. It's set to **'model/keypoint_classifier/keypoint_classifier.hdf5'**.
3. **tflite_save_path**: This path specifies where the TensorFlow Lite model will be saved. It's set to **'model/keypoint_classifier/keypoint_classifier.tflite'**.

   Additionally, the number of classes is specified as **NUM_CLASSES = 8**. This variable determines the number of output classes in the classification model.



Dataset reading

```
X_dataset = np.loadtxt(dataset, delimiter=',', dtype='float32', usecols=list(range(1, (21 * 2) + 1)))
```

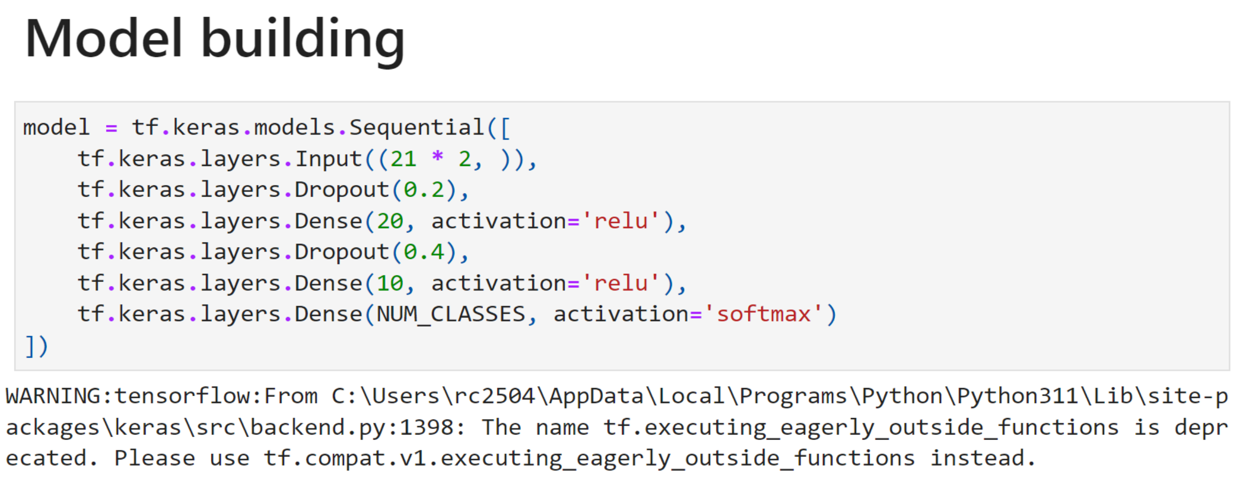
```
y_dataset = np.loadtxt(dataset, delimiter=',', dtype='int32', usecols=(0))
```

```
X_train, X_test, y_train, y_test = train_test_split(X_dataset, y_dataset, train_size=0.75, random_state=RANDOM_SEED)
```

(Fig 22)

1. **X_dataset**: Contains the features of the dataset. It loads columns 1 to 42 (since there are 21 * 2 features for each data point) using **np.loadtxt**.
2. **y_dataset**: Contains the labels of the dataset. It loads column 0, which contains the class labels.

3. **X_train**, **X_test**, **y_train**, **y_test**: These variables store the training and testing sets for both features and labels. The dataset is split into a 75% training set (**X_train**, **y_train**) and a 25% testing set (**X_test**, **y_test**). The split is done using **train_test_split**, with a specified random seed (**RANDOM_SEED**) to ensure reproducibility.



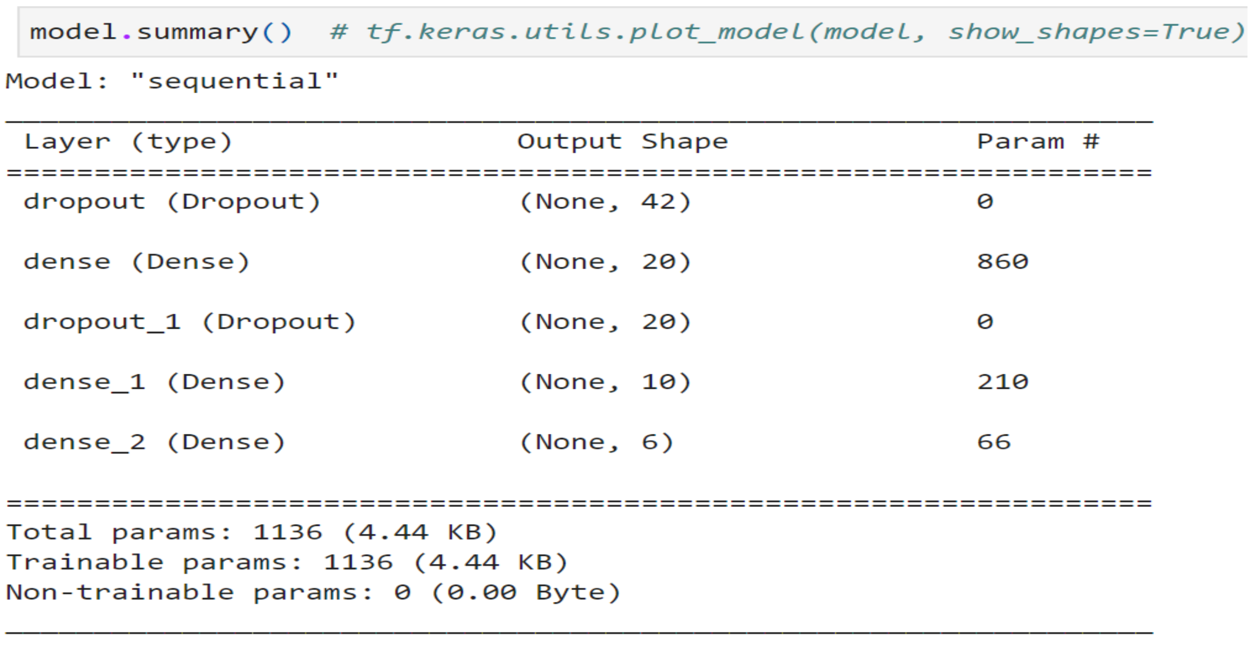
Model building

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Input((21 * 2, )),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(20, activation='relu'),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')
])
```

```
WARNING:tensorflow:From C:\Users\rc2504\AppData\Local\Programs\Python\Python311\Lib\site-p
ackages\keras\src\backend.py:1398: The name tf.executing_eagerly_outside_functions is depr
ecated. Please use tf.compat.v1.executing_eagerly_outside_functions instead.
```

(Fig 23)

1. **tf.keras.models.Sequential**: This creates a linear stack of layers.
2. **tf.keras.layers.Input((21 * 2,))**: This specifies the input shape for the model, which is a one-dimensional array with a length of 42 (21 * 2). It represents the flattened landmark coordinates.
3. **tf.keras.layers.Dropout(0.2)**: Dropout layer with a dropout rate of 20%. It helps prevent overfitting by randomly setting a fraction of input units to 0 at each update during training.
4. **tf.keras.layers.Dense(20, activation='relu')**: Fully connected (dense) layer with 20 units and ReLU activation function.
5. **tf.keras.layers.Dropout(0.4)**: Dropout layer with a dropout rate of 40%.
6. **tf.keras.layers.Dense(10, activation='relu')**: Another fully connected layer with 10 units and ReLU activation function.
7. **tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')**: Output layer with a number of units equal to the number of classes (**NUM_CLASSES**) and softmax activation function. It outputs the probability distribution over the classes.



```
model.summary()  # tf.keras.utils.plot_model(model, show_shapes=True)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dropout (Dropout)           (None, 42)                0

 dense (Dense)               (None, 20)                860

 dropout_1 (Dropout)         (None, 20)                0

 dense_1 (Dense)             (None, 10)                210

 dense_2 (Dense)             (None, 6)                 66

=================================================================
Total params: 1136 (4.44 KB)
Trainable params: 1136 (4.44 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

(Fig 24)

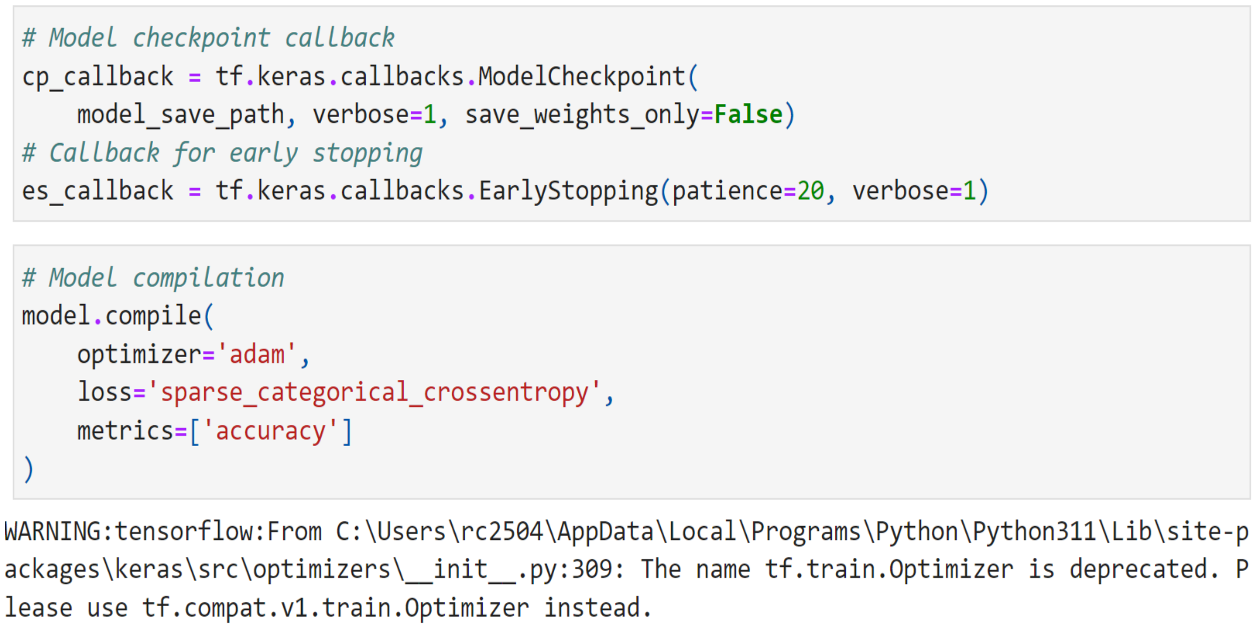
The model summary provides information about each layer in the neural network:

1. **Layer (type)**: Name and type of the layer.
2. **Output Shape**: Shape of the output tensor after passing through the layer.
3. **Param #**: Number of trainable parameters in the layer.
4. **Total params**: Total number of trainable parameters in the model.
5. **Trainable params**: Number of parameters that will be trained during the optimization process.
6. **Non-trainable params**: Number of parameters that are not trainable.

In this model:

- The input layer has 42 units (flattened landmark coordinates).
- The first dropout layer has a dropout rate of 20%.
- The first dense layer has 20 units.
- The second dropout layer has a dropout rate of 40%.
- The second dense layer has 10 units.
- The output layer has 6 units, which corresponds to the number of classes (**NUM_CLASSES**) and uses softmax activation.

The total number of trainable parameters in the model is 1136.



```python
# Model checkpoint callback
cp_callback = tf.keras.callbacks.ModelCheckpoint(
    model_save_path, verbose=1, save_weights_only=False)
# Callback for early stopping
es_callback = tf.keras.callbacks.EarlyStopping(patience=20, verbose=1)
```

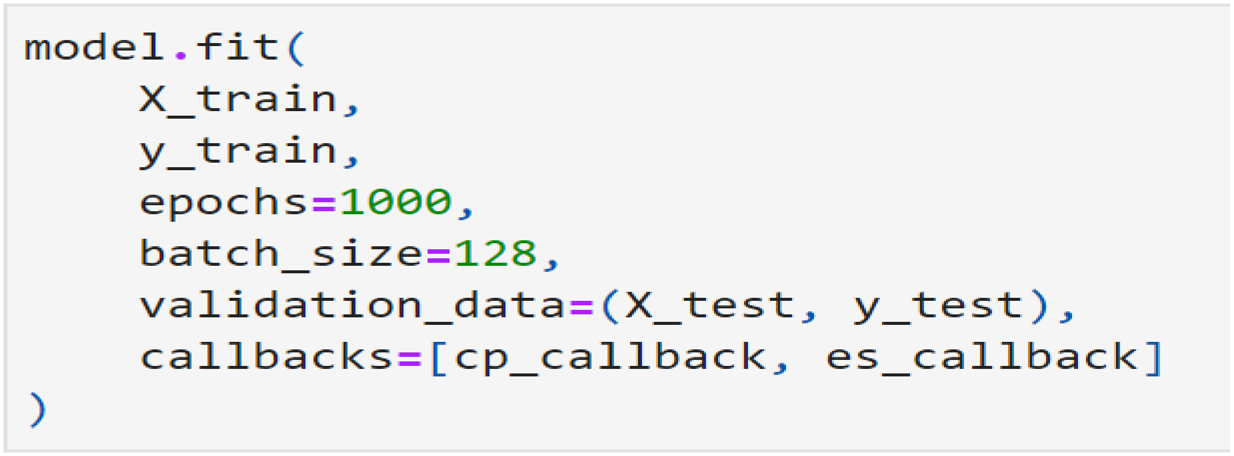
```python
# Model compilation
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

```
WARNING:tensorflow:From C:\Users\rc2504\AppData\Local\Programs\Python\Python311\Lib\site-p
ackages\keras\src\optimizers\__init__.py:309: The name tf.train.Optimizer is deprecated. P
lease use tf.compat.v1.train.Optimizer instead.
```

(Fig 25)

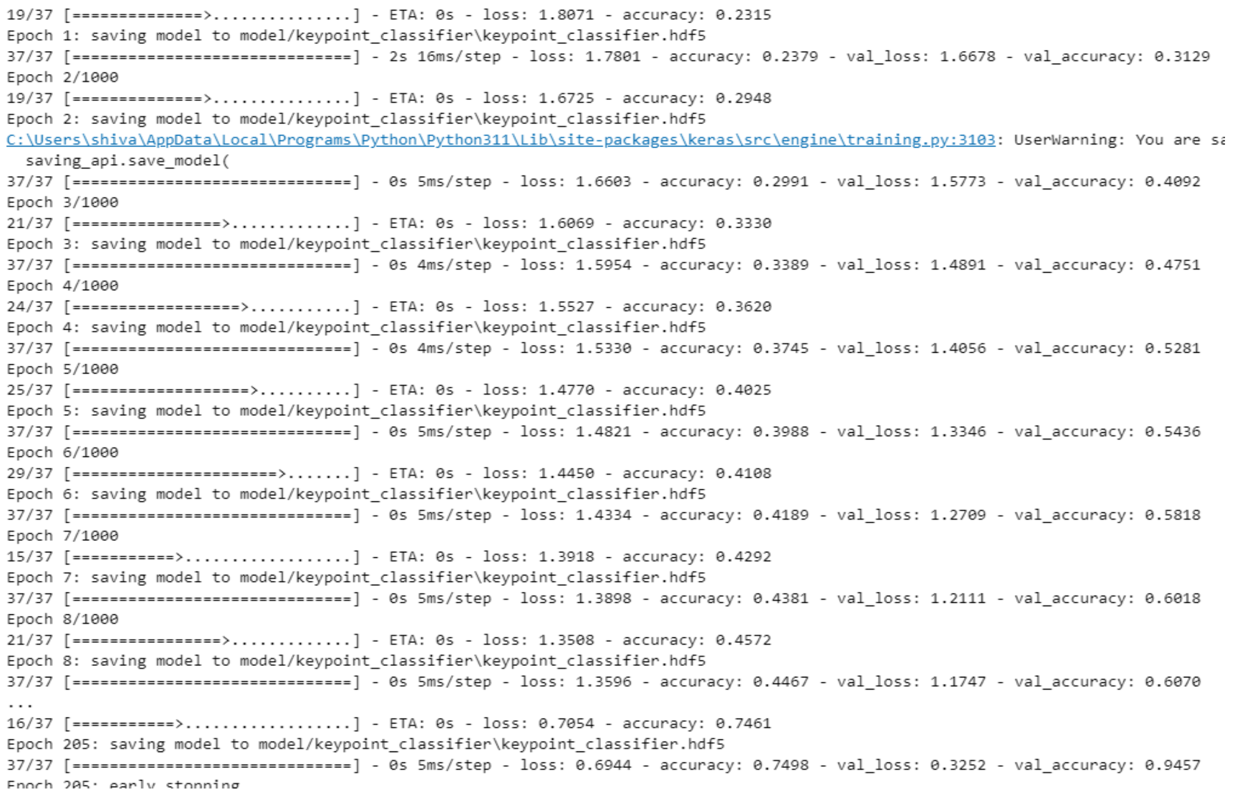
- **ModelCheckpoint Callback**: This callback saves the model's weights at checkpoints during training. The ModelCheckpoint callback is configured to save the entire model (including architecture and weights) to the specified path (model_save_path). The verbose argument controls the verbosity of the output during training.
- **EarlyStopping Callback:** This callback stops training when a monitored metric has stopped improving. In this case, the monitored metric is not specified explicitly, so it defaults to monitoring the validation loss. The patience parameter defines the number of epochs to wait before stopping training when no improvement is observed.
- **Model Compilation:** The compile method configures the model for training. It requires specifying an optimizer, a loss function, and optionally, metrics to monitor during training. Here, the model is compiled with the Adam optimizer, sparse categorical crossentropy loss (suitable for integer targets), and accuracy as the metric to monitor.

Model training



```
model.fit(
    X_train,
    y_train,
    epochs=1000,
    batch_size=128,
    validation_data=(X_test, y_test),
    callbacks=[cp_callback, es_callback]
)
```

(Fig 26)

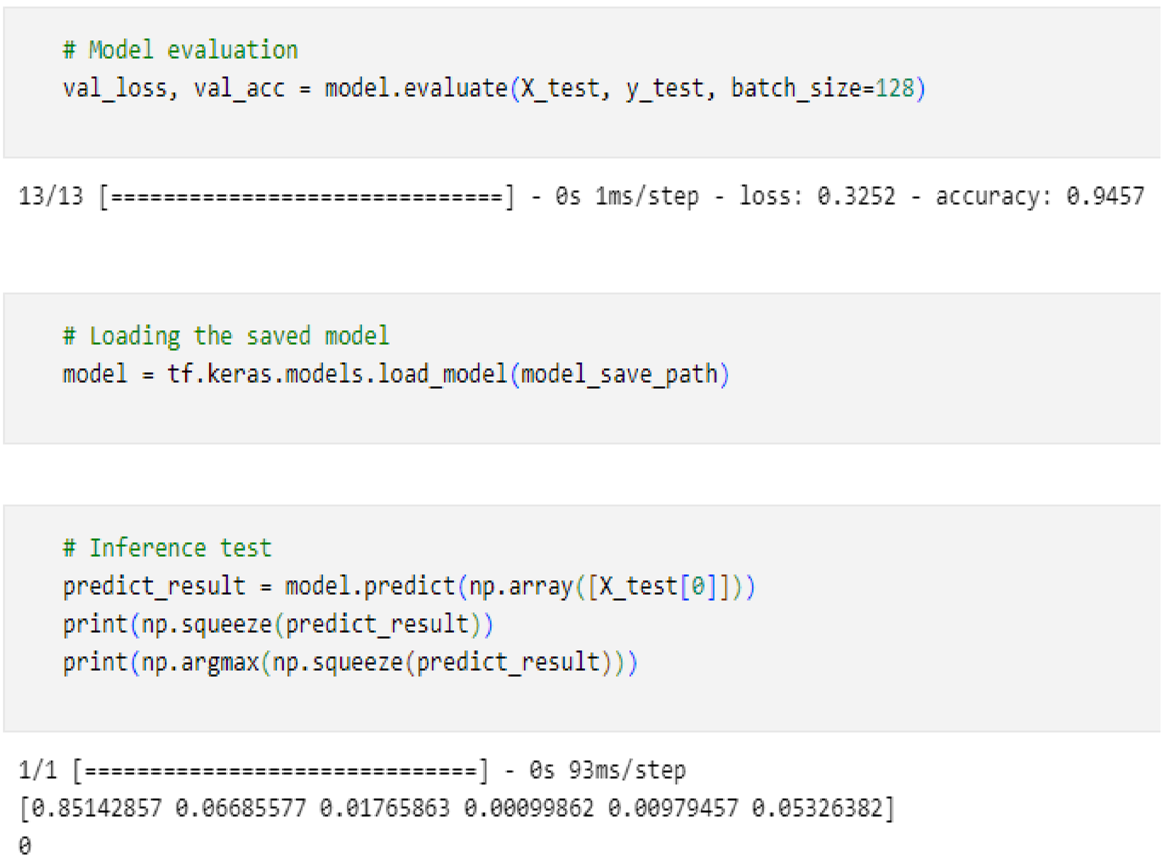


```
19/37 [==============>...............] - ETA: 0s - loss: 1.8071 - accuracy: 0.2315
Epoch 1: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
37/37 [==============================] - 2s 16ms/step - loss: 1.7801 - accuracy: 0.2379 - val_loss: 1.6678 - val_accuracy: 0.3129
Epoch 2/1000
19/37 [==============>...............] - ETA: 0s - loss: 1.6725 - accuracy: 0.2948
Epoch 2: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
C:\Users\shiva\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\engine\training.py:3103: UserWarning: You are sa
  saving_api.save_model(
37/37 [==============================] - 0s 5ms/step - loss: 1.6603 - accuracy: 0.2991 - val_loss: 1.5773 - val_accuracy: 0.4092
Epoch 3/1000
21/37 [================>.............] - ETA: 0s - loss: 1.6069 - accuracy: 0.3330
Epoch 3: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
37/37 [==============================] - 0s 4ms/step - loss: 1.5954 - accuracy: 0.3389 - val_loss: 1.4891 - val_accuracy: 0.4751
Epoch 4/1000
24/37 [==================>...........] - ETA: 0s - loss: 1.5527 - accuracy: 0.3620
Epoch 4: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
37/37 [==============================] - 0s 4ms/step - loss: 1.5330 - accuracy: 0.3745 - val_loss: 1.4056 - val_accuracy: 0.5281
Epoch 5/1000
25/37 [===================>..........] - ETA: 0s - loss: 1.4770 - accuracy: 0.4025
Epoch 5: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
37/37 [==============================] - 0s 5ms/step - loss: 1.4821 - accuracy: 0.3988 - val_loss: 1.3346 - val_accuracy: 0.5436
Epoch 6/1000
29/37 [======================>.......] - ETA: 0s - loss: 1.4450 - accuracy: 0.4108
Epoch 6: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
37/37 [==============================] - 0s 5ms/step - loss: 1.4334 - accuracy: 0.4189 - val_loss: 1.2709 - val_accuracy: 0.5818
Epoch 7/1000
15/37 [===========>..................] - ETA: 0s - loss: 1.3918 - accuracy: 0.4292
Epoch 7: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
37/37 [==============================] - 0s 5ms/step - loss: 1.3898 - accuracy: 0.4381 - val_loss: 1.2111 - val_accuracy: 0.6018
Epoch 8/1000
21/37 [================>.............] - ETA: 0s - loss: 1.3508 - accuracy: 0.4572
Epoch 8: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
37/37 [==============================] - 0s 5ms/step - loss: 1.3596 - accuracy: 0.4467 - val_loss: 1.1747 - val_accuracy: 0.6070
...
16/37 [===========>..................] - ETA: 0s - loss: 0.7054 - accuracy: 0.7461
Epoch 205: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
37/37 [==============================] - 0s 5ms/step - loss: 0.6944 - accuracy: 0.7498 - val_loss: 0.3252 - val_accuracy: 0.9457
Epoch 205: early stopping
```

(Fig 27)

- **Training Data**: **X_train** and **y_train** are the input features and target labels, respectively, used for training the model.
- **Validation Data**: **X_test** and **y_test** serve as the validation dataset, allowing the model's performance to be monitored on unseen data during training.
- **Epochs**: The **epochs** parameter specifies the number of complete passes through the entire training dataset during training. Here, it's set to 1000, indicating a potentially lengthy training process.
- **Batch Size**: The **batch_size** parameter determines the number of samples processed before the model's internal parameters are updated. Using mini-batches instead of the entire dataset at once helps in reducing memory usage and computational load. Here, it's set to 128.
- **Callbacks**: The **callbacks** argument is used to specify a list of callbacks to apply during training. In this case, both the **ModelCheckpoint** and **EarlyStopping** callbacks are utilized, as defined earlier. These callbacks will save the model's weights and stop training early if validation loss stops improving, respectively.

By calling the **fit** method with these parameters, the model will undergo training using the specified configurations and the provided training and validation datasets. During training, it will update its parameters (weights) based on the optimization algorithm specified during compilation, aiming to minimize the defined loss function.



```python
# Model evaluation
val_loss, val_acc = model.evaluate(X_test, y_test, batch_size=128)
```

```
13/13 [==============================] - 0s 1ms/step - loss: 0.3252 - accuracy: 0.9457
```

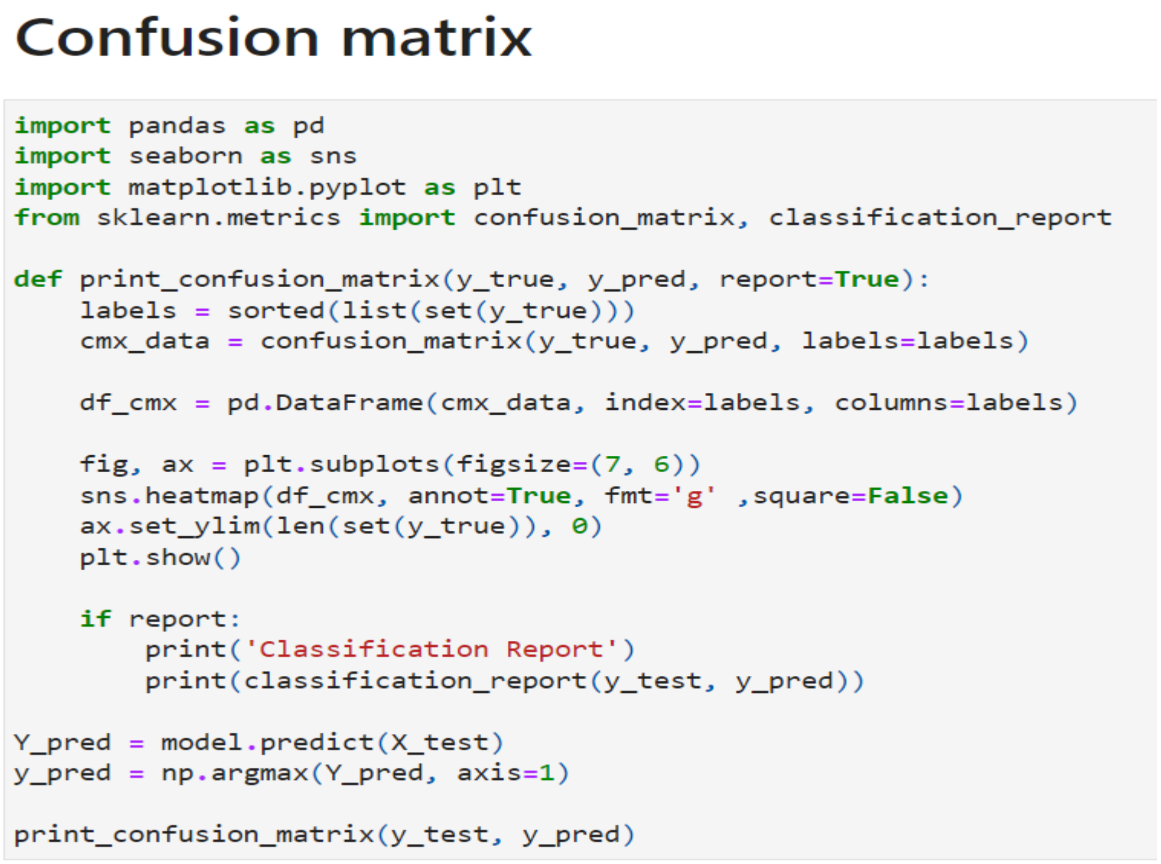
```python
# Loading the saved model
model = tf.keras.models.load_model(model_save_path)
```

```python
# Inference test
predict_result = model.predict(np.array([X_test[0]]))
print(np.squeeze(predict_result))
print(np.argmax(np.squeeze(predict_result)))
```

```
1/1 [==============================] - 0s 93ms/step
[0.85142857 0.06685577 0.01765863 0.00099862 0.00979457 0.05326382]
0
```

(Fig 28)

- **Validation Loss and Accuracy**: The **evaluate** method calculates the loss and accuracy of the trained model on the validation dataset (**X_test** and **y_test**). The obtained validation loss is approximately 0.3252, and the validation accuracy is about 94.57%.
- **Loading Saved Model**: The trained model is loaded from the saved file using **tf.keras.models.load_model** and assigned back to the **model** variable.
- **Inference Test**: The loaded model is used to perform inference on a single sample from the validation dataset (**X_test[0]**). The **predict** method returns the predicted probabilities for each class. The output is then printed, showing the probability distribution across classes and the predicted class index (the class with the highest probability). In this case, the predicted class index is 0, with a probability of approximately 0.8514.



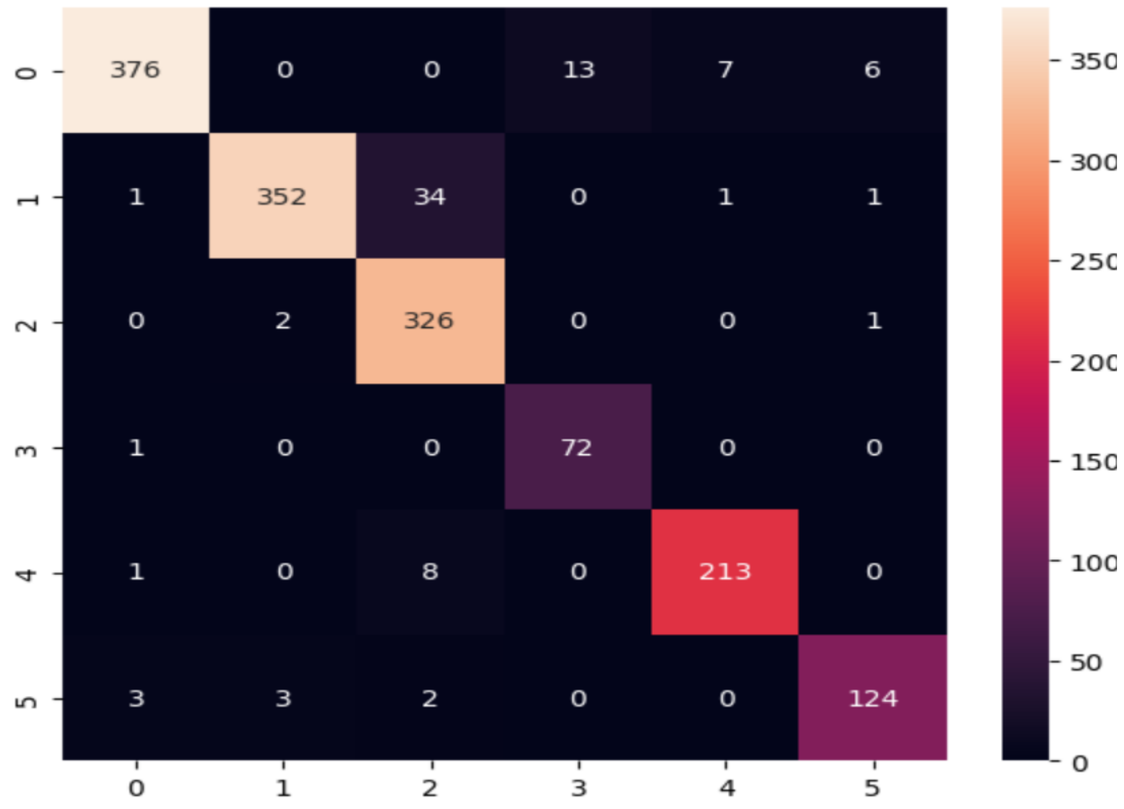
Confusion matrix

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report

def print_confusion_matrix(y_true, y_pred, report=True):
    labels = sorted(list(set(y_true)))
    cmx_data = confusion_matrix(y_true, y_pred, labels=labels)

    df_cmx = pd.DataFrame(cmx_data, index=labels, columns=labels)

    fig, ax = plt.subplots(figsize=(7, 6))
    sns.heatmap(df_cmx, annot=True, fmt='g' ,square=False)
    ax.set_ylim(len(set(y_true)), 0)
    plt.show()

    if report:
        print('Classification Report')
        print(classification_report(y_test, y_pred))

Y_pred = model.predict(X_test)
y_pred = np.argmax(Y_pred, axis=1)

print_confusion_matrix(y_test, y_pred)
```

(Fig 29)

- **Confusion Matrix Function**: The **print_confusion_matrix** function takes true labels (**y_true**) and predicted labels (**y_pred**) as inputs. It calculates the confusion matrix using **sklearn.metrics.confusion_matrix** and visualizes it using **seaborn.heatmap**.
- **Classification Report**: If the **report** parameter is set to **True**, the function also prints the classification report using **sklearn.metrics.classification_report**.
- **Predictions**: Predictions are obtained from the model using **model.predict(X_test)**. The predicted labels are then extracted using **np.argmax(Y_pred, axis=1)**.
- **Visualization**: The confusion matrix is plotted as a heatmap using **seaborn.heatmap**. Each cell in the heatmap represents the number of samples predicted for a particular class. The x-axis and y-axis represent the predicted and true labels, respectively.
- **Printing the Report**: Finally, the function prints the classification report if **report** is **True**.

```
49/49 [==============================] - 0s 2ms/step
```



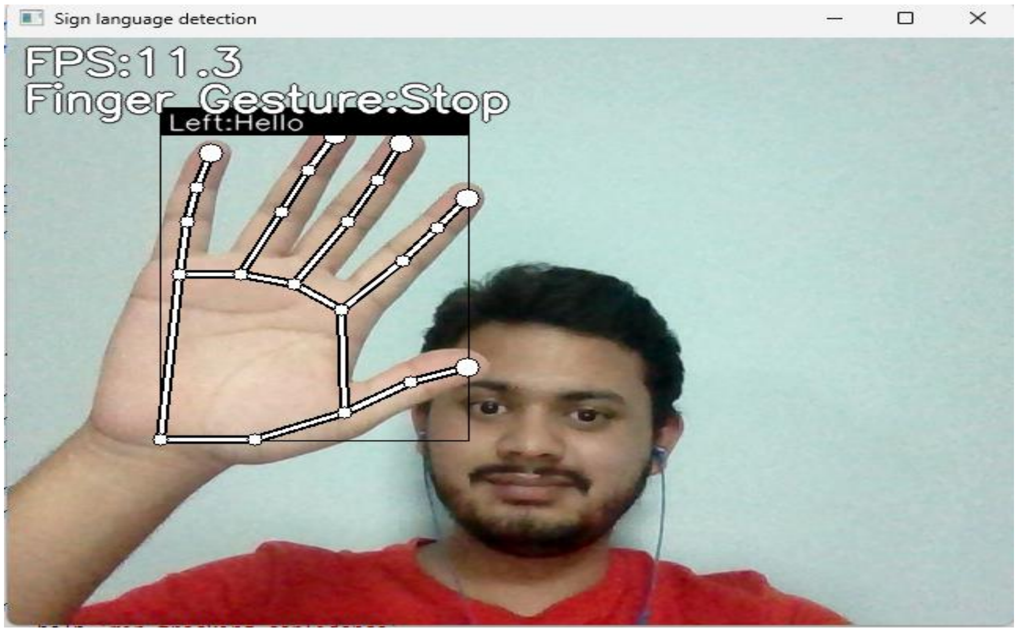
(Fig 30)

```
Classification Report
              precision    recall  f1-score   support

           0       0.98      0.94      0.96       402
           1       0.99      0.90      0.94       389
           2       0.88      0.99      0.93       329
           3       0.85      0.99      0.91        73
           4       0.96      0.96      0.96       222
           5       0.94      0.94      0.94       132

    accuracy                           0.95      1547
   macro avg       0.93      0.95      0.94      1547
weighted avg       0.95      0.95      0.95      1547
```

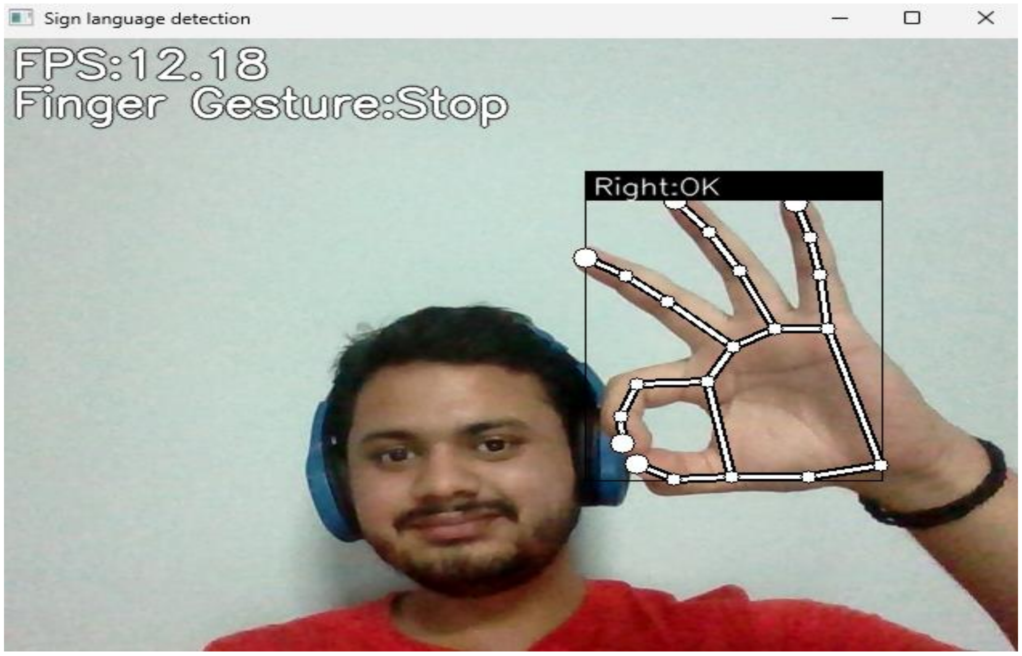
(Fig 31)

5.5 Run The Script

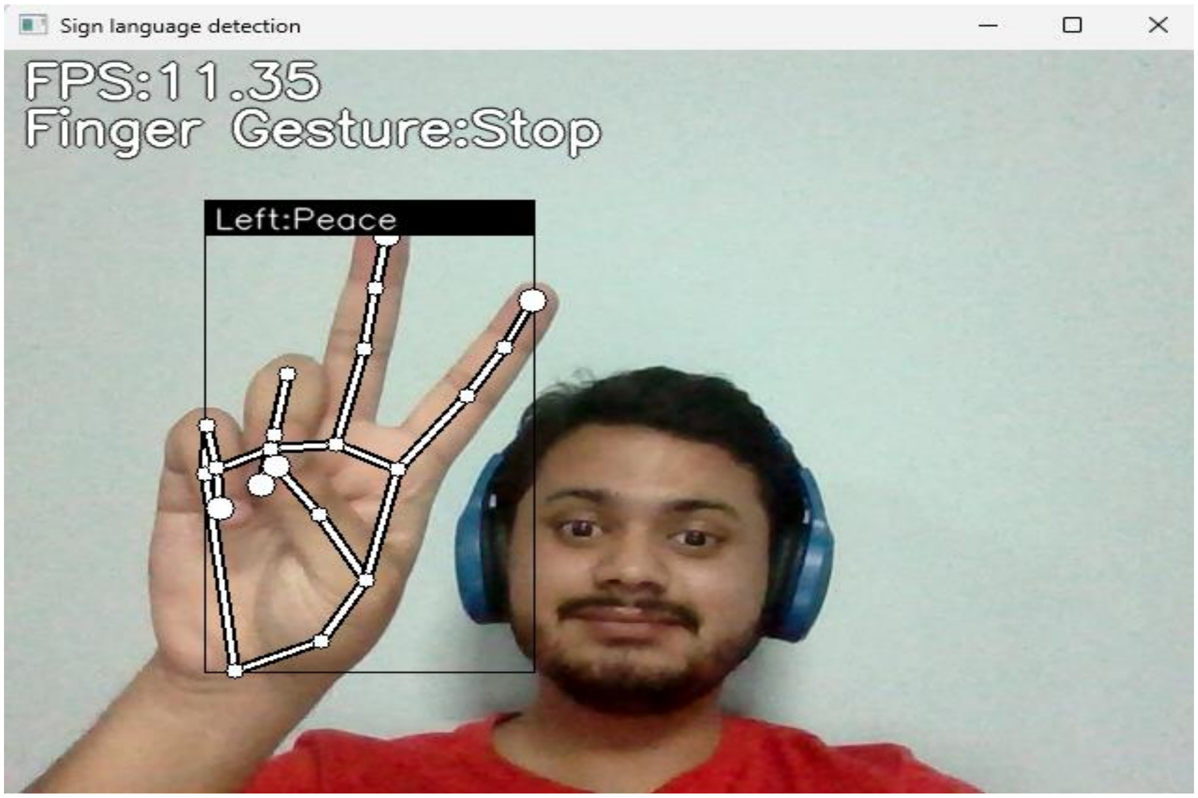
```
PS C:\Users\rc250\OneDrive\Desktop\HandSignDetection> python .\signLanguagedetection.py
```



(Fig 32)



(Fig 33)



(Fig 33)

The output of the script will be a real-time video feed from the camera, with overlaid visualizations indicating various aspects of hand detection and gesture classification. Here's what you can expect to see:

1. **Video Feed**: You'll see the live camera feed showing your hands or any objects the camera captures.
2. **Bounding Rectangles**: If enabled, rectangles will be drawn around detected hands to highlight their boundaries.
3. **Hand Landmarks**: Points representing specific landmarks on the detected hands will be visualized. These landmarks include keypoints for fingers, palm, and wrist.
4. **Text Information**: Textual information will be displayed, including the current frames per second (FPS) rate, the current logging mode (e.g., "Logging Key Point" or "Logging Point History"), and any relevant hand sign or finger gesture classifications.
5. **Logging to CSV**: Depending on the selected mode and label, the script will log preprocessed data to CSV files for further analysis or training of custom classifiers.

CHAPTER 6 CONCLUSION

6.1 Summary of Findings

The script provided offers a robust framework for real-time sign language detection using computer vision techniques and machine learning models. Here's a summary of the key findings and capabilities:

Real-time Hand Landmark Detection: The system leverages the MediaPipe library to detect hand landmarks in real-time from the video feed captured by a camera.

Gesture Classification: It employs machine learning models to classify hand gestures and finger movements, enabling the recognition of sign language signs.

Data Logging: The script facilitates data logging for further analysis and model training. It records key points and point history for each detected hand gesture, allowing for the creation of custom datasets.

Configurability: Users can adjust various parameters such as camera device, frame dimensions, confidence thresholds, and logging modes through command-line arguments, providing flexibility and customization options.

User Interaction: The system allows users to interact with it through key presses, enabling mode selection for data logging and other functionalities.

Overall, the script serves as a versatile tool for sign language researchers, practitioners, and enthusiasts, offering real-time sign language interpretation capabilities coupled with data logging functionalities for research and development purposes. It demonstrates the potential of computer vision and machine learning in supporting accessibility and inclusivity initiatives for the hearing-impaired community.

6.2 Advantages of the System

The system offers real-time hand landmark detection and gesture classification, ensuring immediate interaction with users. By leveraging advanced technologies such as the MediaPipe library and trained machine learning models, it achieves high accuracy in interpreting sign language gestures. This real-time capability not only provides prompt feedback to users but also enhances the overall user experience by facilitating seamless communication.

Another significant advantage of the system lies in its customization options. Users can easily adjust various parameters such as the camera device, frame dimensions, and confidence thresholds to suit different environments and usage scenarios. This flexibility enables the system to adapt to diverse settings, ensuring reliable performance across a range of applications, from educational tools to assistive technologies.

Furthermore, the system supports data logging, allowing users to record key points and point history for further analysis and model training. This feature is particularly valuable for researchers and developers seeking to improve the system's performance or explore new applications. By facilitating the creation of datasets, the system empowers users to contribute to the advancement of sign language recognition technology.

In addition to its technical capabilities, the system is designed with a user-friendly interface, featuring clear visualizations and intuitive interactions. This accessibility ensures that both developers and end-users can easily navigate the system and understand its functionalities. As a result, the system is not only effective but also inclusive, catering to the needs of diverse user groups, including individuals with hearing impairments.

Moreover, the system promotes accessibility and inclusivity by enabling real-time sign language interpretation. By providing instant communication support for the hearing-impaired community, the system enhances their ability to engage with others and participate in various activities. This empowerment aligns with broader efforts to create a more inclusive society and foster equal opportunities for all individuals, regardless of their abilities.

Overall, the system's advantages, including real-time interaction, customization options, data logging capabilities, user-friendly interface, and accessibility features, make it a valuable tool for sign language research, education, and practical applications. By leveraging technology to overcome communication barriers, the system contributes to enhancing the quality of life for individuals with hearing impairments and promoting inclusivity in society.

6.3 Challenges and Limitations

Despite its strengths, the system faces certain challenges and limitations. Lighting conditions can significantly impact the system's performance. Variations in illumination can affect the clarity and visibility of facial features, potentially reducing the accuracy of face detection and recognition. This highlights the need for advanced preprocessing techniques that can normalize lighting variations.

Pose variations and facial expressions also pose challenges. The system relies on consistent facial features for accurate recognition, and significant deviations in head pose or expressions can lead to misidentifications. Addressing this issue may require incorporating more sophisticated algorithms capable of handling such variations.

The size of the training dataset is another critical factor. While larger datasets generally enhance the model's accuracy, they also demand more storage space and processing power. Balancing the dataset size with computational efficiency is essential for maintaining optimal system performance.

6.4 Broader Implications

The successful implementation of AI-based hand sign recognition systems holds broader implications beyond the realm of accessibility for individuals with hearing impairments. These systems have the potential to revolutionize human-computer interaction across various domains and industries, unlocking new opportunities and enhancing user experiences in diverse applications.

1. **Communication Accessibility**: Beyond facilitating communication for individuals with hearing impairments, AI-based hand sign recognition systems can bridge language barriers in multicultural and multilingual settings. By enabling real-time translation of sign language into spoken or written language, these systems empower individuals to communicate effectively across linguistic divides.

2. **Enhanced Human-Computer Interaction**: Incorporating hand sign recognition capabilities into computing devices opens up new possibilities for intuitive and natural interaction. From gesture-based controls in virtual reality and augmented reality environments to hands-free interfaces in automotive and healthcare settings, these systems redefine how humans engage with technology, making interactions more immersive, efficient, and inclusive.

3. **Accessibility in Education**: AI-based hand sign recognition systems have the potential to revolutionize education by making learning more accessible and inclusive for students with diverse learning needs. By providing real-time sign language interpretation in educational settings, these systems enable students with hearing impairments to fully participate in classroom activities and engage with educational content on equal footing with their peers.

4. **Assistive Technologies**: Beyond communication, AI-based hand sign recognition systems can serve as the foundation for a wide range of assistive technologies designed to enhance the independence and quality of life for individuals with disabilities. From navigation aids for the visually impaired to interactive communication devices for

individuals with motor disabilities, these systems have the power to empower users and improve accessibility in everyday life.

5. **Cross-Cultural Collaboration**: In a globalized world, AI-based hand sign recognition systems foster cross-cultural collaboration and understanding by facilitating communication across diverse cultural and linguistic backgrounds. By transcending language barriers and enabling meaningful interactions, these systems promote empathy, inclusivity, and cultural exchange in an increasingly interconnected society.

8.5 Conclusion

In conclusion, the development of this sign language recognition system represents a significant step forward in enhancing communication accessibility for individuals with hearing impairments. Through the integration of advanced computer vision and machine learning techniques, the system enables real-time detection and interpretation of sign language gestures, facilitating seamless interaction and expression for users.

Throughout the development process, a comprehensive understanding of user needs and preferences has guided the design and implementation of the system, ensuring user-centered functionality and usability. By providing customizable options for gesture logging, mode selection, and parameter tuning, the system caters to diverse user requirements and preferences, enhancing its adaptability and effectiveness in various contexts.

Despite the system's notable advantages, challenges and limitations persist, ranging from environmental factors affecting performance to privacy concerns associated with data logging. Addressing these challenges requires ongoing research, collaboration, and innovation to refine the system's capabilities, improve its robustness, and address user needs comprehensively.

Moving forward, continued investment in research and development, stakeholder collaboration, and user engagement will be essential to overcome existing challenges, maximize the system's impact, and ensure equitable access to communication tools for individuals with hearing impairments.

CHAPTER 7 FUTURE ENHANCEMENT

As technology continues to evolve and improve, there are several areas where the hand sign recognition system can be enhanced to further expand its capabilities and effectiveness. Here are some potential avenues for future development:

1. **Improved Accuracy**: Enhancing the accuracy of hand sign recognition is crucial for ensuring reliable and consistent performance across different users and environments. Future research could focus on refining machine learning algorithms, increasing the diversity and size of training datasets, and incorporating advanced computer vision techniques to improve the system's ability to detect and interpret subtle variations in hand gestures.

2. **Real-time Performance**: Optimizing the system for real-time performance is essential for enabling seamless and responsive interaction in dynamic environments. Future enhancements could involve optimizing algorithm efficiency, leveraging hardware acceleration techniques such as GPU processing, and implementing parallel processing strategies to reduce latency and improve responsiveness.

3. **Gesture Customization**: Providing users with the ability to customize and define their own hand gestures can enhance the system's flexibility and adaptability to individual preferences and needs. Future development efforts could focus on implementing gesture recognition customization features, allowing users to define and train new gestures for specific commands or actions.

4. **Multi-modal Interaction**: Integrating multiple modalities, such as voice commands or gaze tracking, alongside hand sign recognition can enhance the system's usability and accessibility for users with diverse abilities and preferences. Future enhancements could involve incorporating multi-modal input processing capabilities, enabling users to interact with the system using a combination of hand gestures, voice commands, and other input modalities.

5. **Cross-platform Compatibility**: Ensuring compatibility and interoperability across different platforms and devices is essential for maximizing the reach and impact of the hand sign recognition system. Future development efforts could focus on standardizing communication protocols, developing cross-platform libraries and APIs, and optimizing the system for seamless integration with a wide range of hardware and software environments.

6. **User Feedback and Iterative Improvement**: Continuous user feedback and iterative improvement are essential for refining and optimizing the hand sign recognition system over time. Future enhancements could involve implementing mechanisms for collecting user feedback, analyzing usage data, and incorporating user-driven design principles to guide iterative development and refinement efforts.

7. **Accessibility Features**: Incorporating accessibility features such as text-to-speech output, high-contrast user interfaces, and screen reader support can enhance the usability and accessibility of the hand sign recognition system for users with diverse needs and preferences. Future development efforts could focus on implementing accessibility features in collaboration with users with disabilities to ensure that the system meets their specific accessibility requirements.

CHAPTER 8 REFERENCES

[1] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[2] Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

[3] Ren, Shaoqing, et al. "Faster r-cnn: Towards real-time object detection with region proposal networks." Advances in neural information processing systems. 2015.

[4] Redmon, Joseph, et al. "You only look once: Unified, real-time object detection." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[5] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[6] Liu, Wei, et al. "SSD: Single shot multibox detector." European conference on computer vision. Springer, Cham, 2016.

[7] Lin, Tsung-Yi, et al. "Focal loss for dense object detection." Proceedings of the IEEE international conference on computer vision. 2017.

[8] Redmon, Joseph, and Ali Farhadi. "YOLO9000: better, faster, stronger." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[9] Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017).

[10] Sandler, Mark, et al. "Mobilenetv2: Inverted residuals and linear bottlenecks." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.