ABSTRACT

Hand gesture recognition is an essential technology that can greatly benefit physically challenged individuals by providing them with an intuitive and accessible means of interacting with computers and devices. This project aims to develop a hand gesture recognition system using the Mediapipe library and a Multi-layer Perceptron (MLP) algorithm, specifically tailored to cater to the needs of physically challenged persons. The proposed system will empower them to control various applications and devices effortlessly through simple hand movements, enhancing their independence and overall quality of life.

This project focuses on creating a communication system called "Sine Language" using hand gesture recognition with Python. The goal is to allow people to communicate using specific hand movements that represent words or ideas. By using computer vision techniques, the system captures hand gestures, interprets them, and converts them into text or sound. This project aims to make communication easier for people from different language backgrounds and could be helpful for travel, deaf-mute communication, and remote teamwork.

**1. INTRODUCTION**

Hand gesture recognition technology has emerged as a powerful tool in human-computer interaction, offering a natural and intuitive way for users to interact with computers and devices. For physically challenged individuals, this technology holds even greater significance, as it can provide them with an accessible means of communication and control over technology. The proposed project aims to develop a hand gesture recognition system using the Mediapipe library and a Multi-layer Perceptron (MLP) algorithm, specifically tailored to cater to the unique needs of physically challenged individuals. By harnessing the capabilities of computer vision and artificial neural networks, this system seeks to empower these individuals, enhancing their independence and enriching their daily lives.

Conventional input methods like keyboards and mice can be challenging or impractical for physically challenged individuals due to motor impairments. Hand gesture recognition technology has the potential to bridge this gap by allowing users to interact with devices through simple hand movements. The Mediapipe library provides an extensive set of tools for real-time hand tracking and landmark detection, which will be leveraged to accurately interpret users' gestures. Coupled with the efficiency of the Multi-layer Perceptron (MLP) algorithm, the system can efficiently recognize a wide range of gestures and translate them into meaningful actions, such as controlling assistive devices, accessing applications, or navigating through virtual environments.

The successful implementation of the proposed hand gesture recognition system could revolutionize the way physically challenged individuals interact with technology. By offering a more inclusive and intuitive interface, the system can enhance their overall quality of life, improve accessibility, and promote independence. Moreover, the project's scope goes beyond mere gesture recognition; it aims to create a user-friendly and reliable solution that can be integrated with various assistive devices and applications, opening up new possibilities for a more inclusive and empowered digital world.

* **AIM OF THE PROJECT**

The scope of this project involves the development and implementation of a hand gesture recognition system tailored specifically for physically challenged individuals. The system will be designed to work with standard cameras commonly found in laptops, tablets, and smartphones, ensuring widespread accessibility. It will focus on recognizing essential hand gestures that can facilitate common tasks and interactions.

* OBJECTIVE OF THE PROJECT

The objectives of the project are as follows.

1. Develop a user-friendly hand gesture recognition system specifically catering to the needs of physically challenged individuals.
2. Implement real-time hand tracking and landmark detection using the Mediapipe library for accurate and responsive gesture recognition.
3. Design and train a Multi-layer Perceptron (MLP) algorithm capable of recognizing a wide range of hand gestures with high accuracy.
4. LITERATURE SURVEY

* **EXISTING SYSTEM**

Existing hand gesture recognition systems for physically challenged individuals are relatively limited and may not be tailored to their specific needs. Some solutions require specialized hardware or cumbersome setups, making them less accessible and cost-effective. Additionally, these systems might not provide the level of accuracy and responsiveness required for smooth interaction, hindering the overall user experience.

* **PROPOSED SYSTEM**

The proposed hand gesture recognition system will utilize the Mediapipe library to track and detect hand landmarks in real-time, making it more accessible for physically challenged individuals. The system will be optimized for efficiency and accuracy, ensuring a smooth user experience. It will interpret various hand gestures and translate them into corresponding actions, such as controlling assistive devices, accessing applications, or performing specific tasks.

**3. ABOUT PROGRAMMING LANGUAGE**

* **PYTHON**

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for :

* Python can be used on a server to create web applications.
* Python can be used alongside software to create workflows.
* Python can connect to database systems. It can also read and modify files.
* Python can be used to handle big data and perform complex mathematics.
* Python can be used for rapid prototyping, or for production-ready software development.
* **FEATURES**
* Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
* Python has a simple syntax similar to the English language.
* Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
* Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
* Python can be treated in a procedural way, an object-oriented way or a functional way.

### 

### **PYTHON SYNTAX COMPARED TO OTHER PROGRAMMING LANGUAGES**

* Python was designed for readability, and has some similarities to the English language with influence from mathematics.
* Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
* Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.
* **SOFTWARE REQUIREMENTS**

**Idle’s interactive shell**

**IDLE** is a simple Python integrated development environment available for Windows, Linux, and Mac OS X. Figure 3.1 shows how to start IDLE from the Microsoft Windows Start menu. The IDLE interactive shell is shown in Figure 3.2. You may type the above one line Python program directly into IDLE and press enter to execute the program. Figure 3.3 shows the result using the IDLE interactive shell.

Since it does not provide a way to save the code you enter, the interactive shell is not the best tool for writing larger programs. The IDLE interactive shell is useful for experimenting with small snippets of Python code.

**IDLE’s editor**. IDLE has a built-in editor. From the IDLE menu, select New Window, as shown in Figure 3.4. Type the simple program as shown above into the editor. Figure 3.5 shows the resulting editor window with the text of the simple Python program. You can save your program using the Save option in the File menu as shown in Figure 3.6. Save the code to a file named simple.py. The actual name of the file is irrelevant, but the name “simple” accurately describes the nature of this program. The extension .py is the extension used for Python source code. We can run the program from within the IDLE editor by pressing the F5 function key or from the editor’s Run menu: Run!Run Module. The output appears in the IDLE interactive shell window.

The editor allows us to save our programs and conveniently make changes to them later. The editor understands the syntax of the Python language and uses different colors to highlight the various components that comprise a program. Much of the work of program development occurs in the editor.

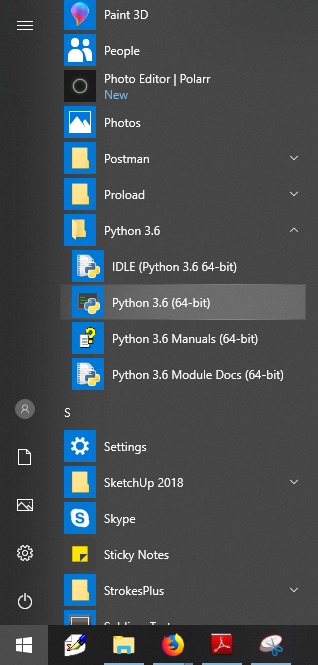


Figure 3.1: Launching IDLE from the Windows Start menu.

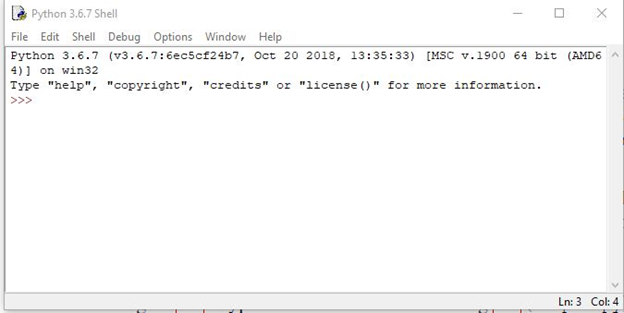


Figure 3.2: The IDLE interpreter Window.

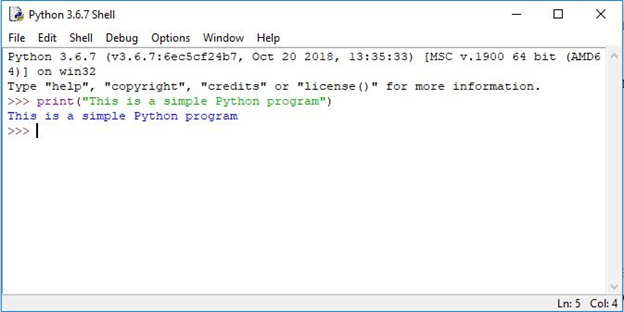


Figure 3.3: A simple Python program entered and run with the IDLE interactive shell.

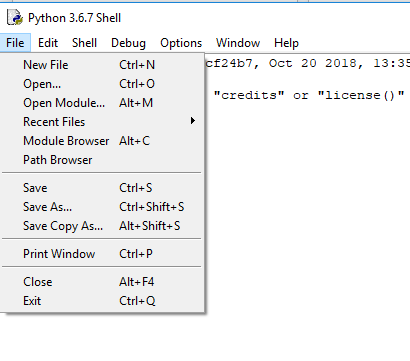


Figure 3.4: Launching the IDLE editor.

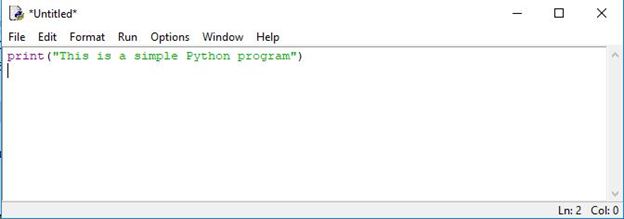


Figure 3.5: The simple Python program typed into the IDLE editor.

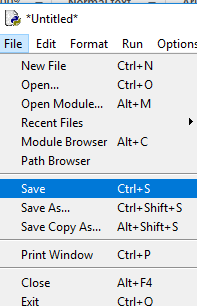
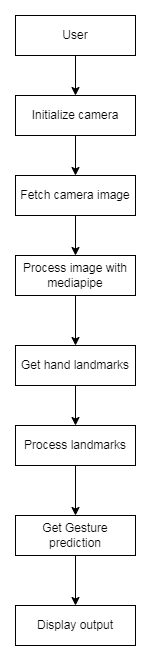


Figure 3.6: Saving a file created with the IDLE editor

simple.py file contains only one line of code: print("This is a simple Python program"). This is a Python statement. A statement is a command that the interpreter executes. This statement prints the message This is a simple Python program on the screen. A statement is the fundamental unit of execution in a Python program. Statements may be grouped into larger chunks called blocks, and blocks can make up more complex statements. Higher-order constructs such as functions and methods are composed of blocks. The statement print("This is a simple Python program") makes use of a built-in function named print. Python has a variety of different kinds of statements that may be used to build programs, and the chapters that follow explore these various kinds of statements.

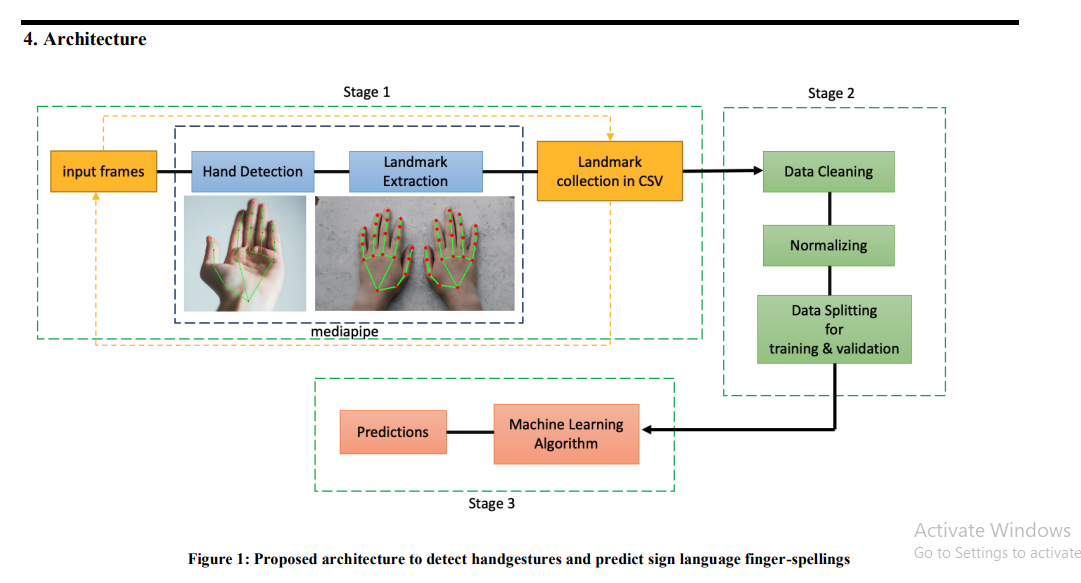
1. DATA FLOW DIAGRAM

****

**Figure 4.1 : Data Flow Diagram**

WORKING

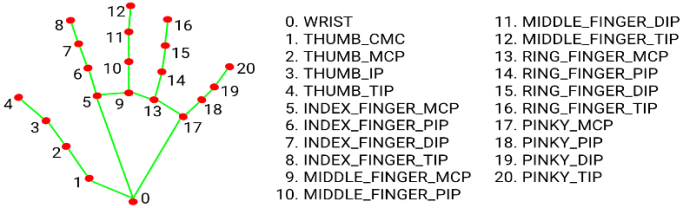
1. User Initialization:
   * Initialize the camera to capture video frames.
   * Set up the display window to show the camera feed and the output of the gesture recognition system.
2. Fetch Camera Image:
   * Continuously capture video frames from the camera.
   * Convert the frames to the appropriate format for further processing.
3. Process Image with Mediapipe:
   * Utilize the Mediapipe library to process the camera frames.
   * Apply the pre-trained hand tracking model from Mediapipe to detect and track the hand in each frame.
   * Obtain the hand landmarks (coordinates of keypoints on the hand) from the detected hand.
4. Get Hand Landmark:
   * Extract the hand landmark data from the Mediapipe output.
   * Preprocess the landmark data if necessary (e.g., normalization, scaling).
5. Process Landmark for Gesture Prediction:
   * Prepare the preprocessed hand landmark data as input to the Multi-layer Perceptron (MLP) algorithm for gesture recognition.
   * Apply the trained MLP model to predict the gesture based on the processed hand landmark data.
6. Display Output:
   * Visualize the camera feed with the detected hand landmarks overlaid on the hand in real-time.
   * Display the recognized gesture as text or visual feedback on the output display window.



**Figure 4.2**  working of Landmarks Detection

**Stage 1**:

Pre-Processing of Images to get Multi-hand Landmarks using MediaPipe In this step we will be making a hands landmarks detection model with the profound library called as mediapipe as base library and for other computer vision pre-processing CV2 library.



**Stage 2:** Data cleaning and normalization

As in stage 1, here we are considering only x and y coordinates from the detector, in this stage each image in the dataset is passed through stage 1 to collect all the data points under one file. This file is then scraped through the pandas' library function to check for any null entries. Sometimes due to blurry images, the detector cannot detect the hand which leads to null entry into the dataset. we need to remove those entries in the dataset. Hence, it is necessary to clean these points or null entries otherwise it will lead to biasness while making the predictive model.

**Stage 3**: Prediction using Machine Learning Algorithm

Predictive analysis of different sign languages are performed using machine learning algorithms.

1. **MEDIAPIPE**

MediaPipe provides a pre-built graph component called "Hand Tracking," which can be used for hand gesture detection.

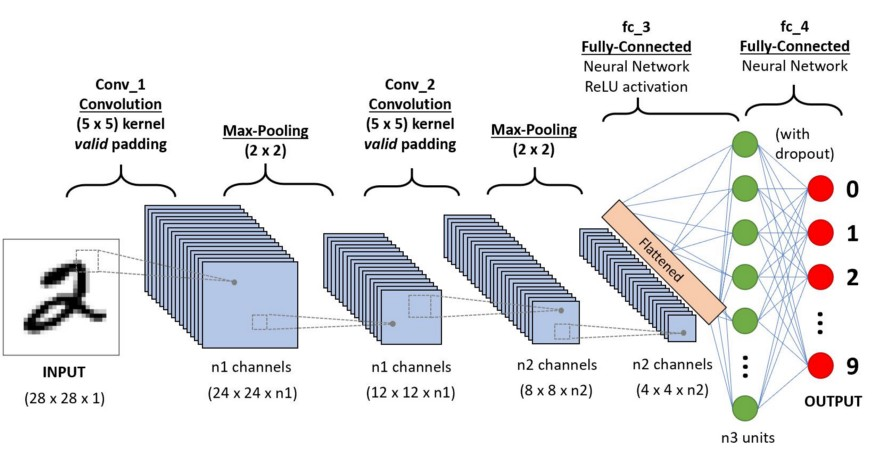
MediaPipe's Hand Tracking component is built on a combination of computer vision and machine learning techniques. It utilizes a machine learning model trained on a large dataset of hand images to detect and track hand landmarks in real-time.

The underlying algorithm used in MediaPipe's Hand Tracking component is based on convolutional neural networks (CNNs).

**5. CONVOLUTIONAL NEURAL NETWORK**

A Convolutional Neural Network (CNN) is a type of deep learning algorithm that is particularly well-suited for image recognition and processing tasks. It is made up of multiple layers, including convolutional layers, pooling layers, and fully connected layers.

The convolutional layers are the key component of a CNN, where filters are applied to the input image to extract features such as edges, textures, and shapes. The output of the convolutional layers is then passed through pooling layers, which are used to down-sample the feature maps, reducing the spatial dimensions while retaining the most important information. The output of the pooling layers is then passed through one or more fully connected layers, which are used to make a prediction or classify the image.



**Figure 6.1** convolution neural network

**THREE LAYERS**

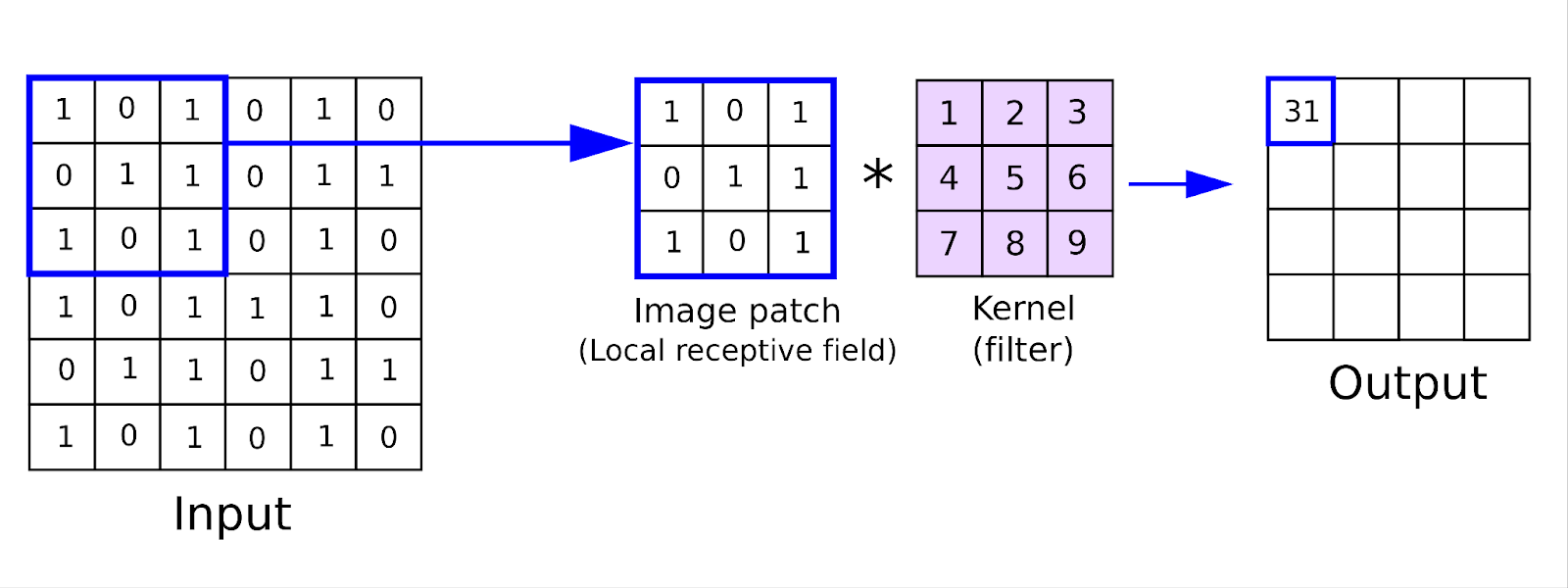
1.Convolutional layer

2.Max pooling

3.Fully connected

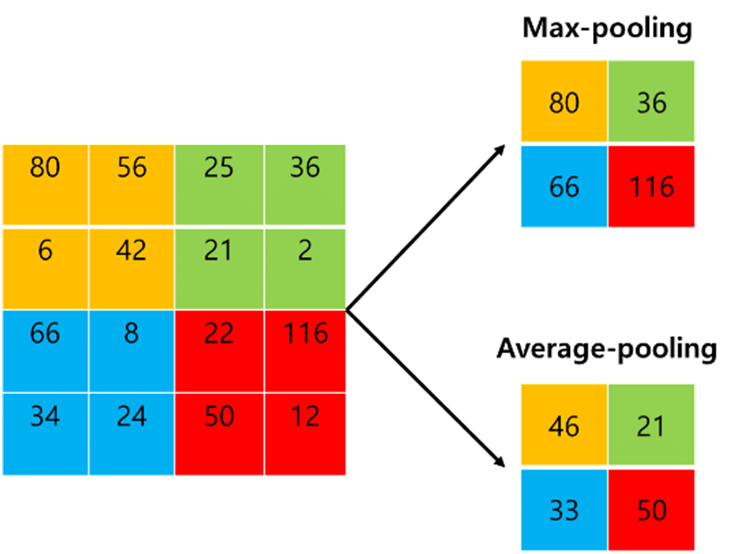
**1.Convolutional layer**

CNN's first building block is the convolutional layer. It takes the features from the input image and extracts them. Convolution mathematically combines the two sets of data. Convolution can be applied to the input data. The future map is created using convolution.



**Figure 6.2** Working of convolutional layer

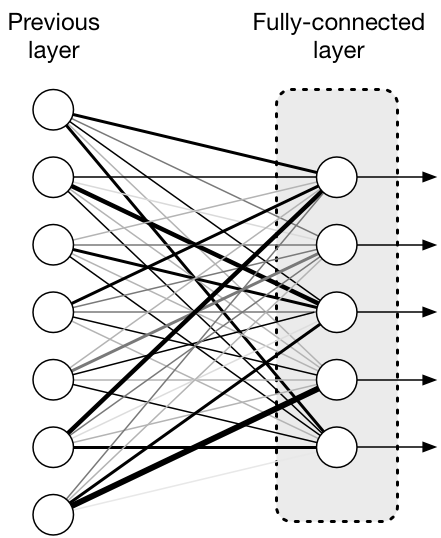
2. Max Pooling

Feature maps are obtained using the convolution layers. By using the pooling layers dimension of the feature maps are reduced by 50%. There are two types of pooling layers i,e average pooling and maximum pooling. 

**Figure 6.3** Max pooling

3.Fully connected

The final feature map outputs or max pooling layer matrix outputs are the input to the fully connected layer. Inputs of the fully connected layers are flattened to one column vectors. The example is as shown below.

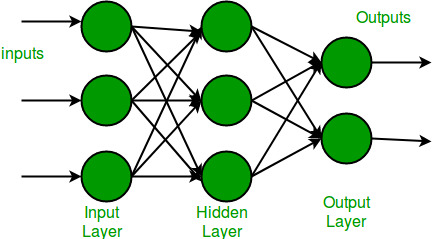


**Figure 6.4** Fully connected layer

1. **MLP**

A multi-layer perception is a neural network that has multiple layers. To create a neural network we combine neurons together so that the outputs of some neurons are inputs of other neurons.

A multi-layer perceptron has one input layer and for each input, there is one neuron(or node), it has one output layer with a single node for each output and it can have any number of hidden layers and each hidden layer can have any number of nodes. A schematic diagram of a Multi-Layer Perceptron (MLP) is depicted below.



**Figure 6.1 MLP(**multi-layer preception)

In the multi-layer perceptron diagram above, we can see that there are three inputs and thus three input nodes and the hidden layer has three nodes. The output layer gives two outputs, therefore there are two output nodes. The nodes in the input layer take input and forward it for further process, in the diagram above the nodes in the input layer forwards their output to each of the three nodes in the hidden layer, and in the same way, the hidden layer processes the information and passes it to the output layer.

1. **CODE AND SCREENSHOTS**

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 time

from pygame import mixer

# Starting the mixer

mixer.init()

mixer.music.set\_volume(0.7)

f1 = True

f2=True

f3=True

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

def main():

    f1 = True

    f2=True

    f3=True

    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

    cap = cv.VideoCapture(cap\_device)

    time.sleep(2)

    cap.set(cv.CAP\_PROP\_FRAME\_WIDTH, cap\_width)

    cap.set(cv.CAP\_PROP\_FRAME\_HEIGHT, cap\_height)

    mp\_hands = mp.solutions.hands

    hands = mp\_hands.Hands(

        static\_image\_mode=use\_static\_image\_mode,

        max\_num\_hands=1,

        min\_detection\_confidence=min\_detection\_confidence,

        min\_tracking\_confidence=min\_tracking\_confidence,

    )

    keypoint\_classifier = KeyPointClassifier()

    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

        ]

    cvFpsCalc = CvFpsCalc(buffer\_len=10)

    history\_length = 16

    point\_history = deque(maxlen=history\_length)

    finger\_gesture\_history = deque(maxlen=history\_length)

    mode = 0

    while True:

        fps = cvFpsCalc.get()

        key = cv.waitKey(10)

        if key == 27:  # ESC

            break

        number, mode = select\_mode(key, mode)

        ret, image = cap.read()

        if not ret:

            break

        image = cv.flip(image, 1)

        debug\_image = copy.deepcopy(image)

        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):

                brect = calc\_bounding\_rect(debug\_image, hand\_landmarks)

                landmark\_list = calc\_landmark\_list(debug\_image, hand\_landmarks)

                pre\_processed\_landmark\_list = pre\_process\_landmark(

                    landmark\_list)

                logging\_csv(number,mode,pre\_processed\_landmark\_list)

                hand\_sign\_id = keypoint\_classifier(pre\_processed\_landmark\_list)

                finger\_gesture\_id = 0

                finger\_gesture\_history.append(finger\_gesture\_id)

                if hand\_sign\_id == 0 and f1:

                    f1 = False

                    f2=True

                    f3=True

                    print(keypoint\_classifier\_labels[0])

                    mixer.music.load("music/sick.mp3")

                    mixer.music.play()

                if hand\_sign\_id == 1 and f2:

                    f2 = False

                    f1  = True

                    f3=True

                    print(keypoint\_classifier\_labels[1])

                    mixer.music.load("music/emergency.mp3")

                    mixer.music.play()

                if hand\_sign\_id == 2 and f3:

                    f2 = True

                    f1  = True

                    f3= False

                    print(keypoint\_classifier\_labels[2])

                    mixer.music.load("music/strangers.mp3")

                    mixer.music.play()

                # drawing

                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])

        else:

            point\_history.append([0, 0])

        debug\_image = draw\_info(debug\_image, fps, mode, number)

        cv.imshow('Hand Gesture Recognition', debug\_image)

    cap.release()

    cv.destroyAllWindows()

def select\_mode(key, mode):

    number = -1

    if 48 <= key <= 57:  # 0 ~ 9

        number = key - 48

    if key == 110:  # n

        mode = 0

    if key == 107:  # k

        mode = 1

    if key == 104:  # h

        mode = 2

    return number, mode

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]

def calc\_landmark\_list(image, landmarks):

    image\_width, image\_height = image.shape[1], image.shape[0]

    landmark\_point = []

    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.append([landmark\_x, landmark\_y])

    return landmark\_point

def pre\_process\_landmark(landmark\_list):

    temp\_landmark\_list = copy.deepcopy(landmark\_list)

    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

    temp\_landmark\_list = list(

        itertools.chain.from\_iterable(temp\_landmark\_list))

    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

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)

    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

    temp\_point\_history = list(

        itertools.chain.from\_iterable(temp\_point\_history))

    return temp\_point\_history

def logging\_csv(number, mode, landmark\_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])

    return

def draw\_landmarks(image, landmark\_point):

    if len(landmark\_point) > 0:

        cv.line(image, tuple(landmark\_point[2]), tuple(landmark\_point[3]),

                (0, 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]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[3]), tuple(landmark\_point[4]),

                (255, 255, 255), 2)

        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)

        cv.line(image, tuple(landmark\_point[9]), tuple(landmark\_point[10]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[9]), tuple(landmark\_point[10]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[10]), tuple(landmark\_point[11]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[10]), tuple(landmark\_point[11]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[11]), tuple(landmark\_point[12]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[11]), tuple(landmark\_point[12]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[13]), tuple(landmark\_point[14]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[13]), tuple(landmark\_point[14]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[14]), tuple(landmark\_point[15]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[14]), tuple(landmark\_point[15]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[15]), tuple(landmark\_point[16]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[15]), tuple(landmark\_point[16]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[17]), tuple(landmark\_point[18]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[17]), tuple(landmark\_point[18]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[18]), tuple(landmark\_point[19]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[18]), tuple(landmark\_point[19]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[19]), tuple(landmark\_point[20]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[19]), tuple(landmark\_point[20]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[0]), tuple(landmark\_point[1]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[0]), tuple(landmark\_point[1]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[1]), tuple(landmark\_point[2]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[1]), tuple(landmark\_point[2]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[2]), tuple(landmark\_point[5]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[2]), tuple(landmark\_point[5]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[5]), tuple(landmark\_point[9]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[5]), tuple(landmark\_point[9]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[9]), tuple(landmark\_point[13]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[9]), tuple(landmark\_point[13]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[13]), tuple(landmark\_point[17]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[13]), tuple(landmark\_point[17]),

                (255, 255, 255), 2)

        cv.line(image, tuple(landmark\_point[17]), tuple(landmark\_point[0]),

                (0, 0, 0), 6)

        cv.line(image, tuple(landmark\_point[17]), tuple(landmark\_point[0]),

                (255, 255, 255), 2)

    return image

def draw\_bounding\_rect(use\_brect, image, brect):

    if use\_brect:

        cv.rectangle(image, (brect[0], brect[1]), (brect[2], brect[3]),

                     (0, 0, 0), 1)

    return image

def draw\_info\_text(image, brect, handedness, hand\_sign\_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)

    return image

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)

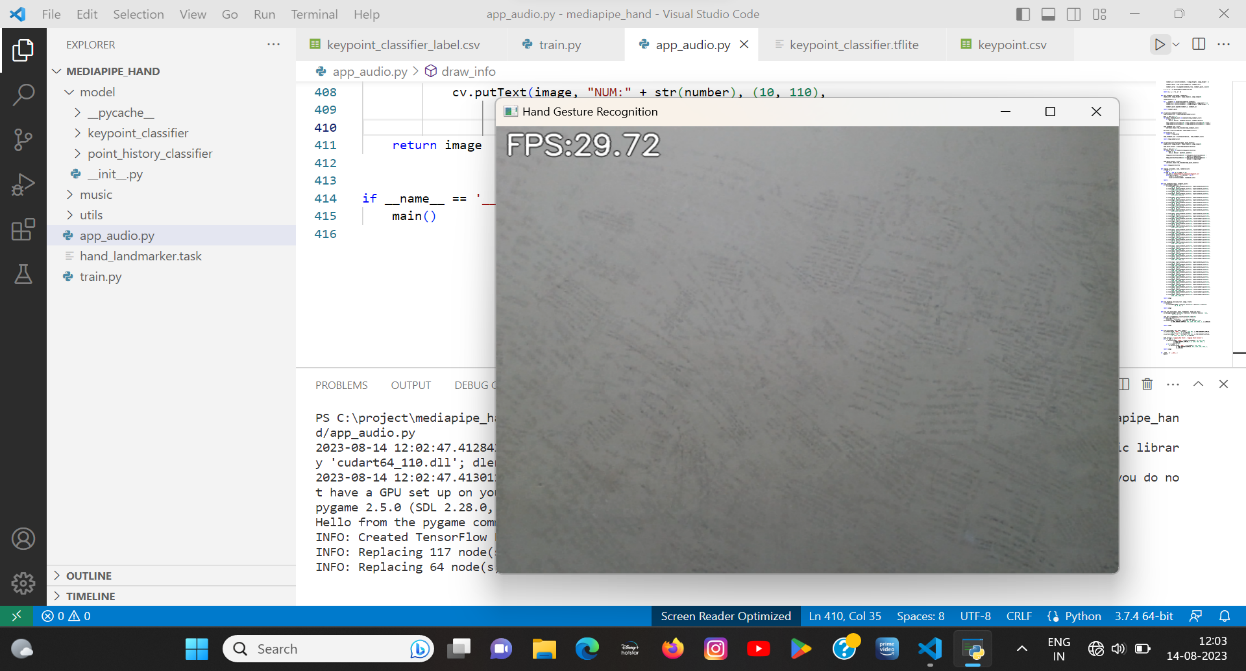
    return image

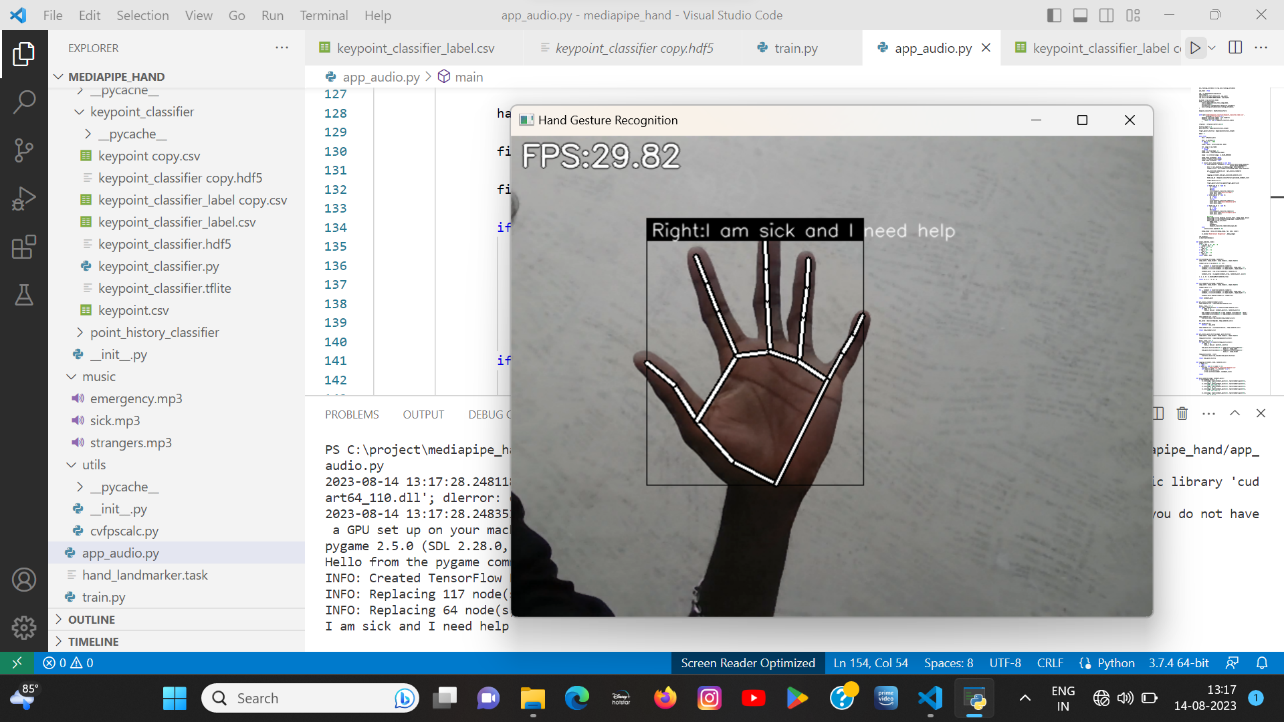
if \_\_name\_\_ == '\_\_main\_\_':

    main()

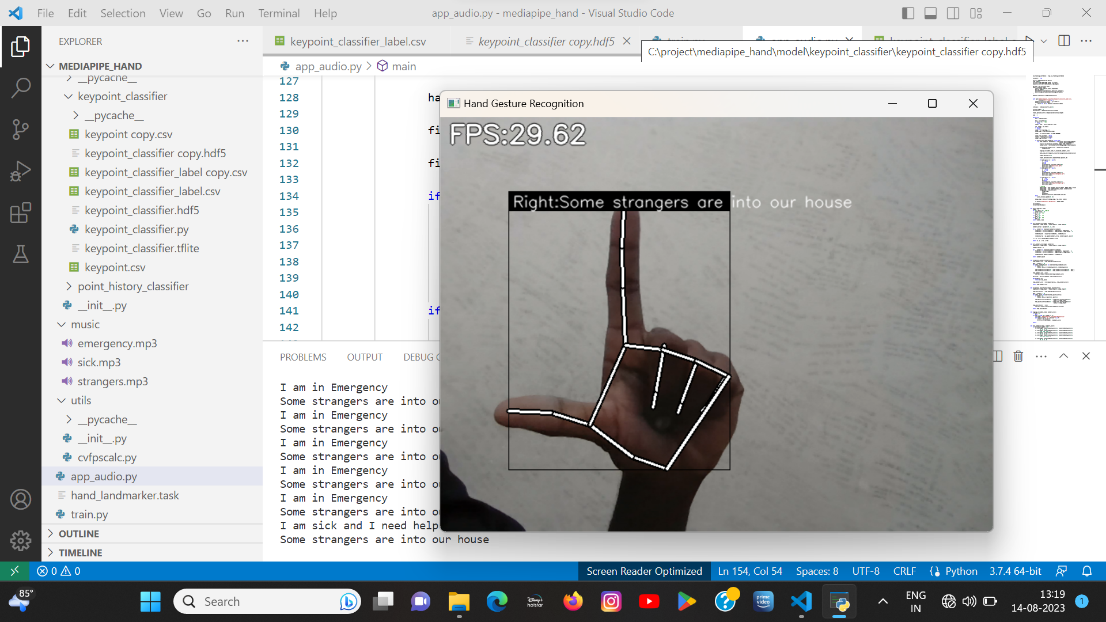
* **CAMERA OPENING**

**Figure8.1 camera opening**

**HAND DETECTION IMAGES**

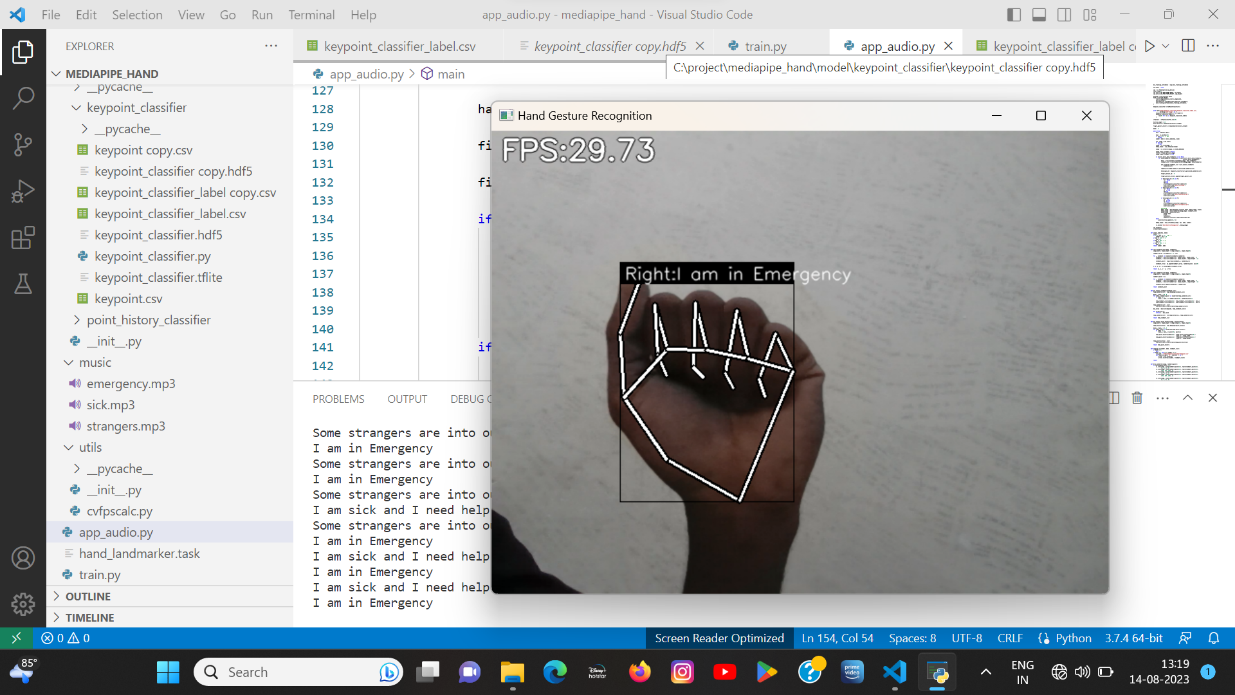
Figure8.2 detecting the sign using hand

(I am sick and I need help)

****

**Figure8.3** detecting the sign using hand

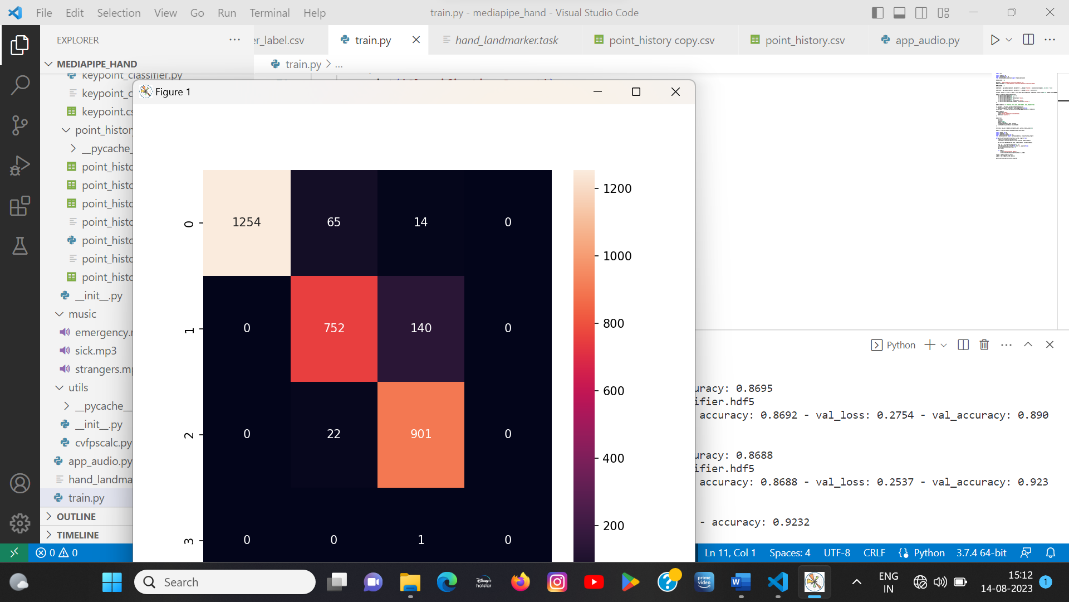
(some strangers are into our house)

****

**Figure8.4** detecting the sign using hand

**(**I am in emergency)

TESTING OF MATRIX FORM

****

**Figure8.5** testing of matrix form

* 1. **CONCLUSION**

In conclusion, the hand gesture recognition system developed using the Mediapipe library and Multi-layer Perceptron (MLP) algorithm presents a significant advancement in assistive technology, providing physically challenged individuals with an intuitive and accessible means of interacting with computers and devices. By accurately interpreting hand gestures in real-time, the system empowers these individuals with increased independence and control, enabling them to seamlessly communicate, access applications, and navigate digital environments. The successful implementation of this system opens new possibilities for inclusive human-computer interaction, fostering a more equitable and connected society where technology becomes a catalyst for improving the lives of all individuals, regardless of their physical abilities.

* **FUTURE SCOPE OF THE PROJECT**

The future scope is as follows

1. Mobile and Embedded Devices: Optimize the hand gesture recognition system to run efficiently on mobile devices, IoT devices, and embedded systems. This would extend its accessibility to a broader range of devices and applications.
2. Gesture-based Gaming: Integrate the hand gesture recognition system into gaming applications, enabling users to interact with virtual environments and control gameplay through gestures. This could lead to more immersive and interactive gaming experiences.

**10.BIBLIOGRAPH**Y

[1] Ustunug A, Cevikcan, Industry 4.0: Managing The Digital Transformation, Springer Series in Advanced Manufacturing, Switzerland. 2018. DOI: https://doi.org/10.1007/978-3-319-57870-5.

[2] Pantic M, Nijholt A, Pentland A, Huanag TS, Human-Centered Intelligent Human-Computer Interaction (HCI2): How Far We From Attaining It?,International Jounal of Autonomous and Adaptive Communications Systems (IJAACS), vol.1 no.2, 2008. pp 168-187. DOI: 10.1504/IJAACS.2008.019799 .

[3] Hamed Al-Saedi A.K, Hassin Al-Asadi A, Survey of Hand Gesture Recognition System. IOP Conferences Series: Journal of Physics: Conferences Series 1294 042003. 2019. DOI: https://doi.org/10.1088/1742-6596/4/042003.

[4] S.Rautaray S, Agrawal A. Vision Based Hand Gesture Recognition for Human Computer Interaction: A Survey. Springer Artificial Intelligence Review. 2012. DOI: https://doi.org/10.1007/s10462-012-9356-9.

[5] Lugaresi C, Tang J, Nash H, McClanahan C, et al. MediaPipe: A Framework for Building Perception Pipelines. Google Research. 2019. https://arxiv.org/abs/2006.10214.