PROJECT - PART B: REPORT

CSI 4133 - COMP. METHODS IN PICTURE PROCESSING AND ANALYSIS

Fall 2022

School of Engineering and Computer Science

University of Ottawa

Course Coordinator: Dr. Jiying Zhao

Teaching Assistant: Christopher McIntyre Garciam

Student Name: Jayden Lachhman

Student Number: 8791694

Submission date: 2022/12/02

Introduction

This report will explore hand detection and tracking, finger detection and tracking, and neural network training with the overarching goal of designing a program capable of hand gesture recognition. This will be achieved by using a program that requires training data to learn properties of certain classifications of hand gestures so that it becomes better at classifying new data accurately.

Approach

Given that hands and fingers come in different shapes, colors, and sizes, it would be impractical to depend on a comprehensive dataset to effectively analyze and correctly classify hand gestures. Instead, a machine learning approach (specifically a deep learning approach) will be used to teach our program how to learn hand gestures. This approach can be summarized by an iterative three-step process; the first involving the collection of training data which depends on the hand landmark model built into MediaPipe (a module within Python) to provide us with a framework from which we can extract information about the hands, the second involving the classification of this training data, and third the live analysis of a hand from the webcam of which comparisons to the training data are made to classify the new data accurately. The code references an existing repository for hand detection accessible here: https://github.com/kinivi/hand-gesture-recognition-mediapipe

Procedure

Step 1: The capture properties are initialized with reference to the device camera's width and height along with the tracking confidence threshold values which help us calibrate the MediaPipe model.

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

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


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
```

Figure 1: Argument Initializations

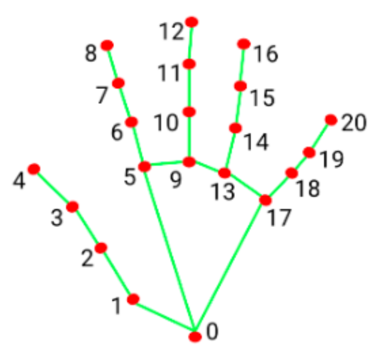
```python
def main():
    # Argument parsing #################################################################
    args = get_args()

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

    # Model load #############################################################
    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,
    )
```

Figure 2: Model Initializations

Hand Landmark Model

After the palm detection over the whole image our subsequent hand landmark model performs precise keypoint localization of 21 3D hand-knuckle coordinates inside the detected hand regions via regression, that is direct coordinate prediction. The model learns a consistent internal hand pose representation and is robust even to partially visible hands and self-occlusions.

To obtain ground truth data, we have manually annotated ~30K real-world images with 21 3D coordinates, as shown below (we take Z-value from image depth map, if it exists per corresponding coordinate). To better cover the possible hand poses and provide additional supervision on the nature of hand geometry, we also render a high-quality synthetic hand model over various backgrounds and map it to the corresponding 3D coordinates.



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

Figure 2: Hand Landmark Model

Step 2: In order for the program to classify the properties of input data, classes need to exist. Thus we create some .csv files which store the labels for each of the different classes the program will refer to when attempting to classify new hand gestures.

```python
    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
        ]

    # FPS Measurement ########################################################
    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)
```

Figure 4: Keypoint Classifier Label

Step 3: Since MediaPipe requires the frames in RGB format and reads a mirrored reflection of the input video file, some presets are required to prepare the environment. A while loop is used since each frame needs to be sequentially acknowledged (meaning of *ret* variable), flipped, and converted from BGR to RGB.

```python
    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 is used to detect if a frame is available to read
        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
```

Figure 5: Environment Initialization

Step 4: MediaPipe and the Hand Landmark Model help us generate numerical information about the input data, but this information needs to be reformatted in order to handle it properly. This starts with the *hand_landmarks* list which reads the x, y, and z coordinates of the detected hand in each frame. These coordinates do not signify a hand gesture independently, but in relation to one another they can. Thus, after creating the region of interest (ROI) referred to by the *brect* variable, the raw pixel coordinates are converted into relative coordinates where eventually the distance between points can be compared to distances between points from the training data in accordance with the Hand Landmark Model. These pre-processed points are then entered into the .csv.

```python
        # hand_landmarks represents the x, y, z coordinates of the detected hand in each frame
        # handedness represents the orientation of a hand to track it under movement
        if results.multi_hand_landmarks is not None:
            for hand_landmarks, handedness in zip(results.multi_hand_landmarks,
                                                  results.multi_handedness):
                # The following print statement prints the x, y, z coordinates of the detected hand in each frame
                # print(hand_landmarks)

                # Bounding box calculation
                brect = calc_bounding_rect(debug_image, hand_landmarks)
                # Landmark calculation
                # landmark_list represents the list of hand_landmarks corresponding to the 21 landmarks on the hand landmark model
                landmark_list = calc_landmark_list(debug_image, hand_landmarks)
                # The following print statement prints the xy-coordinates corresponding to the wrist landmark in the hand landmark model
                # print(landmark_list[0])

                # Conversion to relative coordinates / normalized coordinates
                # This conversion is important because without normalizing the coordinates, the program would be reading exact pixel
                # coordinates instead of interpreting them relative to each other. The distance between landmark points characterizes
                # gestures most distinctly, otherwise, a hand on one side of the screen might be recognized differently to that same
                # hand doing the same gesture on the other side of the screen
                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)
```

Figure 6: Input Pre-Processing

Step 5: Extended form the previous step, pre-processing the coordinates helps us more accurately classify the hand gesture they might be demonstrating. This can be done by assigning a *base* variable to the wrist point on the Hand Landmark Model which serves to be the point the other points refer to. This is a crucial step because the raw coordinates present a large spectrum of potential interpretation, which is proportionally error prone. In this sense, if one were to attempt to classify a gesture solely based on the coordinates of each point independently, hand movement across the camera could produce false-positives/true-negatives. Using the wrist as a base means we measure movement from a relatively static point, which is less error prone.

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

    # Convert to relative coordinates
    # This sets the landmark point corresponding to the wrist point to zero (0) which is our base value. This is important
    # because now the distance of the other points on the hand landmark model are calcualted relative to this wrist point
    # so that our gestures are detectable regardless of where the hand is on the screen, which is more effective than trying
    # to detect gestures using independent pixel values
    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
```

Figure 7: Conversion to Wrist Relative Coordinates

Step 6: Extending again from the previous step, the relative coordinates are more useful than in their previous format, but can be improved still. The landmark displacement values can be made more distinctive by normalizing each to a range between -1 and 1 instead of calculating each across arbitrary ranges, and after this, the pre-processing stage is complete.

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

    # Normalization
    # The list of landmark values are relative to the wrist point, but the format of these values can be made easier
    # still for the program to recognize hand gestures from by normalizing the values to numbers between -1 and 1
    # by dividing their the values by the max in the list. This helps the program interpret more distinctly the points
    # of the landmark model when classifying the gestures.
    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
```

Figure 8: Normalization

Step 7: Each of the distinct hand gestures refers to an index in the *keypoint_classifier_label.csv* file from earlier. This is where they are matched and the resulting hand gesture is returned.

```python
                # Hand sign classification
                # This signifies that the neural network responds with the classification of "pointer" for the index of 2, which corresponds
                # to the index of each gesture classification in the other file
                hand_sign_id = keypoint_classifier(pre_processed_landmark_list)
                if hand_sign_id == 2:  # One 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]],
```

Figure 9: Hand and Finger Classification

```
model > keypoint_classifier > keypoint_classifier_label.csv
Five
Zero
One
Two
Three
Four
```

Figure 10: Classifier Lable CSV

Step 8: Collecting training data is important for supplementing our neural network with more data to use in its pool to invoke when classifying a hand gesture. So instead of importing a dataset, this program allows the efficient creation of a dataset. Below is a figure representing different modes once the camera has begun capturing frames, and inputting 'k' will allow the user to log the hand landmarks currently in frame. Then by pressing the number corresponding to the index of the class the user would like to populate training data for, the coordinates of the landmarks will be logged into the .csv file accessible through the Jupyter notebook where similar data for other classes are stored.

```python
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
```

Figure 9: Capture Mode

Model training

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

```
Output exceeds the size limit. Open the full output data in a text editor
Epoch 1/1000
29/29 [==============================] - 2s 15ms/step - loss: 1.3853 - accuracy: 0.3360 - val_loss: 1.2779 - val_accuracy: 0.4244

Epoch 00001: saving model to model/keypoint_classifier/keypoint_classifier.hdf5
Epoch 2/1000
29/29 [==============================] - 0s 4ms/step - loss: 1.2943 - accuracy: 0.3780 - val_loss: 1.2151 - val_accuracy: 0.4703

Epoch 00002: saving model to model/keypoint_classifier/keypoint_classifier.hdf5
Epoch 3/1000
29/29 [==============================] - 0s 4ms/step - loss: 1.2524 - accuracy: 0.3749 - val_loss: 1.1472 - val_accuracy: 0.5572

Epoch 00003: saving model to model/keypoint_classifier/keypoint_classifier.hdf5
```

Figure 10: Model Training in Jupyter Notebook

The seven (7) steps detailed above describe the most functionally/conceptually critical points of the program but there are a number of helper functions that are instrumental in terms of coherence. These will be summarized below but separated from the linear sequence of principal steps detailed above.

The following code snippet (Figure 11) refers to the *logging_csv()* function where the training data for the hand gesture classification is stored. The more data entered, the more it helps increase the accuracy level of our gesture recognition program.

```
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

```

Figure 11: Logging CSV

The following code snippet (Figure 12) refers to the *draw_landmarks()* function which uses the Hand Landmark Model referenced earlier to identify the landmarks composing each finger.

```
def draw_landmarks(image, landmark_point):
    if len(landmark_point) > 0:
        # Thumb
        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)

        # 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)
```

Figure 12: Landmark Detection

The following code snippet (Figure 13) refers to the *draw_bounding_rect()* function which calculates the rectangle bounding the user's hand. The camera's dimensions include the user's hand, but there is excess information stored there as well, and thus the program needs to focus on extracting only pertinent data. This pertinence can be defined using a rectangular bound around the user's hand.

```python
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


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
```

Figure 13: Region of Interest

The following code snippet (Figure 14) refers to a couple of *draw…()* functions. Each adds a different textual element to the frame when the camera is recording, that element related to the measured frame-rate, mode of capture, etc.

```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


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()
```

Figure 14: Info

The following snippet from the jupyter file shows how the new data inputs are being synthesized with the existing training data to increase the accuracy of this particular hand gesture classification.

```
Epoch 58/1000
21/30 [===================>.........] - ETA: 0s - loss: 0.6496 - accuracy: 0.7612
Epoch 58: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
30/30 [==============================] - 0s 5ms/step - loss: 0.6706 - accuracy: 0.7511 - val_loss: 0.3491 - val_accuracy: 0.8
909
Epoch 59/1000
20/30 [===================>.........] - ETA: 0s - loss: 0.6595 - accuracy: 0.7523
Epoch 59: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
30/30 [==============================] - 0s 5ms/step - loss: 0.6592 - accuracy: 0.7542 - val_loss: 0.3456 - val_accuracy: 0.8
973
Epoch 60/1000
19/30 [==================>..........] - ETA: 0s - loss: 0.6836 - accuracy: 0.7405
Epoch 60: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
30/30 [==============================] - 0s 5ms/step - loss: 0.6667 - accuracy: 0.7495 - val_loss: 0.3440 - val_accuracy: 0.8
973
Epoch 61/1000
22/30 [=====================>........] - ETA: 0s - loss: 0.6711 - accuracy: 0.7454
Epoch 61: saving model to model/keypoint_classifier\keypoint_classifier.hdf5
30/30 [==============================] - 0s 6ms/step - loss: 0.6724 - accuracy: 0.7402 - val_loss: 0.3328 - val_accuracy: 0.9
021
```

Figure 15: Learning

Data

The following three (3) images correspond to the hand gesture classifications determined by the program. Each hand gesture refers to a certain number of fingers raised.

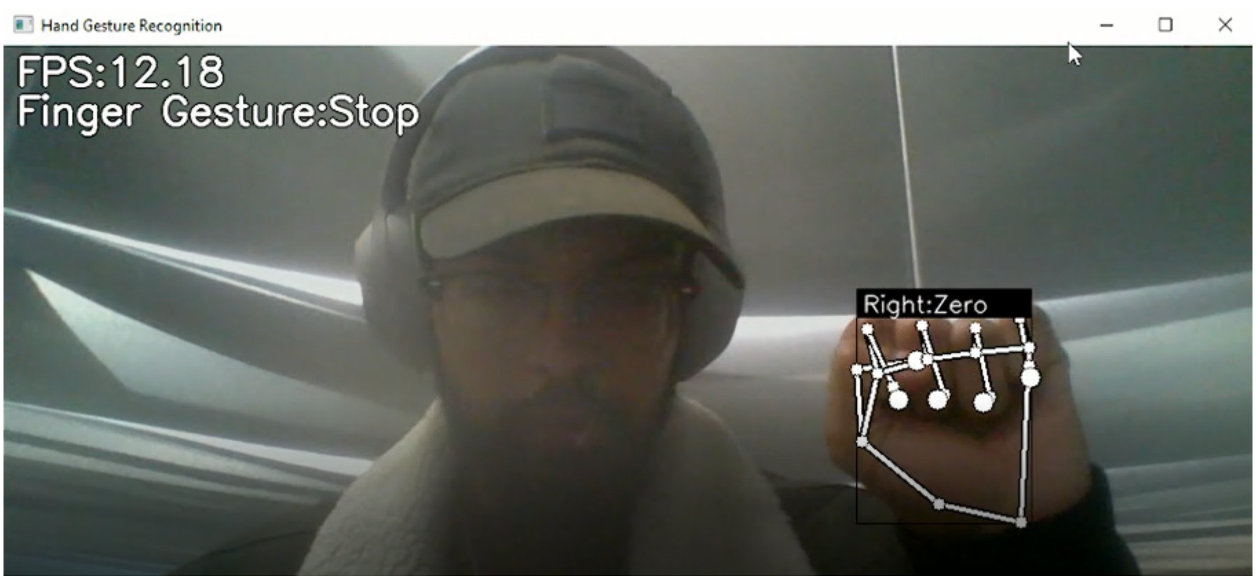


Figure 1: Zero Fingers Raised

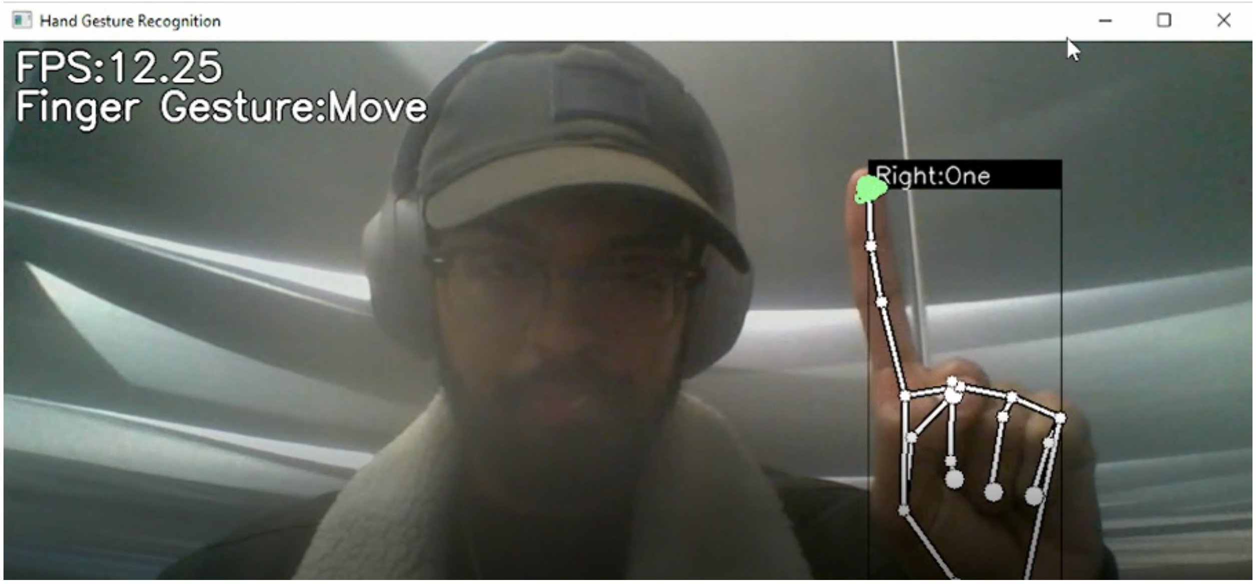


Figure 2: One Finger Raised

Figure 3: Five Fingers Raised

Conclusion

Using the MediaPipe module with reference to the Hand Landmark Model, a couple of .csv files for logging and storing training data for a neural network, and a jupyter notebook for synchronizing new data inputs (frames) with others of the same hand gesture classification, we were able to design a program that helps us analyze hand gestures from a live video feed. The accuracy of