**Code Explanation:**

**Import:**

**Import Libraries**

**import cv2**

**import numpy as np**

**import math**

**import pickle as pkl**

**import os**

**import mediapipe as mp**

* **cv2**: OpenCV, used for capturing video and processing images.
* **numpy**: Provides support for working with arrays, in this case for matrices.
* **math**: Contains mathematical functions (like square root), used in calculations.
* **pickle**: For loading and saving Python objects (like gesture data) in a serialized format.
* **os**: Interacts with the operating system to check file existence.
* **mediapipe**: Google's library for hand landmark detection.

**Video Capture Setup:**

**w, h = 640, 480**

* w and h: Set the width and height of the video feed to 640x480 pixels.

**cam = cv2.VideoCapture(0)**

* cv2.VideoCapture(0): Opens the default camera (usually the webcam). The 0 refers to the default camera.

**cam.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, h)**

**cam.set(cv2.CAP\_PROP\_FRAME\_WIDTH, w)**

**cam.set(cv2.CAP\_PROP\_FPS, 30)**

**cam.set(cv2.CAP\_PROP\_FOURCC, cv2.VideoWriter\_fourcc(\*"MJPG"))**

* CAP\_PROP\_FRAME\_HEIGHT and CAP\_PROP\_FRAME\_WIDTH: Set the resolution of the video to w x h (640x480).
* CAP\_PROP\_FPS: Sets the frame rate to 30 frames per second.
* CAP\_PROP\_FOURCC: Specifies the video codec as "MJPG" (Motion JPEG) to compress the video frames

**Load Gesture Data:**

**keypoints = [0, 4, 5, 8, 9, 12, 13, 16, 17, 20]**

* The keypoints in this code are specific landmarks on the hand that are used to recognize gestures. MediaPipe's hand-tracking model detects 21 landmarks on each hand, but here we are focusing on specific keypoints that are important for gesture recognition.

**Key Points (Landmarks):**

keypoints = [0, 4, 5, 8, 9, 12, 13, 16, 17, 20]

Each of these numbers corresponds to specific points on the hand. These points are the hand landmarks detected by MediaPipe and are labeled from **0** to **20**. Here’s what each of these points represents:

1. **0** - **Wrist**: This is the base point of the hand, located at the wrist.
2. **4** - **Tip of the Thumb**: This is the tip of the thumb.
3. **5** - **Base of the Index Finger**: The point where the index finger meets the palm.
4. **8** - **Tip of the Index Finger**: The tip of the index finger.
5. **9** - **Base of the Middle Finger**: The point where the middle finger connects with the palm.
6. **12** - **Tip of the Middle Finger**: The tip of the middle finger.
7. **13** - **Base of the Ring Finger**: The point where the ring finger connects with the palm.
8. **16** - **Tip of the Ring Finger**: The tip of the ring finger.
9. **17** - **Base of the Little Finger**: The point where the little finger meets the palm.
10. **20** - **Tip of the Little Finger**: The tip of the little finger.

**Why These Keypoints?**

These specific points were chosen because they represent significant parts of the hand that are critical for recognizing gestures:

* **Finger Tips (points 4, 8, 12, 16, 20)**: The tips of the fingers are essential for distinguishing between open and closed gestures.
* **Finger Bases (points 5, 9, 13, 17)**: These are important for calculating finger positioning relative to the palm.
* **Wrist (point 0)**: The wrist is a stable reference point to calculate distances and normalize hand positions.

**Usage in the Code:**

* The **distances** between these key points are calculated using the finddistances function.
* These distances are compared to known gesture templates to recognize specific gestures using the findgesture function.

**tol = 15**

* **tol** is defined as 15, and it plays a role in comparing the error (or difference) between the distances of the detected hand landmarks and the distances of known gestures.

**File checker:**

**file\_name = 'gestures.pkl'**

**if os.path.exists(file\_name):**

**with open(file\_name, 'rb') as f:**

**gestnames = pkl.load(f)**

**knowngesture = pkl.load(f)**

**else:**

**print(f"File '{file\_name}' does not exist.")**

**exit()**

* Opening a file in 'rb' mode means that the file is opened for reading (r) in binary (b) mode
* pkl.load(f): Loads the gesture names (gestnames) and gesture matrices (knowngesture) from the file.

**Handmarks Class:**

**def \_\_init\_\_(self, hc=2, tol1=0.5, tol2=0.5):**

**self.hands = mp.solutions.hands.Hands(static\_image\_mode=False, max\_num\_hands=hc, min\_detection\_confidence=tol1, min\_tracking\_confidence=tol2)**

**self.h\_draw = mp.solutions.drawing\_utils**

* \_\_init\_\_: Constructor for the Handmarks class.
* hc=2: Specifies that up to 2 hands can be tracked.
* **tol1=0.5** (detection confidence threshold):
* This sets the minimum confidence value required for the hand detection to be considered valid.
* **tol2=0.5** (tracking confidence threshold):
* This sets the minimum confidence for tracking the hand landmarks over frames. A value of 0.5 ensures consistent tracking as long as it is reasonably confident.
* **mp.solutions.hands.Hands()**: Initializes the MediaPipe hand-tracking model.
* **static\_image\_mode=False**: This means the hand tracking is intended for real-time video.
* **max\_num\_hands=hc**: Specifies the number of hands to detect (default is 2).
* **min\_detection\_confidence=tol1**: The confidence threshold for initial detection.
* **min\_tracking\_confidence=tol2**: The confidence threshold for tracking the detected hands.
* **self.h\_draw**: Initializes a drawing utility to visualize hand landmarks.

**def h\_marks(self, val):**

**valrgb = cv2.cvtColor(val, cv2.COLOR\_BGR2RGB)**

**results = self.hands.process(valrgb)**

**hand\_marks = []**

**if results.multi\_hand\_landmarks:**

**for marks in results.multi\_hand\_landmarks:**

**hand\_mark = [(int(mark.x \* w), int(mark.y \* h)) for mark in marks.landmark]**

**hand\_marks.append(hand\_mark)**

**return hand\_marks**

**h\_marks**: Detects hand landmarks from the video frame (val).

* **valrgb**: Converts the frame from BGR to RGB format because MediaPipe requires RGB input.
* **results = self.hands.process(valrgb)**: Processes the RGB frame (valrgb) using the MediaPipe hand model, which detects hand landmarks and returns the results in the **results** variable.
* **hand\_marks**: Stores the (x, y) coordinates of each hand landmark after scaling them to the video size (w, h).
* **if results.multi\_hand\_landmarks:**

Checks if any hands have been detected. If hands are detected, they will be stored in **results.multi\_hand\_landmarks**.

* **for marks in results.multi\_hand\_landmarks:**

Iterates through each detected hand's landmarks.

* **hand\_mark = [(int(mark.x \* w), int(mark.y \* h)) for mark in marks.landmark]**:

For each detected hand, this line converts the normalized coordinates of the landmarks into actual pixel coordinates based on the width (w) and height (h) of the video frame.

**mark.x** and **mark.y**: Represent the relative x and y positions of each landmark on the hand. These values are scaled up to match the pixel coordinates of the frame.

* **hand\_marks.append(hand\_mark)**:

Adds the list of landmarks for the current hand to the **hand\_marks** list.

* **return hand\_marks**:

Finally, the method returns a list of landmark positions for each detected hand. If no hands are detected, an empty list is returned.

* **def finddistances(handdata):**

**distmatrix = np.zeros((len(handdata), len(handdata)), dtype='float')**

**palm\_size = ((handdata[0][0] - handdata[9][0]) \*\* 2 + (handdata[0][1] - handdata[9][1]) \*\* 2)**

**for i in range(len(handdata)):**

**for j in range(len(handdata)):**

**distmatrix[i][j] = math.sqrt(((handdata[i][0] - handdata[j][0]) \*\* 2 + (handdata[i][1] - handdata[j][1]) \*\* 2) / palm\_size)**

**return distmatrix**

Let’s say handdata is a list of coordinates for five landmarks:

handdata = [(100, 150), (110, 160), (120, 170), (130, 180), (140, 190)]

The function will create a 5x5 matrix (distmatrix).

It computes the Euclidean distances between every pair of landmarks, normalizes the distances using the palm\_size, and stores the result in distmatrix.

**Visualization of distmatrix (example)**

For 5 landmarks, it would look something like this:

[[0.0, 0.1, 0.2, 0.3, 0.4],

[0.1, 0.0, 0.15,0.25,0.35],

[0.2, 0.15, 0.0, 0.2, 0.3],

[0.3, 0.25, 0.2, 0.0, 0.15],

[0.4, 0.35, 0.3, 0.15, 0.0]]

Each entry represents the distance between landmarks i and j.

**Summary:**

The finddistances function calculates a normalized distance matrix for the hand landmarks, which is later used to compare the current hand gesture with known gestures for recognition purposes.

**def find\_error(knownmatrix, unknownmatrix, keypoints):  
 error = 0  
 for i in keypoints:  
 for j in keypoints:  
 error += abs(knownmatrix[i][j] - unknownmatrix[i][j])  
 return error**

**Purpose**: This function computes the total error between the distances of known and unknown hand gestures.

**Parameters**:

* knownmatrix: Distance matrix of a known gesture.
* unknownmatrix: Distance matrix of the current (unknown) gesture.
* keypoints: Specific hand landmarks used for comparison.

**Process**:

* Initializes error to 0.
* Loops through each pair of keypoints and calculates the absolute difference between their distances in the known and unknown matrices.
* Sums these differences to get the total error.

**Return**: The total error value, which helps determine how closely the unknown gesture matches the known gestures. Lower error indicates a better match.

**def findgesture(unknowngest, knowngest, keypoints, tol, gestnames):**

**errorarray = []**

**for i in range(len(gestnames)):**

**error = find\_error(knowngest[i], unknowngest, keypoints)**

**errorarray.append(error)**

**errormin = min(errorarray)**

**minindex = errorarray.index(errormin)**

**if errormin < tol:**

**gesture = gestnames[minindex]**

**else:**

**gesture = 'UNKNOWN'**

**return gesture**

**findgesture**: Identifies the closest gesture by comparing the unknown gesture with all known gestures.

* **errorarray**: Stores the error values between the unknown gesture and each known gesture.
* **min(errorarray)**: Finds the smallest error.
* **minindex**: The index of the gesture with the smallest error.
* If the error is less than the tolerance (tol), the corresponding gesture is selected; otherwise, the gesture is labeled as **UNKNOWN**.

**handmarks = Handmarks()**

**while True:**

**ret, frame = cam.read**() # Capture a video frame

**if not ret:**

**break** # Exit loop if frame not captured

**marks = handmarks.h\_marks(frame**) # Detect hand landmarks

**if marks:**

**unknownmatrix = finddistances(marks[0])** # Calculate distances between landmarks

**find\_gesture = findgesture(unknownmatrix, knowngesture, keypoints, tol, gestnames)**

# Recognize gesture

**cv2.putText(frame, find\_gesture, (20, 30), cv2.FONT\_HERSHEY\_PLAIN, 2, (0, 255, 255), 2)**

# Display gesture name

**cv2.imshow('Gesture Recognition', frame**) # Show video frame with gesture

**if cv2.waitKey(1) & 0xFF == ord('q'):** # Exit if 'q' is pressed

**break**

**cam.release()**

**cv2.destroyAllWindows()** # Release camera and close windows