Lab 1 Sound Analytics

audio_features.py

This Python script uses Librosa, NumPy, and Matplotlib to analyze a speech/audio signal stored in test.wav. It extracts and visualizes important audio features such as the spectrogram, chroma features, mel-spectrogram, and MFCCs (Mel-Frequency Cepstral Coefficients).

Load the Audio File

- y, sr = librosa.load("test.wav", sr=None)
    - Loads the audio file test.wav and stores its waveform (y) and sample rate (sr).

Compute Spectrogram Magnitude and Phase

- S_full, phase = librosa.magphase(librosa.stft(y))
    - Uses Short-Time Fourier Transform (STFT) to compute the spectrogram.
    - Separates magnitude and phase information.

Plot Time Series and Spectrogram

- ax1.plot(y) # Time-domain representation
- img = librosa.display.specshow(librosa.amplitude_to_db(S_full, ref=np.max), y_axis='log', x_axis='time', sr=sr, ax=ax2)
    - Plots the raw waveform.
    - Displays the spectrogram, which shows how frequency content varies over time.

Compute and Display Chroma Features

- chroma = librosa.feature.chroma_stft(S=S, sr=sr)
    - Extracts chroma features, which represent musical pitch content.

Compute and Display Mel-Spectrogram

- S_mel = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128, fmax=8000)
    - Converts the spectrogram into a Mel scale representation, commonly used in speech and music analysis.

Compute and Display MFCCs (Mel-Frequency Cepstral Coefficients)

- mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=40)
    - Extracts MFCCs, which are widely used in speech and audio recognition.

Filtering_audio.py

This Python script records live audio using PyAudio, applies a bandpass filter, and visualizes the waveform in real-time using Matplotlib. It also measures the execution time of filtering operations.

Set Audio Parameters

```
BUFFER = 1024 * 16          # Samples per frame
FORMAT = pyaudio.paInt16    # 16-bit audio format
CHANNELS = 1                # Single-channel (mono) audio
RATE = 44100                # Sample rate (Hz)
RECORD_SECONDS = 20         # Duration of recording
```

- Configures the microphone settings for audio capture.

Initialize Plot for Real-Time Waveform Visualization

```
fig, (ax1, ax2) = plt.subplots(2, figsize=(7, 7))
line, = ax1.plot(x, np.random.rand(BUFFER), '-', lw=2)
line_filter, = ax2.plot(x, np.random.rand(BUFFER), '-', lw=2)
```

- Creates a Matplotlib figure with two subplots:
  - Raw audio waveform
  - Filtered waveform (after bandpass filtering)

Design a Bandpass Filter

```
def design_filter(lowfreq, highfreq, fs, order=3):
        nyq = 0.5 * fs
        low = lowfreq / nyq
        high = highfreq / nyq
        sos = butter(order, [low, high], btype='band', output='sos')
        return sos
```

- Uses the Butterworth filter to isolate frequencies between 19.4 kHz - 19.6 kHz.

Initialize PyAudio and Open a Stream

```
audio = pyaudio.PyAudio()
stream = audio.open(format=FORMAT, channels=CHANNELS, rate=RATE, input=True,
output=True, frames_per_buffer=BUFFER)
```

- Starts capturing live microphone input.

Important Highlights

- STFT & Spectrogram: Used to analyze frequency variations over time.
- Chroma Features: Identify harmonic content.
- Mel-Spectrogram & MFCCs: Used in speech processing and classification tasks.
- Visualization: Matplotlib is used to plot time-domain, frequency-domain, and feature representations.

Process and Filter Audio in Real-Time

```
data = stream.read(BUFFER)  # Read microphone data
data_int = struct.unpack(str(BUFFER) + 'h', data)  # Convert to integers
yf = sosfilt(sos, data_int)  # Apply bandpass filter
```

- Reads audio, converts it from bytes to integer values, and applies bandpass filtering.

Update the Live Plot

```
line.set_ydata(data_int)
line_filter.set_ydata(yf)
fig.canvas.draw()
fig.canvas.flush_events()
```

- Updates the waveform plot in real time.

Measure Execution Time

```
exec_time.append(time.time() - start_time)
print('average execution time = {:.0f} milli seconds'.format(np.mean(exec_time) * 1000))
```

- Measures and prints the average execution time per frame.

Important Highlights

- PyAudio Stream: Captures real-time audio.
- Bandpass Filtering: Uses a Butterworth filter to retain only frequencies in a specified range.
- Real-Time Visualization: Updates waveform plots dynamically.
- Execution Time Measurement: Evaluates the performance of the filtering process.

Microphone_streaming_with_spectrum.py

This code sets up a real-time audio processing system using PyAudio for capturing audio, NumPy for FFT (Fast Fourier Transform), and Matplotlib for visualization. It records audio from a microphone, processes it, and updates two plots:

- Time-domain waveform (representing the raw audio signal).
- Frequency spectrum (showing the frequency components of the signal).

Parameters

- BUFFER: Defines the number of samples per frame (audio chunk). This determines how much audio data is captured at each iteration.
- FORMAT: Specifies the audio format (16-bit PCM format).
- CHANNELS: Indicates the number of audio channels (mono audio here).
- RATE: The sample rate (44.1 kHz in this case).
- RECORD_SECONDS: The total duration (in seconds) for audio recording.

Plot Setup

- Creates two subplots using matplotlib:
    - ax1: The audio waveform.
    - ax2: The frequency spectrum (FFT).

```
line, = ax1.plot(x, np.random.rand(BUFFER), '-', lw=2)
line_fft, = ax2.plot(xf, np.random.rand(BUFFER//2), '-', lw=2)
```

    - These lines create the initial line objects for the waveform and frequency spectrum.

PyAudio Stream Setup

- The pyaudio.PyAudio() instance is used to set up an audio stream for input and output.

```
stream = audio.open(format=FORMAT, channels=CHANNELS, rate=RATE, input=True, output=True, frames_per_buffer=BUFFER)
```

Audio Data Capture and Processing

- For the duration of RECORD_SECONDS, the code reads chunks of audio (BUFFER samples), processes them, and updates the plots:
    - Converts raw binary data into 16-bit integers.
    - Computes the FFT to analyze the frequency content.

```
yf = fft(data_int)
```

Plot Updates

- After each FFT calculation, the waveform and frequency spectrum are updated in real-time:

```
line.set_ydata(data_int)
line_fft.set_ydata(2.0/BUFFER * np.abs(yf[0:BUFFER//2]))
```

- These lines update the y-data of the waveform and spectrum plots.

Stream Termination:
After the loop ends, the stream is terminated using `audio.terminate()`, and the average execution time for FFT calculation is printed.

Microphone_recognition.py

This script performs speech recognition using two engines:

- Google Speech Recognition API - Recognizes speech and converts it into text.
- Sphinx - An offline speech recognition engine (also known as CMU Sphinx).

Recording from Microphone

- r = sr.Recognizer() # Initializing the Recognizer class
  with sr.Microphone() as source:
  r.adjust_for_ambient_noise(source) # Important step to adjust for ambient noise
  os.system('clear') # Clears terminal screen
  print("Say something!")
  audio = r.listen(source) # Listen to the microphone
    - r = sr.Recognizer(): Creates an instance of the Recognizer class that will be used for speech recognition.
    - with sr.Microphone() as source:: This line opens the microphone to listen to the user's speech.
    - r.adjust_for_ambient_noise(source): Adjusts the recognizer to account for any background noise. This step is important to ensure that only the speech is considered for recognition.
    - os.system('clear'): Clears the terminal screen before prompting the user to speak.
    - print("Say something!"): Informs the user to speak after this message.
    - audio = r.listen(source): This captures the user's speech through the microphone.

Recognition with Google Speech API

```
start_time = time.time()  # start time
try:
        print("Google Speech Recognition thinks you said " + r.recognize_google(audio))
except sr.UnknownValueError:
        print("Google Speech Recognition could not understand audio")
except sr.RequestError as e:
        print("Could not request results from Google Speech Recognition service; {0}".format(e))
print('Time for Google Speech Recognition recognition = {:.0f}
```

- seconds'.format(time.time()-start_time))
- start_time = time.time(): Marks the start time for the recognition process.
- r.recognize_google(audio): Sends the captured audio to Google's Speech Recognition service for transcription. This will return the transcribed text.
- except sr.UnknownValueError:: If the audio is unclear or not recognized, this error is caught.
- except sr.RequestError as e:: Handles any issues with the request to the Google API, like network problems or API key issues.

- print('Time for Google Speech Recognition recognition = {:.0f} seconds'.format(time.time()-start_time)): Calculates and prints the time taken for recognition.

Recognition with Sphinx

```
start_time = time.time()  # start time
try:
    print("Sphinx thinks you said " + r.recognize_sphinx(audio))
except sr.UnknownValueError:
    print("Sphinx could not understand audio")
except sr.RequestError as e:
    print("Sphinx error; {0}".format(e))
print('Time for Sphinx recognition = {:.0f} seconds'.format(time.time()-start_time))
```

start_time = time.time(): Starts timing the Sphinx recognition.

- r.recognize_sphinx(audio): Uses the Sphinx engine to recognize the audio (offline recognition).
- except sr.UnknownValueError:: If Sphinx cannot recognize the audio, this error is caught.
- except sr.RequestError as e:: If there's an issue with Sphinx, like a configuration or processing error, this error is caught.
- print('Time for Sphinx recognition = {:.0f} seconds'.format(time.time()-start_time)): Displays how long it took for the Sphinx engine to process the speech.

Lab 2 Sound Analytics

Image_capture_display.py

- This Python code uses OpenCV to capture video from a webcam and perform real-time color detection on the video stream. It segments the video frame into three color components (Red, Green, and Blue) and displays the original frame alongside the segmented images for each of these colors.
- This code allows real-time segmentation of video frames into red, green, and blue components based on specified color ranges. It captures frames from the webcam, processes them to isolate colors, and displays the results in a combined window. The user can stop the video capture by pressing the 'q' key.

Color Boundaries

- boundaries is a list of RGB (actually BGR for OpenCV) color ranges. These boundaries are used to detect specific colors in the video stream:
    - Red: [17, 15, 100] to [50, 56, 200]
    - Blue: [86, 31, 4] to [220, 88, 50]
    - Green: [25, 90, 4] to [62, 200, 50]

Normalization Function (normalizeImg)

- Converts the image to a float, normalizes the pixel values, and then scales it back to uint8 format (0-255 range) to enhance display.

Main Loop

- The program captures frames from the webcam continuously.
- For each frame, it iterates over the color boundaries, creates a mask, and uses cv2.bitwise_and to segment the colors from the frame.

Color Segmentation

- For each color boundary (Red, Green, and Blue):
    - cv2.inRange(frame, lower, upper) detects the pixels within the specified range.
    - cv2.bitwise_and(frame, frame, mask=mask) applies the mask to the frame, isolating that color.

Image_hog_feature.py

This script captures video frames from a webcam, processes them to extract Histogram of Oriented Gradients (HoG) features, and displays both the original frame and the HoG image side by side in real-time.

Preprocessing the Frame

```
# Converting to gray scale as HOG feature extraction works only on gray scale image
image = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

- cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY): Converts the captured frame to grayscale. HoG feature extraction requires grayscale images.

HoG Feature Extraction

```
(H, hogImage) = feature.hog(image, orientations=9, pixels_per_cell=(8, 8),
cells_per_block=(2, 2), transform_sqrt=True, block_norm="L1",
visualize=True)
```

- feature.hog(image, ...): Extracts the Histogram of Oriented Gradients (HoG) features from the image. Key parameters include:
    - orientations=9: Number of gradient orientation bins.
    - pixels_per_cell=(8, 8): Size of each cell (in pixels).
    - cells_per_block=(2, 2): Number of cells in each block.
    - transform_sqrt=True: Applies square root transformation to the gradient magnitudes.
    - block_norm="L1": Normalization method for blocks.

    - visualize=True: Returns both the HoG descriptor and the HoG image for visualization.

Intensity Rescaling for Visualization

- hogImage = exposure.rescale_intensity(hogImage, out_range=(0, 255))
- hogImage = hogImage.astype("uint8")
    - exposure.rescale_intensity(hogImage, out_range=(0, 255)): Rescales the intensity of the HoG image to the range of 0 to 255 for better visualization.
    - hogImage.astype("uint8"): Converts the image to uint8 type for display (values between 0 and 255).

image_human_capture.py

The code captures video from the webcam, detects people using a HOG (Histogram of Oriented Gradients) person detector, sorts the detected people by their proximity to the center of the frame, and then determines whether the detected person is centered or needs to turn left or right based on their position.

Setup

```
# initialize the HOG descriptor/person detector
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())
# sets how many pixels away from the center a person needs to be before the head stops
center_tolerance = 5
```

- HOG Descriptor: This is a feature descriptor used for detecting objects, in this case, people. It's pre-trained to recognize human shapes.
- center_tolerance: This variable defines how far the detected person can be from the center of the frame before we classify them as needing to move.

Person Detection

```
# detect people in the image
boxes, weights = hog.detectMultiScale(frame, winStride=(1, 1), scale=1.05)
boxes = np.array([[x, y, x + w, y + h] for (x, y, w, h) in boxes])
```

- hog.detectMultiScale(frame): Detects people in the frame, returning the bounding boxes of detected persons.
- The result is a list of boxes containing the coordinates (x, y, w, h).

Sorting Detected People Based on Center Position:

```
centers = []
for box in boxes:
    center_x = ((box[2]-box[0])/2) + box[0]  # Calculate the center of the bounding box
    x_pos_rel_center = (center_x - 70)  # Relative position to the center of the frame
    dist_to_center_x = abs(x_pos_rel_center)  # Distance from the center
    centers.append({'box': box, 'x_pos_rel_center': x_pos_rel_center, 'dist_to_center_x': dist_to_center_x})
```

- center_x: Calculates the center of each bounding box.
- x_pos_rel_center: Calculates the horizontal distance between the center of the detected person and the center of the frame.
- dist_to_center_x: Calculates the absolute distance to the center.

Sorting the Bounding Boxes by Proximity to the Center

```
if len(centers) > 0:
    sorted_boxes = sorted(centers, key=lambda i: i['dist_to_center_x'])  # Sort by distance to center
    center_box = sorted_boxes[0]['box']  # Get the closest box to the center
```

- Sorting: The list centers is sorted by the distance from the center. The person closest to the center is identified.

Drawing Bounding Boxes

```
for box in range(len(sorted_boxes)):
    if box == 0:
        # Display the closest detected person with a green bounding box
        cv2.rectangle(frame, (sorted_boxes[box]['box'][0], sorted_boxes[box]['box'][1]), (sorted_boxes[box]['box'][2], sorted_boxes[box]['box'][3]), (0, 255, 0), 2)
    else:
        # Display the others with a red bounding box
        cv2.rectangle(frame, (sorted_boxes[box]['box'][0], sorted_boxes[box]['box'][1]), (sorted_boxes[box]['box'][2], sorted_boxes[box]['box'][3]), (0, 0, 255), 2)
```

- cv2.rectangle(): Draws the bounding boxes around the detected people, highlighting the closest one with green and others with red.

Centering Logic (Turning Head)

```
Center_box_pos_x = sorted_boxes[0]['x_pos_rel_center']
if -center_tolerance <= Center_box_pos_x <= center_tolerance:
        print("center")  # Person is centered
elif Center_box_pos_x >= center_tolerance:
        print("right")  # Person needs to move right
elif Center_box_pos_x <= -center_tolerance:
        print("left")  # Person needs to move left
```

- Center_box_pos_x: Gets the relative position of the closest person to the center.
- Based on the position, it prints whether the head should stay centered, turn right, or turn left.

Image_face_capture.py

The script captures video frames from the webcam, uses MediaPipe's face mesh model to detect facial landmarks, and displays the results by drawing connections between the landmarks. The program will exit when the user presses the 'q' key.

Initialize MediaPipe Face Mesh

```
# Initialize Mediapipe Face Mesh
mp_face_mesh = mp.solutions.face_mesh
face_mesh = mp_face_mesh.FaceMesh(static_image_mode=False,
                                  max_num_faces=1,
                                  min_detection_confidence=0.5,
                                  min_tracking_confidence=0.5)
```

- FaceMesh: A MediaPipe solution that detects facial landmarks.
  - static_image_mode=False: This indicates that we want to process video frames in real-time.
  - max_num_faces=1: Only the first face is detected in each frame.
  - min_detection_confidence=0.5: A threshold for detecting faces with at least 50% confidence.
  - min_tracking_confidence=0.5: A threshold for tracking the detected face.

```
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles
```

- drawing_utils: A utility module from MediaPipe to draw the landmarks and connections.
- drawing_styles: This provides predefined styles for drawing the landmarks and their connections.

Process the Frame

```
results = face_mesh.process(rgb_frame) # Process the frame
```

- face_mesh.process(rgb_frame): Processes the RGB frame to detect facial landmarks.
- The results object contains the face landmarks if any faces are detected.

Draw Landmarks and Connections

```
if results.multi_face_landmarks:
    for face_landmarks in results.multi_face_landmarks:
        mp_drawing.draw_landmarks(
            image=frame,
            landmark_list=face_landmarks,
            connections=mp_face_mesh.FACEMESH_TESSELATION,
            landmark_drawing_spec=None,
connection_drawing_spec=mp_drawing_styles.get_default_face_mesh_tesselation_style()
        )
        mp_drawing.draw_landmarks(
            image=frame,
            landmark_list=face_landmarks,
            connections=mp_face_mesh.FACEMESH_CONTOURS,
            landmark_drawing_spec=None,
            connection_drawing_spec=mp_drawing_styles.get_default_face_mesh_contours_style()
        )
```

- if results.multi_face_landmarks: Checks if any faces were detected in the frame.
- draw_landmarks(): This function draws the facial landmarks and their connections.
    - FACEMESH_TESSELATION: The connections used for the face mesh tesselation (grid of triangles).
    - FACEMESH_CONTOURS: The connections for facial contours.
    - landmark_drawing_spec=None: This uses the default drawing specifications (e.g., color and thickness).
    - connection_drawing_spec: Specifies the style used to draw the face mesh contours and tesselations.

Lab 3 Video Analytics

optical_flow.py

This code performs real-time optical flow estimation and tracking using OpenCV. It captures video from a webcam and computes motion between consecutive frames using two methods: Lucas-Kanade optical flow and dense optical flow by Gunnar Farneback's method.

Parameters for Lucas-Kanade Optical Flow

```
feature_params = dict(maxCorners=100, qualityLevel=0.3, minDistance=7, blockSize=7)
lk_params = dict(winSize=(15,15), maxLevel=2, criteria=(cv2.TERM_CRITERIA_EPS |
cv2.TERM_CRITERIA_COUNT, 10, 0.03))
```

- feature_params define the parameters for Shi-Tomasi corner detection (used for detecting points to track).

- lk_params define the parameters for the Lucas-Kanade optical flow calculation (e.g., window size, maximum pyramid levels).

Lucas-Kanade Optical Flow Calculation

```
def LucasKanadeOpticalFlow(frame, old_gray, mask, p0):
```

- This function calculates optical flow using the Lucas-Kanade method. It tracks feature points between the old and current frame, draws tracks, and updates the points and previous frame.

Dense Optical Flow (Farneback Method)

```
def DenseOpticalFlowByLines(frame, old_gray):
```

- This function computes dense optical flow using the Farneback method. It estimates the flow at each pixel, visualizes it as streamlines (lines representing flow direction), and overlays them on the frame.

handlankmark.py

The code captures video from a webcam, detects hand landmarks, and visualizes them on the captured frames using MediaPipe. It also counts the number of fingers detected based on the positions of key landmarks and displays this information on the screen.

Set Parameters

```
numHands = 2  # Number of hands to be detected
model = 'hand_landmarker.task'  # Model file
minHandDetectionConfidence = 0.5
minHandPresenceConfidence = 0.5
minTrackingConfidence = 0.5
frameWidth = 640
frameHeight = 480
```

- Parameters are set for detecting a maximum of 2 hands, using a model file, and defining confidence thresholds for detection and tracking.

HandLandmarker Setup

```
base_options = python.BaseOptions(model_asset_path=model)
options = vision.HandLandmarkerOptions(
        base_options=base_options,
        num_hands=numHands,
        min_hand_detection_confidence=minHandDetectionConfidence,
        min_hand_presence_confidence=minHandPresenceConfidence,
        min_tracking_confidence=minTrackingConfidence)
detector = vision.HandLandmarker.create_from_options(options)
```

- This section initializes the HandLandmarker with the specified options to load the model and set detection parameters.

Landmark Detection and Visualization

```
hand_landmarks_list = detection_result.hand_landmarks
for hand_landmarks in hand_landmarks_list:
    points = []
    for landmark in hand_landmarks:
        x, y = int(landmark.x * frame.shape[1]), int(landmark.y * frame.shape[0])
        points.append((x, y))
        cv2.circle(frame, (x, y), 5, DOT_COLOR, -1)  # Draw red dots
    # Draw connections between landmarks
    for connection in HAND_CONNECTIONS:
        start_idx, end_idx = connection
        if start_idx < len(points) and end_idx < len(points):
            cv2.line(frame, points[start_idx], points[end_idx], LINE_COLOR, 2)
```

- Here, the hand landmarks are extracted and visualized as red dots, and connections between them are drawn with green lines. The landmarks are scaled based on the frame size.

Finger Count Detection

```
finger_tips = [4, 8, 12, 16, 20]
finger_bases = [2, 6, 10, 14, 18]
thumb_tip_x, thumb_tip_y = points[4]
thumb_joint_x, thumb_joint_y = points[3]
thumb_base_x, thumb_base_y = points[2]
thumb_extended = abs(thumb_tip_x - thumb_joint_x) > abs(thumb_base_x - thumb_joint_x)

finger_count = sum(points[tip][1] < points[base][1] for tip, base in zip(finger_tips[1:],
finger_bases[1:]))

if thumb_extended:
    finger_count += 1
```

- This section detects whether the thumb is extended by checking the horizontal distance between key thumb landmarks. It also counts the number of extended fingers based on the vertical position of finger tips and bases.

handgesture.py

This code demonstrates hand gesture recognition using the MediaPipe library. It captures video frames from the webcam, processes them with a hand gesture recognition model (gesture_recognizer.task), and then displays recognized hand gestures along with the landmarks on the video feed.

Parameters Setup

```
numHands = 2  # Number of hands to be detected
model = 'gesture_recognizer.task'  # Model file for hand gesture detection
minHandDetectionConfidence = 0.5  # Threshold for detecting hands
minHandPresenceConfidence = 0.5  # Threshold for hand presence
minTrackingConfidence = 0.5  # Threshold for hand tracking confidence
frameWidth = 640  # Width of the captured frame
frameHeight = 480  # Height of the captured frame
```

- Defines key parameters like the number of hands to detect, confidence thresholds, and the model file used for hand gesture recognition.

Result Callback Setup

```
def save_result(result: vision.GestureRecognizerResult,
                unused_output_image: mp.Image, timestamp_ms: int):
    recognition_result_list.append(result)
```

- This callback function stores the recognition results (detected hand gestures) in the recognition_result_list.

Gesture Recognizer Initialization

```
base_options = python.BaseOptions(model_asset_path=model)
options = vision.GestureRecognizerOptions(
    base_options=base_options,
    running_mode=vision.RunningMode.LIVE_STREAM,
    num_hands=numHands,
    min_hand_detection_confidence=minHandDetectionConfidence,
    min_hand_presence_confidence=minHandPresenceConfidence,
    min_tracking_confidence=minTrackingConfidence,
    result_callback=save_result)
recognizer = vision.GestureRecognizer.create_from_options(options)
```

- Initializes the Gesture Recognizer with options that include the model path, number of hands to detect, and confidence thresholds.

Gesture Classification

```
if recognition_result_list:
    for hand_index, hand_landmarks in enumerate(
        recognition_result_list[0].hand_landmarks):

        # Bounding box and gesture classification
        x_min = min([landmark.x for landmark in hand_landmarks])
        y_min = min([landmark.y for landmark in hand_landmarks])
        y_max = max([landmark.y for landmark in hand_landmarks])
        ...
        if recognition_result_list[0].gestures:
            gesture = recognition_result_list[0].gestures[hand_index]
            category_name = gesture[0].category_name
            score = round(gesture[0].score, 2)
            result_text = f'{category_name} ({score})'
            cv2.putText(current_frame, result_text, (text_x, text_y), ...)
```

- For each detected hand, the landmarks are used to calculate the bounding box. The recognized gestures (e.g., "thumbs up") are extracted, and the score is displayed on the frame.

Drawing Landmarks

```
mp_drawing.draw_landmarks(
    current_frame,
    hand_landmarks_proto,
    mp_hands.HAND_CONNECTIONS,
    mp_drawing_styles.get_default_hand_landmarks_style(),
    mp_drawing_styles.get_default_hand_connections_style())
```

- This code draws the hand landmarks and the connections between them on the captured frame using MediaPipe's drawing utilities.

obj_detection.py

```
Setting Parameters
maxResults = 5
scoreThreshold = 0.15
frameWidth = 640
frameHeight = 480
model = 'efficientdet.tflite'
```

- maxResults: The maximum number of objects to be detected.
- scoreThreshold: The minimum detection score (confidence) required to classify an object.
- frameWidth and frameHeight: Resolution for the webcam feed.
- model: The file path to the EfficientDet model (a .tflite file).

Setting Visualization Parameters

```
MARGIN = 2
ROW_SIZE = 30
FONT_SIZE = 1
FONT_THICKNESS = 1
TEXT_COLOR = (0, 0, 0)  # black
```

- These parameters are used to format the overlay text and bounding boxes on the video frames.

Creating a Callback Function for Storing Results

```
def save_result(result: vision.ObjectDetectorResult, unused_output_image: mp.Image,
timestamp_ms: int):
    detection_result_list.append(result)
```

- This callback function appends the detection results to the detection_result_list for later use.

Initializing Object Detection Model

```
base_options = python.BaseOptions(model_asset_path=model)
options = vision.ObjectDetectorOptions(base_options=base_options,
                                       running_mode=vision.RunningMode.LIVE_STREAM,
                                       max_results=maxResults, score_threshold=scoreThreshold,
                                       result_callback=save_result)
detector = vision.ObjectDetector.create_from_options(options)
```

- Here, we initialize the object detection model by setting various options, including the model file path and the result callback function.

Handling Detection Results

```
if detection_result_list:
    for detection in detection_result_list[0].detections:
        # Drawing bounding boxes and text
        bbox = detection.bounding_box
        start_point = bbox.origin_x, bbox.origin_y
        end_point = bbox.origin_x + bbox.width, bbox.origin_y + bbox.height
        cv2.rectangle(current_frame, start_point, end_point, (0, 255, 0), 3)  # Green rectangle
        # Display the label and score
        category = detection.categories[0]
        result_text = category.category_name + ' (' + str(round(category.score, 2)) + ')'
        text_location = (MARGIN + bbox.origin_x, MARGIN + ROW_SIZE + bbox.origin_y)
        cv2.putText(current_frame, result_text, text_location, cv2.FONT_HERSHEY_DUPLEX,
                    FONT_SIZE, TEXT_COLOR, FONT_THICKNESS, cv2.LINE_AA)
```

- When results are available in detection_result_list, we iterate over the detected objects, draw bounding boxes, and display labels with detection scores on the frame.

- The cv2.rectangle() function draws a bounding box around detected objects, and cv2.putText() displays the category and detection score on the frame.

Lab 4 Deep Learning of Edge

This script implements real-time image classification using MobileNetV2 with Post-Training Quantization (PTQ) or Quantization-Aware Training (QAT) to improve performance. The model processes frames from a webcam, classifies them, and outputs the top 5 predictions with their confidence scores.

mobile_net.py

Preprocessing Input Images

```
preprocess = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

- Converts images to tensors for PyTorch processing.
- Normalizes pixel values using ImageNet mean and standard deviation for better model performance.

Selecting Quantization Method

```
QUANTIZATION_METHOD = "PTQ"  # Choose between "PTQ" or "QAT"
```

- PTQ (Post-Training Quantization): Converts a pre-trained floating-point model into an efficient, lower-bit version.
- QAT (Quantization-Aware Training): Trains a model with quantization in mind, giving better accuracy at the cost of more training.

Implementing Post-Training Quantization (PTQ)

```
if QUANTIZATION_METHOD == "PTQ":
    print("Performing Post-Training Quantization...")
    net_fp32 = models.mobilenet_v2(pretrained=True)
    net = quantize_dynamic(net_fp32, {torch.nn.Linear}, dtype=torch.qint8)
    print("PTQ Model:", net)
```

- Loads a pre-trained MobileNetV2 model.
- Applies dynamic quantization to reduce the model size and improve inference speed.

Implementing Quantization-Aware Training (QAT)

```
elif QUANTIZATION_METHOD == "QAT":
    print("Preparing for Quantization Aware Training...")
    net_fp32 = models.mobilenet_v2(pretrained=True)

    net_fp32.fuse_model()
    net_fp32.qconfig = get_default_qat_qconfig('qnnpack')
    net_qat = prepare_qat(net_fp32)
    print("QAT Model Prepared for Training")
```

- Fuses layers (e.g., BatchNorm + Conv layers) for better optimization.
- Configures quantization using qnnpack for mobile-friendly execution.

Fine-Tuning QAT Model (Dummy Training)

```
optimizer = torch.optim.SGD(net_qat.parameters(), lr=0.001)
dummy_input = torch.randn(4, 3, 224, 224)
dummy_target = torch.randint(0, 1000, (4,))

print("Fine-tuning QAT Model...")
for epoch in range(5):
    optimizer.zero_grad()
    output = net_qat(dummy_input)
    loss = torch.nn.CrossEntropyLoss()(output, dummy_target)
    loss.backward()
    optimizer.step()
    print(f"Epoch {epoch + 1}: Loss = {loss.item()}")

net = convert(net_qat.eval())
print("QAT Model Converted to Quantized Version")
```

- Uses dummy data to fine-tune the model for better quantized accuracy.
- Converts the trained model into a fully quantized version.

Loading ImageNet Class Labels

```
weights = MobileNet_V2_QuantizedWeights.DEFAULT
classes = weights.meta["categories"]
```

- Loads class labels (e.g., "cat", "dog", "car") from the pre-trained MobileNetV2 quantized model.

Real-Time Image Classification

```
with torch.no_grad():
    while True:
        ret, image = cap.read()
        if not ret:
            raise RuntimeError("Failed to read frame from webcam.")

        image = image[:, :, [2, 1, 0]]  # Convert BGR (OpenCV) to RGB
        input_tensor = preprocess(image)
        input_batch = input_tensor.unsqueeze(0)

        output = net(input_batch)
```

- Captures frames from the webcam.
- Converts BGR to RGB (since OpenCV reads images in BGR format).
- Prepares the image and runs it through the quantized model.

Displaying Predictions

```
top = list(enumerate(output[0].softmax(dim=0)))
top.sort(key=lambda x: x[1], reverse=True)

print("Top Predictions:")
for idx, val in top[:5]:  # Show top 5 predictions
    print(f"{val.item() * 100:.2f}% {classes[idx]}")
print("=" * 72)
```

- Extracts top 5 predictions from the model.
- Prints the category names along with confidence scores.

Lab 5 MQTT

Mqtt_subscriber.py

This script sets up an MQTT-based image capture and transmission system using OpenCV and the paho-mqtt library. It listens for a capture request message via MQTT, captures an image using the webcam, converts it to Base64 format, and then publishes the encoded image back over MQTT.

Image Encoding and MQTT Transmission

```
if ret:
    _, buffer = cv2.imencode('.jpg', frame)
    image_base64 = base64.b64encode(buffer).decode()
    client.publish(TOPIC_PUBLISH, image_base64)
```

- cv2.imencode('.jpg', frame): Converts the image into a compressed JPEG format to reduce size.
- base64.b64encode(buffer): Converts the binary image into Base64 format, making it text-based and suitable for MQTT.
- client.publish(TOPIC_PUBLISH, image_base64): Sends the encoded image to the MQTT broker.

MQTT Callback Mechanism

```
client.on_connect = on_connect
client.on_message = on_message
```

- These functions handle automatic MQTT message reception.
- on_connect: Subscribes to "capture/image" when connected.
- on_message: Captures and transmits an image when a message is received.

Lab 6 AWS IoT

QTT Client Setup

- Uses paho-mqtt to connect to AWS IoT Core securely via TLS/SSL.
- Loads AWS certificates (rootCA.pem, aws-certificate.pem.crt, aws-private.pem.key).

Threaded Data Publishing

- Starts a new thread (justADummyFunction) that:
    - Sends "Hello from INF2009 RaspberryPi Device#1" to device/data every 5 seconds.
    - Mimics sending sensor data for IoT applications.

Persistent Connection

- Runs client.loop_forever() to maintain connection & process MQTT messages.