Lab Instructions - session 7

Active Face Model, PCA

Introduction

In this lab, we will explore facial modeling techniques by capturing and processing images of your face. You will begin by taking multiple front-view photos of your face with varying expressions. These photos will be used to detect and map facial landmarks, providing a foundation for further analysis and modeling.
For this section, you have the below code. You must complete the code and use the functions in **`utils_main.py`** to save images and landmarks in **`OUTPUT_FOLDER`**:

- Note that the **first image** you take must be **neutral** and need to take at least 30 images.

Webcam_face.py

```
# Constants
OUTPUT_FOLDER = "images_landmarks"
PREDICTOR_PATH = 'shape_predictor_68_face_landmarks.dat'
FRAME_SKIP_RATE = 2
# Initialize Dlib's Face Detector and Predictor
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor(PREDICTOR_PATH)

# Webcam Initialization
vc = cv2.VideoCapture(0)
frame_count = 0
frame_number = 0
# Video Capture and Processing Loop
while True:
    rval, frame = vc.read()
    frame_count += 1
    # Process every 3rd frame
    if frame_count % FRAME_SKIP_RATE == 1:
        processed_frame, landmarks = process_frame(detector, predictor, fram)
        # Display the processed frame
        cv2.imshow('Facial Landmark Detection', processed_frame)
        # Handle Keyboard Input
        key = cv2.waitKey(1) & 0xFF
        if key == ord('q'):
            break
        elif key == ord('s'):
          # TODO:  Save the current frame and landmarks and increment the frame_number
        elif key == ord(' '):
            print('Facial Landmarks:', landmarks)
```

- Explain the code.
- How does changing the **`FRAME_SKIP_RATE`** value affect the performance and output of the program?
- What is the role of the **`process_frame()`** function, and how does it contribute to the facial landmark detection workflow?

o In this lab, our focus is entirely on landmarks, and we will not be working directly with images.

1. Registration

In the last part, we first detect the facial landmarks using the Dlib library. To effectively analyze and compare facial features, it is crucial to align facial images so that corresponding landmarks match.
we need a transformation to normalize the images and convert them to the neutral face. So, we need to find the coefficients A^* and b^* according to the following equations to normalize each face to a neutral face.

$$\text{Face} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{68} \end{bmatrix} \quad \text{where } f_i \in \mathbb{R}^2 \text{ and } f_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

$$\text{Neutral\_Face} = \begin{bmatrix} f'_1 \\ f'_2 \\ \vdots \\ f'_{68} \end{bmatrix} \quad \text{where } f'_i \in \mathbb{R}^2 \text{ and } f'_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

$$A^*, b^* = \arg\min_{A,b} \sum_{i=1}^{68} \|Af'_i + b - f_i\|^2$$

This process, known as registration, involves aligning all captured images relative to a **`Neutral_Face`**. The equations necessary for solving this registration are implemented in the **`class FaceRegister()`**.

Complete the Register Function

This function is designed to take a neutral face and another face as inputs, align the second face to the neutral one, and return the transformed face. To accomplish this, you can create an object from the **`FaceRegister()`** class and use it to perform the registration.

***Task1*.py**

```
def Register(Neutral_Face, Face, Method="affine"):
    # TODO: Implement the registration logic
    # TODO: return transformed_face
      pass
```

- Why is it important to apply similarity registration before computing the average face? How does this step affect the final result?

To visualize the registration results, you need to implement this function. It should display the original face, the neutral face, and the transformed face side by side for comparison. Be sure to utilize the **`plot_face()`** function from **`utils.py`** to achieve this.

***Task1*.py**

```
def plot_transformed_face(Face, Neutral_Face, transformed_face):
    # TODO: Plot 1: Original Face
    # TODO: Plot 2: Neutral Face
    # TODO: Plot 3: transformed Face
      pass
```

2. Averaging Faces

To proceed, complete the **`Task2`** by implementing the logic to apply similarity registration and calculate the average face for all the facial expressions you have recorded. Additionally, use the **`plot_transformed_face()`** function to display the **`Face, Neutral_Face, and transformed_face`**.
Remember, no transformations should be applied to the neutral face.

***Task2*.py**

```
Neutral_Face = np.load('images_landmarks/landmarks_0.npy')
# Folder containing landmarks .npy files
landmarks_folder = 'images_landmarks'
transformed_landmarks_folder = 'transformed_landmarks'
Average_landmarks_folder = 'Average_landmarks'
# Initialize a list to hold the flattened transformed_face arrays
Flattened_Faces = [Neutral_Face.ravel()]
```

```
# TODO: compute Flattened_Faces and plot transformed faces Through all landmarks
# TODO: save transformed face(landmarks)
average_face = # TODO: compute average face(landmarks) from transformed faces

filename = os.path.join(Average_landmarks_folder, "average_face.npy")
np.save(filename, average_face)

filename = os.path.join(Average_landmarks_folder, "Flattened_Faces.npy")
np.save(filename, Flattened_Faces)
# TODO: Plot the average_face
```

- What does the average face represent? Use the image below to illustrate your explanation.

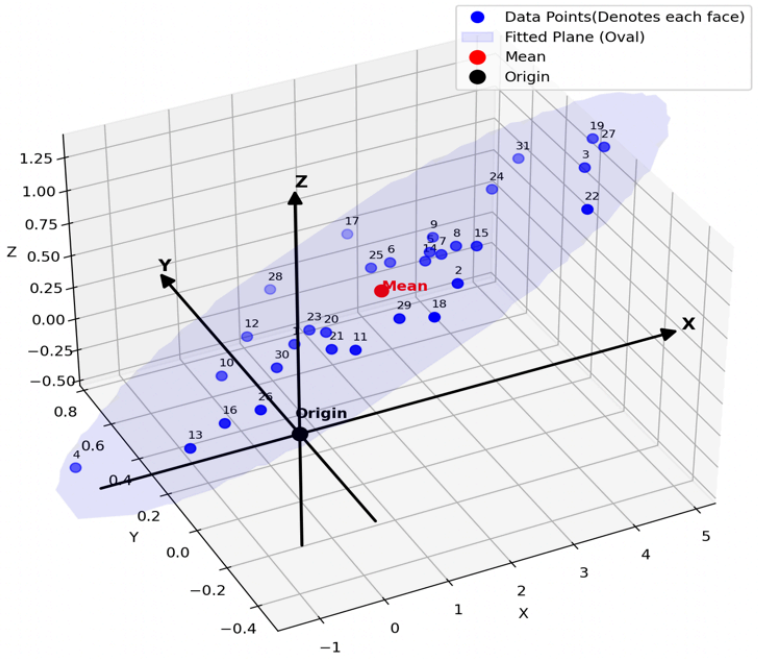

Figure1) Non_normalized Data

3. Face Models and Animating Principal Modes

In this part, you will use all the facial expressions you have created to build a face model. To do this, you need to apply the Principal Component Analysis (PCA) procedure to all the registered faces. The goal is to compute the principal components using Singular Value Decomposition (SVD). You need to:

1. Subtract the average face from the other faces.
2. Apply PCA to compute the principal components, using eigen analysis or SVD.
3. choose the first k principal components that best reconstruct the primary face.

***Task3*.py**

```
# Load the average face and flattened faces
Average_Face = np.load('Average_landmarks/average_face.npy')
Flattened_Faces = np.load('Average_landmarks/Flattened_Faces.npy')  # Shape: (n_samples,
n_features)

Flattened_Faces_centered = # TODO
# TODO: Perform Singular Value Decomposition

num_components = # TODO
U_n = U[:, :num_components]

# TODO: choose the first k principal components and save it
K =

np.save('U_k.npy', U_k)
np.save('S_k.npy', S_k)
```

- what number of *singular vectors (eigenvectors or num_components)* you must take? Why? (you must pay attention to *singular values or eigenvalues.)*
- What roles do eigenvectors and eigenvalues play in Principal Component Analysis (PCA) when applied to facial data?

Task4

This task aims to compute eigenvectors and eigenvalues for Principal Component Analysis (PCA) using Eigen Decomposition (ED) and compare the results with those obtained from the Singular Value Decomposition (SVD) method.

Task5

Animate the first k modes (k=16) of variation. The animation should follow the formula:

$$\text{Model} = \mu + a \cdot U_i$$

where a ranges from -σ to σ and i denotes mode.

Use this formula to generate a dynamic visualization for each mode, clearly demonstrating the impact of the principal components on the face model.

***Task5*.py**

```
# Load the average face and flattened faces
Average_Face = np.load('Average_landmarks/average_face.npy')
U_k = np.load('U_k.npy')
S_k = np.load('S_k.npy')
Mode =

# TODO: loop in ranges for each mode and plot active shape model (use plot_face())
```

- how would you interpret the differences observed between the first few modes versus the later modes?
- What role does the average face serve in the animation regarding the variations from the principal components?