Assignment 2

Batuhan Faik Derinbay derinbay18@itu.edu.tr
150180705

Part 1: Showing the Landmark Points

In the first part, displaying landmarks on given pictures is aimed (Figure 1). To achieve that landmark points need to be found. Luckily landmarks of the animals are given with the homework files in `.npy` format but landmarks for the faces need to be found (predicted).

In order to predict landmarks of the faces the notebook checks whether the predictor model is present in the directory or not. If the model is not available it tries to download the predictor model. Then using the `shape_predictor` function of the `get_frontal_face_detector` class of the `dlib` library, the landmarks of the faces are predicted. After placing circles at the coordinates of the landmarks, all images are concatenated into a big $800x1200x3$ matrix and then displayed, resulting in the Figure 1.

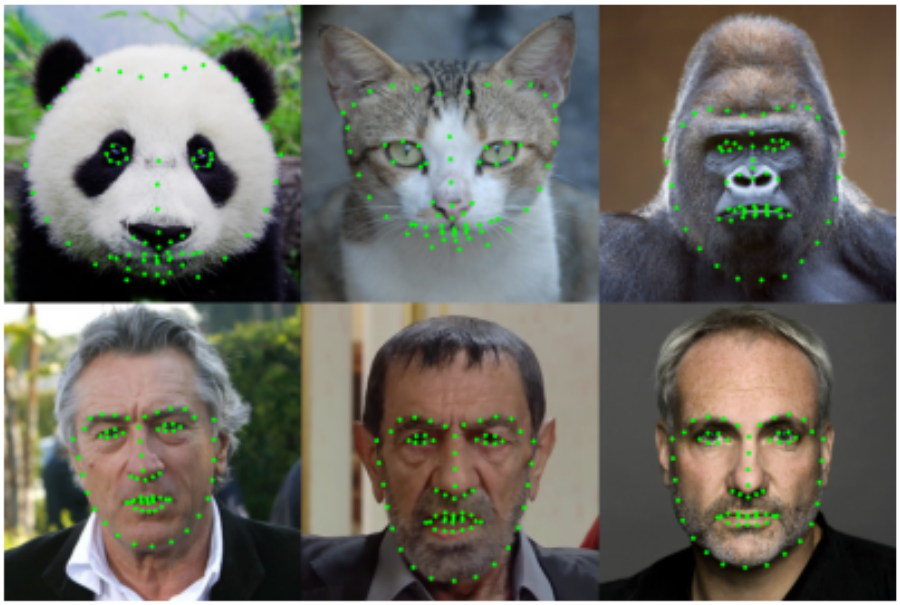

Figure 1: Faces with Landmarks

Part 2: Delaunay Triangulation

In the second part, Delaunay triangulation for the given set of landmark points is aimed. To do that first subdivisions of images are calculated using the `Subdiv2D` function of the Python

`OpenCV` library. Then landmark, corner and edge points are inserted into the subdivision vector. In order to match Delaunay triangulations of two images, triangles of two images should be created relative to one another by matching vertex IDs.

At the first try, in order to match vertex ID's of the target image with the points of the source image, `OpenCV` built-in functions `getVertex` and `findNearest` are used where the `getVertex` returns the ID of the vertex that is supplied by the `findNearest` by giving the source triangle's points. However due to unpredictability (Unexpected return values such as incorrect vertex ID due to equally spaced points etc.) of these built-in functions the implementation of the matching algorithm below fails to work properly for the image of Kim Bodnia. Moreover, since the code functions properly for some cases (at this point one might argue that a properly functioning code shouldn't be failing at any cases) it will be used later just to prove its functionality.

At the second try, rather than obtaining vertex ID's through subdivision vertices, triangles of the source image are matched with the indices of target landmarks. The algorithm can be examined below.

Create Triangles of Target Image Using OpenCV Functions

```
# Doesn't function proparly due to problems with the cv2.getVertex() function
def create_triangles(subdiv_src, subdiv_trg):
    src_triangles = subdiv_src.getTriangleList().astype("uint32")
    trg_triangles = np.empty((len(src_triangles), 6)).astype("uint32")
    for idx in range(len(src_triangles)):
        p1 = subdiv_trg.getVertex(subdiv_src.findNearest(tuple(src_triangles[idx][0:2]))[0])[0]
        # Repeated operations for the second and third corners of the triangle
        points = [p1[0], p1[1], p2[0], p2[1], p3[0], p3[1]]
        trg_triangles[idx:] = np.array(points)
    return trg_triangles
```

Create Triangles of Target Image by Index Matching

```
def create_triangles_using_landmarks(subdiv_src, landmarks_src, landmarks_trg):
    src_triangles = subdiv_src.getTriangleList().astype("uint32")
    trg_triangles = np.empty((len(src_triangles), 6)).astype("uint32")
    for idx in range(len(src_triangles)):
        p1_arr = np.where(np.all(src_triangles[idx][0:2] == landmarks_src, axis=1))
        if p1_arr[0].size == 0:
            p1 = src_triangles[idx][0:2]
        else:
            p1 = landmarks_trg[p1_arr[0][0]]
        # Repeated operations for the second and third corners of the triangle
        points = [p1[0], p1[1], p2[0], p2[1], p3[0], p3[1]]
```

```
12          trg_triangles[idx:] = np.array(points)
13      return trg_triangles
```

By using both of the functions, triangles on the faces are drawn and displayed as in Figure 2. Notice how the triangles of Aydemir Akbas is created using the `create_triangles` function whereas triangles of Kim Bodnia is created using the `create_triangles_using_landmarks` function. For both of the images, source triangles are that is of Deniro's.

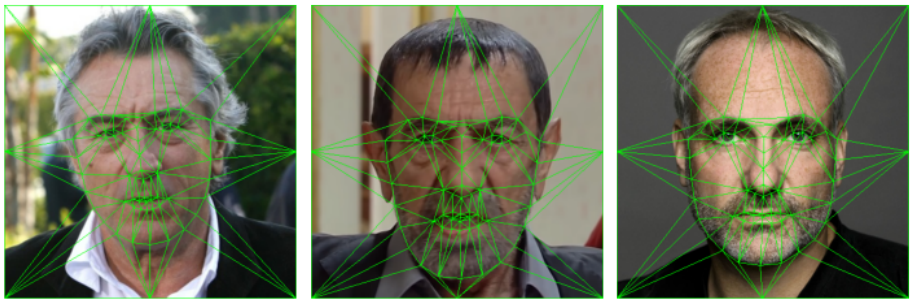

Figure 2: Human Faces with Delaunay Triangles

Part 3: Face Morphing

In this part, by using the triangles obtained from aforementioned functions, source image (image 1) is morphed frame by frame to achieve the target image (image 2). To explain further in detail, let's look at the tagged lines in the code.

(A) `calc_transform()` function is called for each triangle of the images. Return value is inserted at the current index of the `transforms` zero array. This is done because every pair of triangle has to be calculated.

(B) Video frames (images) are generated and appended to the morphs list in this for loop. Where variable `t` is the opacity of the target image frame and ranges between 0 and 1, taking 51 different values, hence 51 frames. By varying the opacity, a smoother transition is achieved.

(C) Inserts points of the triangles on the columns of the matrix and pads the last row with ones hence creating a homogeneous matrix. Eventually achieving the following matrix;

$$\begin{bmatrix} x_1 & y_1 & x_2 & y_2 & x_3 & y_3 \end{bmatrix} \Longrightarrow \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}$$

(D) Obtains the matrix required for coefficient calculation by creating a 6×6 matrix with alternating lines padded with zeros from left and right per point. This matrix will then be used for the transformation. Let's denote the operation as follows, $T' = X \times A$

(E) Psuedo-inverse of the `mtx` (6×6) is multiplicated with the target triangles points (6×1) resulting in the coefficients matrix (6×1). Continuing the same representation, the

operation now becomes, $A = X^{-1} \times T'$ where X is the `mtx`.

$$A = X^{-1} \times T' \implies \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix} \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \end{bmatrix}$$

(F) Affine transformation matrix is obtained from the coefficients matrix and addition of the row $[0, 0, 1]$. Hence the affine transformation matrix becomes $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix}$

(G) These compact lines multiply errors and residuals which are calculated from the coordinates matrix. Then resulting vector (of length number of coordinates) is multiplied with the pixel values present at the given coordinates of the image.
It should also be noted that source code given with the homework files has a very minor mistake. The variables named `top_right` and `bot_left` represent each other and should be changed.

(H) The aforementioned calculations per corner are summed to be rounded to the 0^{th} decimal point and returned by the function. The rounding is used rather than truncation since it is more accurate and pixel values require an unsigned 8-bit integer value. The result is the vectorised form of bilinear interpolation with a shape of `coordinates[1]` $\times 3$.

(I) Homogeneous matrices of triangles are multiplied with their relative opacity values for a smoother transition and summed up (blending).

(J) A mask created to white out pixels inside the triangle in order to be put in the right location on the output image.

(K) Segmentation area (masked with white pixels) is located (indices are returned).

(L) Segmentation indices are concatenated as row vectors and padded with a ones vector at the end.

(M) Intermediary point coordinates are extracted then flattened in column-major order. Preparing it as an argument ready to be passed in to the `calc_transform()` function.

(N) New positions of the mask are mapped in order to get pixel values of each triangle by mutliplying the mask with the transformation matrices.

(O) Bilinear interpolation is applied to images which then returns the pixel values of the previously obtained positions.

(P) Interpolated images are multiplied with their alpha (opacity) value `t` and summed in order to achieve a smoother transition.

Letter Tagged Lines from the Source Code

```
1  transforms[i] = calc_transform(source, target)    # (A)
2  for t in np.arange(0, 1.00001, 0.02):    # (B)
3  homogeneous = np.array([triangle[::2], triangle[1::2], [1, 1, 1]])    ↩
       # (C)
4  mtx = np.array([np.concatenate((source[0], np.zeros(3))),
5                  np.concatenate((np.zeros(3), source[0])),
6                  np.concatenate((source[1], np.zeros(3))),
7                  np.concatenate((np.zeros(3), source[1])),
8                  np.concatenate((source[2], np.zeros(3))),
9                  np.concatenate((np.zeros(3), source[2]))])    # (D)
10 coefs = np.matmul(np.linalg.pinv(mtx), target)    # (E)
11 transform = np.array([coefs[:3], coefs[3:], [0, 0, 1]])    # (F)
12 bot_right = np.multiply(np.multiply(error[0], error[1]).reshape(↩
       coordinates.shape[1], 1), target_img[upper[0], upper[1], :])    # (↩
       G)
13 return np.uint8(np.round(top_left + top_right + bot_left + bot_right))↩
         # (H)
14 homo_inter_tri = (1 - t)*make_homogeneous(triangles1[i]) + t*↩
       make_homogeneous(triangles2[i])    # (I)
15 cv2.fillPoly(polygon_mask, [np.int32(np.round(homo_inter_tri[1::-1, ↩
       :].T))], color=255)    # (J)
16 seg = np.where(polygon_mask == 255)    # (K)
17 mask_points = np.vstack((seg[0], seg[1], np.ones(len(seg[0]))))    # (↩
       L)
18 inter_tri = homo_inter_tri[:2].flatten(order="F")    # (M)
19 mapped_to_img1 = np.matmul(inter_to_img1, mask_points)[:-1]    # (N)
20 inter_image_1[seg[0], seg[1], :] = vectorised_bilinear(mapped_to_img1,↩
       image1, inter_image_1.shape)    # (O)
21 result = (1 - t)*inter_image_1 + t*inter_image_2    # (P)
```

At the end of this part, obtained frames are returned as a list and converted into a video file using the `moviepy` library.