Face Recognition

Alina-Ioana Somcutean
Computer Science
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
Alina.Somcutean@student.utcluj.ro

I. Introduction

Face recognition is an important research problem spanning numerous fields and disciplines. It is a method of identifying or verifying the identity of an individual using their face. Face recognition systems can be used to identify people in photos, video, or in real-time.

Face recognition systems use computer algorithms to pick out specific, distinctive details about a person's face. These details, such as distance between the eyes or shape of the chin, are then converted into a mathematical representation and compared to data on other faces collected in a face recognition database. The data about a particular face is often called a *face template* and is distinct from a photograph because it's designed to only include certain details that can be used to distinguish one face from another.

Face recognition systems vary in their ability to identify people under challenging conditions such as poor lighting, low quality image resolution, and suboptimal angle of view (such as in a photograph taken from above looking down on an unknown person). When it comes to errors, there are two key concepts to understand:

1) *False negative*: when the face recognition system fails to match a person's face to an image that is, in fact, contained in a database. In other words, the system will erroneously return zero results in response to a query;

2) *False positive*: when the face recognition system does match a person's face to an image in a database, but that match is actually incorrect.

All face recognition algorithms consistent of two major parts:

1) *face detection and normalization*
2) *face identification.*

Algorithms that consist of both parts are referred to as fully automatic algorithms and those that consist of only the second part are called partially automatic algorithms. Partially automatic algorithms are given a facial image and the coordinates of the center of the eyes. Fully automatic algorithms are only given facial images.

II. Related work

Much of the work in computer recognition of faces has focused on detecting individual features such as the eyes, nose, mouth, and head outline, and defining a face model by the position, size, and relationships among these features. Beginning with *Bledsoe's* and *Kanade's* early systems, a number of automated or semi-automated face recognition strategies have modeled and classified faces based on normalized distances and ratios among feature points. Recently this general approach has been continued and improved by the recent work of *Yuille et al.*

Such approaches have proven difficult to extend to multiple views, and have often been quite fragile. Research in human strategies of face recognition, moreover, has shown that individual features and their immediate relationships comprise an insufficient representation to account for the performance of adult human face identification [5]. Nonetheless, this approach to face recognition remains the most popular one in the computer vision literature.

III. Method

Step 1: Finding all the faces

The first step of this algorithm is *face detection*. This step is written in the *face_detection.py* file. In order to detect a face, I started by making my image black and white because I don't need color data to find faces.

Then I used the *Haar features* for each image. Haar features are kind of convolution kernels which primarily detect whether a suitable feature is present on an image or not. These Haar features are like windows that are placed upon images to compute a single feature. The feature is essentially a single value obtained by subtracting the sum of the pixels under the white region and that under the black.

The Haar features were loading using the *cascade of classifier*. These consists of stages where each stage consists of a strong classifier. This is beneficial since it eliminates the need to apply all features at once on a window. Rather, it groups the features into separate sub-windows and the classifier at each stage determines whether or not the sub-window is a face. In case it is not, the sub-window is discarded along with the features in that window. If the sub-window moves past the classifier, it continues to the next stage where the second stage of features is applied.

After loading the classifier, I used **detectMultiscale** module from **openCV library**. This function returns a rectangle with coordinates (x, y, w, h) around the detected face. It has 2 important parameters:

- **scaleFactor**: in a group photo, there may be some faces which are near the camera than others. Naturally, such faces would appear more prominent than the ones behind. This factor compensates for that.
- **minNeighbors**: This parameter specifies the number of neighbors a rectangle should have to be called a face.

The next step here, was to loop over all the coordinates returned by *detectMultiscale* function and draw rectangles around them using *OpenCV*.

In the end, the image will be displayed with the face detected.

Step 2: Encoding Faces

This step is written in the *encode_faces.py* file. The first thing to do here was to take the folders list with the persons containing images in order to encode them. The folder name corresponds with the name of the persons of which the pictures are. Also, it was necessarily to initialize 2 lists:

- one for the known encodings
- and one for the known names

These 2 lists will contain the face encodings and corresponding names for each person in the dataset.

Then, for each folder, I saved the name of the person, by extracting it from the path, and then I read all the images using the function *imread* from the *OpenCV* library. I chose to resize each image in order to avoid the cases when the image is too big and it cannot be loaded.

For each iteration of the loop, we're going to detect a face (or possibly multiple faces and assume that it is the same person in multiple locations of the image). So, I used the function *face_locations* from the *face_recognition* library in order to localize the faces and put them in a list of faces.

After that, the bounding boxes of faces are transformed into a list of numbers. This list is known as *encoding* the face into a vector using the *face_encoding* method from the library mentioned above. Then, i went through the list of encodings and I added the current encoding value in the known encoding vector initialized before. Also, I appended in the name's array the current name of the person from the image.

The last thing that I did here was to construct a dictionary with 2 keys: *encodings* and *names*. Then I open a file for writting and I dump the created dictionary to disk for future call.

Step 3: Finding the person's name from the encoding

This last step is actually the easiest step in the whole process. All we have to do is find the person in our database of known people who has the closest measurements to our test image. This step is implemented in the *recognize_faces_image.py* file.

I started this step by opening the file created at the end of the last step. After opening, i take the information from that file and put it into a variable in the current file because I will need this data later during the actual face recognition part.

Then, I read the image for which I want to test the algorithm and to identify the person. As I did in the first step, here I also chose to resize the image if it is necessarily (if the dimension of the image exceeds a setting size) in order to prevent the possibility of not loading the image because of its too high size.

After that, I used again the function *face_locations* from the *face_recognition* library in order to localize the faces from the test image and put them in a list of faces. Then I created again, for the test image too, the vector with the encoding values got after calling the function *face_encoding*.

The next step here was to initialize a list of *names* for each face that is detected. This will be populated a bit later.

Then, I loop over the facial encodings computed from the test image. Afterwards I attempted to match each face from the test image to my known encodings dataset got from the opened file using the function *compare_faces* from the *face_recognition* library. This function returns a list of *True/False* values, one for each image from my dataset. For example, for 50 images in the dataset, this function will return a list with 50 boolean values.

After computing the boolean list, I took another variable in which I hold the name of the person from the test image. Initially, it will get the *Unknown* value.

After having the boolean match list I could compute the number of *True* values associated with each name and select the person's name with the most corresponding positive values. So, in order to do this, I checked to see if I have found a match (True value). If yes, I found the indexes of all matched faces and after that I initialize a dictionary which will hold the person name as the key and the number of times it appears. Then, I started to go through the list of indexed. At each step, I updated the name of the person and I increased with 1 the value for that name in the count dictionary. At the end, I found what name appears most in the count dictionary and I saved as the final name the one with the most occurrences in the count dictionary.

The last thing that I did was to put the text with the name of the person above the rectangle which mark out the face of the person. If the person was in the database, the name of that person will appear. Otherwise, the predefined name (Unknown) will be displayed.

References

[1] https://www.eff.org/pages/face-recognition

[2] A. S. Tolba, A.H. El-Baz, and A.A. El-Harby, ”Face Recognition: A Literature Review”

[3] Matthew A. Turk and Alex P. Pentland, Vision and Modeling Group, The Media Laboratory Massachusetts Institute of Technology, ”Face Recognition Using Eigenfaces”