COMPUTER VISION

Detecting Face Features with Python

Identify faces and extract up to 6 facial features with OpenCV and DLib

Juan Cruz Martinez Jul 4, 2020 · 8 min read ⋆

Today we are going to learn how to work with images to detect faces and to extract facial features such as the eyes, nose, mouth, etc. There's a number of incredible things we can do this information as a pre-processing step like capture faces for tagging people in photos (manually or through machine learning), create effects to "enhance" our images (similar to those in apps like Snapchat), do sentiment analysis on faces and much more.

In the past, we have covered before how to work with OpenCV to detect shapes in images, but today we will take it to a new level by introducing DLib, and abstracting face features from an image.

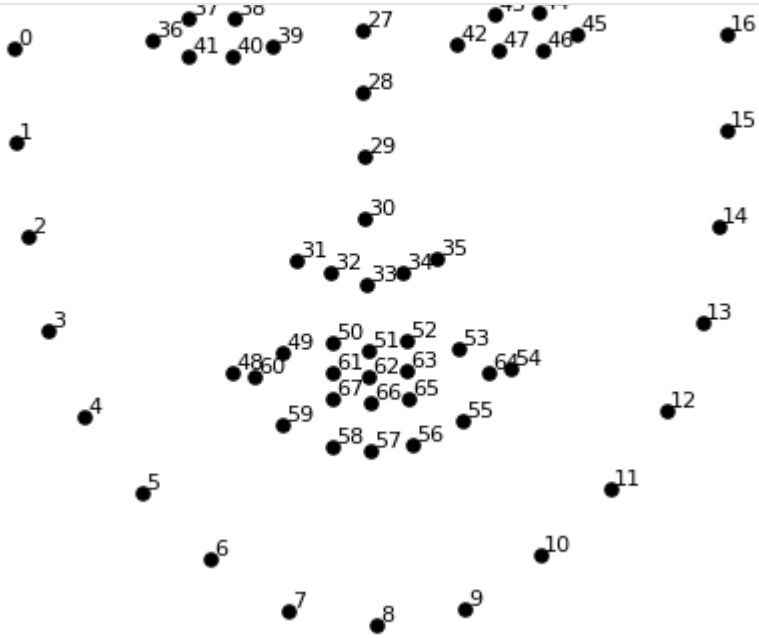
Dlib is an advanced machine learning library that was created to solve complex real-world problems. This library has been created using the C++ programming language and it works with C/C++, Python, and Java.

It worth noting that this tutorial might require some previous understanding of the OpenCV library such as how to deal with images, open the camera, image processing, and some little techniques.

How does it work?

Our face has several features that can be identified, like our eyes, mouth, nose, etc. When we use DLib algorithms to detect these features we actually get a map of points that surround each feature.

This map composed of 67 points (called landmark points) can identify the following features:



Point Map

- Jaw Points = 0–16
- Right Brow Points = 17–21
- Left Brow Points = 22–26
- Nose Points = 27–35
- Right Eye Points = 36–41
- Left Eye Points = 42–47
- Mouth Points = 48–60
- Lips Points = 61–67

Now that we know a bit about how we plan to extract the features, let's start coding.

Installing requirements

you need to start a new Python project and install 3 different libraries:

- opencv-python
- dlib

If you use `pipenv` like I do, you can install all of them with the following command:

```
pipenv install opencv-python, dlib
```

If you are working on Mac, and some versions of Linux you may have some problems installing dlib, if you get compiling errors during the installation make sure you check the CMake library version you are using. In Mac to make sure you have CMake available and with the right version you can run:

```
brew install cmake
```

For other OS, check online for specific support.

Step 1: Loading and presenting an image

We will start small and build on the code until we have a fully working example.

Normally I like to use plots to render the images, but since we have something cool prepared for later in the post, we will do something different, and we will create a window where we are going to show the results of our work.

Let's jump into the code

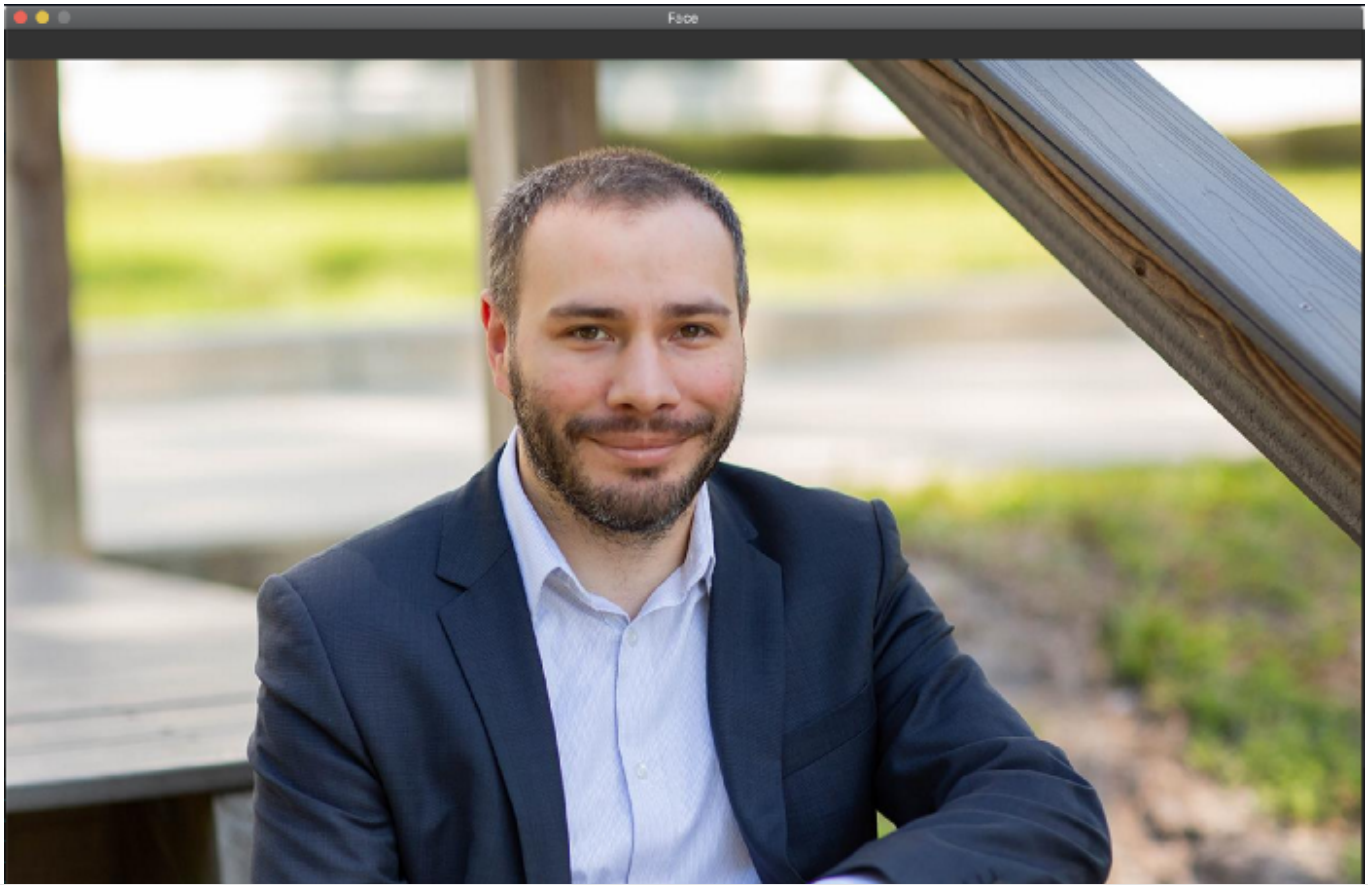
```
import cv2
```

```
# show the image
cv2.imshow(winname="Face", mat=img)

# Wait for a key press to exit
cv2.waitKey(delay=0)

# Close all windows
cv2.destroyAllWindows()
```

Pretty simple, right? We are just loading the image with `imread`, and then telling OpenCV to show the image in a `winname`, this will open the window and give it a title.

After that, we need to pause execution, as the window will be destroyed when the script stops, so we use `cv2.waitKey` to hold the window until a key is pressed, and after that, we destroy the window and exit the script.

If you use the code and added an image named `face.jpg` to the code directory, you should get something like the following:



Original Image

Step 2: Face recognition

So far we haven't done anything with the image other than presenting it into a window, pretty boring, but now we will start coding the good stuff, and we will start by identifying where in the image there is a face.

For this, we will use Dlib function called `get_frontal_face_detector()`, pretty intuitive. There is a caveat though, this function will only work with grayscale images, so we will have to do that first with OpenCV.

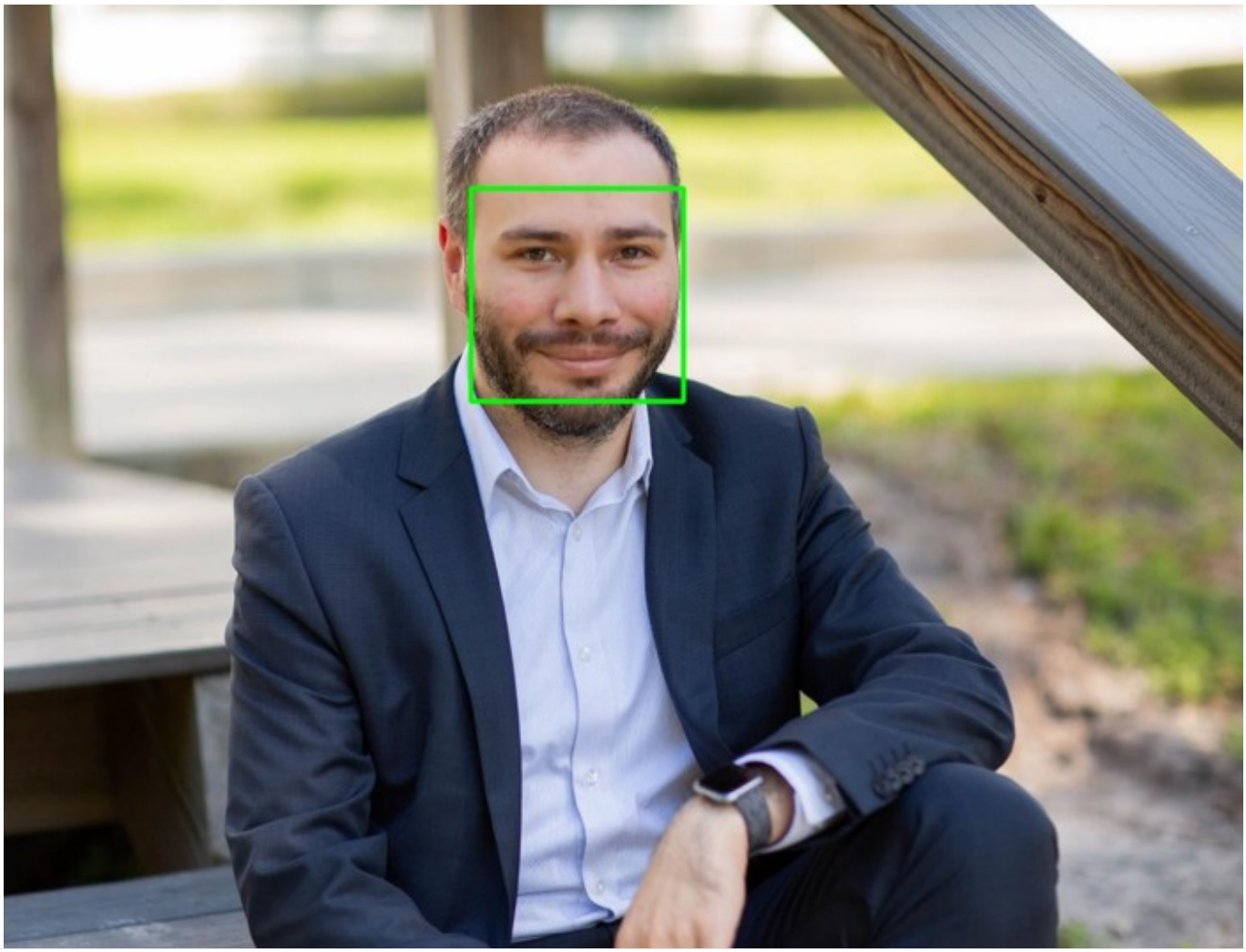
The `get_frontal_face_detector()` will return a `detector` that is a function we can use to retrieve the faces information. Each face is an object that contains the points where the image can be found.

But let's better see it on the code:

```
import cv2
import dlib

# Load the detector
detector = dlib.get_frontal_face_detector()

# read the image
img = cv2.imread("face.jpg")

# Convert image into grayscale
gray = cv2.cvtColor(src=img, code=cv2.COLOR_BGR2GRAY)

# Use detector to find landmarks
faces = detector(gray)
```

```
    x2 = face.right() # right point
    y2 = face.bottom() # bottom point
    # Draw a rectangle
    cv2.rectangle(img=img, pt1=(x1, y1), pt2=(x2, y2), color=(0, 255,
0), thickness=4)

# show the image
cv2.imshow(winname="Face", mat=img)

# Wait for a key press to exit
cv2.waitKey(delay=0)

# Close all windows
cv2.destroyAllWindows()
```

The code above will retrieve all the faces from the image and render a rectangle over each face, resulting in an image like the following:



the features (landmarks). Let's work on that next.

Step 3: Identifying face features

Do you love magic? So far DLib has been pretty magical in the way it works, with just a few lines of code we could achieve a lot, and now we have a whole new problem, would it continue to be as easy?

The short answer is YES! Turns out DLib offers a function called `shape_predictor()` that will do all the magic for us but with a caveat, it needs a pre-trained model to work.

There are several models out there that work with `shape_predictor`, the one I'm using can be downloaded here, but feel free to try others.

Let's see how the new code looks like now

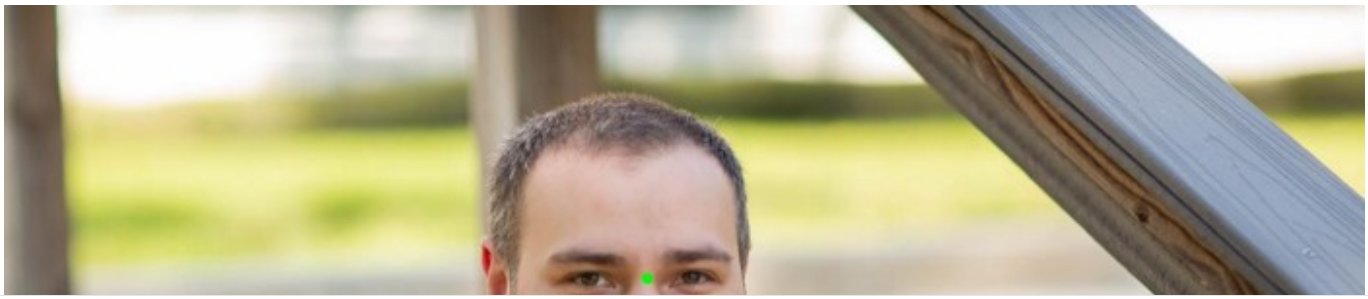
```
import cv2
import dlib

# Load the detector
detector = dlib.get_frontal_face_detector()

# Load the predictor
predictor =
dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

# read the image
img = cv2.imread("face.jpg")

# Convert image into grayscale
gray = cv2.cvtColor(src=img, code=cv2.COLOR_BGR2GRAY)

# Use detector to find landmarks
faces = detector(gray)

for face in faces:
    x1 = face.left() # left point
```

```
    # Look for the landmarks
    landmarks = predictor(image=gray, box=face)
    x = landmarks.part(27).x
    y = landmarks.part(27).y

    # Draw a circle
    cv2.circle(img=img, center=(x, y), radius=5, color=(0, 255, 0),
thickness=-1)

# show the image
cv2.imshow(winname="Face", mat=img)

# Wait for a key press to exit
cv2.waitKey(delay=0)

# Close all windows
cv2.destroyAllWindows()
```

Like before, we are always building on the same code, now using our predictor function for each face to find the landmarks. Now I'm still doing something strange, like what's the number 27 doing there?

```
landmarks = predictor(image=gray, box=face)
x = landmarks.part(27).x
y = landmarks.part(27).y
```

Our predictor function will return an object that contains all the 68 points that conform a face according to the diagram we saw before, and if you pay attention to it, the point 27 is exactly between the eyes, so if all worked out correctly you should see a green dot between the eyes in the face like in here:



We are getting really close, let's now render all the points instead of just the one:

```
import cv2
import numpy as np
import dlib

# Load the detector
detector = dlib.get_frontal_face_detector()

# Load the predictor
predictor =
dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

# read the image
img = cv2.imread("face.jpg")

# Convert image into grayscale
gray = cv2.cvtColor(src=img, code=cv2.COLOR_BGR2GRAY)

# Use detector to find landmarks
faces = detector(gray)
for face in faces:
    x1 = face.left() # left point
    y1 = face.top() # top point
    x2 = face.right() # right point
    y2 = face.bottom() # bottom point
```

```
        # Loop through all the points
        for n in range(0, 68):
            x = landmarks.part(n).x
            y = landmarks.part(n).y

            # Draw a circle
            cv2.circle(img=img, center=(x, y), radius=3, color=(0, 255,
    0), thickness=-1)

    # show the image
    cv2.imshow(winname="Face", mat=img)

    # Delay between every fram
    cv2.waitKey(delay=0)

    # Close all windows
    cv2.destroyAllWindows()
```

Tada! Magic:

But what if you are not interested in all the points? well… you can actually adjust your `range` intervals to get any feature specified in the glossary above, as I did here:

Amazing, but can we do something even cooler?

Step 4: Real-time detection

Yes, you read it right! And yes it's probably what you are thinking! The next step is to hook up our webcam and do real-time landmark recognition from your video stream.

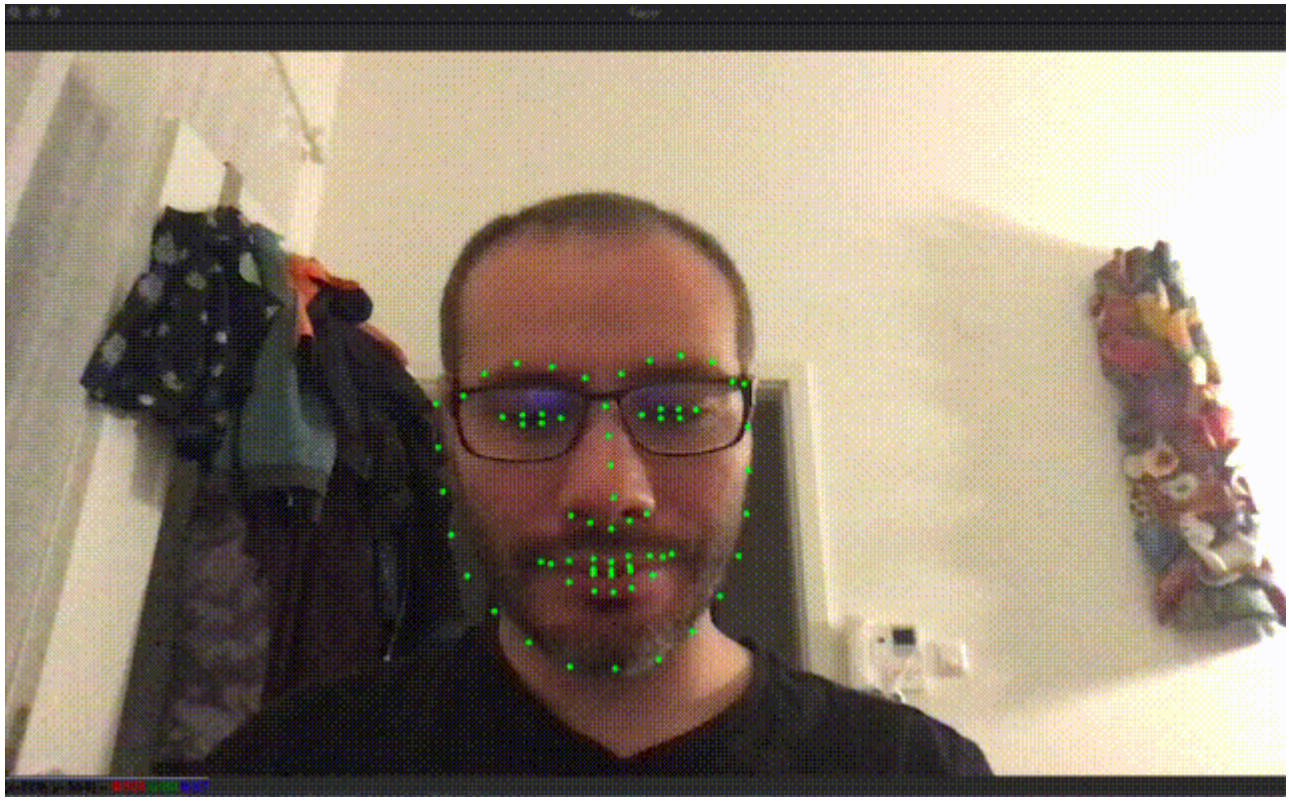
your own camera but for video file make sure to change the number 0 to video path.

If you want to end the window press ESC key on your keyboard:

```
import cv2
import dlib

# Load the detector
detector = dlib.get_frontal_face_detector()

# Load the predictor
predictor =
dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

# read the image
cap = cv2.VideoCapture(0)

while True:
    _, frame = cap.read()
    # Convert image into grayscale
    gray = cv2.cvtColor(src=frame, code=cv2.COLOR_BGR2GRAY)

    # Use detector to find landmarks
    faces = detector(gray)

    for face in faces:
        x1 = face.left()  # left point
        y1 = face.top()  # top point
        x2 = face.right()  # right point
        y2 = face.bottom()  # bottom point

        # Create landmark object
        landmarks = predictor(image=gray, box=face)

        # Loop through all the points
        for n in range(0, 68):
            x = landmarks.part(n).x
            y = landmarks.part(n).y

            # Draw a circle
            cv2.circle(img=frame, center=(x, y), radius=3, color=(0,
255, 0), thickness=-1)

    # show the image
    cv2.imshow(winname="Face", mat=frame)
```

```
# When everything done, release the video capture and video write
objects
cap.release()

# Close all windows
cv2.destroyAllWindows()
```

And the final result is here:



GIF created from the original video, I had to cut frames to make the GIF a decent size.

Even in cases with low light conditions the results were pretty accurate, though there are some errors in the image above, with better lighting works perfectly.

Conclusion

from both of them.

Thanks so much for reading!