CHAPTER 14

Image Analysis and Computer Vision with OpenCV

In the previous chapters, the analysis of data was centered entirely on numerical and tabulated data, while in the previous one we saw how to process and analyze data in textual form. This book rightfully closes by introducing the last aspect of data analysis: *image analysis*.

During the chapter, topics such as computer vision and face recognition will be introduced. You will see how the techniques of deep learning are at the base of this king of analysis. Furthermore, another library will be introduced, called `openCV`, which has always been the reference point for image analysis.

Image Analysis and Computer Vision

Throughout the book, you have seen how the purpose of the analysis is to extract new information, to draw new concepts and characteristics from a system under investigation. You did it with numerical and textual data, but the same can be done with images.

This branch of analysis is called *image analysis* and is based on some calculation techniques applied to them (image filters), which you will see in the next sections.

In recent years, especially because of the development of deep learning, image analysis has experienced huge development in solving problems that were previously impossible, giving rise to a new discipline called *computer vision*.

F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-3913-1_14

In Chapter 9, you learned about artificial intelligence, which is the branch of calculation that deals with solving problems of pure "human relevance". Computer vision is part of this, since its purpose is to reproduce the way the human brain perceives images.

In fact, seeing is not just the acquisition of a two-dimensional image, but above all it is the interpretation of the content of that area. The captured image is decomposed and elaborated into levels of representation that are gradually more abstract (contours, figures, objects, and words) and therefore recognizable by the human mind.

In the same way, computer vision intends to process a two-dimensional image and extract the same levels of representation from it. This is done through various operations that can be classified as follows:

- *Detection*: Detect shapes, objects, or other subjects of investigation in an image (for example finding cars)
- *Recognition*: The identified subjects are then led back to generic classes (for example, subdividing cars by brands and types)
- *Identification*: An instance of the previous class is identified (for example, find my car)

OpenCV and Python

OpenCV (Open Source Computer Vision) is a library written in C ++ that is specialized for computer vision and image analysis (`https://opencv.org/`). This powerful library, designed by Gary Bradsky, was born as an Intel project and in 2000 the first version was released. Then with the passage of time, it was released under an open source license, and since then has gradually becoming more widespread, reaching the version 3.3 (2017). At this time, OpenCV supports many algorithms related to computer vision and machine learning and is expanding day by day.

Its usefulness and spread is due precisely to its antagonist: MATLAB. In fact, those who need to work with image analysis can follow only two ways: purchase MATLAB packages or compile and install the open source version of OpenCV. Well, it is easy to see why many have opted for the second choice.

OpenCV and Deep Learning

There is a close relationship between computer vision and deep learning. Since 2017 was a significant year for the development of deep learning (read my article about it at http://www.meccanismocomplesso.org/en/2017-year-of-deep-learning-frameworks/), the release of the new version of OpenCV 3.3 has seen the enhancement of the library with many new features of deep learning and neural networks in general. In fact, the library has a module called dnn (deep neural networks) dedicated to this aspect. This module has been specifically developed for use with many deep learning frameworks, including Caffe2, TensorFlow, and PyTorch (for information on these frameworks see Chapter 9).

Installing OpenCV

Installing a OpenCV package on many operating systems (Windows, iOS, and Android) is done through the official website (https://opencv.org/releases.html).

If you use Anaconda as a distribution medium, I recommend using this approach. The installation is very simple and clean.

```
conda install opencv
```

Unfortunately for Linux systems there is no official PyPI package (with pip to be clear) to be installed. Manual installation is required and may vary depending on the distribution and version used. Many procedures are present on the Internet, some more or less valid. For those with Ubuntu 16, I recommend this installation procedure (see https://github.com/BVLC/caffe/wiki/OpenCV-3.3-Installation-Guide-on-Ubuntu-16.04).

First Approaches to Image Processing and Analysis

In this section, you will begin to familiarize yourself with the OpenCV library. First you will start to see how to upload and view images. Then you will pass some simple operations to them, add and subtract two images, and see an example of image blending. All these operations will be very useful as they will serve as a basis for any other image analysis operation.

Before Starting

Once the OpenCV library is installed, you can open an IPython session on the Jupyter QtConsole or Jupyter Notebook.

Then before you start programming, you need to import the openCV library.

```
import numpy as np
import cv2
```

Load and Display an Image

First, mainly because OpenCV works on pictures, it is important to know how to load them in a program in Python, manipulate them again, and finally view them to see the results.

The first thing you need to do is read the file containing the image using the OpenCV library. You can do this using the `imread()` method. This method reads the file in a compressed format such as JPG and translates it into a data structure that's made of a numerical matrix corresponding to color gradations and position.

Note You can find the images and files in the source code of this book.

```
img = cv2.imread('italy2018.jpg')
```

If you are interested in more details, you can see the content of an image directly. You will notice an array of arrays, each corresponding to a specific position of the image, and each characterized by numbers between 0 and 255.

In fact, if you see the content of the first element of the image, you will get the following.

```
img[0]
array([[38, 43, 11],
       [37, 42, 10],
       [36, 41,  9],
       ...,
       [24, 37, 15],
       [22, 36, 12],
       [23, 36, 12]], dtype=uint8)
```

Continuing with the code, you will now use the `imshow()` method to create a window with the image loaded in the variable `img`. This method takes two parameters—the window name and the image variable. Once you have created the window, you can use the `waitKey()` method.

```
cv2.imshow('Image', img)
cv2.waitKey(0)
```

Executing this command, a new window opens and shows the image, as shown in Figure 14-1.

Figure 14-1. *The photo of the Italian national football team during training*

The `waitKey()` method starts to display the window and allows you to control the waiting time of the program before continuing with the next command. This example used 0 as an argument, which means that the wait will be infinite as long as you press any key on the keyboard.

If you wanted to keep open the window only for a specific period, you need to write the number of milliseconds as a parameter. Try to replace the value in the program, for example 2000 (two seconds), and run the program.

Note This behavior can vary greatly from system to system. Sometimes the IPython kernel could give problems. Then use `waitKey(0)`.

```
cv2.imshow('Image', img)
cv2.waitKey(2000)
```

The window with the image (as shown in Figure 14-1) should appear and then disappear after two seconds.

However, for examples that are more complex, it is useful to have direct control over the closure of a window, without the use of waiting times. The `destroyWindow()` method allows you to close the desired window (could be several open) by specifying as an argument the name of the window, which in your case is `Image`.

```
cv2.imshow('Image', img)
cv2.waitKey(2000)
cv2.destroyWindow('Image')
```

If there are multiple windows open and you want to close them all at once, you can use a single command, the call to `destroyAllWindows()` method.

Working with Images

Now that you've seen how to view existing images in your file system, you can proceed to the next step: processing the image by performing an operation on it and saving the result to a new file.

Continuing with the previous example, you will use the same code. This time, however, you will perform a simple image manipulation, for example, by decomposing the three RGB channels. Then you will exchange the channels to form a new image. This new image will have all altered colors.

After loading the image, decompose it into the three RGB channels. You can do this easily by using the `split()` method.

```
b,r,g = cv2.split(img)
```

Now reassemble the three channels, but change the order, for example by exchanging the red channel with the green channel. You can easily recombine the channels using the `merge()` method.

```
img2 = cv2.merge((b,g,r))
```

The new image is contained in the `img2` variable. Display it along with the original in a new window.

```
cv2.imshow('Image2', img2)
cv2.waitKey(0)
```

By running the program, a new window appears with altered colors (as shown in Figure 14-2).

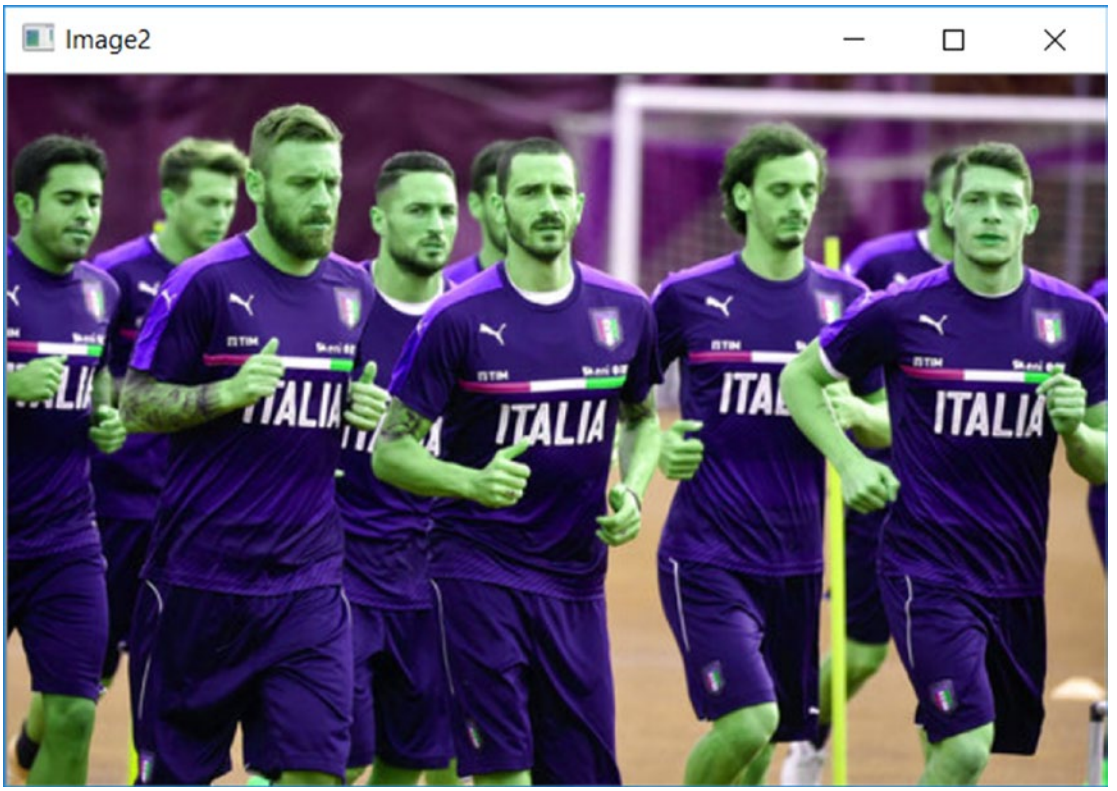

Figure 14-2. *The processed image has altered colors*

Save the New Image

Finally you have to save your new image by saving the file system.

At the end of the program, add an `imwrite()` method with the name of the new file that you want to save, which can also be of another format, such as PNG.

```
cv2.imwrite('italy2018altered.png', img2)
```

Execute this command and you will notice a new `italy2018altered.png` file in the workspace.

Elementary Operations on Images

The most basic operation is the addition of two images. With the `openCV` library, this operation is very simple and you can do it using the `cv2.add()` function. The result obtained will be the combination of the two images.

But do not forget that the two images must have the same dimensions to be added together. In this case, the images are both 512x331 pixels.

The first thing you need to do is load a second image with the same dimensions, in our case `soccer.jpg` (you can find it in the source code).

```
img2 = cv2.imread('soccer.jpg')
cv2.imshow('Image2', img2)
cv2.waitKey(0)
```

By executing the code, you will get the image shown in Figure 14-3.

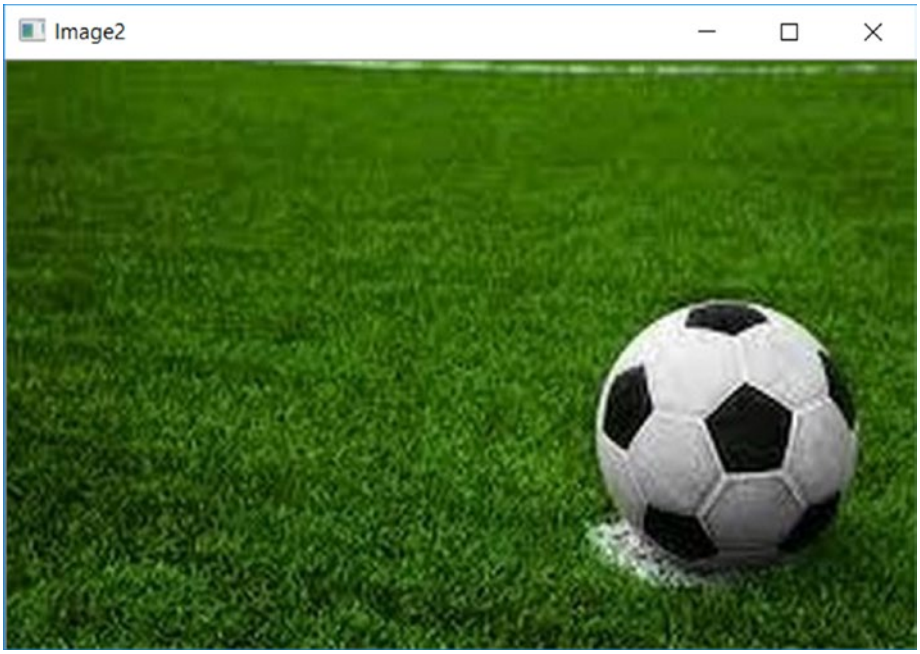

Figure 14-3. *A new image that's the same size (512x331 pixels)*

Now you just have to add the two images using the `add()` function.

```
img = cv2.add(img,img2)
cv2.imshow('Sum',img)
```

By executing this code, you will receive a combination of the two images (as shown in Figure 14-4). Unfortunately, the effect is not very appealing.

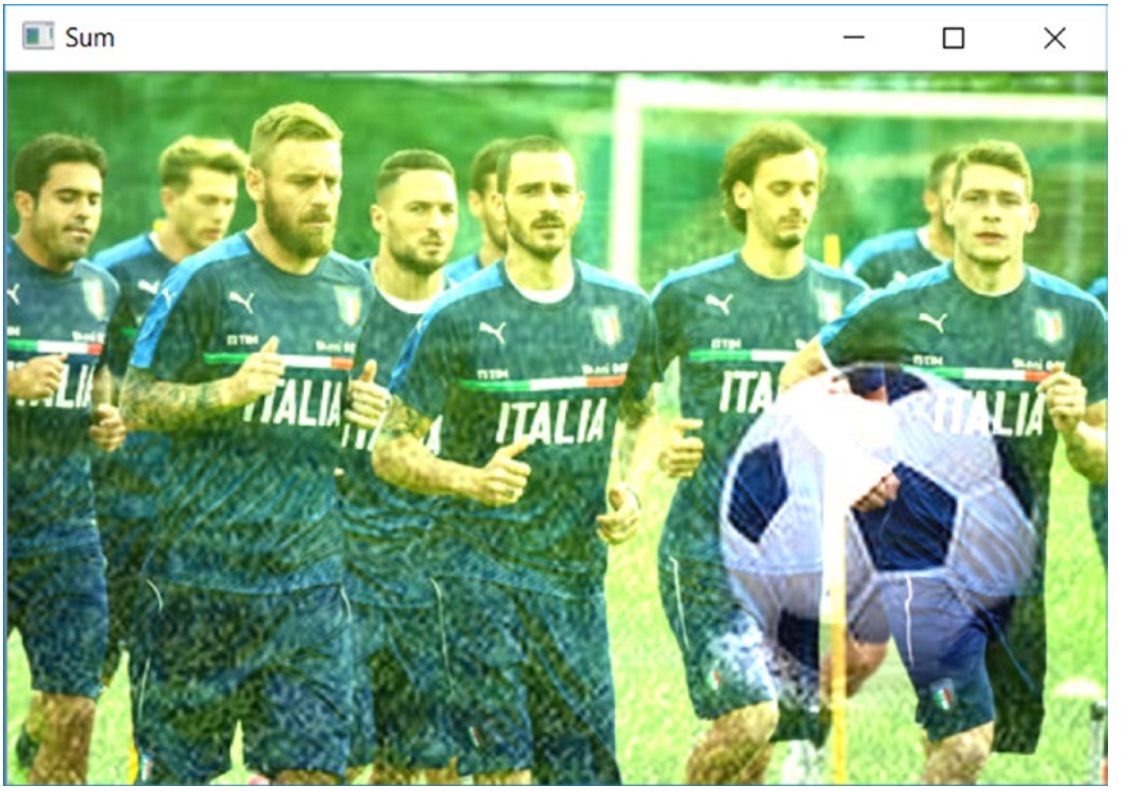

Figure 14-4. *A new image obtained by adding the two images*

The result is not what we expected. The prevalence of white is in fact the result of the simple arithmetic sum of the three RGB values, which is calculated for each individual pixel.

In fact, you know that each of the three RGB components takes values from 0 to 255. Therefore, if the sum of the values of a given pixel is greater than 255 (which is quite likely) the value will still be 255. Therefore, the simple task of adding the images does not lead to an image that's a merger of the two, but instead shifts gradually more and more toward white.

Later you will see how the concept of adding two images to create a new image that is half of the two (it is not the arithmetic sum).

You can do the same thing by subtracting two images. This operation can be performed with the `cv2.subtract()` function. This time we would expect an image that will tend more and more toward the black. Replace the `cv2.add()` function with the following.

```
img3 = cv2.subtract(img, img2)
cv2.imshow('Sub1',img3)
cv2.waitKey(0)
```

By running the program you will find a picture tending to the darkness (even if you do not see much), as shown in Figure 14-5.

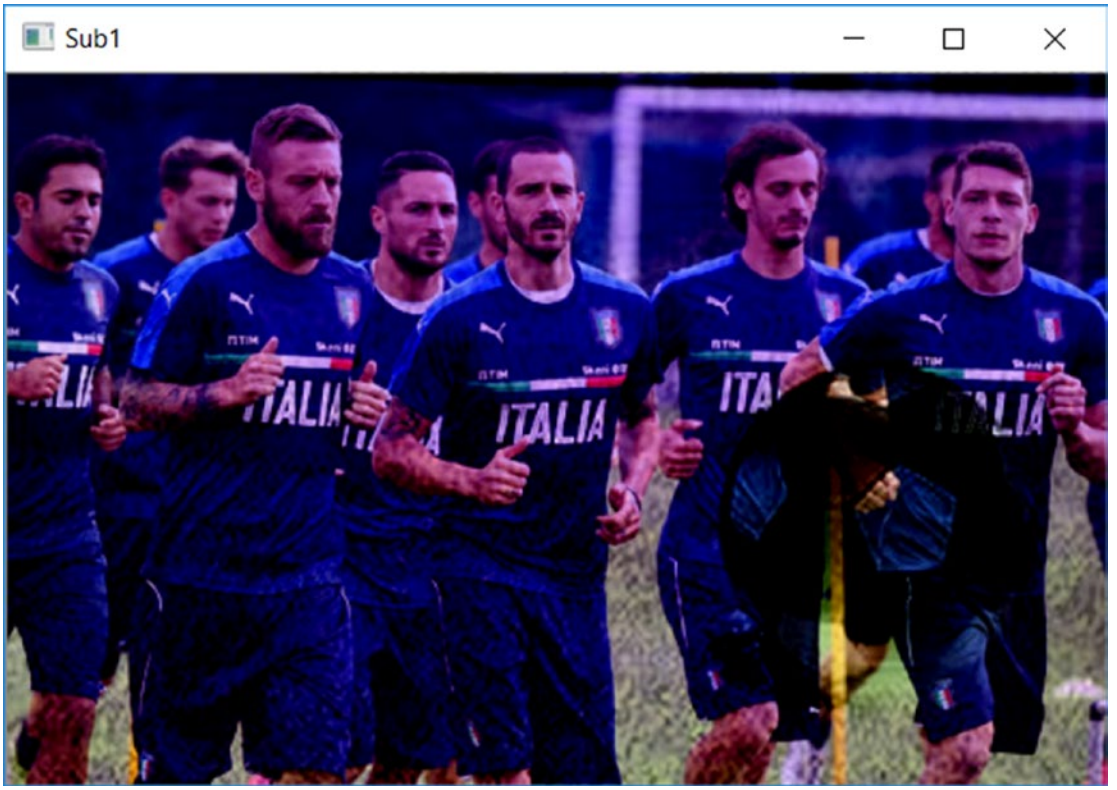

Figure 14-5. *A new image obtained by subtracting one image from another*

Please note that this effect is even worse if you do the reverse.

```
img3 = cv2.subtract(img2, img)
cv2.imshow('Sub1',img3)
cv2.waitKey(0)
```

You get a blackish image, as shown in Figure 14-6.

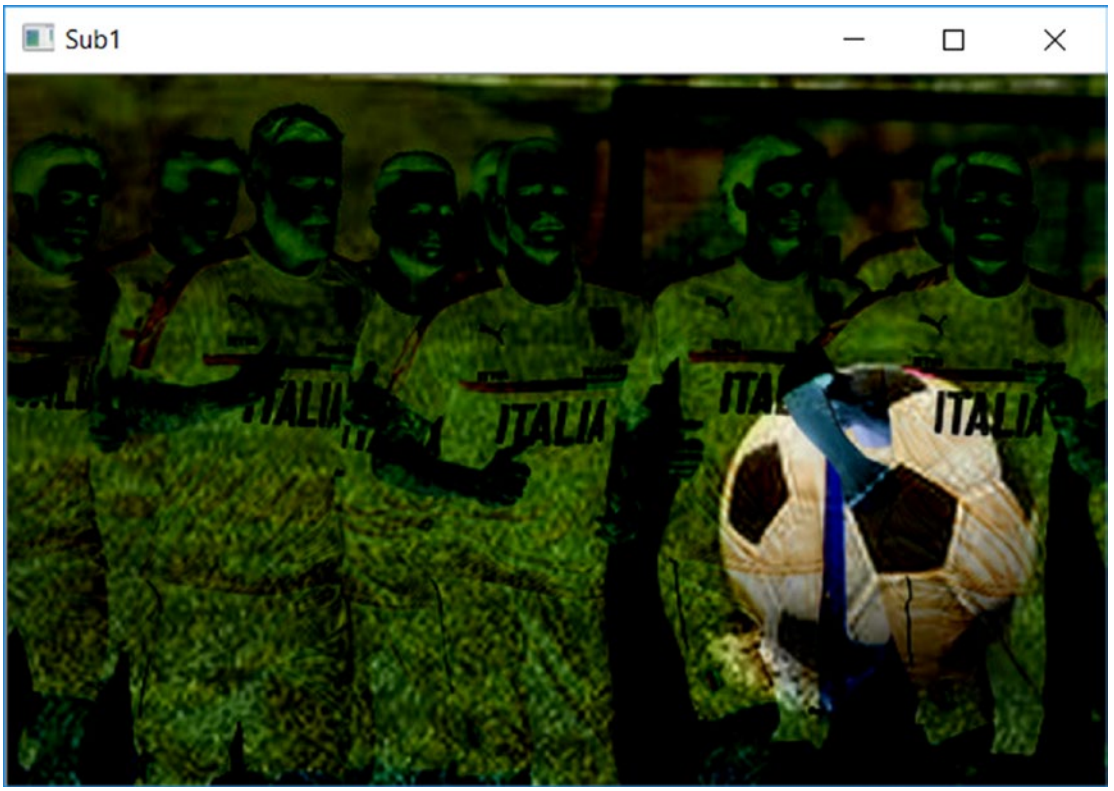

Figure 14-6. *A new image obtained by subtracting one image from another*

However, this is useful to know that the order of the operators is important for the result.

More concretely, you have already seen that an image object created with the OpenCV library is nothing more than an array of arrays that respond perfectly to the canons of NumPy. Thus, you can use the operations between matrices provided by NumPy, such as the addition of matrices. But be careful, the result will certainly not be the same.

```
img = img1 + img2
```

In fact, the `cv2.add()` and `cv2.subtract()` functions maintain the values between 0 and 255, regardless of the value of the operators. If the sum exceeds 255 the result is interpreted differently, thus creating a very strange color effect (maybe as a module of 255). The same thing happens when the removal produces a negative value; the result would be 0. Arithmetic operations do not have this feature.

However, you can try it directly.

```
img3 = img + img2
cv2.imshow('numpy',img3)
cv2.waitKey(0)
```

Executing it, you will get an image with a very strong color contrast (they are the points over 255), as shown in Figure 14-7.

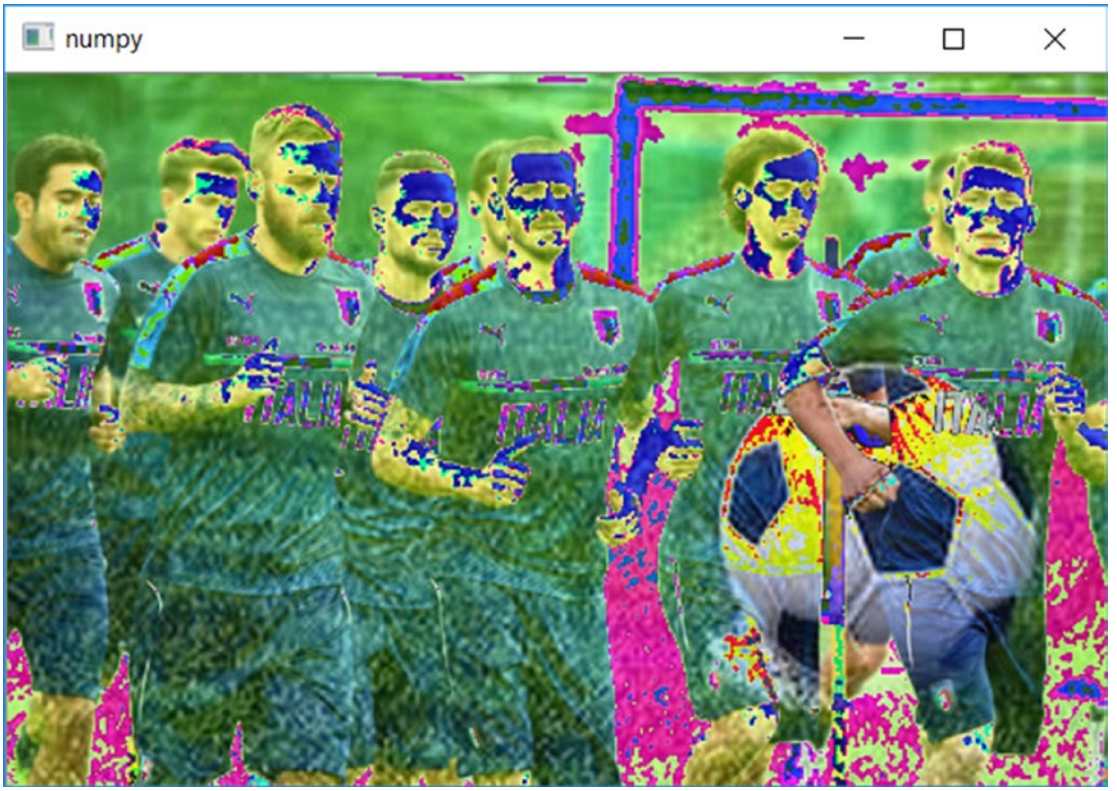

Figure 14-7. *An image obtained by adding two images as two NumPy matrices*

Image Blending

In the previous example, you saw that the addition or subtraction of two images does not produce an intermediate image between the two, but instead changes the coloration toward whites or blacks.

The correct operation is called *blending*. That is, you can consider the operation of superimposing the two images, one above the other, making the one placed above gradually more and more transparent. By adjusting the transparency gradually, you get a mixture of the two images, creating a new one that is the intermediate.

The blending operation does not correspond to a simple addition, but the formula corresponds to the following equation.

$$img = \alpha \cdot img1 + (1 - \alpha) \cdot img2 \qquad \text{with} \quad 0 \geq \alpha \geq 1$$

As you can see from the previous equation, the two images have two numerical coefficients that take values between 0 and 1. With the growth of the α parameter you will have a smooth transition from the first image to the second.

The OpenCV library provides the operation of blending with the `cv2.addWeighted()` function.

Therefore, if you want to create an intermediate image between two source images, you can use the following code.

```
img3 = cv2.addWeighted(img, 0.3, img2, 0.7, 0)
cv2.imshow('numpy',img3)
cv2.waitKey(0)
```

The result will be an image like the one shown in Figure 14-8.

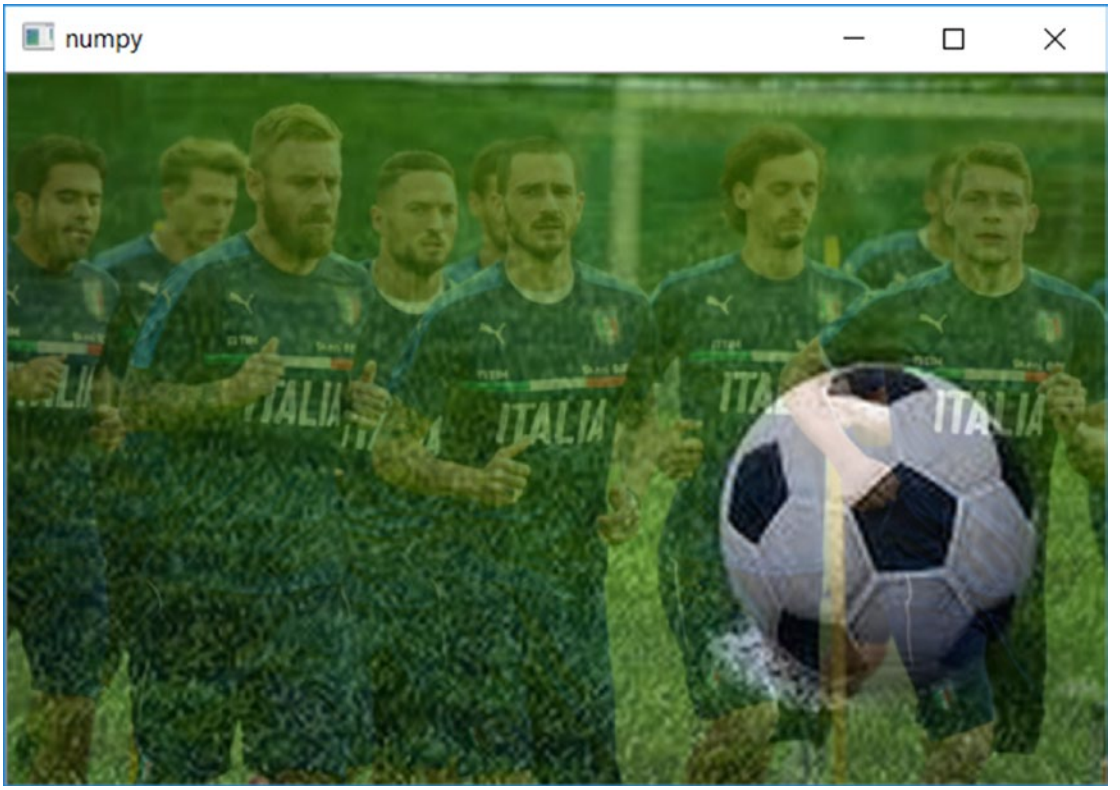

Figure 14-8. *An image obtained with image blending*

Image Analysis

The purpose of the examples in the previous section was to understand that images are nothing but NumPy arrays. As such, these numerical matrices can be processed. Therefore, you can implement many mathematical functions that will process the numbers within these matrices to get new images. These new images, obtained from operations, will serve to provide new information.

This is the concept underlying *image analysis*. The mathematical operations carried out by a starting image (matrix) to a resultant image (matrix) are called image filters (see Figure 14-9). To help you understand this process, you will certainly have to deal with photo editing applications (like Photoshop). In any case, you have certainly seen

that filters that can be applied to photos. These filters are nothing more than algorithms (sequences of mathematical operations) that modify the numerical values in the matrix of the starting image.

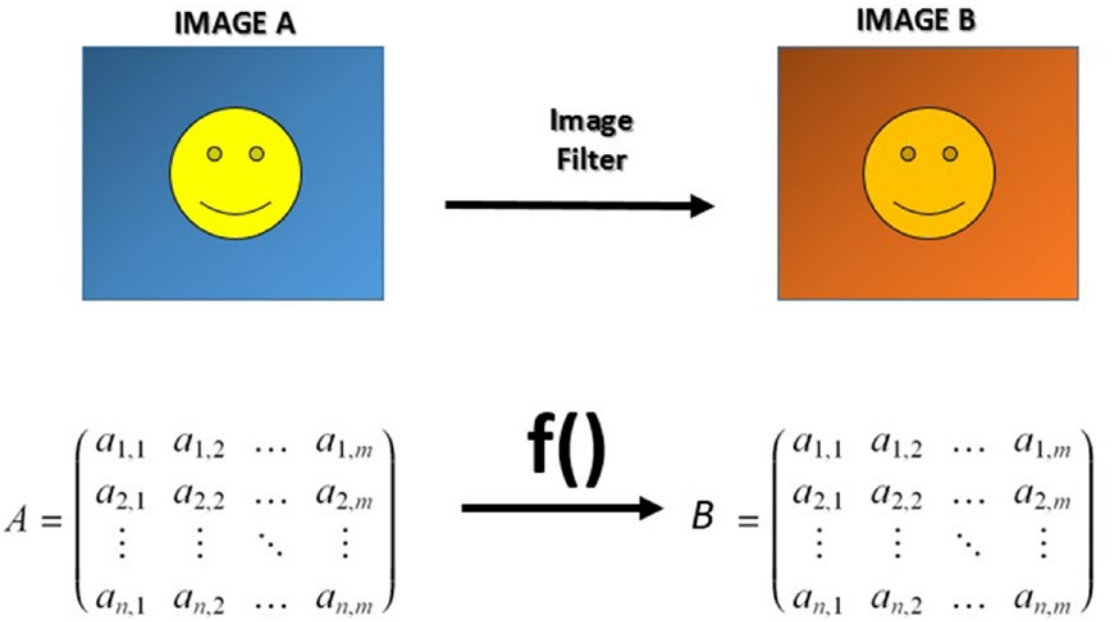

Figure 14-9. *A representation of the image filters that are the basis of image analysis*

Edge Detection and Image Gradient Analysis

In the previous sections, you saw how to perform some basic operations that are useful for image analysis. In this section, you start with a real case of image analysis, called *edge detection*.

Edge Detection

While analyzing an image, and especially during computer vision, one of the fundamental operations is to understand the content of the image, such as objects and people. It is necessary first to understand what possible forms are represented in the image. Nevertheless, to understand the geometries represented, it is necessary to recognize the outlines that delimit an object from the background or from other objects. This is precisely the task of the edge detection.

In edge detection, a great many algorithms and techniques have been developed and they exploit different principles in order to determine the contours of objects correctly. Many of these techniques are based on the principle of color gradients, and exploit the image gradient analysis process.

The Image Gradient Theory

Among the various operations that can be applied to images, there are the *convolutions* of an image in which certain filters are applied to edit the image in order to obtain information or some other utility. You have already seen that an image is represented as a large numerical matrix in which the colors of each pixel are represented by a number from 0 to 255 in the matrix. The convolutions process all these numerical values by applying a mathematical operation (*image filter*) to produce new values in a new matrix of the same size.

One of these operations is the *derivative*. In simple words, the derivative is a mathematical operation that allows you to get the numerical values indicating the speed at which a value changes (in space, time, etc.).

How could the derivative be important in the case of the images? It has to do with color variation, called a *gradient*.

Being able to calculate the gradient of a color is an excellent tool to calculate the edges of an image. In fact, your eye can distinguish the outlines of a figure present in an image, thanks to the jumps between one color to another. In addition, your eye can perceive the depths thanks to the various shades of color ranging from light to dark, which is the gradient.

From all this, it is quite clear that measuring a gradient in an image is crucial to being able to detect the edges of the image. It's done with a simple operation (filter) that is carried out on the image.

To get a better look at it from a mathematical point of view, look at Figure 14-10.

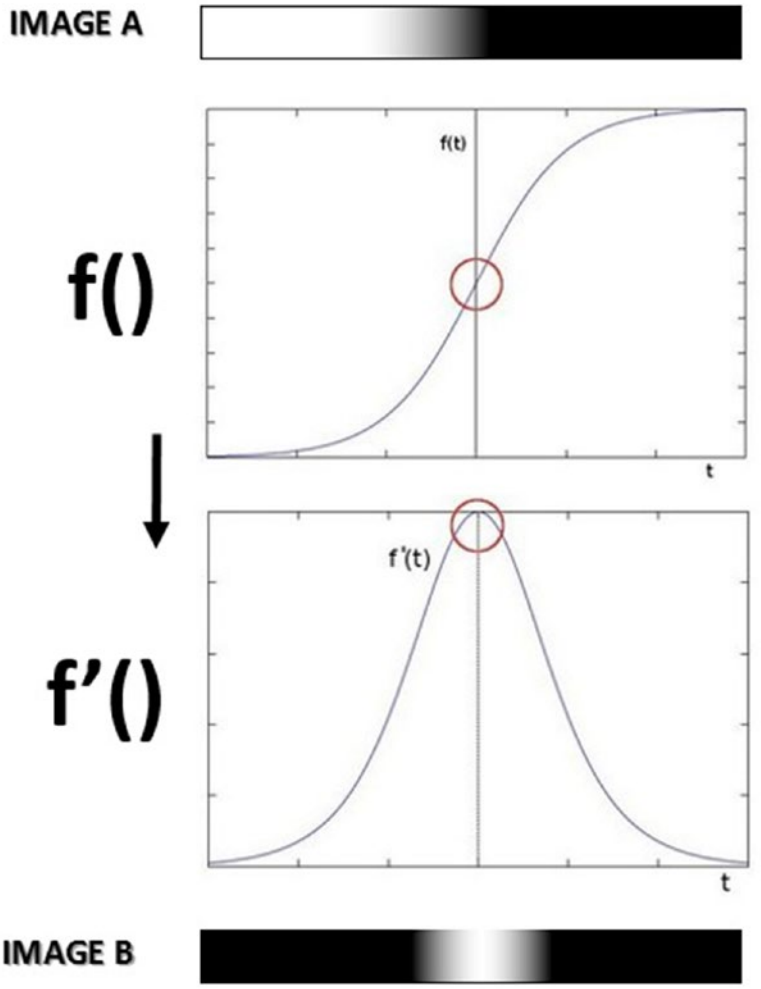

Figure 14-10. *The image gradient theory representation*

As you can see in Figure 14-10, an edge is no more than a quick transition from one hue to another. To simplify, 0 is black and 1 is white. All shades of gray are floating values between 0 and 1.

If you chart all corresponding values to the gradient values, you get the function `f()`. As you can see, there is a sudden transition from 0 to 1, which indicates the edge.

The derivative of the function `f()` results in the function `f'()`. As you can see, the maximum variation of the hue leads to values close to 1. So when converting colors, you will get a figure in which white will indicate the edge.

A Practical Example of Edge Detection with the Image Gradient Analysis

Moving on to the practical part, you will use two images created specifically to test the analysis of the contours, since they have several important characteristics in them.

The first image (as shown in Figure 14-11) consists of two arrows in black and white and corresponds to the `blackandwhite.jpg` file. In this image, the color contrast is very strong and the contours of the arrows have all the possible orientations (horizontal, vertical, and diagonal). This test image will serve to evaluate the effect of edge detection in a black and white system.

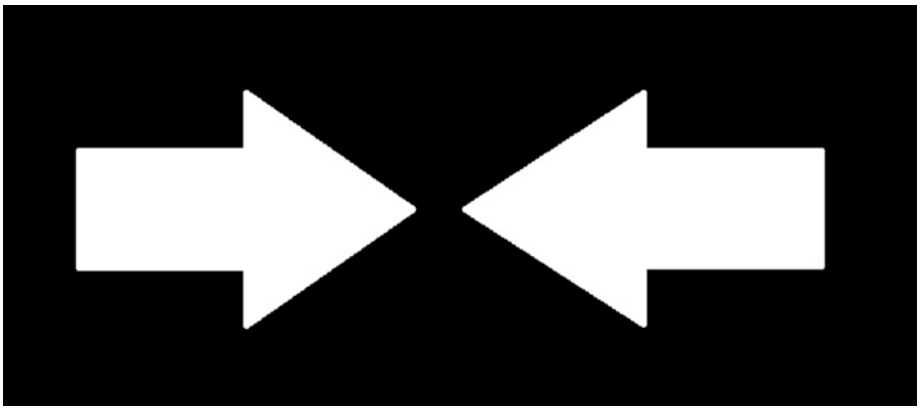

Figure 14-11. *A black and white image representing two arrows*

The second image, `gradients.jpg`, shows different gradients of gray, which, when placed next to each other, create rectangles whose edges have all the possible gradations and combinations of shades (as shown in Figure 14-12). This image is a good test to evaluate the true edge detection capabilities of the system.

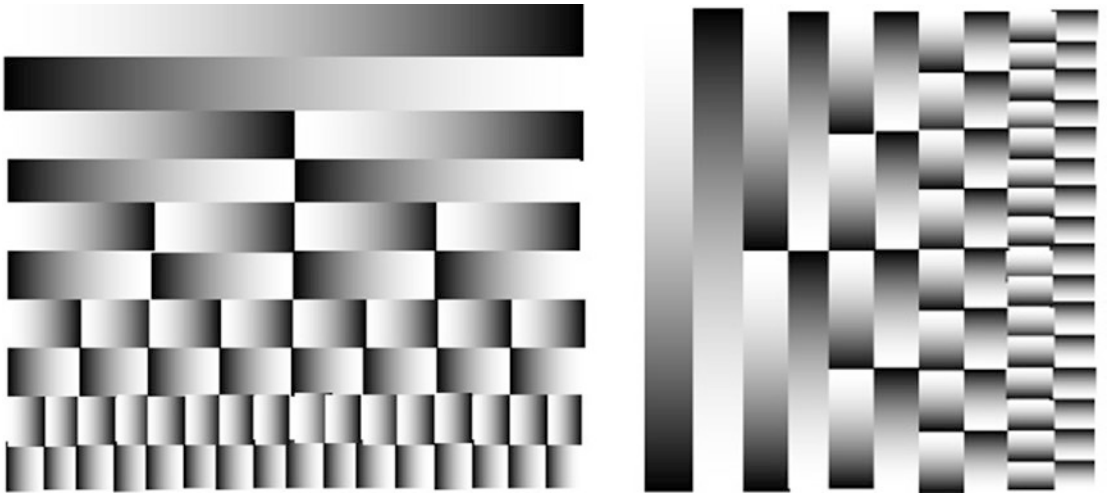

Figure 14-12. *A set of gray gradients placed next to each other*

Now you can start to develop the code needed for edge detection. You will use matplotlib to display different images in the same window. In this test, we will use two different types of image filters provided by opencv: *sobel* and *laplacian*. In fact, their names correspond to the name of the mathematical operations performed on the matrices (images). The openCV library provides `cv2.Sobel()` and `cv2.Laplacian()` to apply these two calculations.

First it starts by analyzing the edge detection applied to the `blackandwhite.jpg` image.

```
from matplotlib import pyplot as plt
%matplotlib inline
img = cv2.imread('blackandwhite.jpg',0)
laplacian = cv2.Laplacian(img, cv2.CV_64F)
sobelx = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)
sobely = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)

plt.subplot(2,2,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,2),plt.imshow(laplacian,cmap = 'gray')
plt.title('Laplacian'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,3),plt.imshow(sobelx,cmap = 'gray')
plt.title('Sobel Y'), plt.xticks([]), plt.yticks([])
```

```
plt.subplot(2,2,4),plt.imshow(sobely,cmap = 'gray')
plt.title('Sobel Y'), plt.xticks([]), plt.yticks([])
plt.show()
```

When you run this code, you get a window with four boxes (as shown in Figure 14-13). The first box is the original image in black and white, while the other three are the result of the three filters applied to the image.

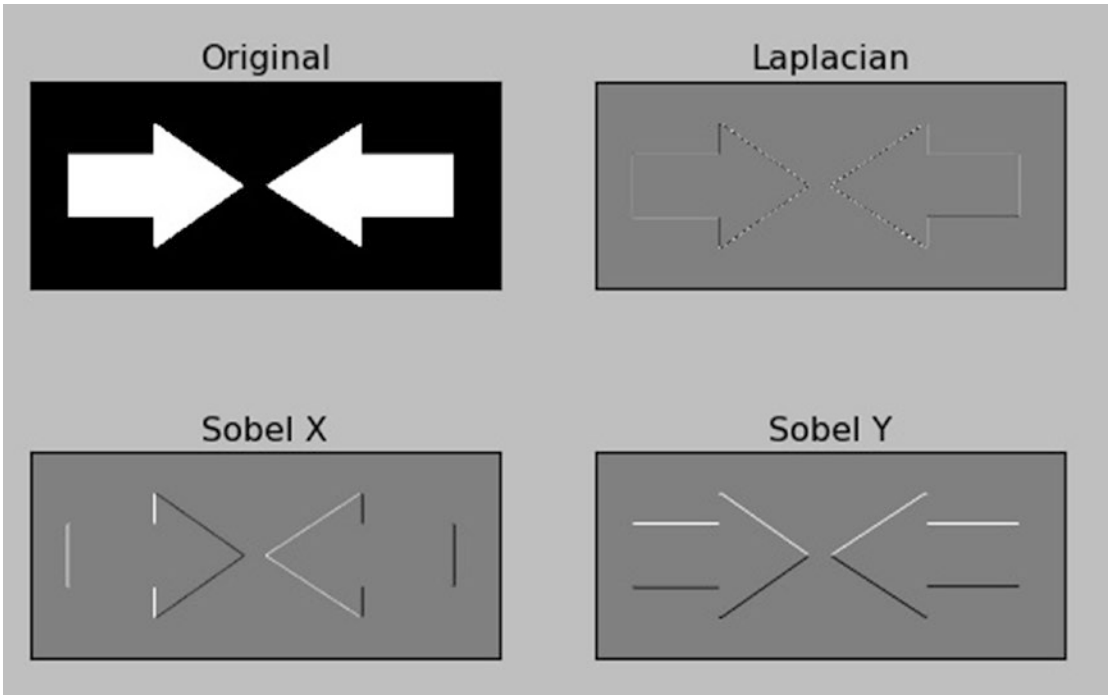

Figure 14-13. *The result from the edge detection applied to the blackandwhite.jpg image*

In regards to the Sobel filters, edge detection is perfect, even if limited horizontally or vertically. The diagonal lines are visible in both cases, since they have both horizontal and vertical components, but the horizontal edges in the Sobel X and those in the vertical Sobel Y are not detected in any way.

Combining the two filters (the calculation of two derivatives) to obtain the Laplacian filter, the determination of the edges is omnidirectional but has some loss of resolution. In fact, you can see that the ripples corresponding to the edges are more subdued.

The coloring in gray is very useful for detecting edges and gradients, but if you are interested in only detecting edges, you have to set as output an image file in `cv2.CV_8U`.

Therefore, you can change the type of output data from `cv2.CV_64F` to `cv2.CV_8U` in the filters function of the previous code. Replace the arguments passed to the two image filters as follows.

```
laplacian = cv2.Laplacian(img, cv2.CV_8U)
sobelx = cv2.Sobel(img,cv2.CV_8U,1,0,ksize=5)
sobely = cv2.Sobel(img,cv2.CV_8U,0,1,ksize=5)
```

By running the code, you will get similar results (as shown in Figure 14-14), but this time only in black and white, where the edges are displayed in white on a black background.

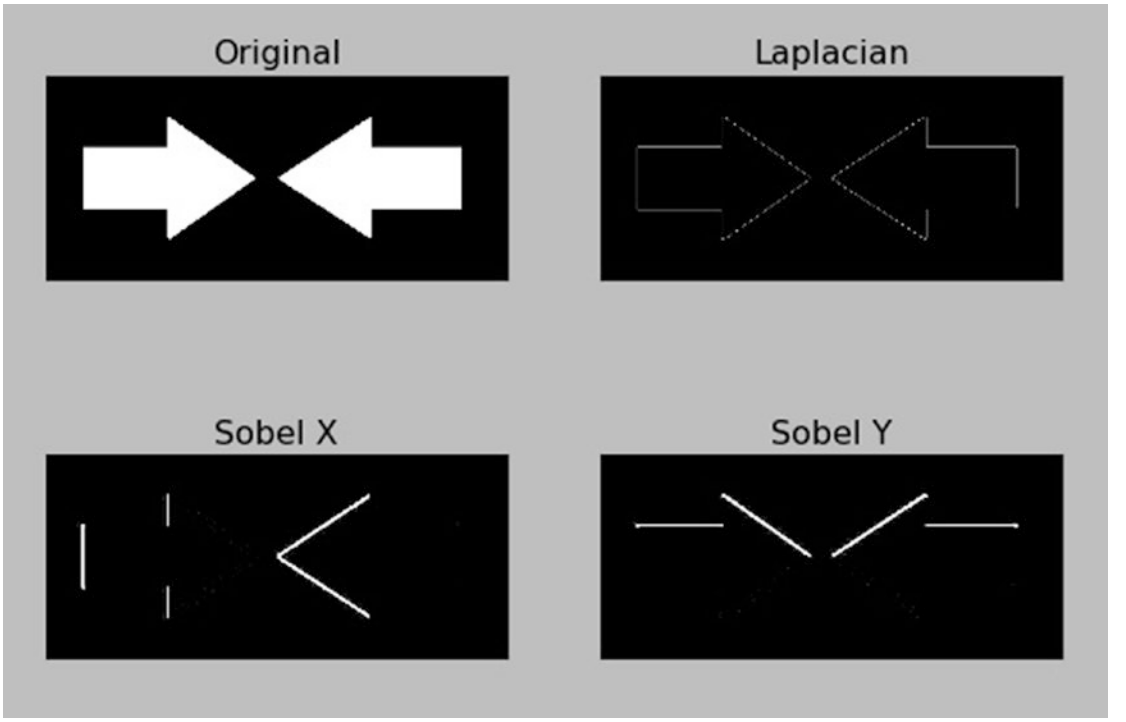

Figure 14-14. *The result from the edge detection applied to the blackandwhite.jpg image*

But if you look carefully at the panels of the Sobel filter X and Y, you will notice right away that something is wrong. Where are the missing edges? Note this issue in Figure 14-15.

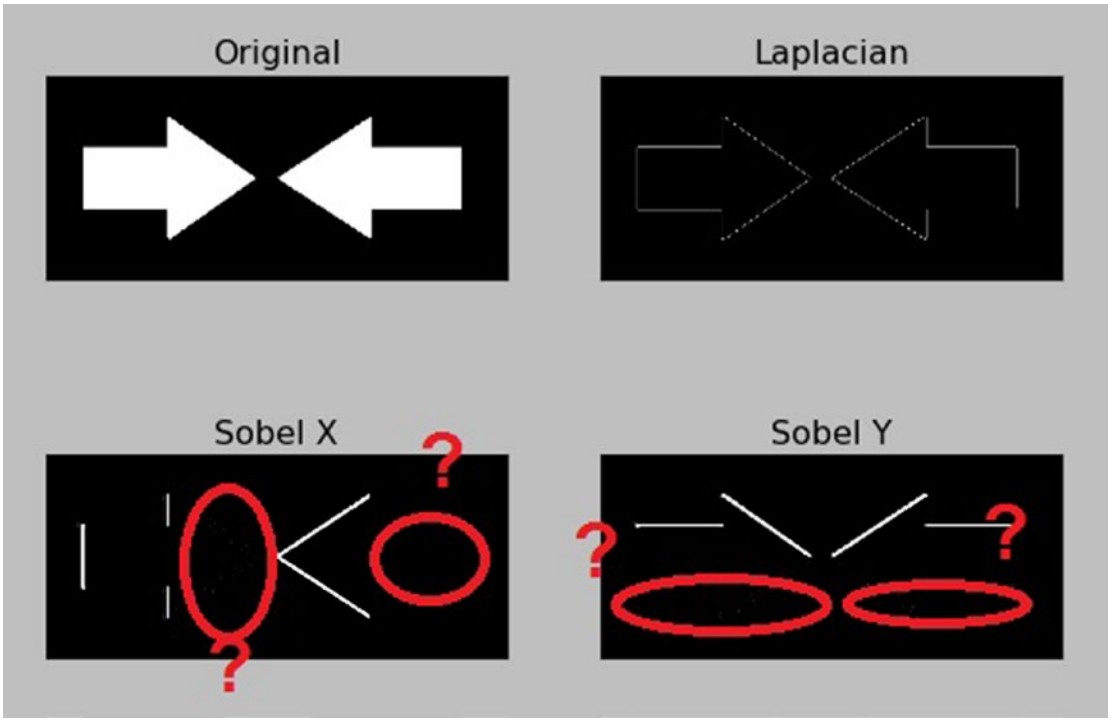

Figure 14-15. *Missing edges in the blackandwhite.jpg image*

In fact, there was a problem while converting the data. The gradients reported in the grayscale with `cv2.CV_64F` values are represented by positive values (positive slope) when changing from black to white. However, they are represented by negative values (negative slope) when switching from white to black. In the conversion from `cv2.CV_64F` to `cv2.CV_8U`, all negative slopes are reduced to 0, and then the information relating to those edges is lost. When the program will display the image, the edges from white to black will not be shown.

To overcome this, you should keep the data in the output of the filter in `cv2.CV_64F` (instead of `cv2.CV_8U`), then calculate the absolute value, and finally do the conversion in `cv2.CV_8U`.

Make these changes to the code.

```
laplacian64 = cv2.Laplacian(img, cv2.CV_64F)
sobelx64 = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)
sobely64 = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)
laplacian = np.uint8(np.absolute(laplacian64))
sobelx = np.uint8(np.absolute(sobelx64))
sobely = np.uint8(np.absolute(sobely64))

plt.subplot(2,2,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,2),plt.imshow(laplacian,cmap = 'gray')
plt.title('Laplacian'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,3),plt.imshow(sobelx,cmap = 'gray')
plt.title('Sobel Y'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,4),plt.imshow(sobely,cmap = 'gray')
plt.title('Sobel Y'), plt.xticks([]), plt.yticks([])
plt.show()
```

Now, if you execute it, you will get the right representation in white on the black edges of the arrows (as shown in Figure 14-16). As you can see, the edges do not appear in Sobel X and Sobel Y because they are parallel to the direction of detection (horizontal and vertical).

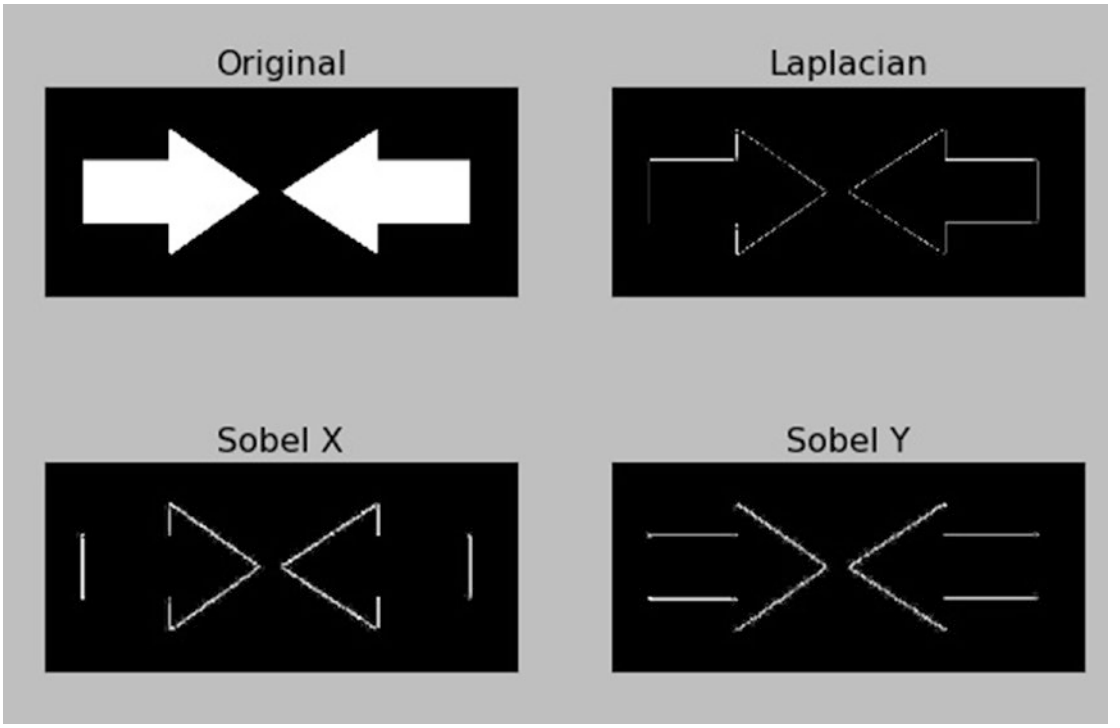

Figure 14-16. *The result from the edge detection applied to the blackandwhite.jpg image*

In addition to the edges, you see that the Laplacian and Sobel filters are also able to detect the level of gradients across a grayscale. Apply what you've seen to the `gradient.jpg` image. You have to make some changes to the previous code, leaving only one image (Laplacian) to be shown.

```
from matplotlib import pyplot as plt
img = cv2.imread('gradients.jpg',0)
laplacian = cv2.Laplacian(img, cv2.CV_64F)
sobelx = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)
sobely = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)

laplacian64 = cv2.Laplacian(img, cv2.CV_64F)
sobelx64 = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)
sobely64 = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)
```

```
laplacian = np.uint8(np.absolute(laplacian64))
sobelx = np.uint8(np.absolute(sobelx64))
sobely = np.uint8(np.absolute(sobely64))

plt.imshow(laplacian,cmap = 'gray')
plt.title('Laplacian'), plt.xticks([]), plt.yticks([])
plt.show()
```

By executing this code, you will get an image showing the white borders on a black background (as shown in Figure 14-17).

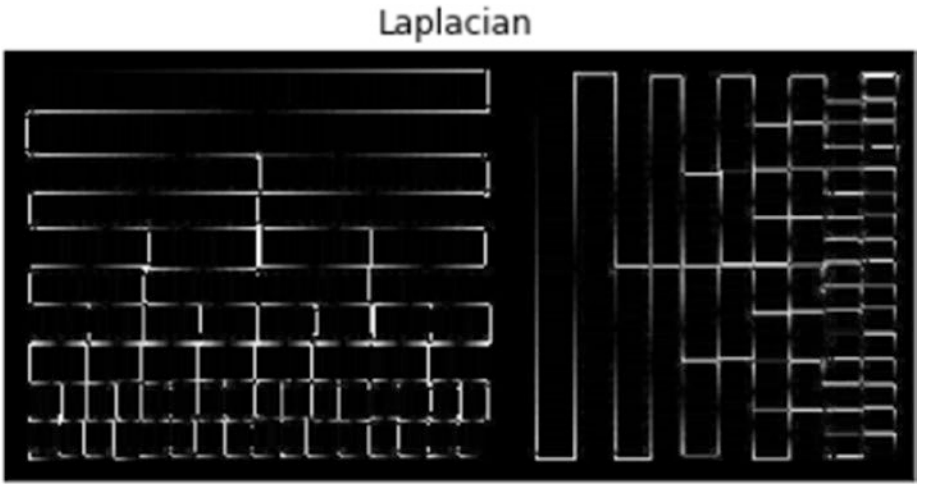

Figure 14-17. *The result from the edge detection applied to the gradients.jpg image*

A Deep Learning Example: The Face Detection

In this last section of the chapter, you will shift your attention to another highly studied and used case in computer vision, face detection.

This is a far more complex case than edge detection, and it is based on identifying human faces in an image. Given the complexity of the problem, face detection uses deep learning. In fact at the base of this technique, there are neural networks that are specially designed to recognize different subjects, including the faces of a person, in a photo. Object detection techniques also work very similarly. So this example will be very useful to fully understand the heart of computer vision, that of interpreting the subjects present in a photo.

In this example, you will use an already learned neural network. In fact, educating a neural network for this kind of problem can be a complex operation and require a great deal of time and resources.

Fortunately, on the web, there are some neural networks already trained to perform these kinds of operations, and for this test you will use a model developed using the `Caffe2` framework (see Chapter 9 for more information).

When you want to use a deep neural network module with Caffe models in the OpenCV environment, you need two types of files, as follows:

- A *prototxt* file, which defines the model architecture (i.e., the layers themselves). You will use a `deploy.prototxt.txt` file downloaded from the web (`https://github.com/opencv/opencv/blob/master/samples/dnn/face_detector/deploy.prototxt`).
- The second type of file is a *caffemodel* file, which contains the weights for the actual layers in the deep neural network. This file is the most important because it contains all the "learning" of that neural network to perform a given task. For your purposes, a caffemodel file is available at `https://github.com/opencv/opencv_3rdparty/tree/dnn_samples_face_detector_20170830`.

Note You can also find these files in the source code of the book.

Now that you have everything you need, start by uploading the neural network model and all the information about your learning.

The OpenCV library supports many deep learning frameworks, and it has many features in it that help you with this. In particular (mentioned at the beginning of the chapter), OpenCV has the `dnn` module, which specializes in these kinds of operations.

To load a learned neural network you can use the `dnn.readNetFromCaffe()` function.

```
net = cv2.dnn.readNetFromCaffe('deploy.prototxt.txt', 'res10_300x300_ssd_
iter_140000.caffemodel')
```

As a test image, you can use the photo with the players of the Italian national team, `italy2018.jpg`. This image is a great example, as there are many faces inside.

```
image = cv2.imread('italy2018.jpg')
(h, w) = image.shape[:2]
```

Another function, called `dnn.blobFromImage()`, takes care of preprocessing the image to be adapted to neural networks. For example, resize the image to 300x300 pixels so that it can be used by the caffemodel file that has been trained for images of this size.

```
blob = cv2.dnn.blobFromImage(cv2.resize(image, (300, 300)), 1.0, (300, 300),
(104.0, 177.0, 123.0))
```

Then define a confidence threshold with an optimal value of 0.5.

```
confidence_threshold = 0.5
```

And finally perform the face detection test.

```
net.setInput(blob)
detections = net.forward()

for i in range(0, detections.shape[2]):
    confidence = detections[0, 0, i, 2]
    if confidence > confidence_threshold:
        box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
        (startX, startY, endX, endY) = box.astype("int")
        text = "{:.2f}%".format(confidence * 100)
        y = startY - 10 if startY - 10 > 10 else startY + 10
        cv2.rectangle(image, (startX, startY), (endX, endY),(0, 0, 255), 2)
        cv2.putText(image, text, (startX, y), cv2.FONT_HERSHEY_SIMPLEX,
        0.45, (0, 0, 255), 2)

cv2.imshow("Output", image)
cv2.waitKey(0)
```

By executing the code, a window will appear with the results of processing the face detection (shown in Figure 14-18). The results are incredible, since the faces of all the players have been detected. You can see the faces surrounded by a red square that

highlights them in the image with a percentage of confidence. Confidence percentages are all greater than 50% for the `confidence_threshold` parameter that we specified at the start of the test.

Figure 14-18. *The faces of the national football players have all been accurately recognized*

Conclusions

In this chapter, you saw some simple examples of techniques that form the basis of image analysis and in particular of computer vision. In fact, you saw how images are processed through image filters, and how some complex techniques can be built using edge detection. You also saw how computer vision works by using deep learning neural networks to recognize faces in an image (face detection).

I hope this chapter is a good starting point for your further insights on the subject. If you are interested, you will find in-depth information on this topic on my website at `https://meccanismocomplesso.org`.