Saurabh Kapur

Computer Vision with Python 3

Image classification, object detection, video processing, and more

Packt

Computer Vision with Python 3

Image classification, object detection, video processing, and more

Saurabh Kapur

Packt

BIRMINGHAM - MUMBAI

Computer Vision with Python 3

First published: August 2017

Production reference: 1210817

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78829-976-3

www.packtpub.com

Credits

Author
Saurabh Kapur

Reviewer
Will Brennan

Commissioning Editor
Aaron Lazar

Acquisition Editor
Chandan Kumar

Content Development Editor
Deepti Thore

Technical Editor
Sneha Hanchate

Copy Editor
Laxmi Subramanian

Project Coordinator
Shweta H Birwatkar

Proofreader
Safis Editing

Indexer
Pratik Shirodkar

Graphics
Tania Dutta

Production Coordinator
Melwyn Dsa

About the Author

Saurabh Kapur is a computer science student at Indraprastha Institute of Information Technology, Delhi.

His interests are in computer vision, numerical analysis, and algorithm design. He often spends time solving competitive programming questions. Saurabh also enjoys working on IoT applications and tinkering with hardware.

He likes to spend his free time playing or watching cricket. He can be reached at saurabhkapur96@gmail.com.

About the Reviewer

Will Brennan is a C++ and Python developer based in London, with experience of working on high-performance image processing and machine learning applications.

You can read more about Will at `https://github.com//WillBrennan`.

Customer Feedback

Table of Contents

Preface

Computer vision has gone through significant advancements over the last few years. These advancements have not only made a great impact of computer science but also of other fields, such as medicine, space exploration, and defense. From microscopic organisms to celestial particles light-years away, using computer vision techniques researchers are able to analyze their images to make progress in these fields. Computer vision is now being used as a tool to facilitate research and development in many other fields.

Going forward, the impact of computer vision will have an even greater impact—the most recent application being driverless cars. To be able to be a part of this ongoing revolution, it is important that we are able to understand and implement computer vision algorithms. This book will introduce the reader to three computer vision libraries written for Python—Pillow, Scikit-image, and OpenCV. Through examples and code snippets, the book will help the reader understand the basics of image processing, morphological operations, and eventually, complex feature detection algorithms.

What this book covers

`Chapter 1`, *Introduction to Image Processing*, as the name suggests, introduces the reader to the basics of image processing. Starting with some common use cases of image processing, the chapter goes on to explain how to install different image processing libraries. The next few sections in the chapter explain how to read/write an image and perform basic image manipulation operations.

`Chapter 2`, *Filters and Features*, gives the reader an overview of what filters and features mean in the context of computer vision. We start with convolution, which forms the basis of applying any filter to an image. Then we look at some common filters, such as Gaussian Blur and Median Blur. The second half of the chapter explains the basic image features and how they are implemented using Python.

`Chapter 3`, *Drilling Deeper into Features – Object Detectio*n, walks the reader through some of the sophisticated image feature extraction algorithms, such as Local Binary Pattern and ORB. These algorithms help to identify objects in an image and match them with other images that have the same objects in them. Such matching algorithms form the basis of the most complex computer vision algorithms.

`Chapter 4`, *Segmentation – Understanding Images Better*, has a different theme than the last two chapters. This chapter looks at different image segmentation algorithms, namely, contour detection, superpixels, watershed, and normalized graph cut. These algorithms are fairly easy to implement and run in almost real time. Image segmentation can be use in real-world applications such as background subtraction, image understanding, and scene labeling. Recent advances in machine learning, especially deep learning, have enabled more sophisticated methods of image segmentation that involve almost no manual tuning of parameters.

`Chapter 5`, *Integrating Machine Learning with Computer Vision*, brings together two different fields together. This chapter shows how machine learning algorithms can be implemented for images. We implement a classic image classification program for digit recognition.

`Chapter 6`, *Image Classification Using Neural Networks*, is an extension of the last chapter. In this chapter, we implement digit classification using advanced machine learning technique called neural networks. We install a new library called Keras to implement the neural network.

`Chapter 7`, *Introduction to Computer Vision Using OpenCV*, introduces the readers to a new computer vision library called OpenCV. In this chapter, we revisit all the image processing concepts and algorithms that we have read so far in the book and implement them using OpenCV.

`Chapter 8`, *Object Detection Using OpenCV*, explains different feature extraction algorithms and we will be using OpenCV to implement all the algorithms.

`Chapter 9`, *Video Processing Using OpenCV*, explains how to work with videos instead of images. The chapter uses code snippets to walk the reader through how to capture and save a video. It then explains how to perform operations such as resizing and changing the color space in videos. In the last section, we see how to implement object tracking in videos.

`Chapter 10`, *Computer Vision as a Service*, is the last chapter and it provides an overview of how production-scale computer vision systems are built. The chapter focuses on the infrastructure that is needed for computer vision algorithms. A simple computer vision service is implemented, giving the readers a flavor of how services such as Google Image search are built.

What you need for this book

The software required for this book are as follows:

- Python 3.5
- Pillow 4.0
- Scikit-image (Skimage) 0.13.0
- OpenCV 3.2
- Sklearn 0.18
- Keras 2.0
- Flask 0.12.2

Who this book is for

The book is ideal for developers who have basic knowledge of Python and want to build a strong foundation in implementing computer vision algorithms. The book is also suitable for developers with theoretical knowledge of computer vision but who lack the experience of implementing the algorithms.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"To save or write an image we can use the `imsave()` function."

A block of code is set as follows:

```
>>> from PIL import Image
>>> img = Image.open("image.png")
>>> img.getpixel((100,100))
output
(150, 188, 233, 255)
>>> img.convert("L").getpixel((100,100))
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
>>> from skimage import io
 >>> img = io.imread("image.png")
 >>> io.imshow("image.png")
 >>> io.show()
```

Any command-line (including commands in the R console) input or output is written as follows:

```
$: pip install Pillow
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes, for example, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important to us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply email `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your email address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Computer-Vision-with-Python-3`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/ComputerVisionwithPython3_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

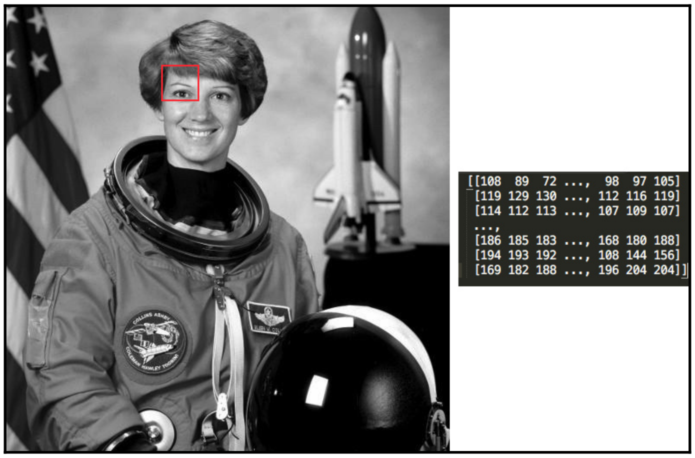
We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introduction to Image Processing

Before diving straight into image processing, let's understand images first. An image, as humans see it, is a two-dimensional grid with each cell in the grid filled with a color value, otherwise called a pixel value. Each cell of the grid is formally called a picture element (commonly abbreviated to pixel). A computer also sees the image in the same way. An image on a computer is a two-dimensional matrix of numbers with each cell in the matrix storing the corresponding pixel value(s) in the image. The following figure is an example of an image matrix. The matrix of the portion of the image in the red box is shown on the right:



```
[[108  89  72 ...,  98  97 105]
 [119 129 130 ..., 112 116 119]
 [114 112 113 ..., 107 109 107]
 ...,
 [186 185 183 ..., 168 180 188]
 [194 193 192 ..., 108 144 156]
 [169 182 188 ..., 196 204 204]]
```

Figure 1: This is the image matrix (right), as stored on a computer, of a small portion of the image (left) in the red box.

Image processing is the field of studying and analyzing images. There is a lot of hidden information in an image that we unconsciously process. For example, what are the different objects in the image?, Is there a car in the image? What are the similarities between any two images? Answers to these questions might feel simple to us humans, but for a computer, to answer such questions is extremely difficult. Through the course of this book, we aim to implement some of the algorithms that can help us answer some of these questions

The essence of image processing is to use the different properties of an image such as color, co-relations between different pixels, object placements, and other fine details to extract meaningful information such as edges, objects, and contours, which are formally called image features. These features can then be used in different applications such as medicine, security, social media services, and self-driving cars, some of which will be covered in the following chapters.

Image processing - its applications

Let's take a look at some common applications of image processing:

- **Medicine**: In recent years, the field of medicine has seen rapid advancements. For example, more sophisticated imaging techniques and better techniques to detect the nature of tumors in MRI/PET scans. The interdisciplinary research between biology and image processing played an important role. The following image illustrates how image processing algorithms are being used to detect tumors. This has helped in early diagnosis of diseases and a more effective treatment:

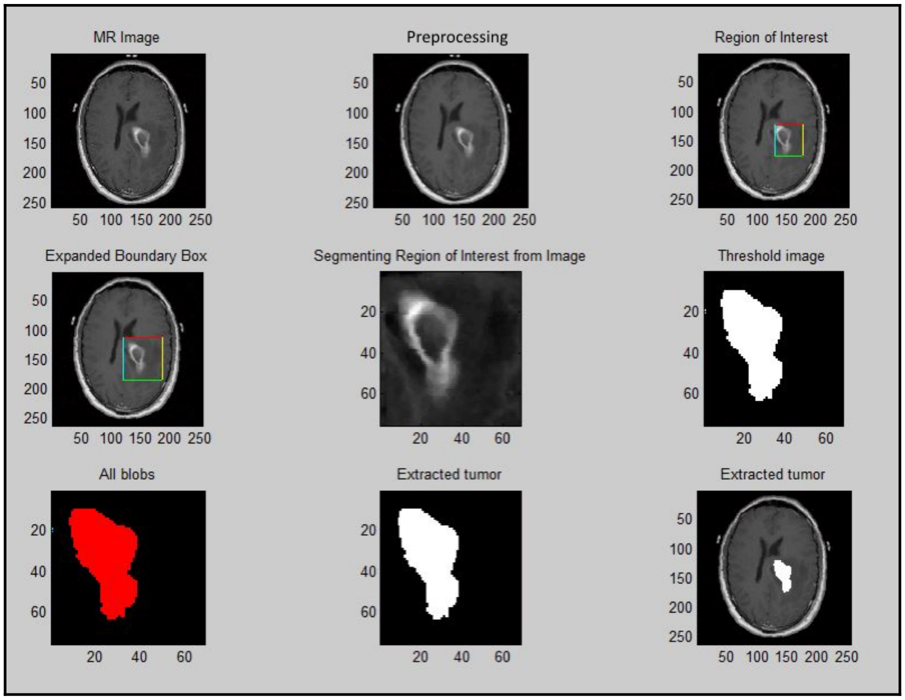


Figure 2 : The image shows how image processing can be used to detect tumors.

- **Security image processing**: This has helped in developing efficient security/surveillance systems. Advancements in this field have impacted a lot of different consumer products as well as enterprises. Fingerprint unlock systems and biometric security systems (face or iris recognition) are now being used in small devices such as mobile phones and even in smart buildings. With the use of these techniques, unlocking devices has become simpler and easier compared to remembering and typing passwords or even carrying **Radio Frequency Identification** (**RFID**) security cards. These concepts have been extended to home security systems as well. Work in the field of human body detection and recognition has led to smarter intrusion detection systems.
- **Social media**: Various social media websites such as Facebook, Instagram, and Snapchat use some form of computer vision techniques to enhance the user experience. For example, Facebook's autotag feature recognizes faces in the pictures that users upload and suggests you an appropriate name tag for the person in the picture. Another application is the Google image search. It searches for visually similar images over the World Wide Web, which is a non-trivial task.

These are few of the applications of computer vision (image processing). There are countless more such applications in the real world, which are outside the scope of this book.

Image processing libraries

There are several image processing libraries written in Python for computer vision. For the purpose of this book we will look into scikit-image and pillow. These libraries will be used throughout this book to implement the algorithms that will be discussed. In the next section, you will be shown how to install these libraries and do some basic image processing operations to prepare you for the next chapters.

Pillow

Pillow is an open source library that has been forked from the **Python Imaging Library** (**PIL**). Pillow is a very good starting point for beginners who want to start with implementing some basic algorithms before diving into the more complex ones. The book will use Pillow version 4.0.

You can find more information on pillow at `https://python-pillow.org/`.

Installation

In this section, we will see how to install Pillow on different operating systems:

- **Windows**: Pillow can be installed on windows using pip. Open the command-line tool on your Windows machine and type in the following command and press *Enter*:

```
$: pip install Pillow
```

pip already comes installed with Python 2>=2.7.9 and Python 3>=3.4. In case you do not have pip installed, follow the official instructions given at `https://pip.pypa.io/en/stable/installing/#do-i-need-to-install-pip`.

- **OSX/macOS**: For OSX/macOS, we will use Homebrew to install Pillow.

Go to `https://brew.sh/` for instructions on how to install Homebrew in case you do not have it installed.

Open the terminal on your Mac. First, install the dependencies and then pillow using the following commands:

```
$: brew install libtiff libjpeg webp little-cms2
$: pip install Pillow
```

If you have both Python2 and Python3 installed, then to install Pillow for Python3, use the following command:

```
$: python3 -m pip install Pillow
```

- **Linux**: Use the `pip` command to install pillow on a Linux operating system:

```
$: pip install Pillow
```

Getting started with pillow

This section will walk you through the basics of pillow using relevant code snippets.

Reading an image

To read an image from a jpg or a png file saved on your computer, Pillow's image module provides a `read()` function (`Image.open`). This function returns an image object, which contains information such as pixel type, image size, and image format. The following is an example of how to read an image. Note that the import statement is only run once at the beginning of the program:

```
>>> from PIL import Image
>>> img = Image.open("image.png")
```

To display the image on your screen, use the `show()` function as follows:

```
>>> img.show()
```

Writing or saving an image

To write or save an image to a file on your computer, use the `save()` function associated to the image object. It takes in the absolute or relative file path to where you want to store the image:

```
>>> img.save("temp.png") # Example showing relative path
>>> img.save("/tmp/temp.png") # Example showing absolute path
```

Cropping an image

Cropping an image means to extract a particular region of the image, which is smaller than the original image. This region in some books/references in called the **Region of Interest** (**ROI**). The concept of ROI is sometimes useful when you want to run your algorithm only on a particular part of the image and not the entire image. The image object has a `crop()` function that takes two coordinates--the upper-left corner and the bottom-right corner of the rectangle that you are interested in--and returns the cropped image:

```
>>> from PIL import Image
>>> dim = (100,100,400,400) #Dimensions of the ROI
>>> crop_img = img.crop(dim)
>>> crop_img.show()
```

The following images shows the `crop` function as used in the preceding code:

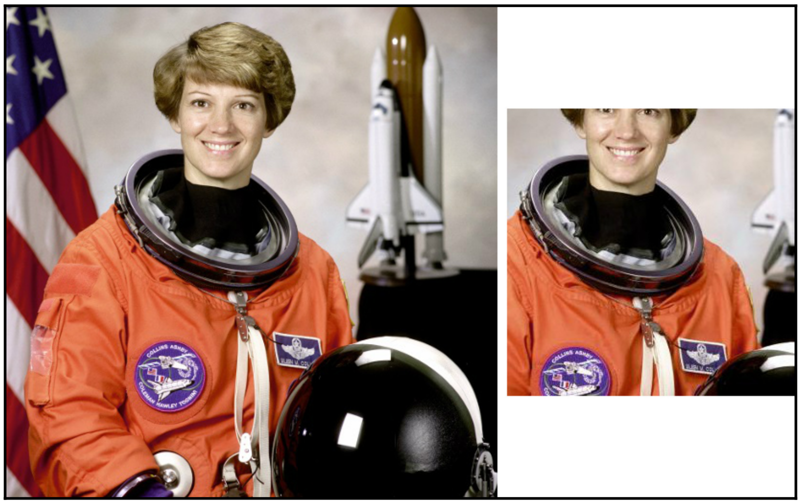


Figure 3: (Left) Original image and (right) a cropped region of the original image

Changing between color spaces

Color spaces and channels

Like in the world of mathematics, we have different coordinate system: for example, a 2-D cartesian plane and 2-D polar coordinates. A point could be stored as (x, y) or (r, theta). Each coordinate system has a specific use case, which makes calculations easier. Similarly, in the world of image processing, we have different color spaces. An image can store its color values in the form of Red, Blue, Green (RGB) or it could as Cyan, Magenta, Yellow, Key(black) (CMYK). Some examples of other color spaces are HSV, HSL, CMY, and it goes on. Each value in the color space is called a color channel. For example, in the RGB color space we say that Red, Blue, and Green each are channels of the image. An image can be represented in many different modes (color spaces) such as RGB, CMYK, Grayscale, and YUV. The colors in the image that we see are derived by the mixture of the colors in each color channel of the color space. Let's look at some of the common color spaces in detail:

- **Grayscale**: This is one of the simplest color spaces both in terms of understanding and storing on a computer. Each pixel value in a grayscale image is a single value between 0 and 255, with 0 representing black and 255 representing white. Keep in mind that the value 255 is not a fixed value but depends on the depth of the image (image depth is covered in the next section). Grayscale images are also sometimes called black and white images but it is not entirely accurate. A black and white image means that the pixel values can only be either 0 or 255 and nothing in between.

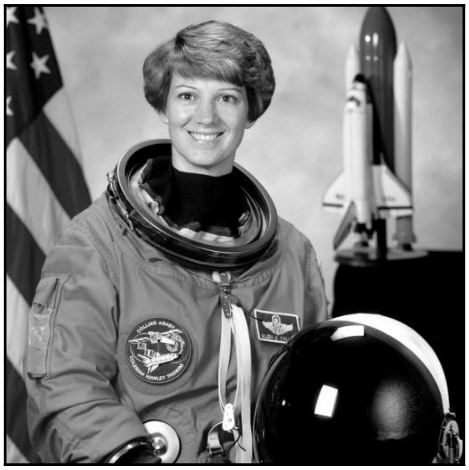


Figure 4 Example of a grayscale image

- **Red, Green, Blue** (**RGB**): This is one of the most common color spaces that is used in the image processing world and elsewhere. Most images that you view over the internet or in your books are in the RGB space. In a typical RGB image, each pixel is a combination of three values, each representing a color in red, green, and blue channels. White color in RGB space is written as (255, 255, 255) and black is written as (0, 0, 0). Red, green, and blue are represented by (255, 0, 0), (0, 255, 0), and (0, 0, 255) respectively. Any other color is just a combination of some values of red, green, and blue. Remember your painting class as a kid where you used to mix the primary colors to create a new color. It's that simple!
- **Hue, Saturation, Value** (**HSV**): This is a cylindrical coordinate system where we project RGB values onto a cylinder. *Figure 5* further illustrates this concept. The HSV color space was designed keeping mind the unintuitive nature of the RGB space. There is no clear intuition to how the color progresses in the RGB space. The HSV scale handles this perfectly in the sense that you can fix the hue and then generate different shades of that hue by just varying values and saturation:

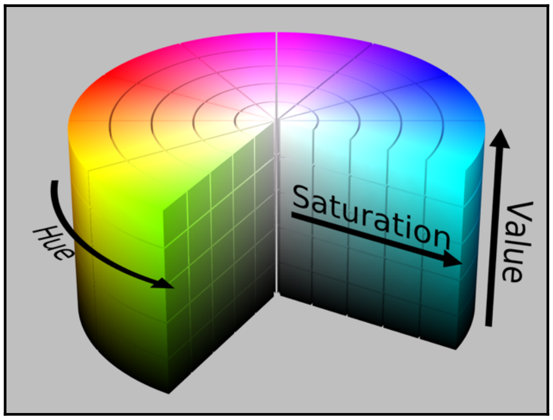


Figure 5: Illustration of HSV color space

Image source: By HSV_color_solid_cylinder.png: SharkDderivative work: SharkD Talk - HSV_color_solid_cylinder.png, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=9801673

At the beginning of the chapter, we said that an image is stored in the form of a 2D matrix. So how do we accommodate for the multiple channels in the image? Simple, we have multiple 2D matrices for each channel. A little exercise--how many matrices will a grayscale image have?

If you try to print the pixel value of a grayscale image, you will only get one value, but if you try to print the pixel value of an RGB image, then you will get three values; this shows that RGB has three channels, red, green, and blue and grayscale images have only one value.

In the following code snippet, we print the pixel values of an RGB image and a grayscale image:

```
>>> from PIL import Image
>>> img = Image.open("image.png")
>>> img.getpixel((100,100))
output
(150, 188, 233, 255)
>>> img.convert("L").getpixel((100,100))
```

This is the following output:

```
181
```

The following image shows the different color channels in an RGB image:

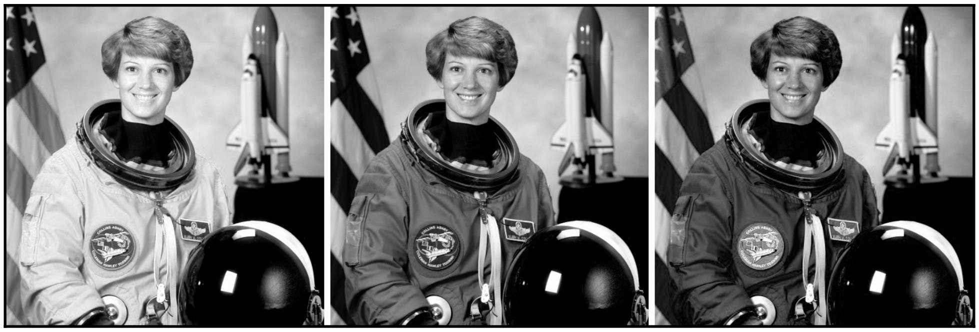


Figure 6: Red, green, and blue respectively

Image depth

Image depth or the color depth is the number of bits used to represent a color of a pixel. The image depth determines the range of colors an image can have. For example, if we have an image with a depth of 4 bits, then the pixel value will range from 0 to 15 (which is the biggest number we can store using 4 bits - 2^4 -1 = 15). Whereas if we use 8 bits, then the value will range from 0 to 255, providing a finer color spectrum. Another way of thinking about image depth is that the number of bits also determines the number of colors, which can be used in an image. For example, 1 bit implies two colors, 2 bits - four colors, and 8 bits - 256 colors.

Images can be converted from one color space to another using the convert function of the image module. To convert an image from RGB color space to grayscale color space, use the *L* mode. There are various other modes available such as *1* which is 1-bit pixel mode, *P*-8 bit pixel mode, *RGB*-3X8 bit pixel, and *RGBA*-4X8 bit pixel.

The following code snippet shows how to convert a color image to grayscale:

```
>>> from PIL import Image
>>> grayscale = img.convert("L")
>>> grayscale.show()
```

The link to the documentation of the Pillow library is `http://pillow.readthedocs.io/en/3.1.x/reference/Image.html#PIL.Image.Image.convert`.

The following image shows the result of the preceding code (converting an image from RGB mode to grayscale mode):

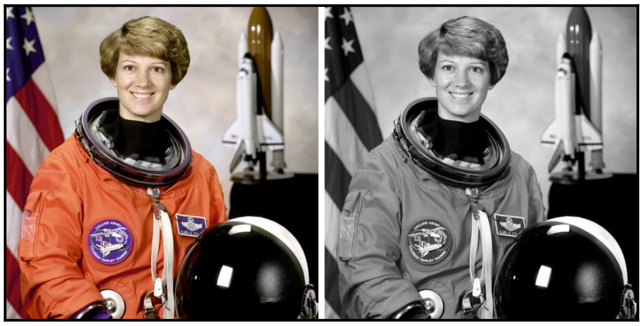


Figure 7: Output after converting from RGB mode to grayscale

Geometrical transformation

There are times when you need to perform different types of transformations to images such as resize, rotate, and flip. Pillow provides direct functions to perform these transformations, saving you from having to write the code from scratch:

- **Resize**: To resize an image, use the `resize()` function, which takes a tuple of the new size as an argument:

```
    >>> from PIL import Image
    >>> resize_img = img.resize((200,200))
    >>> resize_img.show()
```

- **Rotate**: To rotate an image, use the `rotate()` function, which takes in the degrees to be rotated (counter clockwise) as an argument:

```
>>> from PIL import Image
>>> rotate_img = img.rotate(90)
>>> rotate_img.show()
```

The result of the preceding code is shown in the following image:

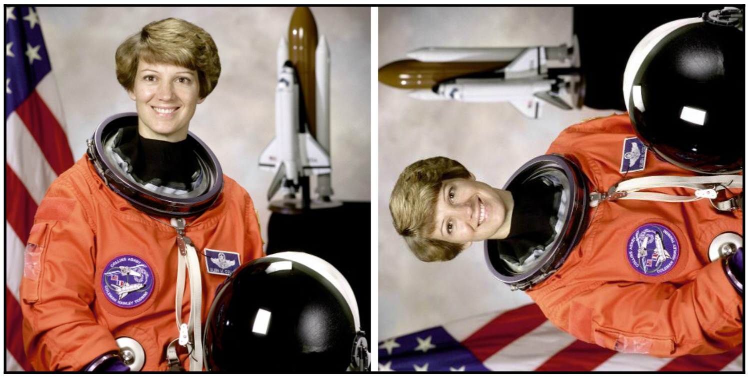


Figure 8: Output after rotating the image by 90 degrees

Image enhancement

Image enhancement involves operations such as changing the contrast, brightness, color balance, or sharpness of an image. Pillow provides an `ImageEnhance` module, which has functions that can help you perform the earlier mentioned operations.

We will begin with importing the `ImageEnhance` module using the following code:

```
>>> from PIL import ImageEnhance
```

After importing the library, let us see how to use the functions available in the library. First we will see how to change the brightness of an image:

- **Change brightness of an image**: We will use the following code to change the brightness:

```
>>> enhancer = ImageEnhance.Brightness(img)
>>> enhancer.enhance(2).show()
```

The `enhance()` function takes a float as an argument, which describes the factor which we want to change the brightness of the image. A factor value less than 1 will decrease the brightness and a factor value greater than 1 will increase the brightness of the image. A factor value equal to 1 will give the original image as output. The output of the `enhance()` function is an image with the changed brightness:

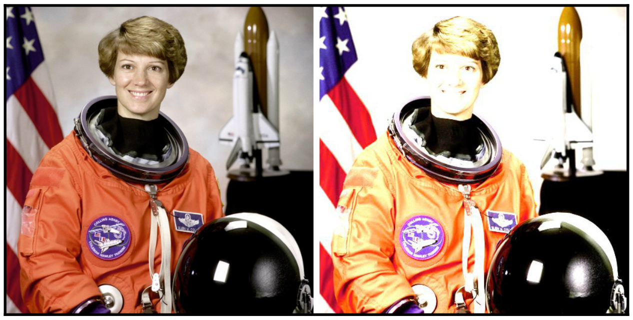


Figure 9: This image shows the increase in the brightness of the image - the image to the left is the original picture and the image to the right is the enhanced one

Next we will see how to change the contrast of an image.

- **Change the contrast of the image**: The following code snippet shows how to enhance the contrast of a given image:

```
>>> enhancer = ImageEnhance.Contrast(img)
>>> enhancer.enhance(2).show()
```

Again the `enhance()` function takes a float argument. A factor equal to 1 will give you the original image, while a factor value less than 1 will decrease the contrast and greater than 1 will increase the contrast:

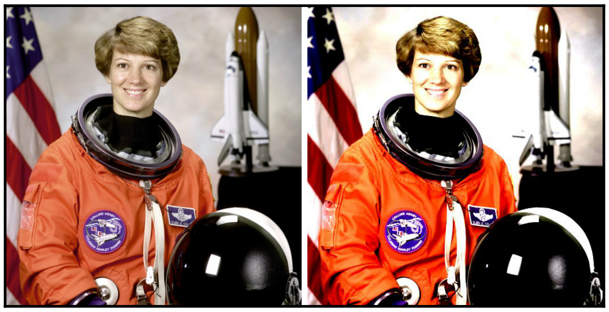


Figure 10: This figure shows the change in the contrast of the image - the image to the left is the original picture and the image to the right is the enhanced image

Accessing pixels of an image

Sometimes for performing tasks such as thresholding (which will be covered later in the book), we have to access the individual pixels in an image. Pillow provides a `PixelAccess` class with functions to manipulate image pixel values. `getpixel()` and `putpixel()` are some of the functions in the `PixelAccess` class:

- `getpixel()`: This function returns the color value of the pixel at the (*x*, *y*) coordinate. It takes a tuple as an argument and returns a tuple of color values:

```
>>> img.getpixel((100,100))
```

This is the following output:

```
(150, 188, 233, 255)
```

- `putpixel()`: This function changes the color value of the pixel at the (*x*, *y*) coordinate to a new color value. Both the coordinates and the new color value are passed as an argument to the function. If the image has more than one band of colors, then a tuple is passed as an argument to the function:

```
>>> img.putpixel((100,100),(20,230,145))
>>> img.getpixel((100,100))
```

This is the following output:

```
(20, 230, 145,255)
```

Introduction to scikit-image

So far we have looked at only integer values for the colors. Some libraries also work with float images where the pixel value lies between 0 and 1.

In this section, we will learn about another Python library for image processing, scikit-image, also represented as **Skimage**. An scikit-image provides more advanced operations as compared to Pillow and is suitable for building enterprise-scale applications.

Here is the official website for scikit-image: `http://scikit-image.org/`

Installation

In this section we look at how to install scikit-image for Python 3 on different operating systems.

- **OSX/macOS**: For installing scikit-image on OSX/macOS, we will use pip. We have already seen how to use `pip` while installing pillow:

```
$: python3 -m pip install -U scikit-image
$: python3 -m pip install scipy
$: python3 -m pip install matplotlib
```

- **Linux (Ubuntu)**: We will use the command-line interface in Linux systems to install scikit-image. Open the default terminal on your computer and type in the following command:

```
$: sudo apt-get install python3-skimage
```

- **Windows**: Similar to what we did for the Linux operating system, for Windows we will also use the command-line interface. Open the the command-line tool and type in the following line to install `skimage` on Windows:

```
pip install scikit-image
```

Getting started with scikit-image

In this section, we will walk through some basic operations that can be performed using the scikit-image library:

- **Reading an image**: As you know, reading an image is the most fundamental operation you would like to perform. In scikit-image, the image can be read using the `imread()` function in the io module of the library. It returns an *ndarray*. An ndarray in Python is an *N* dimensional array. The following is an example:

```
>>> from skimage import io
>>> img = io.imread("image.png")
>>> io.imshow("image.png")
>>> io.show()
```

- **Writing/saving an image**: To save or write an image we can use the `imsave()` function. It takes the absolute or relative path of the file where you want to save the image and the image variable as input:

  ```
  >>> from skimage import io
  >>> img = io.imread("image.png")
  >>> io.imsave("new_image.png", img)
  ```

- **Data module**: This module provides some standard test images which one can work on like a grayscale camera image, grayscale text image, coffee cup, and so on. These images can be used as great examples to demonstrate some of the algorithms in image processing.
  For example, in the following code, `skimage.data.camera()` returns an image array:

  ```
  >>> from skimage import data
  >>> io.imshow(data.camera())
  >>> io.show()
  ```

The following image is the output of the code; that is, the image returned by `skimage.data.camera()`:

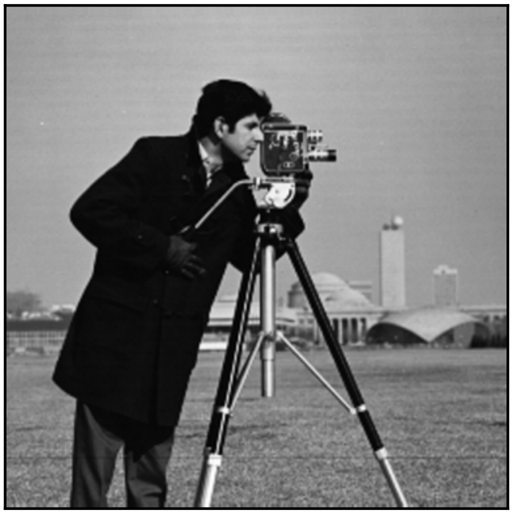


Figure 11: Image returned by the camera() function

Similar to the camera image, we have another image provided by scikit-image. `skimage.data.text()` returns an image which has handwritten text in it:

```
>>> from skimage import data
>>> io.imshow(data.text())
>>> io.show()
```

The following image is the image returned by `skimage.data.text()`:

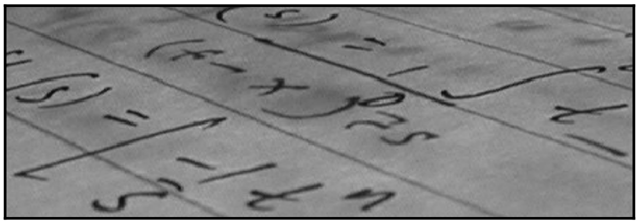


Figure 12:Image returned by the text() function and it can used as an example for corner detection

- **Color module**: This module of the library contains functions for converting the image from one color space to another. Two such functions are shown as follows:
    - Convert RGB to gray: The `rgb2gray()` function in the module can be used to convert a RGB image to a grayscale image. It takes the RGB image array as input and returns the grayscale image array. The following code snippet is an example:

```
>>> from skimage import io, color
>>> img = io.imread("image.png")
>>> gray = color.rgb2gray(img)
>>> io.imshow(gray)
>>> io.show()
```

The following figure is the output of the code:

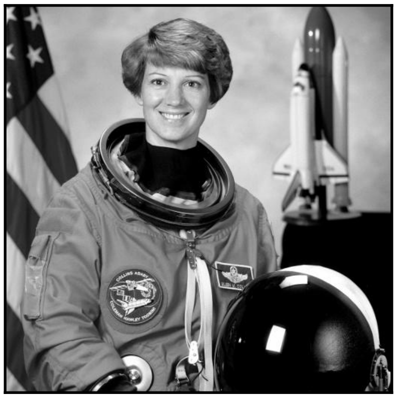


Figure 13: Example of a grayscale image

- Convert RGB to HSV: The `rgb2hsv()` function in the module can be used to convert an RGB image to an HSV image. It takes the RGB image array as input and returns the HSV image array. The following code shows how to convert RGB to HSV:

```
>>> from skimage import data
>>> img = data.astronaut()
>>> img_hsv = color.rgb2hsv(img)
```

There are other functions which can be seen at `http://scikit-image.org/docs/dev/api/skimage.color.html#module-skimage.color`

- **Draw module**: The draw module has various functions to draw different shapes such as circles, ellipses, and polygons. Let's look at each of them one by one:
    - **Circles**: To draw a circle on an image, `skimage` provides a `circle()` function. It takes the center coordinates and the radius as input and returns all the pixel coordinates, which lie within the circle of the given coordinates and radius. After getting the pixels within the circle, assign them the value 1 in the 2D matrix and all the other points make it 0.

```
>>> import numpy as np
>>> from skimage import io, draw
>>> img = np.zeros((100, 100), dtype=np.uint8)
>>> x , y = draw.circle(50, 50, 10)
>>> img[x, y] = 1
>>> io.imshow(img)
>>> io.show(
```

The preceding code snippet would give you a circle as shown here:

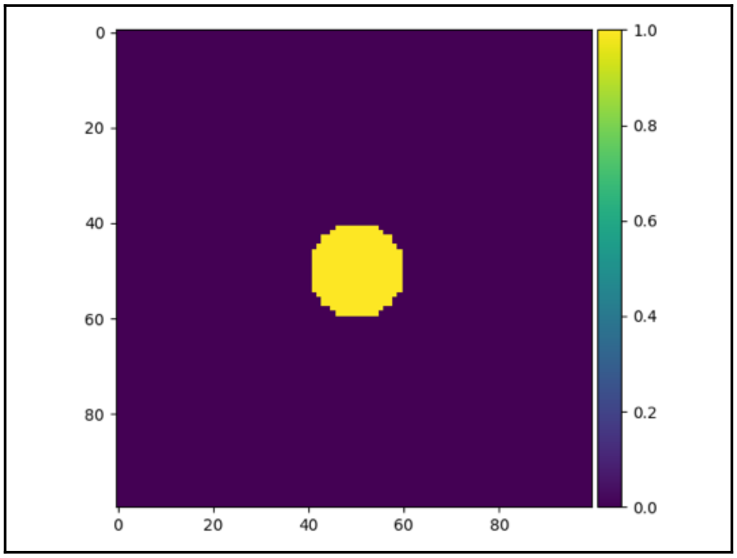


Figure 14: Circle of radius 10 and Centre (50, 50)

- **Ellipses**: To draw an ellipse on an image, `skimage` provides an `ellipse()` function. This function of the draw module can be used to get the coordinates of the pixels within the ellipse of given parameters. Then, these pixels can be distinguished from others by increasing the pixel value:

```
>>> import numpy as np
>>> from skimage import io, draw
>>> img = np.zeros((100, 100), dtype=np.uint8)
>>> x , y = draw.ellipse(50, 50, 10, 20)
>>> img[x, y] = 1
>>> io.imshow(img)
>>> io.show()
```

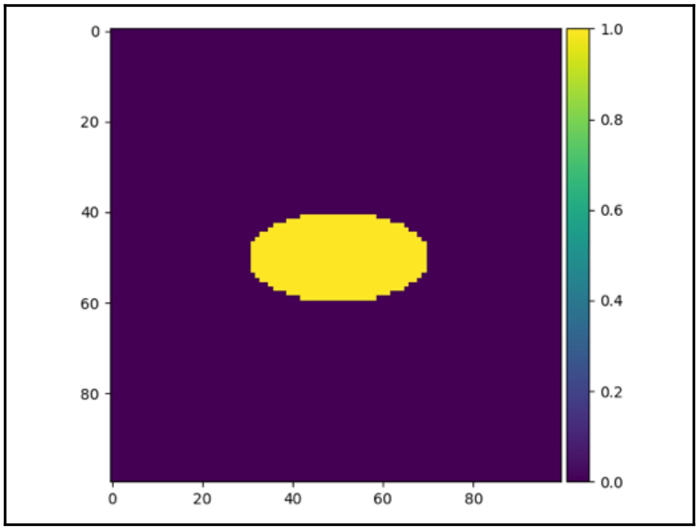


Figure 15: A circle

- **Polygons**: The `polygon()` function takes the array of x and y coordinates of the vertices and returns the pixel coordinates which lie within the polygon:

```
>>> import numpy as np
>>> from skimage import io, draw
>>> img = np.zeros((100, 100), dtype=np.uint8)
>>> r = np.array([10, 25, 80, 50])
>>> c = np.array([10, 60, 40, 10])
>>> x, y = draw.polygon(r, c)
>>> img[x, y] = 1
>>> io.imshow(img)
>>> io.show()
```

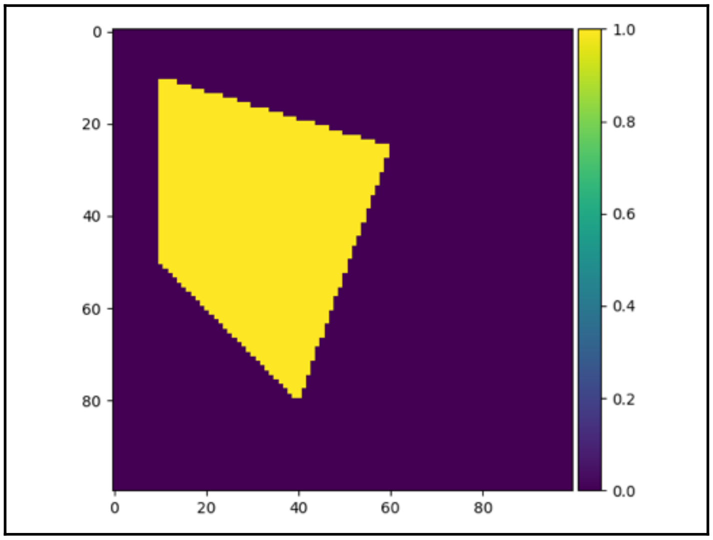


Figure 16: A polygon

A point worth noting in this section is that the (0, 0) point is not at the bottom left of the image but at the top right of the image. This is a standard convention followed in Computer Vision.

Summary

In this chapter, we saw what images are and how are they interpreted by a computer. Then we looked at the basics of image processing and its various applications in medicine, security/surveillance, and social media. Further, two image processing libraries, pillow and scikit-image, were introduced. We saw how we could perform basic operations such as reading/writing an image, converting the image between color spaces, and finally, we ended with how to draw some basic geometrical figures using scikit-image. This chapter forms the foundation of the chapters that follow.

In the next chapter, we will look at some more complex image processing algorithms, such as edge detection, and also some commonly used filters.

2
Filters and Features

After having understood the basics of image processing and their libraries (pillow and skimage), in this chapter, we will extend our understanding by looking at some fundamental concepts such as kernels, convolution, filters, and basic image features. We will learn about the different types of image filters such as Gaussian filter and Sobel. We will also learn about edge detection and Hough transformations that will come in handy later on as essential preprocessing steps in a larger computer vision application. Going forward, the reader is free to use either library (pillow or skimage) of their choice. We provide examples for both of these in this chapter.

Image derivatives

An image derivative is defined as the change in the pixel value of an image. The rate of change of a function is defined as:

$$\boxed{\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}}$$

Using this definition in the context of images, calculate the change in the pixel values of an image and, since pixels are discrete image derivatives, they are defined as *f(x+1) – f(x)/1*. To calculate the derivative at any point, we can use finite difference methods to calculate the derivatives such as forward difference, backward difference, and central difference. Finite difference methods are defined as follows:

- **Forward difference**:

  $f(x+1) - f(x)$

- **Backward difference**:

  f(x) – f(x-1)

- **Central difference**:

  f(x+1) – f(x-1)

Given an image matrix, we can find the derivative using another matrix called mask or kernel. For example, the derivative masks for forward, backward, and central difference are as follows:

[1 -1]

[-1 1]

[1 0 -1]

Another example of a derivative mask is:

$$\boxed{\begin{bmatrix}1 & 0 & -1\\ 1 & 0 & -1\\ 1 & 0 & -1\end{bmatrix}}$$

The preceding matrix is to calculate the derivative of a 2D image matrix in the *x* direction. Similarly, a derivative can also be calculated in the *y* direction.

In order to calculate the derivative at any point in the image, place the derivative mask on the image matrix with the center of the mask at the point on which you want to calculate the derivative. Then, add the product of all the overlapping terms cell by cell. This will give you the derivative of the image at that point. *Figure 1* further illustrates how to calculate the derivative of an image:

$$\underset{\text{Image Matrix}}{\begin{bmatrix}40 & 50 & 60 & 70\\ 40 & 50 & 60 & 70\\ 40 & 50 & 60 & 70\\ 40 & 50 & 60 & 70\end{bmatrix}} \qquad \underset{\text{Derivative Mask}}{\frac{1}{3}\begin{bmatrix}1 & 0 & -1\\ 1 & 0 & -1\\ 1 & 0 & -1\end{bmatrix}} \qquad \underset{\text{Derivative}}{\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 20 & 20 & 20 & 0\\ 0 & 20 & 20 & 20 & 0\\ 0 & 20 & 20 & 20 & 0\\ 0 & 0 & 0 & 0 & 0\end{bmatrix}}$$

Figure 1

The derivative mask is placed on the value 50 at (2,2). The red portion shows the overlapping pixels of the matrix and the derivative mask. After calculation, we get the output seen in green in the derivative image; it is the derivative of the pixel at (2,2) from the original image.

Kernels

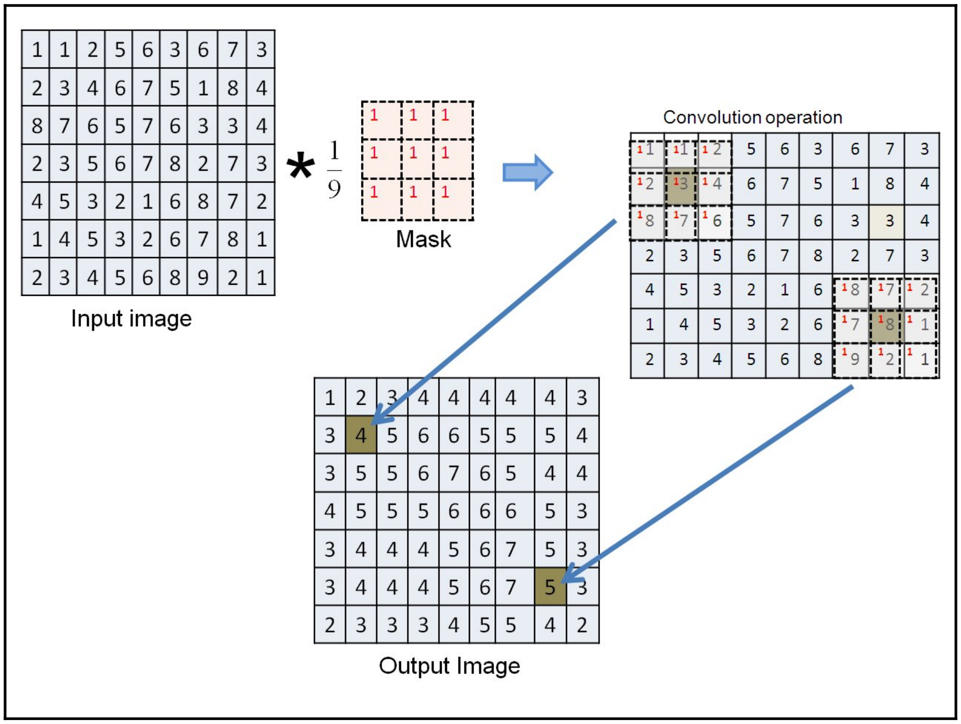
As you saw in the previous section, we used a derivative mask to an calculate image derivative. Before going further into the chapter, let's formally define what these masks are. A lot of times in texts/research papers/books related to image processing, we use the terms mask, kernel, and filter interchangeably. What these essentially mean is a square matrix of numbers that is used to compute various properties or characteristics in an image. You have already seen an example of an image derivative. Some other common examples of such kernels/filters/masks are edge detection, image blurring, and more. As you read through this chapter, you will see various examples of kernels that will help you understand this better.

Convolution

Convolution in the context of image processing is defined as the sum of the product of the corresponding elements of a kernel matrix to an image matrix. Let's try to understand what this means. Given a kernel (matrix), multiply the corresponding elements of the image matrix and kernel matrix, and sum the multiplied values centered around a particular pixel in the image. In a new empty (black) image, set the corresponding pixel from the original image to the sum of multiplied values. Now, perform this operation for all the pixels in the original image. This is **image convolution**!

There are slight variations to the image convolution depending on different applications. Sometimes the kernel matrix is X, Y flipped before it is multiplied with the original image. X flip means to flip the order of the rows in the kernel. For example, the last row of the kernel will become the first row of the flipped matrix and the first row will become the last row, and similarly for all the other rows. Similarly, for Y flip, we flip the columns instead of the rows. The new kernel matrix after the X and Y flip is used for performing convolutions. In most cases, the kernel matrix is symmetric; therefore, there is no need to perform the X or Y flip; we directly multiply the corresponding elements and add them.

The following figure gives us an example of an image convolution:



1	1	2	5	6	3	6	7	3
2	3	4	6	7	5	1	8	4
8	7	6	5	7	6	3	3	4
2	3	5	6	7	8	2	7	3
4	5	3	2	1	6	8	7	2
1	4	5	3	2	6	7	8	1
2	3	4	5	6	8	9	2	1

1	1	1
1	1	1
1	1	1

1	2	3	4	4	4	4	4	3
3	4	5	6	6	5	5	5	4
3	5	5	6	7	6	5	4	4
4	5	5	5	6	6	6	5	3
3	4	4	4	5	6	7	5	3
3	4	4	4	5	6	7	5	3
2	3	3	3	4	5	5	4	2

Figure 2 Example of convolution given 3*3 mask

More examples of convolution follow:

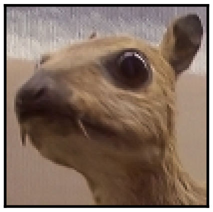
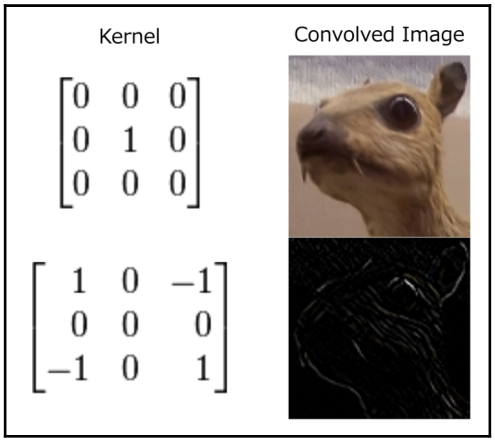


Figure 3: Original image

The following figure shows the examples of the convolution of the image with the given kernels:



$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

Figure 4

Understanding image filters

Enhancing an image by applying some function on the pixel values is called filtering. The process of filtering focuses on the values of the neighborhood of a pixel and uses some to modify the value of the pixel. This is done by convolving the image matrix with a kernel. Therefore, for different filters, you can create different types of kernels. By convolving the image matrix with the kernel, you are basically taking a weighted average of the neighboring values. This method can be used to reduce noise in an image, create effects, and so on. Filtering can also be used in reducing noise in an image as it takes a weighted average and by averaging the noise in a particular pixel also reduces noise. Types of filtering are as follows:

- Gaussian blur
- Median filter
- Dilation and erosion
- Customs filters
- Image thresholding

Gaussian blur

The Gaussian blur is one the most used filters in image processing. It uses the Gaussian distribution bell curve defined by the following function:

$$\boxed{f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/\left(2\sigma^2\right)}}$$

When we use the preceding formula and plot the input and the output values, we get something similar to the image shown in *Figure 5*. What this means is that when we create a kernel that follows a Gaussian distribution, the center pixel gets the most weight and its neighboring pixels get lesser weight when performing convolution. The pixel which has to be modified will have the highest weight in the kernel and the weight decreases for the pixels which are far away.

The following image shows different Gaussian curves for a sigma value equal to 1:

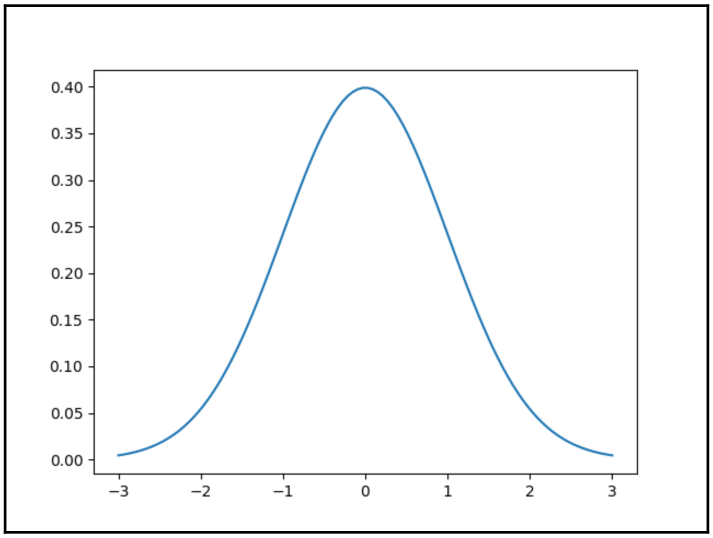


Figure 5

The point to note here is that the Gaussian distribution formula that we saw previously is a continuous function whereas images are discrete. Hence, we discretize the values from the Gaussian distribution before making a kernel matrix out of it.

Let's look at a code `sample`, which applies a Gaussian blur to an image.

- Here is an example using pillow:

```
>>> from PIL import Image
>>> from PIL import ImageFilter
>>> img = Image.open("image.png")
>>> blur_img = img.filter(ImageFilter.GaussianBlur(5))
>>> blur_img.show()
```

The `ImageFilter` library has an inbuilt function for the Gaussian blur. It takes a blur radius as input. The blur radius controls the number of neighboring pixels around the center pixels that are considered while applying the Gaussian blur.

The following is the output of the preceding code:

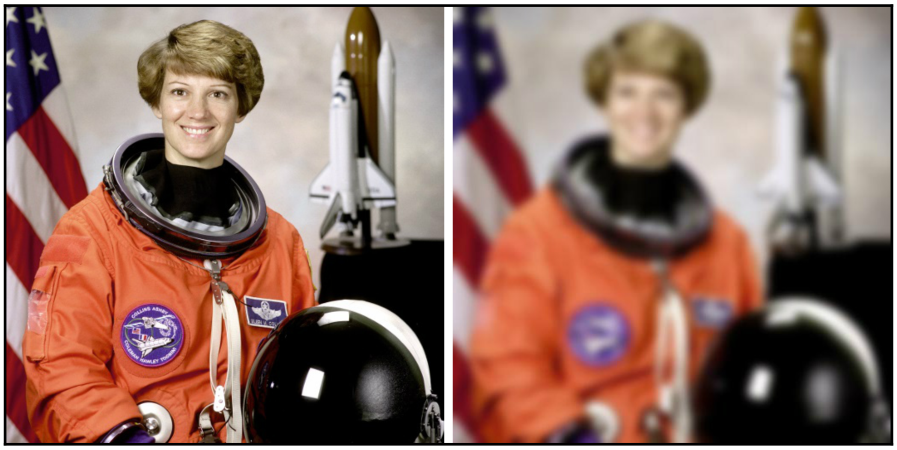


Figure 6: Output of the Gaussian blur using Pillow; the image on the left is the original image and the image on the right is the result of the Gaussian blur

- Here is an example using skimage:

  The filter module of the library provides a Gaussian filter, which takes the image and the standard deviation (sigma) of the Gaussian kernel:

```
>>> from skimage import io
>>> from skimage import filters
>>> img = io.imread("image.png")
>>> out = filters.gaussian(img, sigma=5)
>>> io.imshow(out)
>>> io.show()
```

The following figure is the output of the code with the sigma equal to 5:

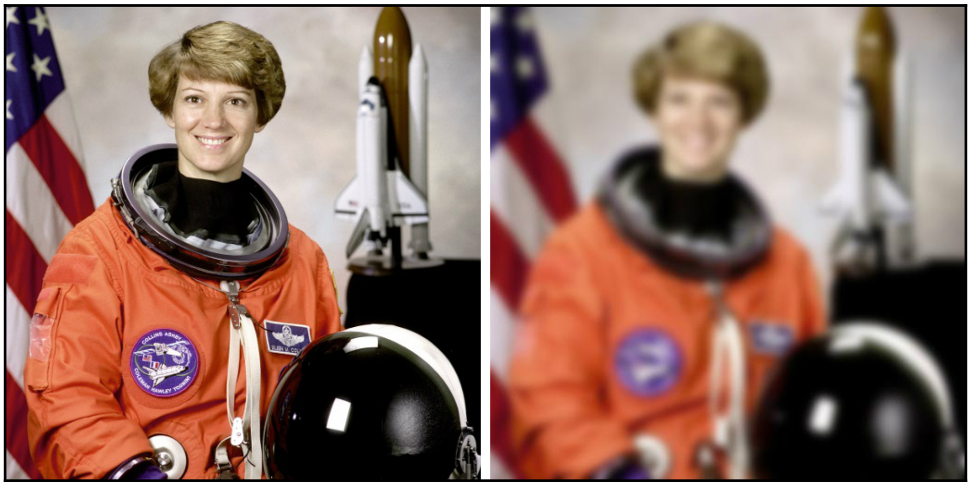


Figure 7: Output of the Gaussian blur using Pillow; the image on the left is the original image and the image on the right is the result of the Gaussian blur

Median filter

This is a very simple filter that returns the median value from the pixel and its neighbors. Both pillow and skimage provide built-in functions for this filter.

The following code sample applies a median filter over an image using pillow:

```
>>> from PIL import Image
>>> from PIL import ImageFilter
>>> img = Image.open("image.png")
>>> blur_img = img.filter(ImageFilter.MedianFilter(7))
>>> blur_img.show()
```

Similarly, using skimage, the following code applies a median filter to a given image:

```
>>> from skimage import io
>>> from skimage import filters
>>> img = io.imread("image.png")
>>> out = filters.median(img, disk(7))
>>> io.imshow(out)
>>> io.show()
```

Dilation and erosion

Morphological operations on images are operations that use the inherent structure or features of an image and processes the image while maintaining the overall structure. The most common examples of morphological operators are erosion and dilation. Let's understand each of them in detail in the following section.

Erosion

Erosion, like in geology, means the removal of the top layer of soil or earth by wind, water, and so on. In image processing too, it means to remove parts of the image. Like the top layer of the soil starts depleting, applying erosion to an image makes the objects in the image to shrink while maintaining the overall structure and shape of the image. But why do we want to shrink objects in an image? Consider a scenario where you have two objects in an image and they are really close by, and you do want your algorithm to assume that they are the same object. Hence you shrink both the objects to mark a clear distinction between the two objects. Another use case of erosion is to remove noise from the image. Erosion might not be the best option to remove all kinds of noise but an upcoming image shows a typical example of noise that can be treated with erosion.

The following figure is an example of a matrix after an erosion operation:

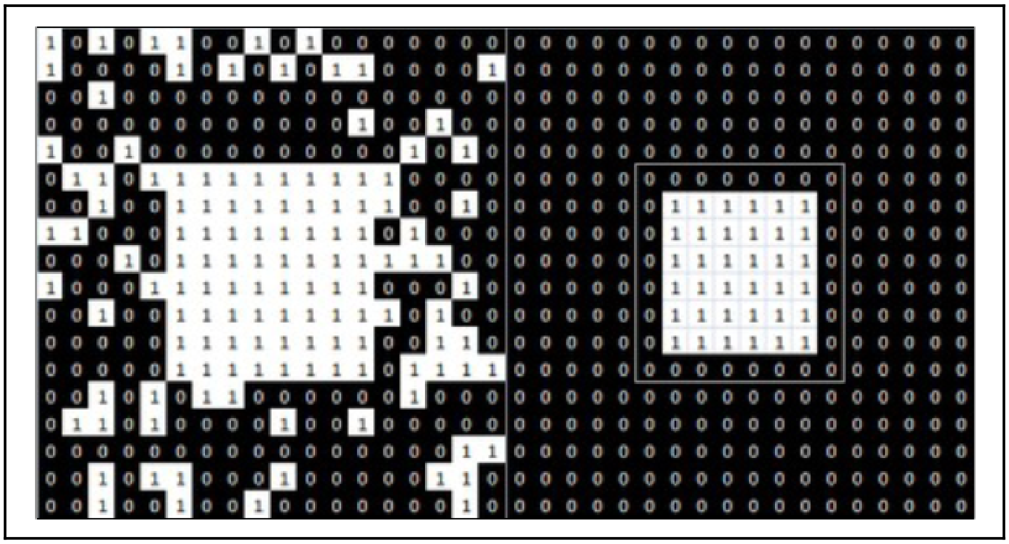


Figure 8: Example of a matrix after an erosion operation; the original image is on the left and the result is on the right

Skimage provides a `binary_erosion()` function for erosion in its morphology module. This function sets the value of the pixel to the min value of its neighboring pixels.

An example usage of the function is as follows:

```
from skimage import morphology
from skimage import io

img = io.imread('image.png')
eroded_img = morphology.binary_erosion(img)

io.imshow(eroded_img)
io.show()
```

The following figure is an example of erosion of an image:

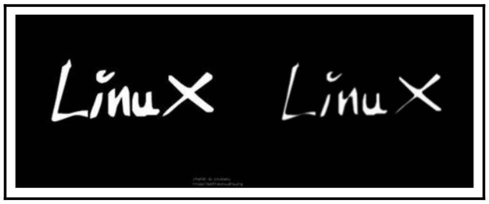


Figure 9: An example of erosion; the original image is on the left and the result is on the right

As you can see in the preceding image, all the letters got shrunk a bit after erosion.

Dilation

Dilation is just the opposite of erosion. While in erosion we shrunk parts of the image, here we try to expand the parts of the image. Dilation finds its use in situations where we want to magnify small details of the image. It is also helpful in situations where you want to fill up unwanted gaps/holes in an image (refer to *Figure 10*). Again, the point to note is that a dilation operation maintains the structure and shape of the original image.

The following figure shows the example of a matrix after a dilation operation:

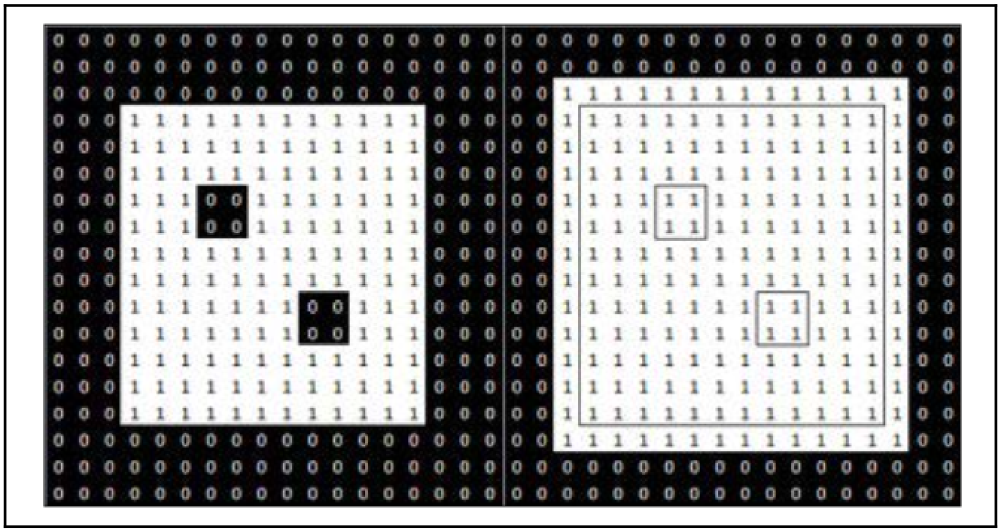


Figure 10: Dilation operation on a given matrix (left)

Skimage provides a `binary_dilation()` function for dilation in its morphology module.

The following code shows how to use this function:

```
from skimage import morphology
from skimage import io

img = io.imread('image.png')
dilated_img = morphology.binary_dilation(img)

io.imshow(dilated_img)
io.show()
```

The following figure is an example of dilation of an image:

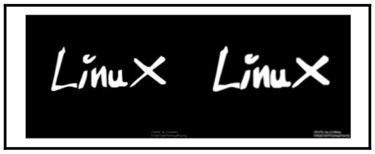


Figure 11: An example of dilation; the original image is on the left and the result is on the right

Custom filters

So far we have seen filters that are used commonly in image processing and are widely accepted by researchers and developers. But there are times when you want to design your own filter. The good news is you do not have to write the entire convolution process again from scratch. Skimage and pillow both provide the option of applying custom filters on images.

The following function can be used to create a kernel for a filter in pillow:

```
>>> from PIL import ImageFilter
>>> kernel = ImageFilter.Kernel((3,3), [1,2,3,4,5,6,7,8,9])
```

The `kernel` function takes the size, the sequence of kernel weights, the scale, and the offset as parameters, where size is the size of the matrix, scale is the value by which the result of the pixel is divided, and offset is the value that is added to result after scaling. The default scale value is the sum of the weights of the kernel. The following code shows how to apply the kernel on an image:

```
>>> from PIL import Image
>>> from PIL import ImageFilter
>>> img = Image.open("image.png")
>>> img = img.convert("L")
>>> new_img = img.filter(ImageFilter.Kernel((3,3),[1,0,-1,5,0,-5,1,0,1]))
>>> new_img.show()
```

The `filter` function takes the kernel as input.

The following figure is the output of the preceding code:

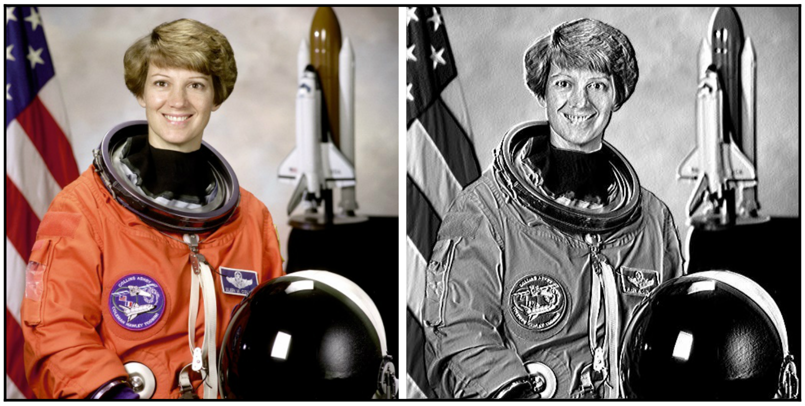


Figure 12: The image on the left is the original image and the image on the right is the result of applying the filter

The same thing can be done using skimage too.

Image thresholding

Thresholding in image processing means to update the color value of a pixel to either white or black according to a threshold value. If the pixel value is greater than the threshold value, then set the pixel to WHITE, otherwise set it to BLACK. There are variations to thresholding as well. One of them is inverse thresholding, where we flip greater than to lesser than and everything else remains the same.

This is one of the simplest ways of filtering images. Using the concept of getting pixel values and setting pixel values from `Chapter 1`, *Introduction to Image processing*, it is fairly simple to write a code for image thresholding. Let's look at how we can implement image thresholding using scikit-image.

The following piece of code implements image thresholding:

```
from skimage.filters import threshold_otsu, threshold_adaptive
from skimage.io import imread, imsave
from skimage.color import rgb2gray

img = imread('image.jpg')
img = rgb2gray(img)

thresh_value = threshold_otsu(img)
thresh_img = img > thresh_value
```

Image thresholding, as explained earlier, works well in simple cases where the background of the image is uniform. But there can be scenarios when the background of the image is not uniform and this happens more often than not. To effectively threshold the image, instead of using a global threshold value, we compute different threshold values for different parts of the image. This is known as adaptive thresholding. As the name suggests, we adapt the threshold value according to different parts of the image. The code for an adaptive threshold is left as an exercise for the reader.

Edge detection

Detecting edges in an image is an important concept and has many applications. Detecting edges can help us know more about the structure and the boundary of the objects in an image. Edges are defined as parts of images where there is significant change in the pixel values while traversing the image. What this means is, say we are scanning an image from left to right, and as we scan the image, we notice that the pixel value of the previous pixel is way less than the pixel value of the current pixel. This implies that the current pixel can be a probable edge pixel. As we saw earlier in the chapter, an image derivative is the way of finding the change in pixel values and hence it could be a naive way of finding edges in an image. Using an image derivative, we can find pixels where there is change in the pixel values among the neighboring pixels and that pixel will probably be a part of an edge. Using only image derivatives is not very robust though. Images with a lot of noise will produce a lot of false edges, which will reduce the overall quality of your vision system.

There are more sophisticated edge-detection techniques such as the Sobel and Canny edge detectors that we will cover in this section, which are more robust and produce fewer false results.

Sobel edge detector

The idea behind the Sobel edge detector is to find the pixels with a large magnitude of gradient values. We are now not only interested in the change, but also the magnitude of change (which is also the gradient). The magnitude of the gradient is calculated by finding the square root of the sum of squares of the image derivative in the *x*-direction and the derivative in the *y*-direction. The equation for the gradient is as follows:

$$\boxed{\nabla f = \frac{\partial f}{\partial x} i + \frac{\partial f}{\partial y} j + \frac{\partial f}{\partial z} k}$$

To determine how large values should be considered, we set a threshold. So after finding the gradient, we take all the gradient values, which exceed that particular threshold.

Why have pixels with large gradient values?

Consider a solid black box (refer to *Figure 13*). All the pixels within the black box have similar pixel values, whereas, the pixel values on the boundary or the edge vary significantly from their neighboring pixels. Therefore, it makes sense to consider pixels with large gradient values:

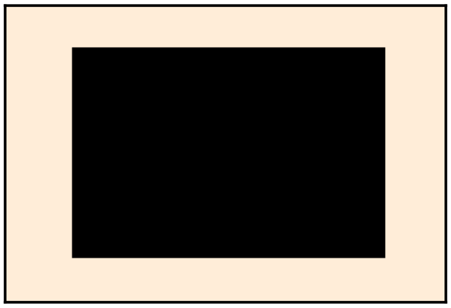
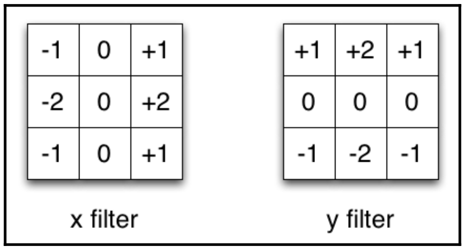


Figure 13: Image to depict why pixels with large values are selected

The kernel used in the Sobel edge detector algorithm is as follows:



-1	0	+1
-2	0	+2
-1	0	+1

x filter

+1	+2	+1
0	0	0
-1	-2	-1

y filter

Here is a code to find edges using the scikit-image:

```
>>> from skimage import io
>>> from skimage import filters
>>> from skimage import color
>>> img = io.imread("image.png")
>>> img = color.rgb2gray(img)
>>> edge = filters.sobel(img)
>>> io.imshow(edge)
>>> io.show()
```

The following figure is the result of the Sobel edge detector:

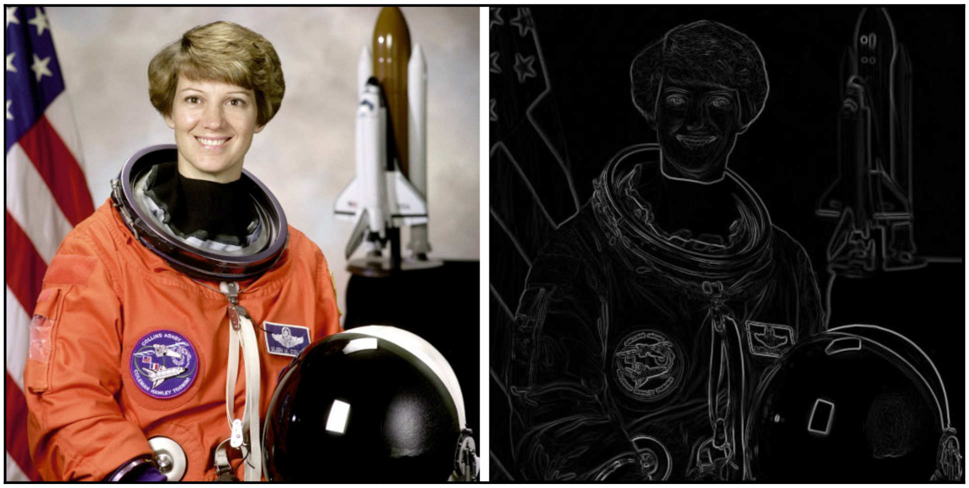


Figure 14: The image on the left is the original image and the image on the right is the output Sobel edge detector

Note that the `Sobel` function in the filter module takes a 2D array as input; therefore, you will have to convert the image into a grayscale image first.

Canny edge detector

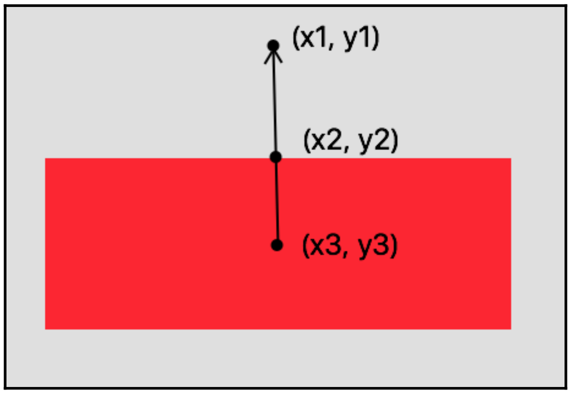
The Canny edge detector is another very important algorithm. It also uses the concept of gradients like in the Sobel edge detector, but in Sobel we only considered the magnitude of the gradient. In this we will also use the direction of the gradient to find the edges.

This algorithm has four major steps:

1. **Smoothing**: In this step, the Gaussian filter is applied to the image to reduce the noise in the image.

2. **Finding the gradient**: After removing the noise, the next step is to find the gradient magnitude and direction by calculating the *x*-derivative and *y*-derivative. The direction is important, as the gradient is always perpendicular to the edge. Therefore, if we know the direction of the gradient, we can find the direction of the edges as well.

3. **Nonmaximal suppression**: In this step, we check whether the gradient calculated is the maximum among the neighboring points lying in the positive and negative direction of the gradient; that is, whether it is the local maxima in the direction of the gradient. If it is not the local maxima, then that point is not part of an edge. In the following figure, point (x_2, y_2) is the local maxima, as this point has the highest change in the pixel values and it is part of the edge whereas the other two points on the line do not have a large change in the pixel values and are not the local maxima:



4. **Thresholding**: In this algorithm, we use two threshold values--the high threshold and low threshold, unlike in Sobel where we just used one threshold value. This is called *hysteresis thresholding*. Let's understand how this works. We select all the edge points, which are above the high threshold and then we see if there are neighbors of these points which are below the high threshold but above the low threshold; then these neighbors will also be part of that edge. But if all the points of an edge are below the high threshold, then these points will not be selected.

So these were the basic steps to detect an edge using the Canny edge detector algorithm. Now we will see how to find edges using this algorithm in Python3.

The scikit-image library provides a function for Canny edge detection in the feature module:

```
>>> from skimage import io
>>> from skimage import feature
>>> from skimage import color
>>> img = io.imread("image.png")
>>> img = color.rgb2gray(img)
>>> edge = feature.canny(img,3)
>>> io.imshow(edge)
>>> io.show()
```

The function takes the image and standard deviation for the Gaussian filter as the input. The following figure shows the output:

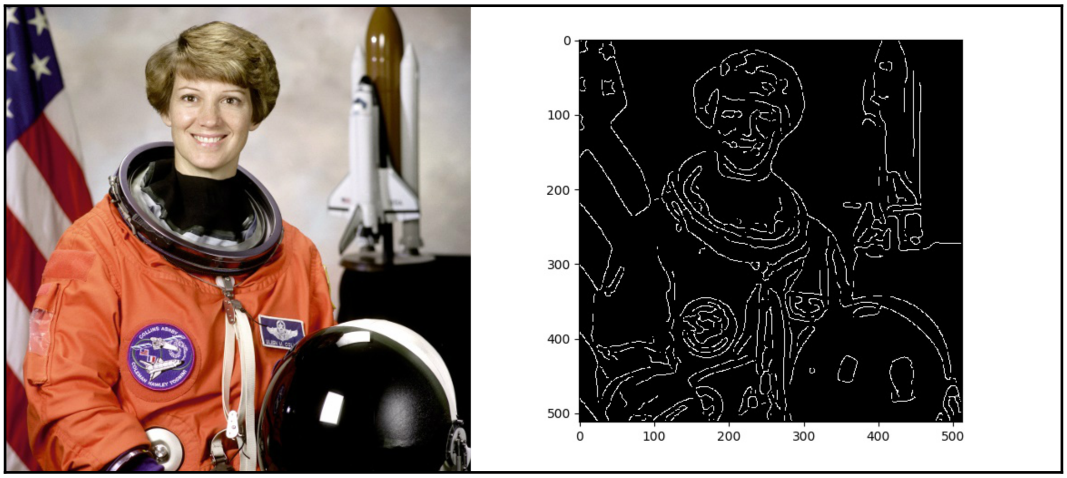


Figure 15: The image on the left is the original image and the image on the right is the output Canny edge detector Hough transformation

Let's recap! So far we have been able to apply filters to find edges. Can we detect certain shapes in an image such as circles, straight lines, or even ellipses? In this section, we will look at how to detect certain rigid shapes in an image using what is known as the *Hough transform*. Hough transformation is a general framework that takes in the parameterized equations of rigid shapes and detects these shapes in an image. For the purpose of this book, we will only look at straight lines and circles, but this technique can be extended to any other shape that you fancy. Let's start with detecting straight lines and then go on to circles.

Hough line

In mathematics, we define a line using two parameters--a slope and a constant (the point of intersection with the y axis). In this algorithm, we exploit the same concept and try to find the slope and constants of the lines (if any) in an image. Given any two points in the image, we substitute in the equation of the line (as shown next) and solve for the slope and intercept of the line. For example, let the two given points be (x_1, y_1), (x_2, y_2).

The equation of a line is defined as follows:

$$\boxed{y = mx + c}$$

At (x_1, y_1):

$$\boxed{y_1 = mx_1 + c}$$

At (x_2, y_2):

$$\boxed{y_2 = mx_2 + c}$$

We will solve these equations to find out (*m*, *c*) and keep a count of the number of points which satisfy this (*m*, *c*). After we have run the algorithm over the entire image, we will know how many points satisfy a given pair of (*m*, *c*) values. Let's call these values a score for a given pair. Using a manually set threshold, we only select the (*m*, *c*) pairs that have a score greater than the threshold. The algorithm returns these pairs and the user can then draw a line using the slope and the constant value. An interesting point to note here is that using just the slope and the constant value, you will never be able to guess the endpoints of the line and you will never be able to accurately draw a line on the image. To solve this problem, you can, along with the score, also maintain the list of points that correspond to a given (*m*, *c*) pair and calculate the endpoints using them.

There is another variation of the Hough line called the probabilistic Hough line. It essentially does the same thing but uses a different approach in calculating the line parameters. It uses more complex mathematics.

The following code shows how to use Hough lines using skimage:

```
from skimage.transform import (hough_line, probabilistic_hough_line)
from skimage.feature import canny

#Read an image
image = io.imread('image.png')

#Apply your favorite edge detection algorithm. We use 'canny' for this
example.
edges = canny(image, 2, 1, 25)

#Once you have the edges, run the hough transform over the image
lines = hough_line(image)
probabilistic_lines = probabilistic_hough_line(edges, threshold=10,
line_length=5, line_gap=3)

#As an exercise you can compare the results of both the methods.
```

Always run Hough transformations on only the edges. (can you think why?)

Hough circle

The Hough circle is similar to the Hough line; only the equation changes here. For the Hough circle, we use the following equation of a circle, where (*h, k*) is the center of the circle and *r* is the radius :

$$\boxed{(x-h)^2+(y-k)^2=r^2}$$

Now, instead of the slope and intercept, the algorithm will find the coordinates of the center of the circle and its radius using the points in the image.

Summary

In this chapter, we first learned about image filters and convolutions. Then we went on to see some examples of filters such as the Gaussian blur, median filter, dilation, and erosion. These are mostly used as preprocessing steps in a larger computer vision system and are seldom used in isolation. Then we looked at more interesting concepts such as edge detection, where we looked at the Canny and Sobel edge detectors. Edge detection also plays a very important role in computer vision and having a thorough understanding of these concepts will help you understand even more complex algorithms that we will see further in the book. Finally we learned how to detect rigid shapes in an image using the Hough transformations.

As we will see in the next chapter, some feature detection algorithms exploit corners in images to extract useful features.

3
Drilling Deeper into Features - Object Detection

A lot of times in a real-world situation, we have to compare two images, or search for an image of an object in a large database of images. An example could be Google Image Search, searching for a person's fingerprint in a defense organization's fingerprint database. Taking a simple difference of the image with all other images from the database will not give a desired solution as there could be distortions or other small variations in the image that will give non-zero output even for matching images. So how do we solve this problem? We need ways in which we are able to describe an image that is independent of the colors, scale, rotation angle, and affine transformations. In the context of computer vision, we essentially want to describe an image in terms of its features. In the last chapter, we looked at some basic features in an image such as gradients and edges but these are not sufficient to describe an image in a unique manner and are very susceptible to changes in brightness, contrast, and so on. Using only edges, we would also not be able to distinguish, between, say a clock and a steering wheel, since both will return a set of points forming a circle. This shows the need for better feature extraction techniques that can help us in such situations. In this chapter, we will expand our knowledge of the features and look at more advanced image features that are able to extract finer details in the image, and are more robust and invariant to scale, color, and rotation. The following is the list of features that we will cover in this chapter:

- Corner detection (Harris corner)
- Cascade classifier—**Local Binary Pattern** (**LBP**)
- Oriented FAST and Rotated BRIEF (ORB)

Before we get into the aforementioned algorithms, let us briefly go through what image features are and why they are important to what we are trying to achieve.

Revisiting image features

Image gradients and edges give us a lot of information about the shapes of different objects in the image. Using edges, we might also be able to determine the orientation of the objects in the image sometimes. But these are weak features and cannot be relied upon all the time. They are very sensitive to variations in brightness, contrast, and backgrounds. We need features that are more stable than just gradients or edges. To solve this problem, we use more sophisticated feature descriptors such as corners, Local Binary Pattern (LBP), BRISK, and Oriented FAST and Rotated BRIEF. Let's understand what is different in these feature descriptors that makes them better than using just edges and image gradients.

To call a feature descriptor a good feature, it should be invariant to changes in scale, rotations, and translations. What this means is that if we are able to describe a car in an image in a particular way, then we should also be able to describe the car in the same way, even when the image is scaled down to half its size (scale invariant) or is rotated by 90 degrees. Having this characteristic in a feature descriptor makes it more robust to variations in the image and handle real-life situations effectively. With advancements in machine learning, researchers and people in the industry now use neural networks to extract features from an image. Results have shown significant improvement over the traditional algorithms that we will see in this chapter. Having said that, it is important to understand the more traditional algorithms to fully understand and appreciate how feature extraction actually works.

We will start off with the most basic feature that can be defined in an image - corners. Simply put corners are nothing but points of intersection of two edges. We aim to find all the corners in an image and use them as a way of describing an image. For example, say we have a picture of a table top. We compute four corner points in the image. We are given another image of a table top and are asked a question—"Is this image also of a table top?" Now what do we know about a table top? We know that it has four corners. So we calculate the number of corners in the new image that we are given. We see that this image also has four corner and we say that the new image is also of a table top. Do you see a problem with this approach? Yes! A lot of objects can have four corners. So maybe corners are not the best way to describe an image but they surely are an integral part of the algorithms that we will study later in this chapter. On the other hand, there are use cases where corners prove very helpful and we will learn about them in the next section.

Harris corner detection

A very crude way to find corners in an image is to first find all the edges in the image and then pairwise check if the edges intersect. This might work well in some cases but will be very inefficient and impractical in real situations. Let us look at a faster corner detection algorithm—Harris corner.

Corners are considered as important points in an image. They are used in many applications such as image correlation, video stabilization, and 3D modeling. Harris corner is one of the most used techniques in corner detection. The Harris corner detector uses a sliding window over the image to calculate the variation in intensity. Since corners will have large variations in the intensity values around them, we are looking for positions in the image where the sliding windows show large variations in intensity in all directions. *Figure 1* illustrates this concept. The Harris corner detector uses mathematical equations to determine which case holds from *Figure 1*:

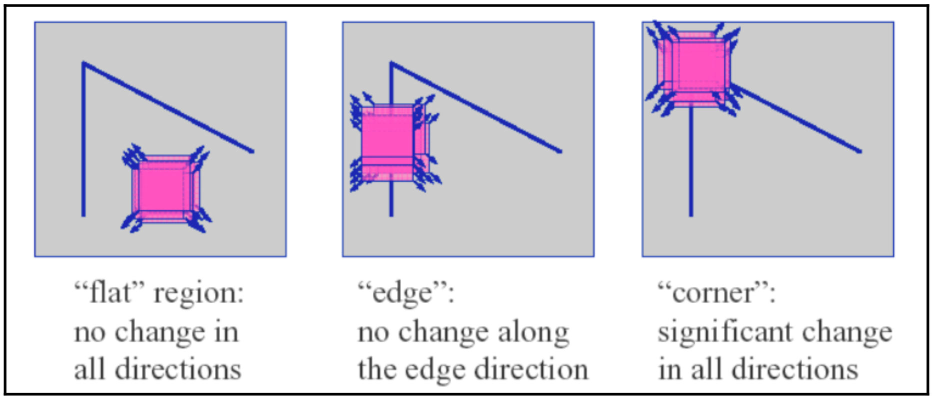


Figure 1: The different cases for the Harris corner detector; a corner is defined as a point where the image changes significantly in all directions

We try to maximize the following value for detecting a corner:

$$\boxed{\sum\left[I(x+u,y+v)-I(x,y)\right]^2}$$

Here, *I* is the image, *u* is the shift in the sliding window in the horizontal direction, and *v* is the shift in the vertical direction.

The following is an implementation of Harris corner using scikit-image:

```
from matplotlib import pyplot as plt
from skimage.io import imread
from skimage.color import rgb2gray
from skimage.feature import corner_harris, corner_subpix, corner_peaks

#Read an image
image = imread('test.png')
image = rgb2gray(image)

#Compute the Harris corners in the image. This returns a corner measure
response for each pixel in the image
corners = corner_harris(image)

#Using the corner response image we calculate the actual corners in the
image
coords = corner_peaks(corners, min_distance=5)

# This function decides if the corner point is an edge point or an isolated
peak
coords_subpix = corner_subpix(image, coords, window_size=13)

fig, ax = plt.subplots()
ax.imshow(image, interpolation='nearest', cmap=plt.cm.gray)
ax.plot(coords[:, 1], coords[:, 0], '.b', markersize=3)
ax.plot(coords_subpix[:, 1], coords_subpix[:, 0], '+r', markersize=15)
ax.axis((0, 350, 350, 0))
plt.show()
```

The high-level organization of the code stays the same as for the previous demonstrations. We first read the image that we want to run the algorithm on. Then, we calculate the Harris corner response using the `corner_harris()` function. The Harris detector algorithm essentially is written inside the `corner_harris()` function and rest are just helper functions that make the output more interpretable to the developer. Let us look at each of them one by one.

- `corner_harris`: This is the primary function which, for each pixel in the image, calculates a measure of how probable it is that the pixel is a corner pixel. The exact mathematics behind The Harris corner detector is beyond the scope of this book but as we saw earlier it tries to find patches in an image which shows significant variation in all directions. And the center pixels of the patches are marked as probable corner pixels.

- `corner_peaks`: Using the output from the `corner_harris()` function, corner_peaks tries to find the actual corner points in the image using the measure provided by the `carner_harris()` function. This function returns the corner pixel coordinates which can be used to plot on the image.
- `corner_subpix`: This is another helper function that helps to fine tune the results that we got from `corner_peaks`. For example, it tries to differentiate if the point is the intersection of two edges or it is really a corner (like of a square). It is not necessary to use this function unless your application really demands this level of classification.

The following is the output of the given code:

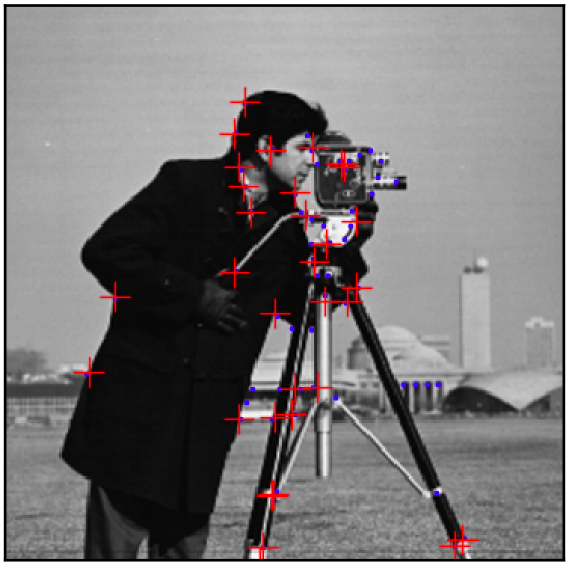


Figure 2: The red cross signs show the corners detected by the Harris corner detector

A point to note here is that Harris corners are rotation and transformation invariant. Even if the corner is moved to a new place in the image, or is rotated by an angle, the algorithm will still be able to detect the corner.

Local Binary Patterns

The **Local Binary Patterns** (**LBP**) cascade is a type of cascade classifier that is used widely in computer vision. But before we go any further into understanding LBP, let us first understand what cascade classifiers in general are. Classifiers are like black boxes where we input an image and the classifier outputs a label for the input image based on some model (this is usually a model trained using a lot of training images). An example of a simple classifier is a digit classifier that we will actually implement in `Chapters 5`, *Integrating Machine Learning with Computer Vision*, and `Chapter 6`, *Image Classification Using Neural Networks*. The word "cascade" means to form a chain of sorts. In the current context, it means to make a chain of classifiers. Output of one classifier is passed onto as input to the next classifier. Two famous examples of cascade classifier are Haar Cascades and Local Binary Patterns (LBP). In this section, we will only look at LBP.

In LBP, an eight-bit binary feature vector is created for each pixel in the image by considering the eight neighboring pixels (top-left, top-right, left, right, bottom-left, and bottom-right). For every neighboring pixel, there is a corresponding bit, which is assigned a value 1 if the pixel value is greater than the center pixel's value, otherwise it is 0. The eight-bit feature vector is treated as a binary number (later convert it to a decimal value) and using the decimal values for each pixel, a 256-bin histogram is computed. This histogram is used as a representation of the image.

LBP features have some primitives coded in them, as shown in the following figure:

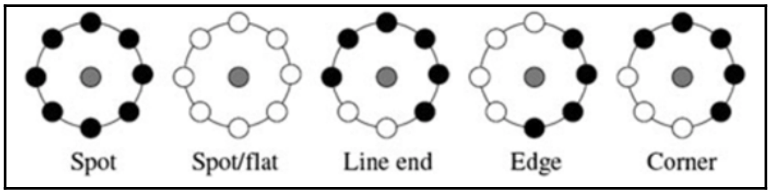


Figure 3: Example of texture primitives

Using the eight-bit binary feature vector that we described in the last paragraph we can identify a set of primitives that are shown in the *Figure 3*. A hollow circle implies that the neighboring pixel was greater than the center pixel and a black filled circle means that the neighboring pixel's value was more than the center pixel.

The following piece of code implements the LBP cascade classifier:

```
from skimage.transform import rotate
from skimage.feature import local_binary_pattern
from skimage import data
from skimage.color import label2rgb
import numpy as np

# Get three different images to test the algorithm with
brick = data.load('brick.png')
grass = data.load('grass.png')
wall = data.load('rough-wall.png')

# Calculate the LBP features for all the three images
brick_lbp = local_binary_pattern(brick, 16, 2, 'uniform')
grass_lbp = local_binary_pattern(grass, 16, 2, 'uniform')
wall_lbp = local_binary_pattern(wall, 16, 2, 'uniform')

# Next we will augment these images by rotating the images by 22 degrees
brick_rot = rotate(brick, angle = 22, resize = False)
grass_rot = rotate(grass, angle = 22, resize = False)
wall_rot = rotate(wall, angle = 22, resize = False)

# Let us calculate the LBP features for all the rotated images
brick_rot_lbp = local_binary_pattern(brick_rot, 16, 2, 'uniform')
grass_rot_lbp = local_binary_pattern(grass_rot, 16, 2, 'uniform')
wall_rot_lbp = local_binary_pattern(wall_rot, 16, 2, 'uniform')

# We will pick any one image say brick image and try to find
# its best match among the rotated images
# Create a list with LBP features of all three images

bins_num = int(brick_lbp.max() + 1)
brick_hist = np.histogram(brick_lbp, normed=True, bins=bins_num, range=(0,
bins_num))

lbp_features = [brick_rot_lbp, grass_rot_lbp, wall_rot_lbp]
min_score = 1000 # Set a very large best score value initially
idx = 0 # To keep track of the winner

for feature in lbp_features:
    histogram, _ = np.histogram(feature, normed=True, bins=bins_num,
range=(0,bins_num))
    p = np.asarray(brick_hist)
    q = np.asarray(histogram)
    filter_idx = np.logical_and(p != 0, q != 0)
    score = np.sum(p[filter_idx] * np.log2(p[filter_idx] / q[filter_idx]))
    if score < min_score:
        min_score = score
        winner = idx
    idx = idx + 1

if idx == 0:
    print('Brick matched with Brick Rotated')
elif idx == 1:
    print('Brick matched with Grass Rotated')
elif idx == 2:
    print('Brick matched with Wall Rotated')
```

Let us break down the preceding code. In the code, we are using Local Binary Patterns to find the best match for a given image. For that we first load three different images provided by Skimage—Brick, Wall, and Grass. After loading the images, we calculate the LBP features for all the three images using the `local_binary_pattern()` function. The parameters to this function are the radius and the number of points we want to consider around any pixel in the image. We set a value of 2 and 16 for the radius and number of points respectively. These are not definitive numbers and can be changed according to the scenario. Once we have the LBP features for all the three images, we will rotate the original images by a random angle (22 degrees in case of the preceding code).

We will now compute the LBP features for the rotated image and use these features to find the best match for one of the original images. For our example, we use the brick image to find its best match from the rotated image. In a `for` loop, we match the brick LBP feature with all the rotated image features one by one. Then we calculate the **Kullback Leibler Divergence** to compute the match score between the LBP features. The following lines of code calculate the score:

```
    p = np.asaray(brick_hist)
    q = np.asarray(histogram)
    filter_idx = np.logical_and(p != 0, q != 0)
    score = np.sum(p[filter_idx] * np.log2(p[filter_idx] / q[filter_idx]))
```

Using this score, we find the best match.

Compared to Haar cascades, LBP cascades deal with integers rather than double values because we just set the value to either 0 or 1. So, both training and testing is faster with LBP cascades and hence is preferred while developing embedded applications. Another important property of LBP is their tolerance against illumination variations.

Oriented FAST and Rotated BRIEF (ORB)

Oriented FAST and Rotated BRIEF (**ORB**) was developed at OpenCV labs by Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary R. Bradski in 2011, as an efficient and viable alternative to SIFT and SURF. ORB was conceived mainly because SIFT and SURF are patented algorithms. ORB, however, is free to use.

ORB performs as well as SIFT on the task of feature detection (and is better than SURF), while being almost two orders of magnitude faster. ORB builds on the well-known FAST keypoint detector and the BRIEF descriptor. Both these techniques are attractive because of their good performance and low cost. ORB's main contributions are as follows:

- The addition of a fast and accurate orientation component to FAST
- The efficient computation of oriented BRIEF features
- Analysis of variance and correlation of oriented BRIEF features
- A learning method for decorrelating BRIEF features under rotational invariance, leading to better performance in nearest-neighbor applications

oFAST – FAST keypoint orientation

FAST is a feature detection algorithm that is widely recognized due to its fast computation properties. It doesn't propose a descriptor to uniquely identify features. Moreover, it does not have any orientation component, so it performs poorly to in-plane rotation and scale changes. We will take a look at how ORB added an orientation component to FAST features.

FAST detector

First, we detect FAST keypoints. FAST takes one parameter from the user, the threshold value between the center pixel and those in a circular ring around it. We use a ring radius of nine pixels as it gives good performance. FAST also produces keypoints that are along edges. To overcome this, we use the Harris corner measure to order the keypoints. If we want *N* keypoints, we keep the threshold low enough to generate more than *N* keypoints, and then pick the topmost *N* based on the Harris corner measure.

FAST does not produce multiscale features. ORB employs a scale pyramid of the image and produces FAST features (altered by Harris) at each level in the pyramid.

Orientation by intensity centroid

To assign orientation to corners, we use the intensity centroid. We assume that the corner is offset from the intensity centroid and this vector is used to assign orientation to a keypoint.

To compute the coordinates of the centroid, we use moments. Moments are calculated as follows:

$$\boxed{m_{pq} = \sum_{x,y} x^p y^p I(x, y)}$$

The coordinates of the centroid can be calculated as follows:

$$\boxed{C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)}$$

We construct a vector OC from the keypoint's center, O, to the centroid, C. The orientation of the patch is obtained as follows:

$$\boxed{\theta = atan2(m_{01}, m_{10})}$$

Here, *atan2* is the quadrant-aware version of arc tan. To improve the rotation in-variance of this measure, we make sure that the moments are computed with x and y remaining within a circular region of radius r. We empirically choose r to be the patch size so that *x* and *y* run from *[−r, r]*. As |C| approaches *0*, the measure becomes unstable; with FAST corners, we have found that this is rarely the case. This method can also work well in images with heavy noise.

rBRIEF – Rotation-aware BRIEF

BRIEF is a feature description algorithm that is also known for its fast speed of computation. However, BRIEF also isn't invariant to rotation. ORB tries to add this functionality, without losing out on the speed aspect of BRIEF. The feature vector obtained by *n* binary tests in BRIEF is as follows:

$$\boxed{f(n) = \sum_{1<i<n} 2^{i-1} \tau(p; x_i, y_i)}$$

$$\boxed{\begin{array}{l} \text{Where } \tau(p;x,y) \text{ is defined as :} \\ \qquad\qquad \tau(p;x,y) = \begin{cases} 1 & : p(x) < p(y) \\ 0 & : p(x) \geq p(y) \end{cases} \\ p(x) \text{ is the intensity value at pixel } x. \end{array}}$$

Steered BRIEF

The matching performance of BRIEF falls off sharply for in-plane rotation of more than a few degrees. ORB proposes a method to steer BRIEF according to the orientation of the keypoints. For any feature set of n binary tests at location (*xi*, *yi*), we need the 2 x n matrix:

$$\boxed{S = \begin{pmatrix} x1, \dots x_n \\ y1, \dots y_n \end{pmatrix}}$$

We use the patch orientation θ and the corresponding rotation matrix R_θ, and construct a steered version S_θ of S:

$$\boxed{S_\theta = R_\theta S}$$

Now, the steered BRIEF operator becomes:

$$\boxed{g_n(p,\theta) = f_n(p) \mid (\mathrm{x}_i, \mathrm{y}_i) \in S_\theta}$$

We discretize the angle to increments of $2\pi/30$ (12 degrees), and construct a lookup table of precomputed BRIEF patterns. As long as the keypoint orientation θ is consistent across views, the correct set of points S_θ will be used to compute its descriptor.

Variance and correlation

One of the properties of BRIEF is that each bit feature has a large variance and a mean near 0.5. A mean of 0.5 gives a maximum sample variance of 0.25 for a bit feature. Steered BRIEF produces a more uniform appearance to binary tests. High variance causes a feature to respond more differently to inputs.

Having uncorrelated features is desirable as in that case, each test has a contribution to the results. We search among all the possible binary tests to find ones that have a high variance (and a mean close to 0.5) as well as being uncorrelated.

ORB specifies the rBRIEF algorithm as follows.

Set up a training set of some 300 keypoints drawn from images in the PASCAL 2006 set. Then, enumerate all the possible binary tests drawn from a 31 x 31 pixel patch. Each test is a pair of 5 x 5 subwindows of the patch. If we note the width of our patch as wp = 31 and the width of the test subwindow as wt = 5, then we have N = (wp – wt)2 possible subwindows. We would like to select pairs of two from these, so we have (N) 2 binary tests. We eliminate tests that overlap, so we end up with N = 205,590 possible tests. The algorithm is as follows:

1. Run each test against all training patches.
2. Order the tests by their distance from a mean of 0.5, forming the vector T.
3. Perform a greedy search:
    1. Put the first test into the result vector R and remove it from T.
    2. Take the next test from T and compare it against all tests in R. If its absolute correlation is greater than a threshold, discard it; otherwise add it to R.
    3. Repeat the previous step until there are 256 tests in R. If there are fewer than 256, raise the threshold and try again.

rBRIEF shows signicant improvement in the variance and correlation over steered BRIEF. ORB outperforms SIFT and SURF on the outdoor dataset. It is about the same on the indoor set; note that blob detection keypoints, such as SIFT, tend to be better on graffiti-type images.

The following code implements ORB using `skimage`:

```
from skimage import data
from skimage import transform as tf
from skimage.feature import (match_descriptors, corner_harris,
 corner_peaks, ORB, plot_matches)
from skimage.color import rgb2gray
```

```
import matplotlib.pyplot as plt

#Read the original image
image_org = data.astronaut()

#Convert the image gray scale
image_org = rgb2gray(image_org)

#We prepare another image by rotating it. Only to demonstrate feature
mathcing
image_rot = tf.rotate(image_org, 180)

#We create another image by applying affine transform on the image
tform = tf.AffineTransform(scale=(1.3, 1.1), rotation=0.5,
 translation=(0, -200))
image_aff = tf.warp(image_org, tform)

#We initialize ORB feature descriptor
descriptor_extractor = ORB(n_keypoints=200)

#We first extract features from the original image
descriptor_extractor.detect_and_extract(image_org)
keypoints_org = descriptor_extractor.keypoints
descriptors_org = descriptor_extractor.descriptors

descriptor_extractor.detect_and_extract(image_rot)
keypoints_rot = descriptor_extractor.keypoints
descriptors_rot = descriptor_extractor.descriptors

descriptor_extractor.detect_and_extract(image_aff)
keypoints_aff = descriptor_extractor.keypoints
descriptors_aff = descriptor_extractor.descriptors

matches_org_rot = match_descriptors(descriptors_org, descriptors_rot,
cross_check=True)
matches_org_aff = match_descriptors(descriptors_org, descriptors_aff,
cross_check=True)

fig, ax = plt.subplots(nrows=2, ncols=1)

plt.gray()

plot_matches(ax[0], image_org, image_rot, keypoints_org, keypoints_rot,
matches_org_rot)
ax[0].axis('off')
ax[0].set_title("Original Image vs. Transformed Image")

plot_matches(ax[1], image_org, image_aff, keypoints_org, keypoints_aff,
matches_org_aff)
ax[1].axis('off')
ax[1].set_title("Original Image vs. Transformed Image")

plt.show()
```

In the preceding code, we match the ORB features extracted from the original image with a rotated version of the same image and a fine-wrapped image. This demonstrates how ORB features are invariant to transformations such as rotation, change in scale, and perspective.

The following is the output of the preceding code:

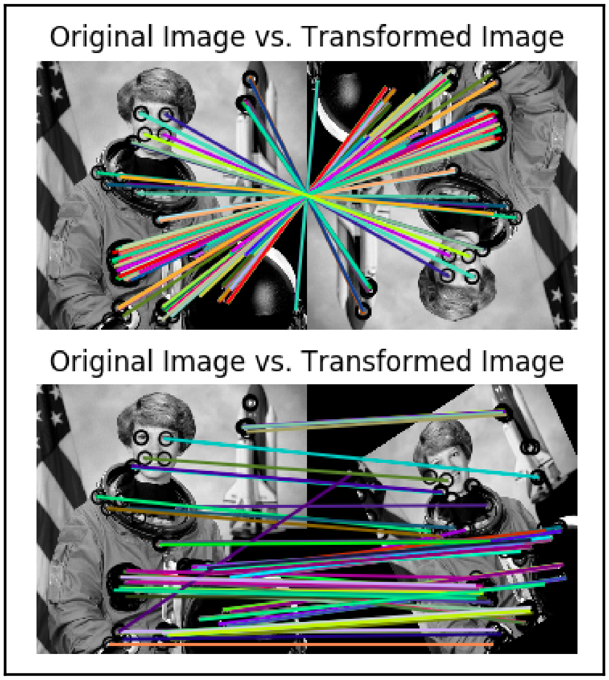

Original Image vs. Transformed Image
Original Image vs. Transformed Image

Figure 4: The top image shows feature matching between the original image and a rotated version of the same image, while the image at the bottom shows feature matching between the original image and an affine transformed version

Image stitching

Most mobile phone cameras today have the feature of capturing a panorama shot. Ever wondered how it works? Panorama images are based on the concept of image stitching, where we capture multiple overlapping images and join them together my matching the common parts of the images. To further illustrate this look at *Figure 5*. Both images shown in the figure have a common region between them. Our task in this section is to stitch the images together to form one bigger image:

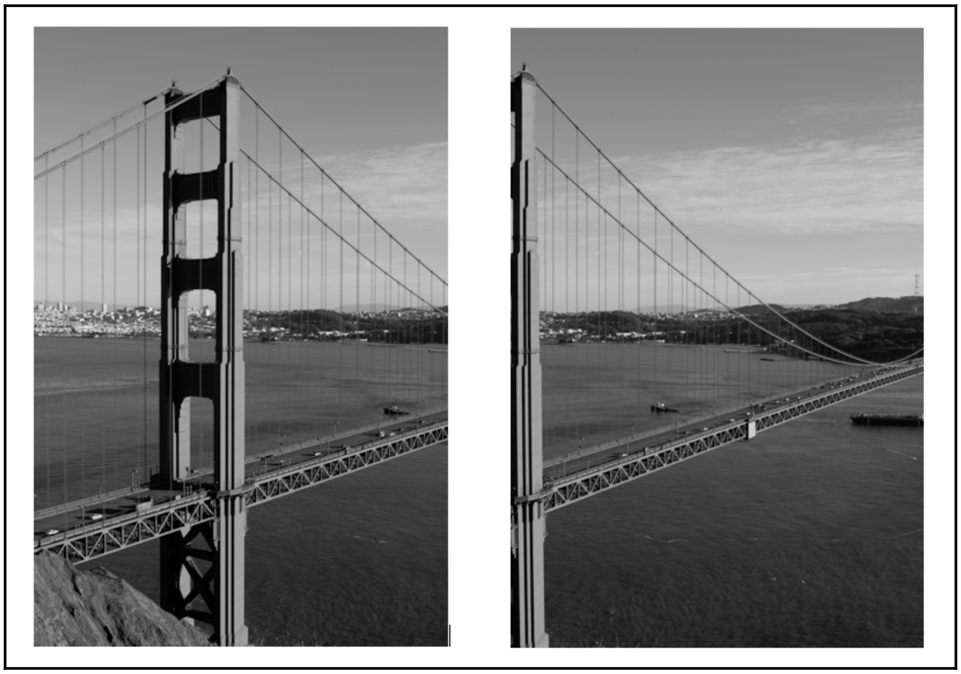


Figure 5: We want to combine these two images to form a bigger image. Notice the common area between the images. Common area will be used to stitch the images.

Image stitching is an interesting application of image feature extraction and matching. Let us use the knowledge we gained in the previous sections and apply it to a real-life situation. To stitch two images together, the first thing is to find common points between the images. As in *Figure 5*, we want to match the tip of the pillar holding the bridge in the two images. Similarly we want to match as many points as possible. Once we have the matching points, we want to align the two images on top of each other using the matched points. In the last section, we saw how ORB was able to extract features in an image and find corresponding features in another image of the same object. We will use the same technique to find these matching points.

The following code given stitches the two images together and outputs a combined image (see *Figure 5*):

```
from skimage.feature import ORB, match_descriptors
from skimage.io import imread
from skimage.measure import ransac
from skimage.transform import ProjectiveTransform
from skimage.color import rgb2gray
from skimage.io import imsave, show
from skimage.color import gray2rgb
from skimage.exposure import rescale_intensity
from skimage.transform import warp
from skimage.transform import SimilarityTransform
import numpy as np


image0 = imread('goldengate1.png')
image0 = rgb2gray(image0)

image1 = imread('goldengate2.png')
image1 = rgb2gray(image1)

orb = ORB(n_keypoints=1000, fast_threshold=0.05)

orb.detect_and_extract(image0)
keypoints1 = orb.keypoints
descriptors1 = orb.descriptors

orb.detect_and_extract(image1)
keypoints2 = orb.keypoints
descriptors2 = orb.descriptors

matches12 = match_descriptors(descriptors1,
 descriptors2,
 cross_check=True)

src = keypoints2[matches12[:, 1]][:, ::-1]
dst = keypoints1[matches12[:, 0]][:, ::-1]

transform_model, inliers = \
 ransac((src, dst), ProjectiveTransform, min_samples=4,
residual_threshold=2)

r, c = image1.shape[:2]

corners = np.array([[0, 0],
 [0, r],
 [c, 0],
 [c, r]])

warped_corners = transform_model(corners)

all_corners = np.vstack((warped_corners, corners))

corner_min = np.min(all_corners, axis=0)
corner_max = np.max(all_corners, axis=0)

output_shape = (corner_max - corner_min)
output_shape = np.ceil(output_shape[::-1])

offset = SimilarityTransform(translation=-corner_min)

image0_warp = warp(image0, offset.inverse, output_shape=output_shape,
cval=-1)

image1_warp = warp(image1, (model_robust + offset).inverse,
output_shape=output_shape, cval=-1)

image0_mask = (image0_warp != -1)
image0_warp[~image0_mask] = 0
image0_alpha = np.dstack((gray2rgb(image0_warp), image0_mask))

image1_mask = (image1_warp != -1)
image1_warp[~image1_mask] = 0
image1_alpha = np.dstack((gray2rgb(image1_warp), image1_mask))

merged = (image0_alpha + image1_alpha)
alpha = merged[..., 3].
merged /= np.maximum(alpha, 1)[..., np.newaxis]

imsave('output.jpg', merged)
```

In the preceding code, we read the two images shown in *Figure 5* into `image0` and `image1` variables. After that, the first thing we do is to find ORB features in the two images (see the last section for details on the ORB algorithm). Once we have computed the ORB features for the two images, we match the features from the two images. The matching features are stored in the `matches12` variable. In the next two lines, we extract the matching features from the two images and store them in `src` and `dst` variables.

```
    src = keypoints2[matches12[:, 1]][:, ::-1]
    dst = keypoints1[matches12[:, 0]][:, ::-1]
```

The next task is to find a projection model that will calculate the project of the destination image over the source image. What this means is that we want to know how the destination image align with the source image. It could happen that the destination image is rotated by 30 degrees. Using the projection transformation, we will know that we have to rotate the destination image by 30 degrees in the opposite direction to align it with the source image. Now we compute the size of the final image after stitching the two images by estimating the size of the overlapping part of the images. For that we take the difference between the two farthest matching corner points. The difference of the corner points is stored in out_shape. Further, we warp the images according to the projection transformation that we computed earlier. Warping an image means to distort the image in a particular way. For example, in our case we will warp the image by shrinking the matching side of the two images (you will notice this in the final result—*Figure 6*). Finally, we add the two warped images together after adding the alpha channels to the images. Alpha channels are added to make the two images blend together properly. The final output of the preceding code is shown in *Figure 6*:

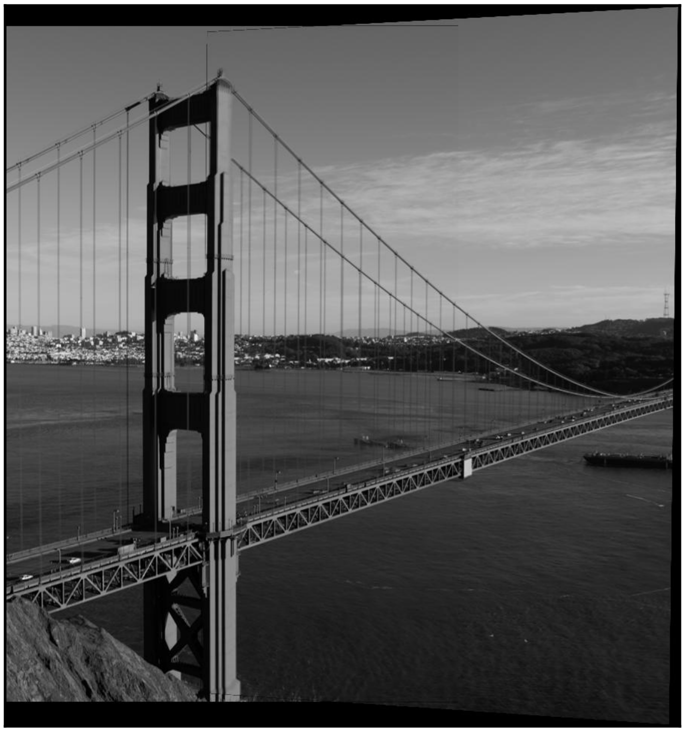


Figure 6: The stitched version of the two images shown in Figure 5

For this example, we used only two images, but this technique can be extended to stitch as many overlapping images we want. The technique will remain the same. This is also how your mobile phone camera captures a panorama shot.

Summary

In this chapter, we looked at different feature detection algorithms such as Harris Corner Detection, Local Binary Classifiers, and ORB. These algorithms make it possible to perform image matching in a real-world scenario. Algorithms such as LBP, ORB compute features in an image that are insensitive to rotation, translation, and other minor distortions. Detecting corners in an image is helpful in applications such as image stitching where we want to correlate two images by finding the same points in different images. In the next chapter, we will look at image segmentation and its applications.

4
Segmentation - Understanding Images Better

In the previous chapter, we looked at techniques to find image features that help us describe various objects in the images. We also looked at Haar Cascades that helped us to detect faces in images. An interesting thing about these algorithms was their local nature. All algorithms look at points in their neighborhood to mark a certain pixel as a feature pixel. In this chapter, we will take things a bit further and try to analyze a given picture from a different perspective. We will look at techniques that will help us look at larger regions of the image and draw meaningful conclusions from that. This will help us segment images into regions such as background, foreground, water region, and grass region.

The following is the list of algorithms and techniques that we will look at in this chapter:

- Contour detection
- Watershed algorithm
- Superpixels
- Graph cut

Introduction to segmentation

What exactly do we mean by segmentation? Segmenting an image is the process of breaking down an image into smaller regions that individually hold meaningful information and help us understand the overall content of the image. For example, let's take a look at the following images (*Figure 1*). On the left you see the original image and on the right there is the corresponding segmented image. As we can see, the algorithm was successful in grouping together similar parts of the image. Like the entire background with bushes was grouped to dark green color. The yellow grass and the grass in front of the animal was colored as one. Throughout the chapter, we will look at different techniques that will help us achieve similar results. Not all techniques will work in all situations, so it is helpful to know a couple of techniques:

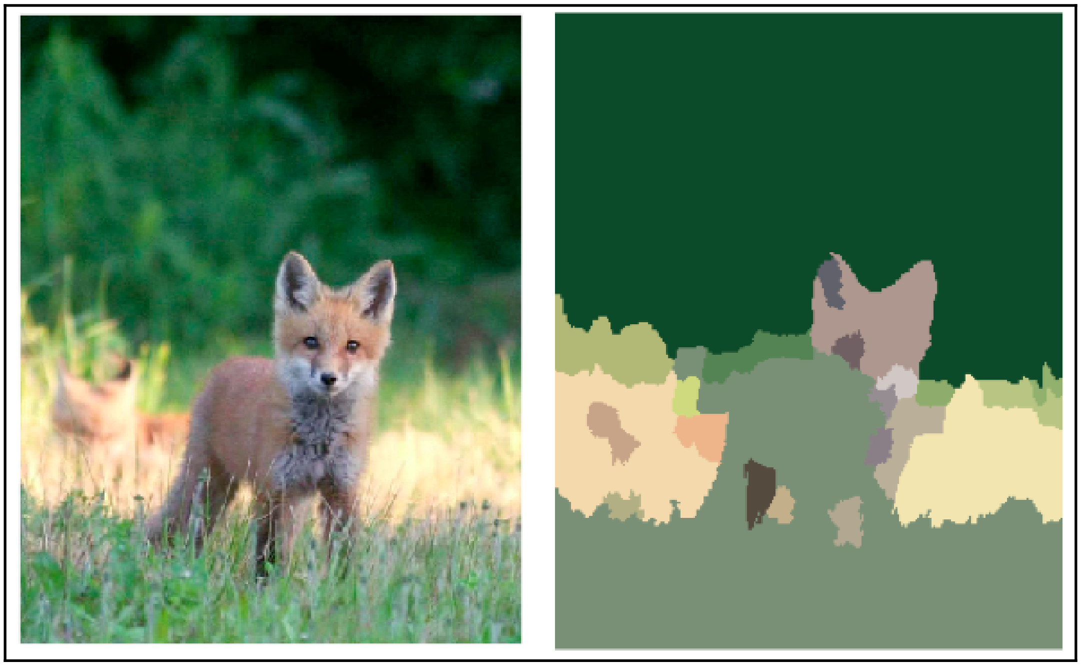


Figure1: Left and right images show the original and the segmented image respectively

Before we look into each algorithm specifically, a nice observation is how all of these algorithms internally use clustering of pixels based on their color values. The only difference between these algorithms is the parameters that are used for clustering. Some use just the Euclidean distance and some use more complicated formulas that we will see in the coming sections.

Contour detection

Let's start with one of the easiest techniques of segmentation—contours. Simply put, contours are nothing but boundaries of objects in an image. Say, for example, you have different types of bottles in an image and you want to segment out each one of them. The contour detection algorithm will try to trace out the boundaries for each bottle and form a closed loop. Each closed loop in the image represents a contour. You might wonder, aren't contours similar to edges? There is a very subtle difference between them—contours always form closed loops whereas edges can remain open. A contour detection algorithm will try to group edges together that will result in a closed loop.

The following is the code used to extract contours in an image. For better results, we first convert the image to grayscale and run a sobel edge detection over it:

```
from skimage import measure
from skimage.io import imread
from skimage.color import rgb2gray
from skimage.filters import sobel
import matplotlib.pyplot as plt

#Read an image
img = imread('contours.png')

#Convert the image to grayscale
img_gray = rgb2gray(img)

#Find edges in the image
img_edges = sobel(img_gray)

#Find contours in the image
contours = measure.find_contours(img_edges, 0.2)

# Display the image and plot all contours found

fig, ax = plt.subplots()
ax.imshow(img_edges, interpolation='nearest', cmap=plt.cm.gray)

for n, contour in enumerate(contours):
    ax.plot(contour[:, 1], contour[:, 0], linewidth=2)

ax.axis('image')
ax.set_xticks([])
ax.set_yticks([])

plt.show()
```

An example output is shown next:

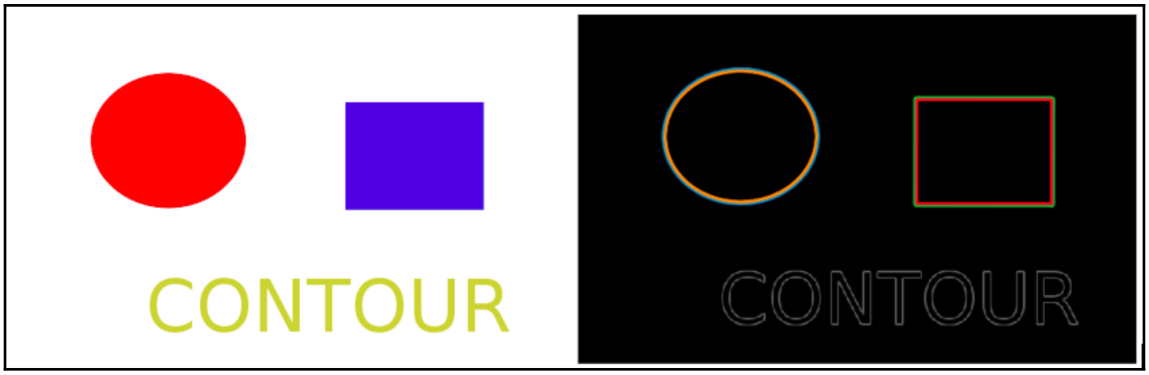


Figure 2: This shows the output (right) of running contour detection over an image (left)

As we can see, the algorithm was able to detect the circle and the square perfectly. The two-color boundary that you see in the output consists of the inner and the outer edges for each shape.

As the image gets more complicated, the results are not that good. You can try an experiment with Lenna's image and see the results yourself. So how do we tackle more complicated images? Simple! By using more sophisticated algorithms. Let's look at the Watershed algorithm next.

The Watershed algorithm

This is an interesting algorithm in the sense that it draws analogy from the physical world. The most common way in which this algorithm is explained in research papers and other texts is in comparison to geographical reliefs. Imagine an area which has a lot of craters (the surface of the moon, for example) and we want to fill each of these craters with water of different colors. We initially start with marking the center of each crater and then keep filling the crater with colored water until the water level reaches to a point where it just touches the boundary of an adjacent crater (assume all craters are close by). After we have filled all craters with colored water, we have successfully segmented out each crater on our surface. Simple enough!

Now, let's try to imagine this in the world of images. To begin with, imagine that your image is the surface that we are trying to segment. All the objects, background, and foreground in the image are craters. As we know, our next task is to identify the center of each of these craters (essentially objects, background, and foreground). Here is the trick—in images, it is not so easy to identify the center of the objects because if we knew what our object was, I probably would not have written this chapter! So how do we solve this problem? We find out the local gradients of the image. By doing so, we will identify all the local minimums in our images. These local minimums will give us an approximate idea of where the objects could possibly be located. In technical terms, these local minimums are called markers. We assign each marker with a unique color and then start filling these colors until we reach the boundary of an adjacent marker.

While doing so we were essentially filling out the objects/regions in the image with a unique color. Let's look at the output of the algorithm to have a better understanding:

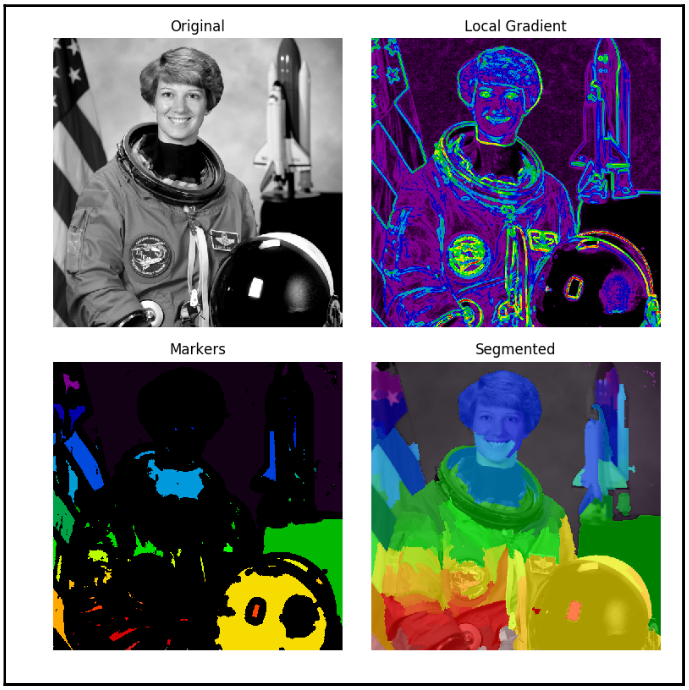


Figure 3: A step-by-step (left to right) illustration of the Watershed algorithm

A step-by-step explanation of the algorithm is as follows:

1. Read the image that you want to segment.
2. Convert it into grayscale (only if it is not in grayscale already).
3. Convert the image pixel values to unsigned int using the `img_as_ubyte()` function. This is because the gradient function expects the image in a certain format.
4. Calculate the local gradients of the image.
5. Apply the Watershed algorithm.

The following is the code for the Watershed algorithm:

```
from scipy import ndimage as ndi
from skimage.morphology import watershed, disk
from skimage import data
from skimage.io import imread
from skimage.filters import rank
from skimage.color import rgb2gray
from skimage.util import img_as_ubyte

img = data.astronaut()
img_gray = rgb2gray(img)

image = img_as_ubyte(img_gray)

#Calculate the local gradients of the image
#and only select the points that have a
#gradient value of less than 20
markers = rank.gradient(image, disk(5)) < 20
markers = ndi.label(markers)[0]

gradient = rank.gradient(image, disk(2))

#Watershed Algorithm
labels = watershed(gradient, markers)
```

The output label contains the pixel-wise label of which object that pixel belongs to.

The watershed algorithm was an improvement over the contour detection algorithm that we saw in the last section. But this is not it. We can further improve the results of segmentation by using k-means clustering and finally using a graph cut over the clusters.

Superpixels

Images are always dealt with by the granularity of a pixel. But this can sometimes be computationally expensive. You do not always want to iterate through all the pixels in the image. As an attempt to remove redundancy in the pixels of an image, we try to combine pixels closer to each other that have the same color value into a cluster and then call those clusters superpixels. The advantage of doing this is that now instead of going through a few pixels we just go through one superpixel, which is nothing but a combination of these pixels.

The following code computes the superpixels in an image:

```
from skimage import segmentation, color
from skimage.io import imread
from skimage.future import graph
from matplotlib import pyplot as plt
img = imread('test.jpeg')

img_segments = segmentation.slic(img, compactness=20, n_segments=500)
superpixels = color.label2rgb(img_segments, img, kind='avg')
```

The following is the output of the preceding code:

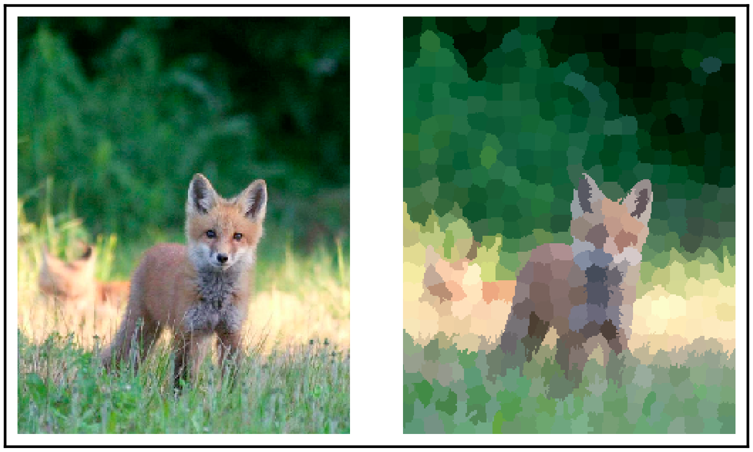


Figure 4: The original image on the left and the superpixels image on the right

As we can see in the output, pixels that were of a similar color and close to each other were clustered together into one blob of mean color. Each of these blobs are called superpixels. Superpixels are used as the starting point for many image segmentation algorithms as they increase the efficiency of the algorithms. An example is a graph cut technique that we will look at in the next section. It uses superpixels to form a graph.

Normalized graph cut

This is one of the most popular image segmentation techniques today. The simplest explanation of the graph cut technique is that each pixel in the image is treated as a node. Apart from these nodes, we have some extra nodes that each represent, say an object in the image. All the pixels are connected to all of its adjacent pixels and each to the object nodes. The following diagram will make the explanation more clear:

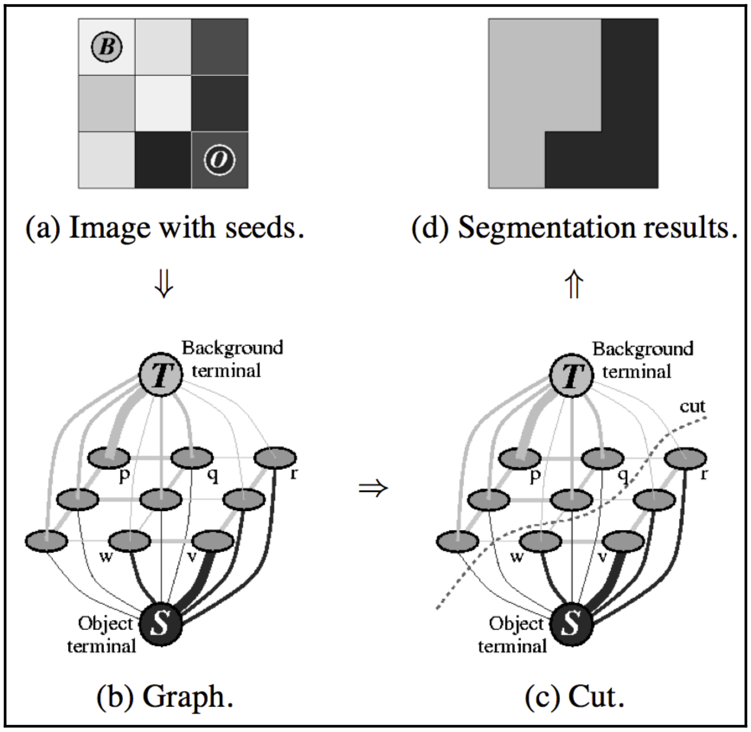


Figure 5: How the graph cut algorithm works--an image graph is created and using it the seed cuts are made in the graph. As a result, we get a well-segmented image

After we have defined our graph, we iteratively start cutting the edges in the graph to obtain subgraphs. After a certain time, we will reach a point where we cannot further cut the graph into subgraphs and that is when we say that we have completed segmenting our image. The result of this would be that each pixel in the image will be connected to one object (defined earlier). This way we can label each pixel in the image to which object it belongs. The interesting part of the algorithm is deciding what edges should be cut in the process. There are many different ways of cutting the edges and each works well for certain types of images. For the purpose of this book, we will look at normalized cuts.

The normalized cuts technique was published in the paper: Shi, J.; Malik, J.,"Normalized cuts and image segmentation", Pattern Analysis and Machine Intelligence, IEEE Transactions, vol. 22, no. 8, pp. 888-905, August 2000

Understanding the exact algorithm is beyond the scope of this book, but if you are curious, you should read the aforementioned paper.

The following is a step-by-step explanation of the implementation using scikit-image:

1. Read the image.
2. Perform k-means clustering over color values. In our implementation, we use the SLIC method for clustering.
3. Using the clustered pixels from the previous step, we create a weighted graph over these clusters. The weight of each edge is determined by how similar two regions are.
4. We apply the normalized graph cut technique over the graph obtained in the last step.

Although the algorithm seems quite easy, if we try to implement it from scratch the mathematics behind it can get really complicated. But we need not worry. Scikit-image provides out-of-the-box functions for each of these steps. Phew!

The following is the code for this algorithm:

```
from skimage import data, segmentation, color
from skimage.io import imread
from skimage import data
from skimage.future import graph

img = data.astronaut()
img_segments = segmentation.slic(img, compactness=30, n_segments=200)
out1 = color.label2rgb(img_segments, img, kind='avg')

segment_graph = graph.rag_mean_color(img, img_segments, mode='similarity')
img_cuts = graph.cut_normalized(img_segments, segment_graph)
normalized_cut_segments = color.label2rgb(img_cuts, img, kind='avg')
```

The preceding code is pretty straightforward. Let's look at two functions `graph.rag_mean_color` and `graph.cut_normalized` in detail.

`graph.rag_mean_color` is used to generate the graph of clusters that we got by using the k-means clustering (see step 2). This function takes in the clusters and a mode as input to generate the graph. The mode can have two values: distance and similarity. Distance is nothing but the Euclidian distance between the mean color of the two clusters. It calculates the weight using the following formula:

$$\boxed{\text{edge weight} = e^{-d^2/sigma}}$$

Here, d = | c_1 - c_2 |
where sigma is a constant provided by the user, and c_1 and c_2 are the mean color values of the two clusters.

`graph.cut_normalized` is the implementation of the research paper mentioned earlier. The exact documentation of the parameters can be found at `http://scikit-image.org/docs/dev/api/skimage.future.graph.html#skimage.future.graph.cut_normalized.`

A sample output of the code is as follows:

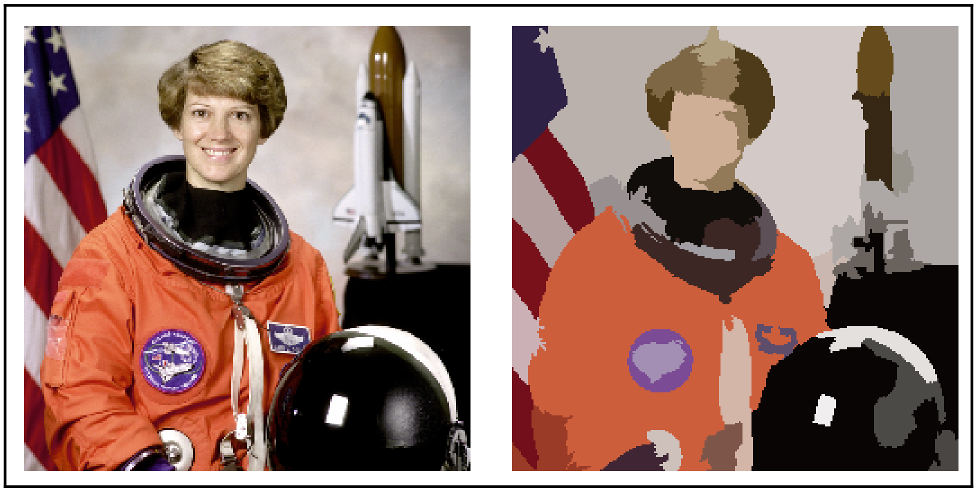


Figure 6: To the left is the original image and to the right is the segmented image

As you can see this technique outperforms all other techniques that we have discussed so far. Let's try to look closely at the output. We can observe that a part of the hat is same in color with the background wall. The algorithm beautifully combines them as one. Also, if we look at the face, as we go down from the forehead of the person towards the nose, we can see a gradual change in the color. The algorithm was able to pick up this change. We can play around with the parameters however we want and get a result that suits best to our need.
Almost all computer vision algorithms that we have seen so far are driven by manual tuning of certain parameters. This is often a bottleneck when you are trying to build computer vision applications for the real world, where you have limited knowledge of the kind of image that your system might encounter. It can so happen that the parameters that you have set in your system work exceptionally well for the input image, but don't be surprised if it completely fails. This problem is being handled by researchers using machine learning. In the coming chapters, we will see how machine learning helps us to avoid dealing with manual parameters and makes our computer vision systems more robust and relevant for the real world.

Summary

In this chapter, we looked at different image segmentation algorithms, namely, contour detection, superpixels, watershed, and normalized graph cut. These algorithms are fairly easy to implement and run almost real time. Image segmentation has tremendous use in real-world applications like background subtraction, image understanding, and scene labeling. Recent advances in machine learning, especially deep learning, have enabled more sophisticated ways of image segmentation that involve almost no manual tuning of parameters.

In the coming chapters, we will look at some of the machine learning techniques and how they are relevant to computer vision.

5
Integrating Machine Learning with Computer Vision

Machine learning is one of the most studied topics in computer science. Every major technology company is spending a sizable amount of their budget on advancing the state of the art for tasks such as image classification, object recognition, and more complex tasks such as activity recognition. In this chapter, we lay the foundation of machine learning in the context of computer vision. We will learn about basic algorithms such as linear regression, support vector machines, and decision trees, while keeping in mind the task of image classification (handwriting recognition). In this chapter, we will also look at a new open source library, sklearn.

Let's begin by understanding the various applications of machine learning for computer vision and then look at specific algorithms.

The following topics will be covered in this chapter

- Introduction to machine learning and sklearn
- Applications of machine learning
- Logistic regression
- Support vector machines
- K-means clustering

Introduction to machine learning

As we saw in the previous section, a system is capable of learning handwriting, understanding an image, or even make a car drive on its own. But how does all of this happen? Let's take an example of detecting different shapes such as circles, squares, and rectangles. We begin with collecting a lot of images for these shapes in different colors and sizes. We should try to have as much diversity in our data as possible. Then we pass this data to our program and, using a machine learning algorithm, it learns what different shapes look like by specifically learning about their characteristic properties such as a circle has no corners, while a square and a rectangle both have four corners, but a square has all its sides of equal length. All of this happens within the algorithm and the developer is not required to learn about these specific properties. Once the program has learned about these shapes, we can now give an unknown shape as input to the program and it will give as output the correct name for that shape.

The concept is fairly simple to understand, but certain sophisticated algorithms require rigorous understanding of mathematics to make sense of what they are doing. The task of learning shapes is an example of supervised machine learning where we give the data and its corresponding label as input to the system to learn. It is like showing an image of a circle and then telling the computer that this is a circle (the same as how you learned about shapes back in preschool). Another vertical of machine learning is unsupervised machine learning, where we only provide the data without any labels to the system and it implicitly tries to group data that it feels belongs to the same label/category. In this book, we will only look at algorithms from both verticals.

An important aspect of machine learning is the training data--the data through which we make the system learn. The quality and quantity of the data both matter equally. If there is not enough data, you will never be able to make the system robust. If you do not have quality data, say you have almost similar data without much diversity or variation, the system might just end up memorizing the data and may not be able to infer new/unknown data. There are a lot of openly available datasets for various tasks. Some of the famous ones are CIFAR and IMAGENET for image classification.

Another important aspect of machine learning is data preprocessing. Let's look at it in more detail.

Data preprocessing

Data preprocessing plays an important role in machine learning. Suppose you are given images of any object that you are trying to learn and all the images are aligned vertically. When after the learning phase you pass an image of the same object at an angle, there are chances that program will not be able to detect what the object is. You could also learn the object with a certain size, but while testing you provide an image with the object either smaller or bigger than what we trained it for. These are just some of the many problems that you can face in terms of the training data. A more formal term used to describe such artifacts is called making an object invariant to translation, rotation, and scale. This means no matter where the object is in the image, at what orientation, and how small or big the object is, the program will be able to detect the object correctly. In this section, we will list some of the preprocessing steps that you can take to make the program more robust.

Image translation through random cropping

Image translation means that the X, Y position of the image changes in the input image. For example, we train our program with images of a pen at the center of the image and while testing (giving an unknown image as input), we give an image with pen at one of the corners of the image. As we read earlier, it is possible that the program might not detect the object as a pen. To tackle this problem, we try to move the object to different positions in the image by randomly cropping the image at different positions.

Image rotation and scaling

This is similar to image translation, only this time we rotate a given image with random angles. This helps the program to learn the object at different orientations. We also scale the image and make it smaller or bigger before we pass it to our machine learning program.

These simple techniques are a way of artificially expanding the original datasets and go a long way in making the program more robust and invariant. There are other preprocessing techniques such as color averaging that could be used too.

But before diving into machine learning algorithms, let's install a new library that we will use in this chapter—scikit-learn.

Scikit-learn (sklearn)

Like scikit-image, scikit-learn is another open source library that provides easy-to-use APIs for most of the machine learning algorithms that we are going to use in this chapter.

Installing sklearn is fairly simple. The following steps show how to install it on various platforms:

- **Windows**: Since you have pip installed already (hopefully you installed scikit-image during the first chapter), run the following command in your terminal.

  ```
  pip3 install sklearn
  ```

  To verify you have successfully installed `sklearn`, run the following command in your terminal:

  ```
  python3 –c "import sklearn"
  ```

  If you see no errors, you are good to go.

- **Linux/macOS**: Both Linux and macOS have the same steps to install `sklearn`. Assuming you have pip installed already, run the following command in your Terminal window:

  ```
  pip3 install sklearn
  ```

  To verify you have successfully installed `sklearn`, run the following command in your Terminal:

  ```
  python3 –c "import sklearn"
  ```

If you see no errors, you are good to go.

Applications of machine learning for computer vision

Machine learning is used in situations when we want the computer to operate, infer situations, or make decision without any human intervention. Just like a human being, to make a computer capable of these features, we need to make the computer learn these situations in advance before the computer can understand these situations independently. An example of this could be to recognize faces in an image.

We need to tell the computer in advance what is a face what it looks like, and what are some specific features that can help us detect a face in an image. The following are some more examples:

- **Handwriting recognition**: This is one of the most common applications of machine learning in computer vision. This finds use in number plate detection, building applications that can translate sign boards written in a language you do not know, and more.
- **Image detection and classification**: Here we try to identify where in the image is an object (detection) and what is the object that is being shown in the image (classification). Is it a mobile phone, a car, or a pen. Many online retail e-commerce websites now have a feature where you click an image of an object and it looks for it on their platform. This is nothing but image classification. We will look at this in the next chapter in more detail regarding how these applications are built.
- **Scene labeling**: An interesting application of machine learning in the context of computer vision, is where given an image the computer tries to label the image. For example, a working machine learning system labeled the following image as *a baseball player swinging a bat on a field*:

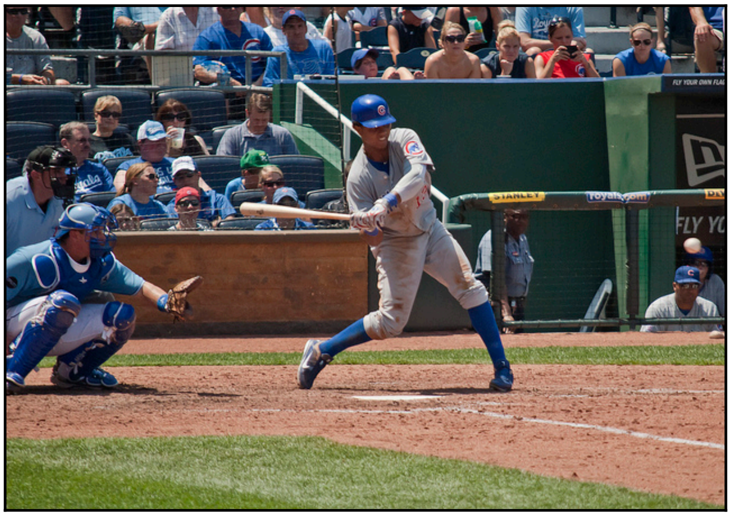


Figure 1: Autogenerated caption using machine learning - "a baseball player swinging a bat on a field" Image source: `http://cs.stanford.edu/people/karpathy/neuraltalk2/demo.html`

- **Self-driving cars**: Using machine learning and computer vision, researchers and technology companies are building autonomous cars that can navigate their way through traffic and obey traffic rules. These cars are fitted with cameras that continuously monitor their surroundings and detect people, cars, and other objects on roads. A lot of different concepts that we have seen so far come together to build a system like this.

Logistic regression

Given a set of data points (in multiple dimensions), logistic regression tries to fit a curve between the data points that best represent it. A more formal definition is; it is a technique that finds relationships between a set of independent variables and a dependent variable—here, the independent variables being the input data and the dependent variable being the labels corresponding to the data. The mathematics behind logistic regression involves probability. Given a data point, we calculate the probability of that data point belonging to a particular label.

Using logistic regression, let's try to build a digit classification program. Given an image of a digit, it will output what digit it is. For this task, we will use the MNIST dataset. MNIST has 60,000 training samples and 10,000 testing samples of digit images each 28 x 28 in size. The official website for the dataset is `http://yann.lecun.com/exdb/mnist/`.

The following is an image showing a sample of the MNIST dataset:

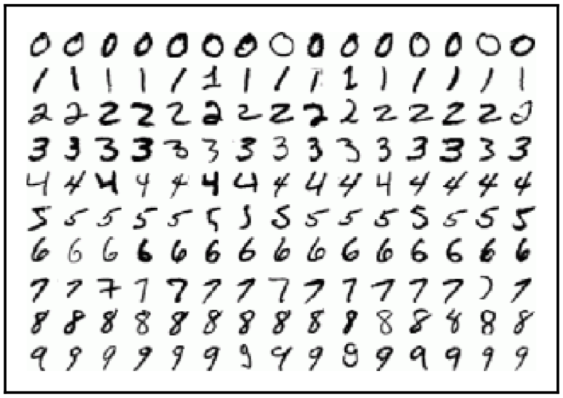


Figure 2: A few examples of digits from the MNIST Dataset

The following is a code for logistic regression using `sklearn`:

```
from sklearn import datasets, metrics
from sklearn.linear_model import LogisticRegression


mnist = datasets.load_digits()

images = mnist.images

data_size = len(images)

#Preprocessing images
images = images.reshape(len(images), -1)
labels = mnist.target

#Initialize Logistic Regression
LR_classifier = LogisticRegression(C=0.01, penalty='l1', tol=0.01)

#Training the data on only 75% of the dataset. Rest of the 25% will be used
in testing the Logistic Regression
LR_classifier.fit(images[:int((data_size / 4) * 3)], labels[:int((data_size
/ 4) * 3)])

#Testing the data
predictions = LR_classifier.predict(images[int((data_size / 4)):])
target = labels[int((data_size/4)):]

#Print the performance report of the Logistic Regression model that we
learnt
print("Performance Report: \n %s \n" %
(metrics.classification_report(target, predictions)))
```

Yes, that's it! All that we read until now about machine learning, we could do that in about 30 lines of code. Let's break down the code.

We first load the dataset using `datasets.load_digits()`. Sklearn has a module called datasets that makes it easy for the developer to use standard datasets such as MNIST without having the problem of downloading the dataset manually and preprocessing it for your use.

After loading the dataset, we transform the images in a way that is compatible to the input of the logistic regression module. The MNIST dataset by default is in a 2D array (like a matrix) but the logistic regression module only accepts arrays in 1D. For that we reshape the array using `images.reshape(len(images), -1)`. Once we have the shape of the input in the correct format, we can pass out digits and labels into the logistic regression module. We define a new object for `LogisticRegression` and set the hyper parameters (tolerance, inverse of regularization strength, and more). It's OK if you do not understand all the hyper parameters. It can be a bit overwhelming to understand all of them. Play around with the values and you will get a feel of what works for your task and what doesn't.

In our code, we divide the given MNIST dataset into 75-25, where we use 75% of the images for training and the remaining 25% for testing. There are no rules to this ratio. You can even use 50-50 for your task. In the preceding code, this line does the actual training for us: `LR_classifier.fit(images[:int((data_size / 4) * 3)], labels[:int((data_size / 4) * 3)])`. After we complete the training phase, we are in a position to test how well the program learned the digits. We test our program by getting the predictions on the remaining 25% images that we had earlier kept aside for testing. This line gets the predictions for us: `predictions = LR_classifier.predict(images[int((data_size / 4)):])`.

After we have the predictions, we get some numbers on our result using `metrics.classification_report(target, predictions)`. Here is the output of running of this program:

```
Performance Report:
                 precision    recall  f1-score   support
         0          0.94        0.99      0.97       131
         1           0.91        0.88      0.90       137
         2          0.98        0.97      0.97       131
         3          0.97        0.87      0.91       136
         4          0.98        0.94      0.96       139
         5          0.94        0.97      0.96       136
         6          0.97        o.97      0.97       138
         7          0.93        0.97      0.95       134
         8          0.86       0.82      0.84       130
         9          0.81        0.90      0.86       136
avg / total        0.93        0.93      0.93      1348
```

We can see for each digit how well the logistic regression worked.

But wait! Where did we use actual images here? Can we not provide our own handwritten image and see what the programs output? Yes, we can certainly do that, and the following code shows how to use a custom image and make the logistic regression predict the digit:

```
from sklearn import datasets, metrics
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from skimage import io, color, feature, transform

mnist = datasets.load_digits()
images = mnist.images
data_size = len(images)

#Preprocessing images
images = images.reshape(len(images), -1)
labels = mnist.target

#Initialize Logistic Regression
LR_classifier = LogisticRegression(C=0.01, penalty='l1', tol=0.01)

#Training the data on only 75% of the dataset. Rest of the 25% will be used
in testing the Logistic Regression
LR_classifier.fit(images[:int((data_size / 4) * 3)], labels[:int((data_size
/ 4) * 3)])

#Load a custom image
digit_img = io.imread('digit.png')

#Convert image to grayscale
digit_img = color.rgb2gray(digit_img)

#Resize the image to 28x28
digit_img = transform.resize(digit_img, (8, 8), mode="wrap")

#Run edge detection on the image
digit_edge = feature.canny(digit_img, sigma=5)

digit_edge = digit_edge.flatten()

#Testing the data
prediction = LR_classifier.predict(digit_edge)

print(prediction)
```

Most of the code that you see here is similar to the code snippet prior to this one. In this code snippet, we add the functionality for the user to load a custom image and pass it through the logistic regression module. There is a bit of preprocessing that needs to be done before you can use the image. We first convert the image into grayscale and resize it to 8 x 8. Then we compute edges using the canny edge detector (you can use whatever edge detection technique you want).

The following diagram was used to test the preceding code:

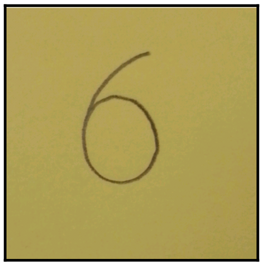


Figure 3: This is a test image of digit 6; it is written with a pencil on a yellow post-it (sticky note)

Support vector machines

Another widely used supervised machine learning algorithm is support vector machines. Like in logistic regression, we tried to fit a curve that passes through the data points, but in SVMs we try to find hyperplanes that divide the given data into regions, with each region representing a particular label.

What are hyperplanes? They are nothing but a generalization of a plane. For example, in one-dimension, it is a point, in two-dimensions, it is a line, in three-dimensions, it is a plane, and for even higher dimensions, we just call them hyperplanes.

The following diagram provides a visual of how linear SVMs work:

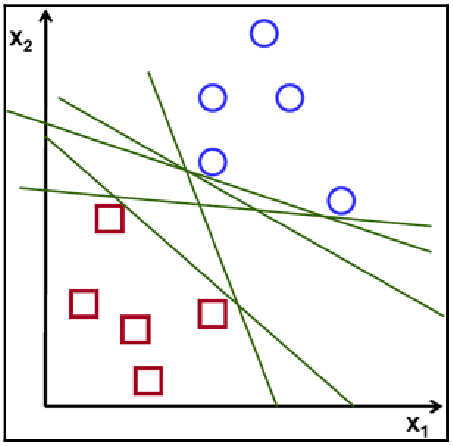


Figure 4: Hyperplanes (green lines) divide the dataset into appropriate regions; these hyperplanes divide the red squares and blue circles into separate regions on the graph

Image source: `http://docs.opencv.org/2.4/_images/separating-lines.png`

Given a set of data points, we find hyperplanes for each of the labels and further optimize them to reduce any generalization errors. If we have n labels in our data, we will try to find n-1 hyperplanes that separate the labels. The mathematics behind support vector machines is beyond the scope of this book, but the ones who are interested can read the following blog post: `https://www.svm-tutorial.com/2017/02/svms-overview-support-vector-machines/`.

Let's apply SVMs for the same task of digit classification. The following code implements classification using SVMs:

```
from sklearn import datasets, metrics, svm

mnist = datasets.load_digits()

images = mnist.images

data_size = len(images)

#Preprocessing images
images = images.reshape(len(images), -1)
labels = mnist.target

#Initialize Support Vector Machine
SVM_classifier = svm.SVC(gamma=0.001)
```

```
#Training the data on only 75% of the dataset. Rest of the 25% will be used
in testing the
Support Vector Machine
SVM_classifier.fit(images[:int((data_size / 4) * 3)],
labels[:int((data_size / 4) * 3)])

#Testing the data
predictions = SVM_classifier.predict(images[int((data_size / 4)):])
target = labels[int((data_size/4)):]

#Print the performance report of the Support Vector Machine model that we
learnt
print("Performance Report: \n %s \n" %
(metrics.classification_report(target, predictions)))
```

As you can see, this is quite similar to the one we wrote for logistic regression. The only major change here is the classifier. We use `svm.SVC` instead. We divide the data into a training set and test set. The result of the preceding code can be seen as follows:

```
Performance Report:
              precision    recall  f1-score   support
          0       1.00      0.99      1.00       131
          1       0.99      1.00      1.00       137
          2       1.00      1.00      1.00       131
          3       0.99      0.95      0.97       136
          4       0.99      0.98      0.99       139
          5       0.98      0.99      0.99       136
          6       0.99      1.00      1.00       138
          7       0.99      1.00      1.00       134
          8       0.96      0.99      0.98       130
          9       0.99      0.99      0.99       136
avg / total       0.99      0.99      0.99      1348
```

We can modify this code to take as input a custom image, as we did earlier for logistic regression. It will be a good exercise to modify the code to make it work for your needs.

Just a side note, support vector machines are known to perform very well with high dimensional data.

You might wonder what happens if the data that is provided to SVMs is not linearly seperable? Can we not use SVMs for such datasets? The answer is we can. There are many variations in SVMs that are capable of finding nonlinear hyperplanes that perform quite well. Just to give you an example, sklearn in its official documentation shows the variations in SVMs. The following screenshot is taken from the sklearn documentation that illustrates different SVMs:

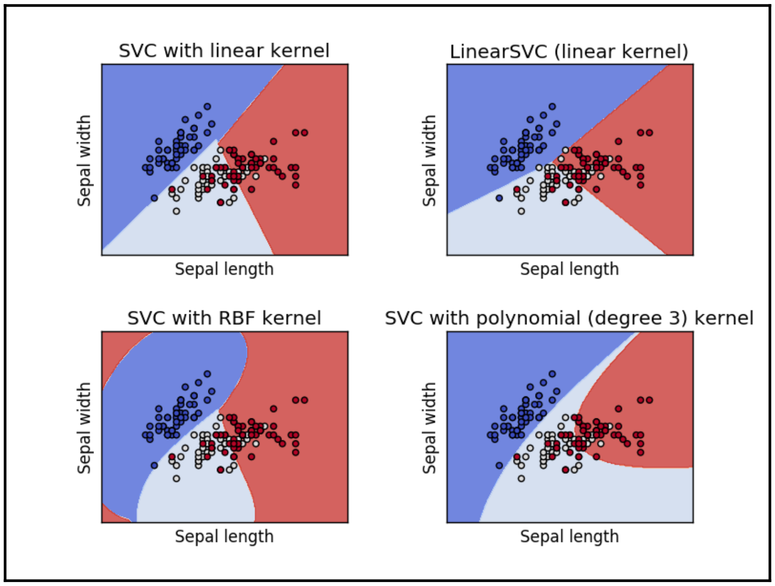


Figure 5: Different versions of SVMs where the data is not really linearly separable

The `svm.SVC` function provides an option to select the kind of hyperplanes that we want to compute. The `kernel` parameter in the function can take values such as `linear`, `poly`, `rbf`, or `sigmoid`. By default, it uses `rbf`. These different kernel choices help the user in finding the most appropriate kernel for the kind of data that they are dealing with. If we know that our data is linearly separable, then we can opt for a `linear` kernel, otherwise use a nonlinear kernel.

Just to aid your understanding of support vector machines, let us visualize the digits data that we used for training our SVM. The images in the MNIST dataset are 28x28, which makes it data 784 dimensions. Does this mean we can never visualize such data? To solve this problem, we use dimensionality reduction. This helps us in converting a 784-D data to 2-D or 3-D which then makes it possible to visualize.

One of the common dimensionality reduction techniques is Principle Component Analysis. The following code will convert the MNIST data into a reduced dimension data:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets, decomposition

digits = datasets.load_digits(n_class=6)
X = digits.data
y = digits.target
n_samples, n_features = X.shape
n_neighbors = 30

X_pca = decomposition.TruncatedSVD(n_components=2).fit_transform(X)

fig, plot = plt.subplots()
plot.scatter(X_pca[:, 0], X_pca[:, 1])
plot.set_xticks(())
plot.set_yticks(())
```

The output of the code is as follows:

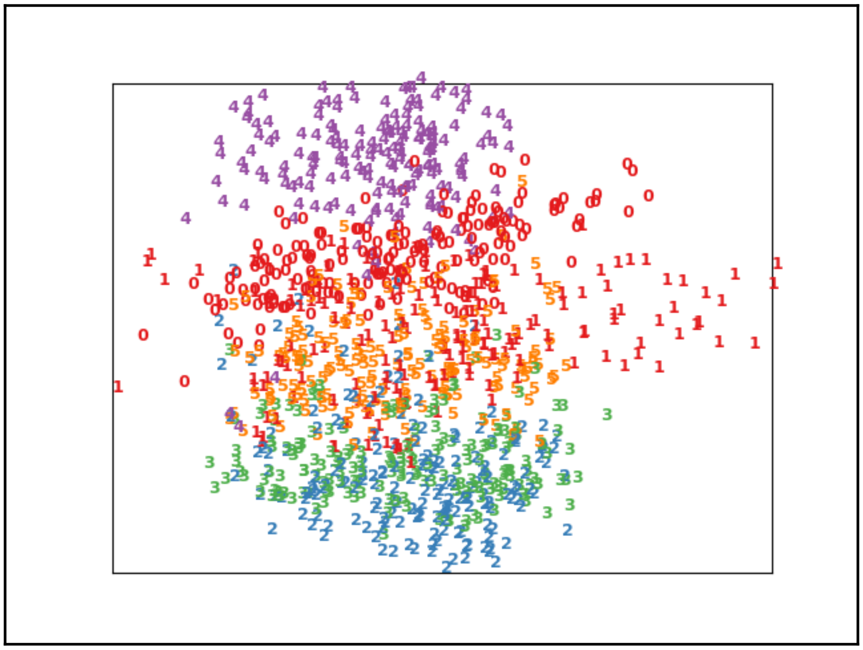


Figure 6: How different digits group together. It is because of this grouping that SVMs are successful in classifying them with high accuracy.

In *Figure 6*, we can see how similar digits are positioned in close proximity to each other. Just by visual inspection, we are able to draw rough lines between different regions occupied by the digits. This implicit grouping of digits helps machine learning techniques such as SVM, logistic regression and even K-means (we will read about this in the next section) to perform classification with such high accuracy.

Apart from PCA, there are other techniques such as t-SNE that do a much better job at helping visualize data. The following code can be used to for t-SNE:

```
tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
X_tsne = tsne.fit_transform(X)
```

Instead of using PCA in the previous code snippet, you can replace it by the preceding code snippet. The output for t-SNE is the following:

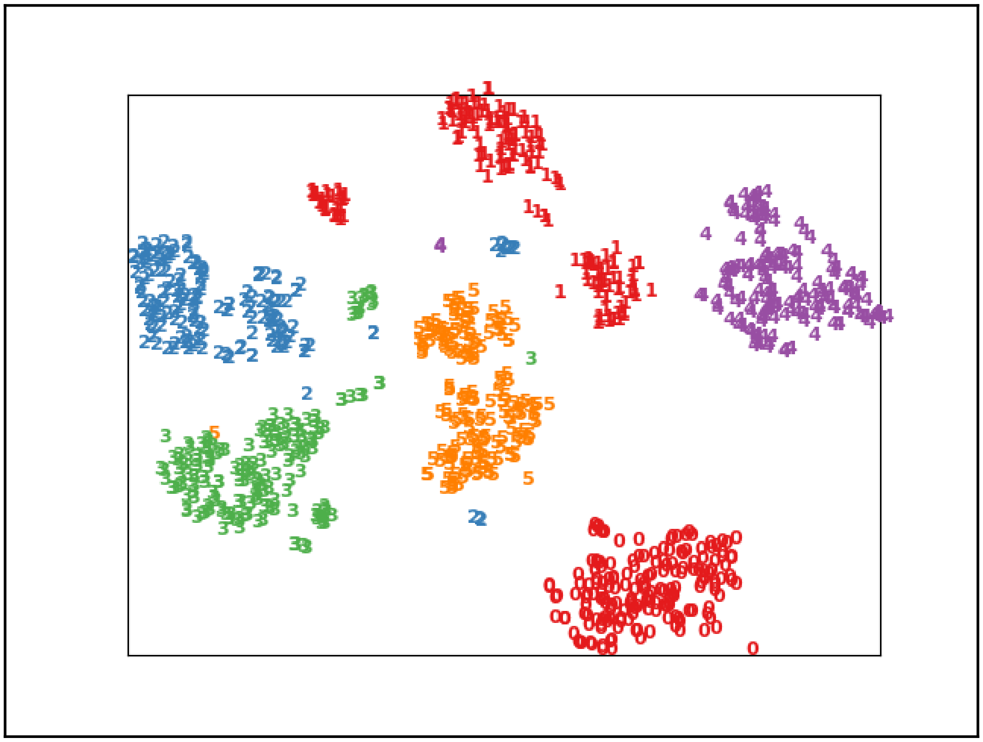


Figure 7: Output using t-SNE visualization algorithm

Figure 7 shows the grouping between the digits in a much clearer way. Data visualization is very important when we want see how our data exactly is positioned in the higher dimensions. This helps us in making the right decision about the kind of machine learning techniques that we want to use.

K-means clustering

K-means clustering is a type of semi-supervised or unsupervised machine learning, which works with partially labeled or even unlabeled data. As the name suggests, this is a type of clustering algorithm, which tries to form clusters of data points based on a similarity function. This algorithm is quite often confused with k-nearest neighbors. Although both algorithms share the same spirit, they are different from each other.

In k-means clustering, we form k clusters using the given data points based on a similarity metric. The most common form of a similarity metric is the distance between two points in the given space. Points closer to each other are clustered together. Initially, when the algorithm begins, we randomly select k points that represent the center of each of the k clusters. Then, iteratively, we keep updating these k center points such that they form the mean of the final k clusters.

For example, consider we are given satellite images of a country and we are asked to draw a rough boundary of the cities in that country. Our approach to the problem could define what a city might look like and then try to locate that in the given images. For simplicity, we say parts of the country that have a high density of buildings is where the cities are. For a moment, assume that we are capable of identifying buildings in an image. Now, the only task left is to group these buildings together into clusters (cities). This is a perfect scenario of using k-means clustering with the similarity metric being the distance between two buildings. The closer the buildings, the more chances they have of being in the same city.

You must be wondering how exactly could this be useful for computer vision? Imagine the same task of classifying digits. (I keep coming back to the same problem of digit classification so that we get a sense of how similar or different are all the algorithms that we are looking at in this chapter.) We are given a lot of images of size 28 x 28 for all the digits. Imagine this to be a 784-dimensional space with each image being a point in that space. (This is difficult to visualize, but try!) For a particular digit, all of its images will lie in a particular part of the 784-dimensional space since almost all of them will have similar pixels with black color. So when we calculate the distance between them, which is nothing but the squared sum of differences of each pixel value, the distance between images of the same digit will have lower value as compared to the distance between images of different digits because most of the differences will become zero for the same digits.

Now that we have some understanding of K-means clustering, let's look at a code snippet that implements the same using `sklearn`:

```
from sklearn import datasets, metrics
from sklearn.cluster import KMeans

mnist = datasets.load_digits()
images = mnist.images
data_size = len(images)

#Preprocessing images
images = images.reshape(len(images), -1)
labels = mnist.target

#Initialize Logistic Regression
clustering = KMeans(n_clusters=10, init='k-means++', n_init=10)

#Training the data on only 75% of the dataset. Rest of the 25% will be used
in testing the KMeans Clustering
clustering.fit(images[:int((data_size / 4) * 3)])

#Print the centers of the different clusters
print(clustering.labels_)

#Testing the data
predictions = clustering.predict(images[int((data_size / 4)):])
```

From the preceding code, let's look at the `KMeans` function. It takes a number of parameters as input. The first is the number of clusters, which is also the value of K in the algorithm. The next is `init`, where `init` means the algorithm that is used to initialize the first K centers for each cluster. The different options that a developer has are random, `k-means++`, or an array with custom K centers chosen by the developer. The `k-means++` initialization is based on Lyod's algorithm of finding evenly-spaced points in a Euclidean space. The Wikipedia page for Lyod's algorithm explains the exact concepts very well. The last parameter, `n_init`, that we used, means the number of different initial points we want to use. 10 means that we ran the k-means algorithm with 10 different initial points and then pick the one that performed the best.

Does something look out of place in this code? Yes, you got it right. We did not calculate the performance report this time. But why? Since this is a form of unsupervised learning, there is no way for us to co-relate the labels that are given by the k-means code and the labels that we have. There are ways in which you can manually co-relate the labels that you have with the labels that are output by the k-means algorithm and then compute the performance.

Summary

In this chapter, we looked at the basics of machine learning by understanding the various applications of machine learning, preprocessing techniques, and then three algorithms. Logistic regression and support vector machines are types of supervised machine learning algorithms, which require the developer to pass labeled training data. Then we looked at an unsupervised machine learning algorithm called k-means, which requires us to give just the data as input. Both types of learning algorithms are useful depending on the kind of data we have and the application that we are trying to build. There are more sophisticated techniques in machine learning, such as neural networks, which we will look at in more depth in the next chapter.

6
Image Classification Using Neural Networks

As we saw in the last chapter, machine learning can be a very useful tool in making applications more sophisticated and robust. In this chapter, we will add to our arsenal of machine learning techniques by understanding what neural networks are and how they can be used in the context of machine learning. We will look at the basics of neural networks and then, later in the chapter, talk about the state-of-the-art networks that are being used today in applications and services that we use everyday. One of the most studied tasks in computer vision is image classification. Given an image, can the computer tell what the image is of? Say, for example, if the computer is shown an image of a car, can it identify whether it's a car or not. Towards the end of this chapter, we will be able to build such a system with good accuracy. (Nothing is ever perfect!)

The following broad topics will be covered in the chapter:

- Introduction to neural networks
- Convolutional neural networks
- Challenges in machine learning

Let's begin with understanding what neural networks are, their history, and how they are relevant today.

Introduction to neural networks

The whole idea of machine learning is to build systems that given an input are able to predict the correct output with respect to a certain predefined context. For example, we want to build an image classification system, which is capable of telling the users what the given input image is of (see *Figure 1*). We briefly discussed this in the previous chapter. In this chapter, we will look at the context of neural networks:

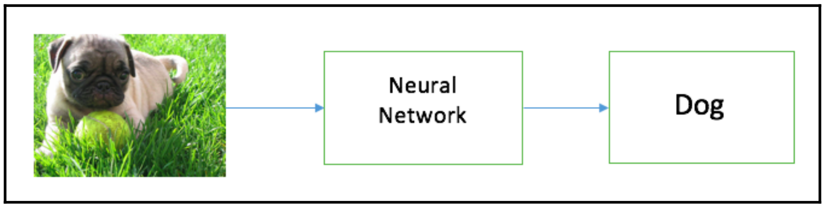


Figure 1: A neural network takes in an image and returns what that image is of

Think of a neural network as a polynomial function (just like any other mathematical function, say $f(x) = x + 2$), which is very difficult to devise manually using techniques of mathematics. With respect to computer vision, x in $f(x)$ is the image and the output is the image label. So the task at hand is to find out this function. Like we saw in the previous chapter, we will feed this polynomial function with our training data and look at the output of the function. We know what the correct output should be. We find the error in the output of the function and the correct output and modify the coefficients of the polynomial function such that the output error reduces. We keep feeding data to our function until we reach an error rate that is within the acceptable range set by us.

This is a very effective technique in finding functions that fit high-dimensional data. Doing the same task manually would require a lot of effort and time with no guarantee of a suitable result.

Design of a basic neural network

Neural networks are said to be inspired from the human brain. Just as our brains are made up of several neurons connected to each other, neural networks are made up of programmatically designed neurons that are connected to each other. We call the programmatically designed neurons—perceptrons.

Perceptrons form the atomic unit of neural networks. They take as input a set of numbers (one or many), multiply the input with some weights, and return an output.

The following diagram illustrates this concept:

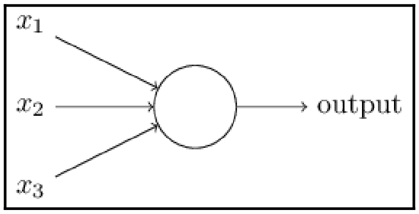


Figure 2: A perceptron

In *Figure 2*, it takes input as three values and returns an output. What happens inside is controlled by the developer. The most usual operation in each input is multiplied by a weight and the normalized sum of all products is returned.

Depending on the application and the data, the perceptron can be programmed to perform other operations. For example, after the sum of the products is calculated, the perceptron returns a 1 or a 0 if the sum is over a certain threshold.
To form a neural network, we connect such perceptrons to each other. One such network is shown in *Figure 3* as follows:

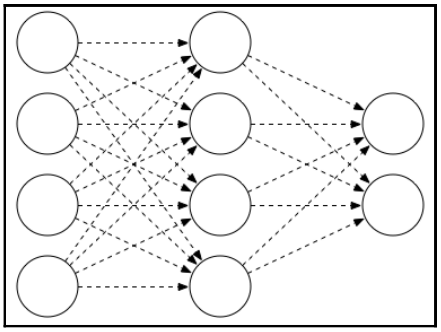


Figure 3: An example of a neural network

Here, each of the individual circles represent a perceptron. Each perceptron takes in a set of inputs and returns an output to the next layer of perceptron. What you see in *Figure 3* is just one way of creating a neural network. Each perceptron in a particular layer is connected to all the perceptrons in the previous layer. This is called a fully connected neural network. We will see in this chapter, some other types of networks that are more commonly used in computer vision these days.

There are a lot of different ways in which we can connect the perceptrons. We can change the number of layers we have in our neural networks. Different networks have different advantages. There is no standard network that works well for all cases. In machine learning, a lot depends on the kind of data you have and the nature of application that you are trying to develop.

Now that we know what a neural network is, let's take a step further and delve into some formal definitions.

The first layer of a neural network is usually called the input layer and similarly the last layer of the neural network is called the output layer. All other layers between them are known as the hidden layers. An input layer and an output layer is a must in a network but the number of hidden layers can vary from zero to as many as you want (as you will see later in the chapter, the greater the number of layers adds a significant amount of time in training the network, thus striking a good balance is very important).

What about the size of each of these layers? The size of the input layer is decided by the size of the images. Say our image size is *28 x 28*. The size of the input layer will be *784 (28 * 28)*. Then comes the hidden layer. Its size is decided by the user. One thing to keep in mind is to not make the size of the hidden layer extremely large or small as compared to the input layer. And, finally, the size of the output layer depends on the number of labels we have. Say we are classifying images into digits, which means we can have 10 different labels. The work of the output layer is to have a perceptron for each of the labels. Hence, in this case, the size of the output layer would be 10. Each perceptron in the output layer will return the probability of the input being in a particular label. The perceptron with the highest probability is selected and its associated label is returned as the output.

Training a network

Once we have the network design in place, we can now start training the network. The training phase of any neural network comprises two parts: first, feed forward the input and second, backpropagate the error. Let's understand each of them separately.

Feed forward means to take the input and pass it through the perceptrons in our network and calculate the output using the perceptrons. The input values are multiplied with the weights values of the perceptrons and an output is generated.

While backpropagating, we take the feed forward output (from the last step) and find its difference from the actual output (ground truth). Using this error, we modify the weights of the perceptron. You can think of the weights of each perceptron as coefficients of the polynomial function. For training the network which has an acceptable error count, we need to pass the entire input dataset multiple times to the network. One pass of the entire dataset is known as an epoch.

MNIST digit classification using neural networks

To understand all these concepts better, let's implement our own neural network using the `sklearn` library. We will use the MNIST digits dataset for our task. The steps involved in training our network are:

1. Preprocess the dataset by normalizing the pixel values of the images between 0, 1 or -1, and 1 (to make the mean 0).
2. Prepare the dataset. Split the dataset into two sets—training set and testing set.
3. Start training the dataset over the test data.
4. Compute your network's performance over the test dataset.

The following code trains a neural network for classification of handwritten digits (MNIST dataset):

```
from sklearn.datasets import fetch_mldata
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import normalize
from sklearn.model_selection import train_test_split

#Get MNIST Dataset
print('Getting MNIST Data...')
mnist = fetch_mldata('MNIST original')
print('MNIST Data downloaded!')

images = mnist.data
labels = mnist.target

#Preprocess the images
images = normalize(images, norm='l2') #You can use l1 norm too

#Split the data into training set and test set
images_train, images_test, labels_train, labels_test =
train_test_split(images, labels, test_size=0.25, random_state=17)

#Setup the neural network that we want to train on
```

```
nn = MLPClassifier(hidden_layer_sizes=(100), max_iter=20, solver='sgd',
learning_rate_init=0.001, verbose=True)

#Start training the network
print('NN Training started...')
nn.fit(images_train, labels_train)
print('NN Training completed!')

#Evaluate the performance of the neural network on test data
print('Network Performance: %f' % nn.score(images_test, labels_test))
```

Let's take a look at the code and understand what is really happening. The code follows the four steps that were mentioned at the beginning of this section. We first download the MNIST dataset using the helper functions from the `sklearn` library. Once we have that data, we then go on to normalize the dataset using `sklearn.preprocessing.normalize`. This scales the data down to the range of (0, 1). The next step is to split the data into a test set and training set. We use the `sklearn.model_selection.train_test_split` function for this. Now we finally come to our neural network. Using the `MLPClassifier` class, we create our own network. One particularly important parameter to look at is `hidden_layer_sizes`. This is where we get to choose the size of our network. As we saw earlier, input and output layers are mandatory for a neural network, but the number of hidden layers are in our control. In the preceding example, there is one hidden layer of size 100. In the next section, we will look at how the performance of the network changes as we change the number and size of the hidden layers. The `max_iter` parameter sets the maximum number of iterations that we will go through before we end the training. This does not mean that we will always iterate over the network that many times. If we reach an error rate that is within our acceptable range, then we can end the training early too.

The following is the output of the the preceding code:

```
NN Training started...
Iteration 1, loss = 2.29416218
Iteration 2, loss = 2.25190395
Iteration 3, loss = 2.20543191
Iteration 4, loss = 2.15313552
Iteration 5, loss = 2.09424290
Iteration 6, loss = 2.02753398
Iteration 7, loss = 1.95293486
Iteration 8, loss = 1.87160113
Iteration 9, loss = 1.78508449
Iteration 10, loss = 1.69547767
Iteration 11, loss = 1.60492990
Iteration 12, loss = 1.51560190
Iteration 13, loss = 1.42952528
Iteration 14, loss = 1.34794722
Iteration 15, loss = 1.27173197
Iteration 16, loss = 1.20135274
Iteration 17, loss = 1.13696783
Iteration 18, loss = 1.07838018
Iteration 19, loss = 1.02531269
Iteration 20, loss = 0.97732106
NN Training completed!
Network Performance: 0.810229
```

As you can see, with iteration, the loss value (error value) of the network keeps decreasing. This means that the network is learning the input data well over time. Network performance means that out of every 100 images provided to the neural network for testing, 81 images were classified correctly and the other 19 images were misclassified. In the next section, we will see how we can improve the network performance.

Even though the idea of training a neural network sounds difficult, using sklearn, you can get your network up and running within minutes. Although a point to note is that sklearn is probably not the best choice for building large-scale machine learning applications. There are other more advanced libraries that are more capable of handling large amounts of data efficiently.

Playing with hidden layers

In the example used in the last section, we tried to train our network with one hidden layer of size 100. Let's play around with that and see what happens.

First we will increase the size of the hidden layer from 100 to 200:

```
nn = MLPClassifier(hidden_layer_sizes=(200), max_iter=20, solver='sgd',
learning_rate_init=0.001, verbose=True)
```

The network performance is as follows:

```
Network Performance: 0.816800
```

We see that there is no significant improvement in the result. Now, let's try with increasing the number of hidden layers. We will train our network with three hidden layers of size 100 each:

```
nn = MLPClassifier(hidden_layer_sizes=(100, 100, 100), max_iter=20,
solver='sgd', learning_rate_init=0.001, verbose=True)
```

The result is as follows:

```
Network Performance: 0.857829
```

As we can see, there was an improvement of 5%. This means that if we increase the number of hidden layers, the performance improves. But this also does not mean that you will get a linear increase in the performance of the network as you keep improving the size of the network. There is no hard-and-fast rule as to how many hidden layers should a network have.

Until now, we have only seen fully connected networks. Let's take a look at another commonly used network architecture—convolutional neural networks.

Convolutional neural networks

Convolutional neural networks (**CNNs**) were first introduced somewhere around 1998 and since then have evolved a lot. CNNs are a variant of the traditional neural networks, where unlike the neural networks, not all perceptrons are connected to each other. In CNNs, the connections between perceptrons are sparse. Apart from that, each layer in a CNN can behave in a different manner. Let's take an example of a basic CNN and use that as a reference to explain the different concepts involved. The architecture that we will look at is called **LeNet**, which was proposed by Yann LeCun and others. This research effort was the beginning of the field of CNNs. Let's understand what they are and how they are different from traditional neural networks:

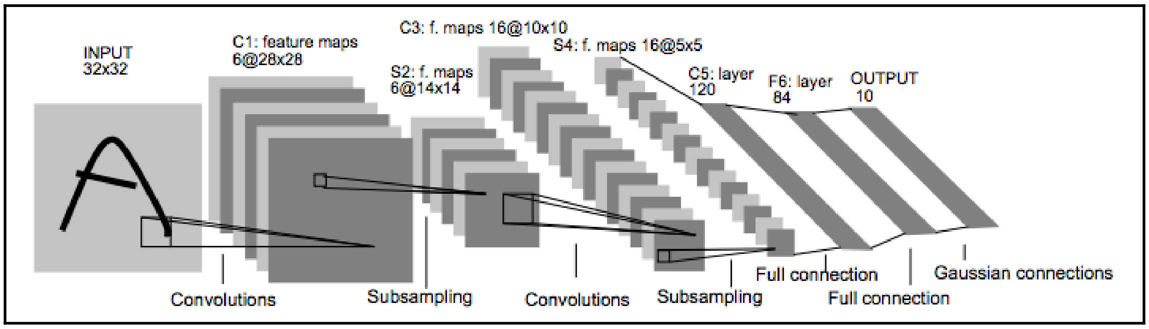


Figure 4: This is one of the most basic convolutional architectures - LeNet (LeCun et. al. 1998); (Source: `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`)

In `Chapter 1`, *Introduction to Image Processing*, we studied the concept of convolutions in images. Let's refresh our memory here a bit. Convolution is the technique of passing a kernel/filter over the entire image and producing a new image that may be smaller in size from the original image. While passing the filter over the image, what we are essentially doing is multiplying each cell in the kernel/filter by the corresponding cell in the image, summing all the products, and populating the output image with the value of the product.

As the name suggests, in convolutional neural networks, we take the input, apply convolution over the image using a randomly initialized filter, and produce a new image, which is smaller in size than the input image. Then, we repeat the same process with the output image from the first layer. We convolve the output image with a different randomly initialized filter and generate a new output image. We repeat this twice. Each time we take an image and apply convolution over it using a filter, we call it the **convolutional layer**. After every convolutional layer, the size of the output is less than the size of the input image. This concept is know as **subsampling**. As you can see in *Figure 4*, we have two convolutional layers and between the convolutional layers, we have subsampling. In *Figure 4*, after the second subsampling, we have a full connection. This is also called a fully connected layer in some books/papers. **Fully connected layers** are nothing but the traditional neural networks. As in neural networks, we have all perceptrons connected to each other, and we start to call these fully connected layers in CNNs. Fully connected layers help us to transform the subsampled images into scores for each label. The output from the fully connected layer is the probability score for each class.

Why do we have to use a convolutional layer and not build a traditional neural network with five layers? The problem with traditional neural networks is that they do not scale well for larger images. As the size of the image starts to increase, the number of perceptrons in the neural network would also start to increase and it will exponentially increase the time it takes to train the networks. On the other hand, in convolutional neural networks, the convolutional layers do not have to perform as many computations as a layer of perceptrons in NN, it is faster to train CNNs as compared to NNs; of the same size and CNNs also perform better.

Now that we know what CNNs are, let's try to implement LeNet using the `sklearn` library. The following code is an implementation of LeNet. Before we dive into the code, there are few libraries that we need to install:

- `keras`: This is a library for machine learning that will help us in implementing various components of a convolutional neural network. To install, run `pip install keras`.
- OpenCV: Follow the installation instructions given in `Chapter 7`, *Introduction to Computer Vision Using OpenCV*.

Once we have these installed, we are ready to run the following code:

```
from keras.models import Sequential
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.layers.core import Activation, Flatten, Dense
from keras.optmizers import SGD
from keras.utils import np_utils
from sklearn import datasets

#Config values
num_classes=9
img_depth=1
img_height=28
img_width=28

#Creating the LeNet model

model = Sequential()

#Adding the first convolutional layer
model.add(Convolution2D(20, 5, 5, border_mode="same",
input_shape=(img_depth, img_height, img_width)))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

#Adding the second convolutional layer
model.add(Convolution2D(50, 5, 5, border_mode="same"))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

#Adding the fully connected layers
model.add(Flatten())
model.add(Dense(500))

#Load MNIST data

model.add(Activation("relu"))

#Adding a softmax layer
model.add(Dense(num_classes))
model.add(Activation("softmax"))

mnist = datasets.fetch_mldata("MNIST Original")

#MNIST data is a flat array of size 784.
#We need to reshape it be in 28x28 images as we have to feed it to a
convolutional layer
mnist.data = mnist.data.reshape((mnist.data.shape[0], 28, 28))
mnist.data = mnist.data[:, np.newaxis, :, :]
mnist.data = mnist.data / 255.0 #Normalize the images to [0, 1.0]

#Split the data into train and test set
train_data, test_data, train_label, test_label =
train_test_split(minist.data, mnist.target, test_size=0.25)
train_label = np_utils.to_categorical(train_label, 10)
test_label = np_utils.to_categorical(test_label, 10)

#Set the loss funtions and evaluation metrics
model.compile(loss="categorical_crossentropy", optimizer=SGD(lr=0.0001),
metrics=["accuracy"])

#Train the LeNet model
model.fit(train_data, train_label, batch_size=32, no_epoch=30, verbose=1)

#Test the model
loss, accuracy = model.evaluate(test_data, test_label, batch_size=64,
verbose=1)
print("Accuracy: %".format(accuracy * 100))
```

Let's break down the code and see how it compares to what we read so far about CNNs. We start with defining configuration parameters. Image depth, width, height, and number of classes. In our case, we are again using the MNIST dataset that we have been using throughout the book.

We then go on and define the LeNet model as described earlier in the text. We create two convolutional layers followed by a fully connected layer.

This is one of the most simple CNNs. The more advanced CNN architectures such as ResNet and Inception are way more accurate and powerful in tasks of image classification. Implementing these architectures are beyond the scope of this book.

A nice exercise here would be to build a small application that takes in a random image, converts it to a grayscale image, resizes it to 28 x 28, and feeds it to the CNN model that we just implemented. This technique is used in post offices to detect zip codes and it works fairly well.

Challenges in machine learning

The most important challenge that researchers in machine learning face is that of the data. How do you decide whether the data that you have is good enough? When do you say the amount of data you have is enough? These are some questions that are very difficult to answer. The main purpose of building machine learning systems is to make them as general as possible for a given scenario. Say we are trying to build a handwritten digit classification system. Can we be sure that any image of digit 5 that we provide to the system will be able to output the correct output? Suppose our training data has hand samples from five people and we test the system on digits written by a sixth person. Will it perform as good? An extension to this problem is the lack of labeled data. One of the most laborious parts of collecting data is labeling data. Over time, with collaborative efforts from researchers, there are some good quality datasets available. But if you want to apply machine learning to a new field that has not been explored much before, collecting and labeling data is a challenging task that cannot be neglected. The system that you build will only be as good as the data that you have. It does not mean that if you have good data, you will have a good ML system. You of course need to have a good algorithm too.

Summary

In this chapter we learned about an important area within machine learning: neural networks. Neural networks form the basis of many state-of-the art machine learning systems. We began the chapter by understanding what neural networks are and how they work. Using the example of digit classification, we trained our own basic neural network which was capable of classifying an image into one of the 10 digits. After that we trained a more complex neural network - LeNet. LeNet is an example of a convolutional neural network, which has over time proved to perform better than traditional neural networks. Towards the end, we saw some challenges faced by researchers and developers in machine learning. In the next chapter, we will learn about another image processing library - OpenCV. This will help the readers expand their arsenal of tools for building computer vision applications.

7
Introduction to Computer Vision using OpenCV

OpenCV is an open source library for image processing and computer vision. Throughout the book, we have looked at scikit-image and pillow as tools for implementing various applications. In this chapter, we will learn about OpenCV and how to implement basic operations in image processing, including:

- Morphological operations
- Edge detection
- Contour detection
- Filters
- Template matching

OpenCV is a widely used library with many academic and commercial products using it at scale. In this chapter, we will revisit some of the algorithms and applications that we have seen so far in the book and re-implement them using OpenCV for Python 3.

Installation

In this section, we will cover the steps to install OpenCV in various operating systems.

macOS

The following steps will install OpenCV on macOS:

1. The first step is to install Xcode on your system. You can install Xcode from the App Store for free. Install the open Terminal and execute this command to accept the license:

   ```
   sudo xcodebuild -license
   ```

2. Now, type `agree` at the end of the document. Then, you need to install command-line tools using Xcode by executing this command:

   ```
   sudo xcode-select -install
   ```

3. The next step is to install OpenCV using the `homebrew` command. To install using `homebrew`, execute these commands:

   ```
   brew tap homebrew/science
   brew install opencv3 --with-contrib --with-python3 --without-python
   ```

4. Now, the final step is to link OpenCV and Python. For doing this, first you need to change the filename in `/usr/local/opt/opencv3/lib/python3.5/site-packages/` from `cv2.cpython-35m-darwin.so to cv2.so` by executing these commands:

   ```
   cd/usr/local/opt/opencv3/lib/python3.5/site-packages/
   mv cv2.cpython-35m-darwin.so cv2.so
   ```

5. Then finally execute this command to complete the installation:

```
        echo /usr/local/opt/opencv3/lib/python3.5/site-packages >>
/usr/local/lib/python3.5/site-  packages/opencv3.pth
```

Windows

OpenCV in Windows can be installed using the `pip` command:

```
pip install opencv_python-3.2.0-cp36-cp36m-win32.whl
```

Linux

The `pip` command can also be used to install OpenCV in Linux. The command for this is:

```
pip install opencv-python
```

OpenCV APIs

Now that we have OpenCV 3.2 installed on our machine, let's begin exploring the different APIs in OpenCV and how they can be used to build our computer vision application.

Reading an image

OpenCV has the `imread()` function to read an image. It takes the name of the file and returns the image matrix. The `imshow()` function can be used to display the image. The `imshow()` function takes the title and the image matrix as parameters. The following is an example of reading an image:

```
>>> import cv2
>>> img = cv2.imread("image.jpg")
>>> cv2.imshow("image",img)
```

The following figure is the output of the preceding code:

Figure 1

Writing/saving the image

The `imwrite()` function can be used to save the image to the disk. It takes the image name and the image matrix as input. The following is an example of writing an image:

```
>>> import cv2
>>> img = cv2.imread("image.jpg")
>>> cv2.imwrite("saved_image.jpg", img)
```

Changing the color space

OpenCV provides the `cvtColor()` function to convert the image from one color space to another. It takes the image as input and also the color space conversion code. The following are a few conversion codes:

- `COLOR_BGR2GRAY`
- `COLOR_BGR2HSV`
- `COLOR_HSV2BGR`
- `COLOR_BGR2YUV`
- `COLOR_GRAY2BGR`

The following is an example code for converting a BGR image to a grayscale image. We can change the second parameter of the `cvtColor()` function by any of the earlier mentioned parameter values:

```
>>> import cv2
>>> img = cv2.imread("image.jpg")
>>> gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
>>> cv2.imwrite("gray_image.jpg", gray)
>>> cv2.imshow("image",gray)
```

The following figure is the output of the preceding code:

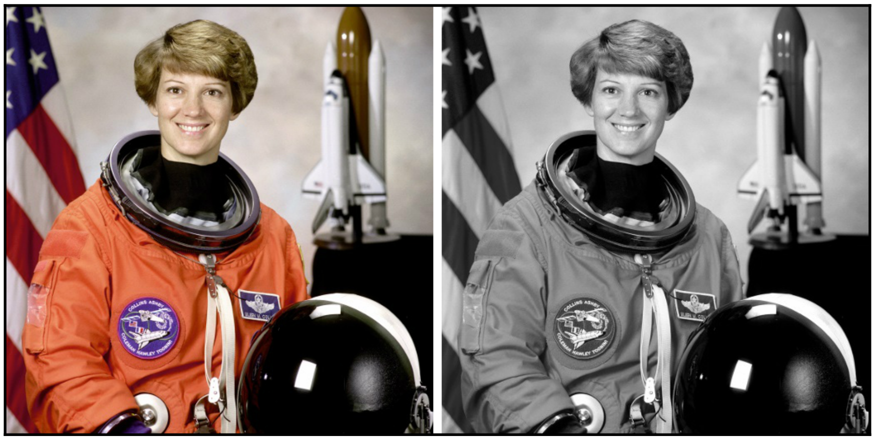


Figure 2: The original image is on the left and the output is on the right

Scaling

In order to resize the image, OpenCV has a `resize()` function, which takes the image, dimensions, and interpolation algorithm as input. Various interpolation algorithms can be used to interpolate the new pixel values. The following are the interpolation algorithms, which can be used to resize the image:

- `cv2.INTER_AREA`: This algorithm is preferred for shrinking the image
- `cv2.INTER_CUBIC`: This algorithm is preferred for zooming (slow)
- `cv2.INTER_LINEAR`: This algorithm is preferred for zooming
- `cv2.INTER_LINEAR`: This is the default algorithm

The following is an example of resizing an image:

```
>>> import cv2
>>> img = cv2.imread("image.jpg")
>>> r,c = img.shape[:2]
>>> new_img = cv2.resize(img, (2*r,2*c), interpolation = cv2.INTER_CUBIC)
>>> cv2.imwrite("resize_image.jpg", new_img)
>>> cv2.imshow("resize", new_img)
```

The `img.shape` attribute in this code returns the dimensions of the image. The following figure shows the output of the preceding code:

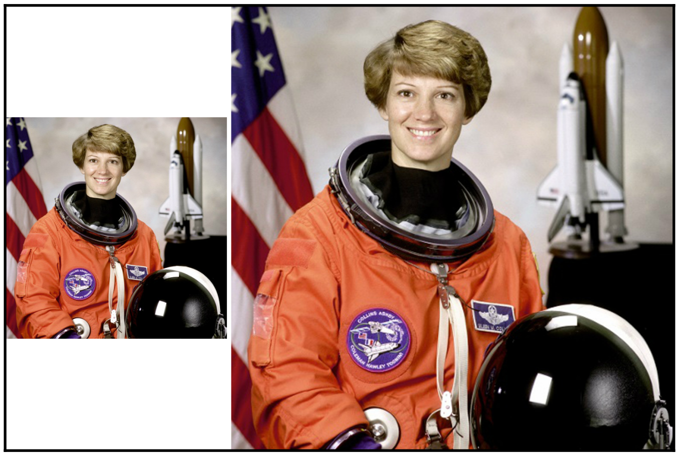


Figure 3: The original image is on the left and the output is on the right

Cropping the image

Cropping an image in OpenCV is very easy. It can be done by slicing the image array. Slicing an array is just taking the array values within particular index values. Consider the following example:

```
>>> import cv2
>>> img = cv2.imread("image.jpg")
>>> img_crop = img[0:200, 150:350]
>>> cv2.imwrite("crop_img.jpg", img_crop)
>>> cv2.imshow("crop", img_crop)
```

The following figure is the output of the preceding code:

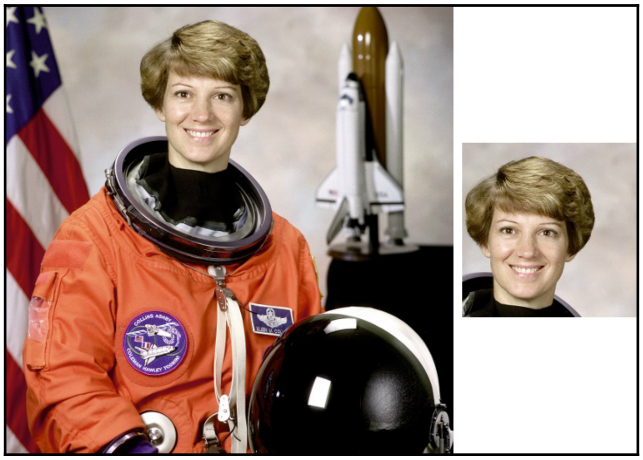


Figure 4: The original image is on the left and the output is on the right

Translation

For geometric transformation, OpenCV provides the `wrapAffine()` function, which takes the image, transformation matrix, and dimension of the image as input. The transformation matrix for the translation is as follows:

$$\boxed{M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}}$$

Here, (*tx*, *ty*) tells us the amount of shift in the image. For example, the following code shifts the image by (100,100):

```
>>> import cv2
>>> import numpy as np
>>> img = cv2.imread("image.jpg")
>>> r,c = img.shape[:2]
>>> M = np.float32([[1,0,100],[0,1,100]])
>>> new_img = cv2.warpAffine(img,M,(c,r))
>>> cv2.imwrite("translation.jpg", new_img)
>>> cv2.imshow("translation", new_img)
```

The following figure is the output of the preceding code:

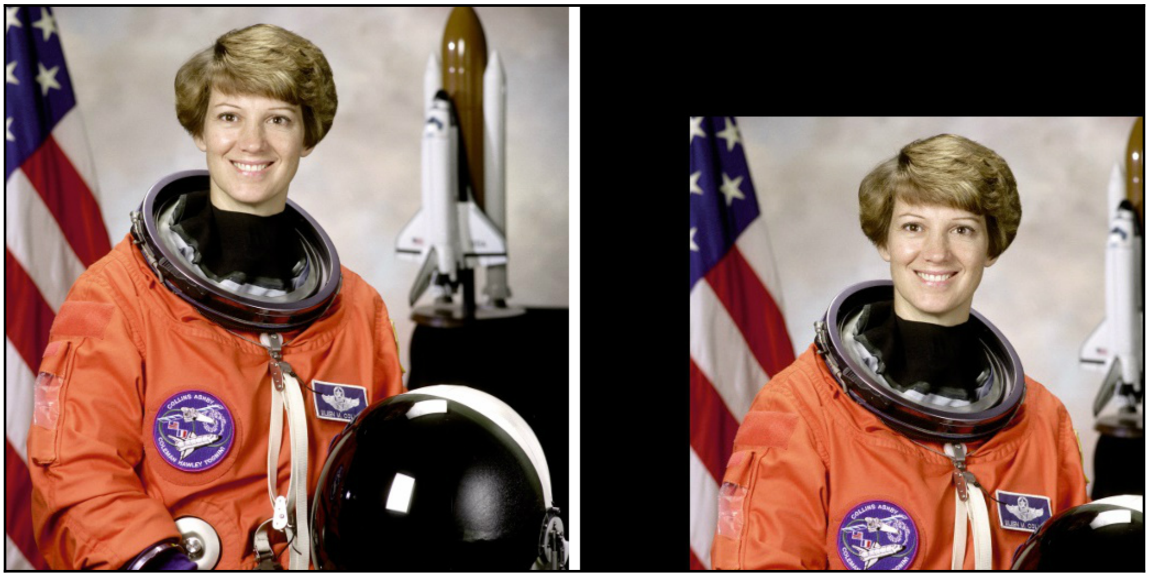


Figure 5: The original image is on the left and the output is on the right

Rotation

Rotation can also be done using `wrapAffine()`--only the transformation matrix changes. The transformation matrix for the rotation is as follows:

$$\boxed{M = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}}$$

Here, *theta* is the angle by which you want to rotate the image. The following code is an example of rotating the image by 90 degrees:

```
>>> import cv2
>>> img = cv2.imread("image.jpg")
>>> r,c = img.shape[:2]
>>> M = cv2.getRotationMatrix2D((c/2,r/2),90,1)
>>> new_img = cv2.warpAffine(img,M,(c,r))
>>> cv2.imwrite("rotate_img.jpg", new_img)
>>> cv2.imshow("rotate", new_img)
```

In this code, we use the `getRotationMatrix2D()` function to generate the transformation matrix. It takes the center for rotation, angle of rotation, and scaling factor as input. The following figure is the output of the preceding code:

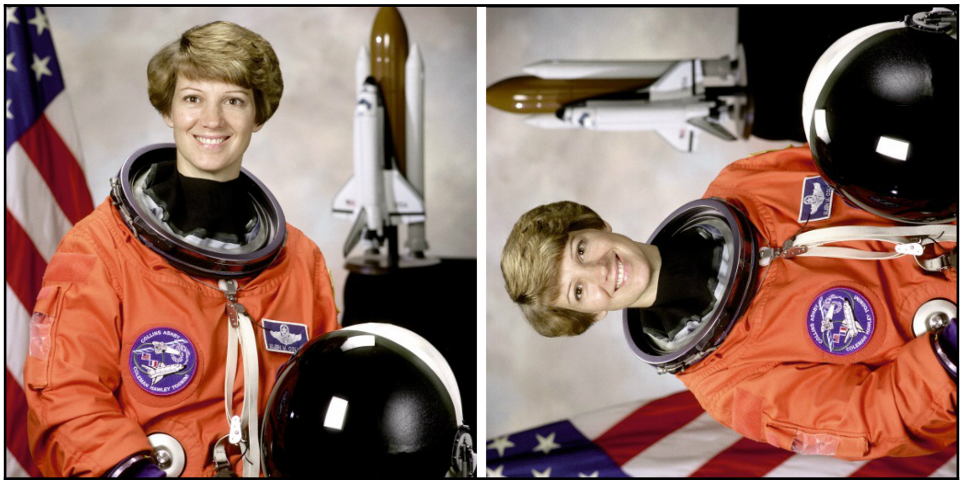


Figure 6: The original image is on the left and the output is on the right

Thresholding

We learned about thresholding in `Chapter 2`, *Filters and Features*. Now, let's see how it can be implemented using OpenCV. OpenCV has an inbuilt `threshold()` function, which takes a grayscale image, threshold value, and new value to be assigned if the value is greater than the threshold and type of `thresholding` as input. The types of thresholding are:

- `cv2.THRESH_BINARY`
- `cv2.THRESH_BINARY_INV`
- `cv2.THRESH_TRUNC`
- `cv2.THRESH_TOZERO`
- `cv2.THRESH_TOZERO_INV`

The following code is an example for `thresholding`:

```
>>> import cv2
>>> img = cv2.imread("image.jpg")
>>> gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
>>> new_img = cv2.threshold(gray,120,255,cv2.THRESH_BINARY)
>>> cv2.imwrite("thresholding.jpg", new_img[1])
>>> cv2.imshow("thresholding", new_img[1])
```

In the given example, all the pixels in the grayscale image, which are above 120, are set to white and the others are set to black, as you can see in *Figure 7*, which is the output of the preceding code:

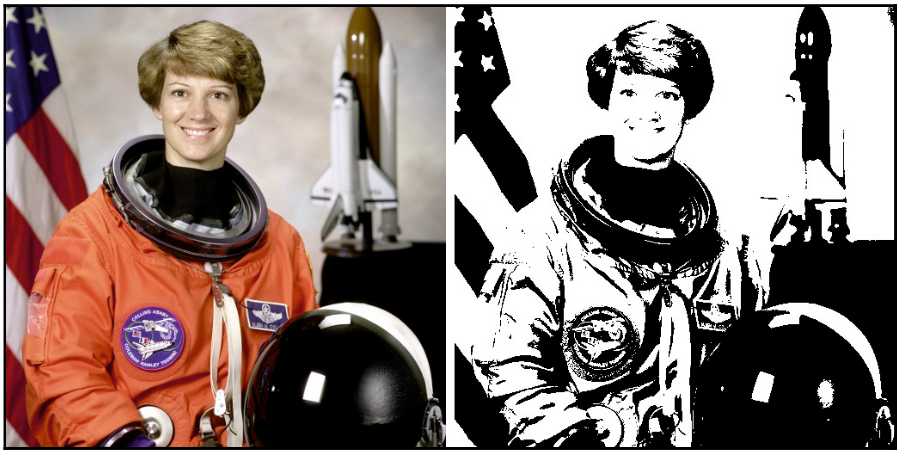


Figure 7: The original image is on the left and the output is on the right

Filters

In this section, we will see implementations of a few filters that we saw in `Chapter 2`, *Filters and Features*, using the OpenCV library. As we saw in the previous chapter, filters are created by convolution of a kernel with an image. To do this operation, OpenCV provides the `cv2.filter2D()` function, which takes the image, destination image depth, and kernel as input. Using this we can create our own filter.

Consider the following example:

```
>>> import cv2
>>> import numpy as np
>>> img = cv2.imread("image.jpg")
>>> ker = np.array([[1, 1, 1],
... [1, 1, 1],
... [1, 1, 1]])
>>> new_img = cv2.filter2D(img,-1,ker)
>>> cv2.imwrite("filter.jpg", new_img)
>>> cv2.imshow("filter", new_img)
```

The kernel is used as follows:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The following figure is the output of the preceding code:

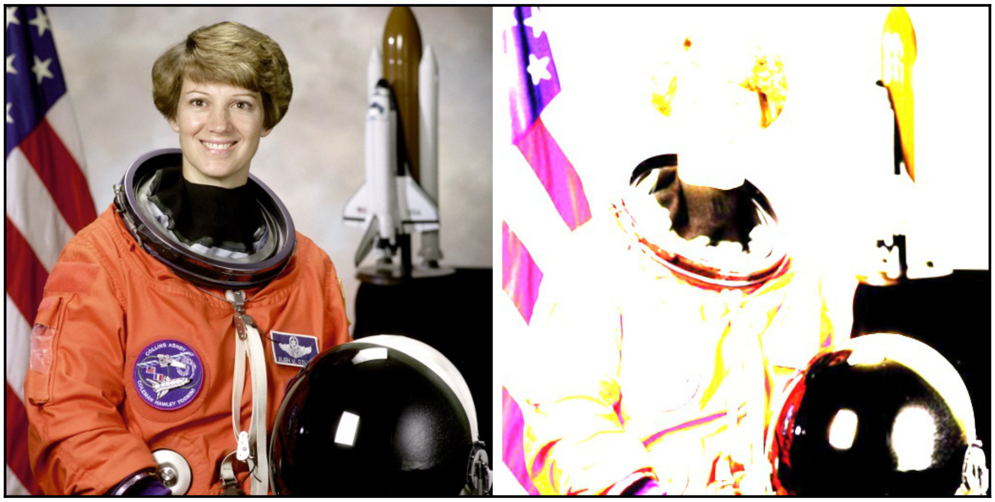


Figure 8: The original image is on the left and the output is on the right

Now, let's look at other inbuilt functions of filters in OpenCV.

Gaussian blur

There is an inbuilt function, `GaussianBlur()`, which takes the image, dimension of the kernel, and standard deviation as input. If the input standard deviation is zero, then the standard deviation is calculated from the size of the kernel:

```
>>> import cv2
>>> img = cv2.imread("image.jpg")
>>> new_img = cv2.GaussianBlur(img,(5,5),0)
>>> cv2.imwrite("gaussian_blur.jpg", new_img)
>>> cv2.imshow("gaussian_blur.jpg", new_img)
```

The following figure is the result of applying the Gaussian blur filter:

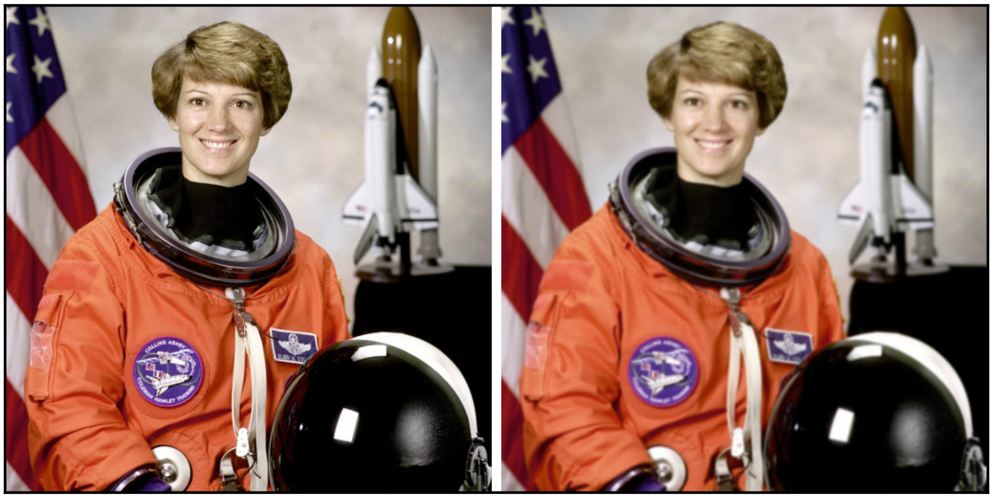


Figure 9: The original image is on the left and the output is on the right

Median blur

OpenCV has the `medianBlur()` function, which takes the image and size of the kernel as input (positive odd number):

```
>>> import cv2
>>> img = cv2.imread("image.jpg")
>>> new_img = cv2.medianBlur(img,5)
>>> cv2.imwrite("median_blur.jpg", new_img)
>>> cv2.imshow("median_blur", new_img)
```

The following figure is the result of applying the median blur filter:

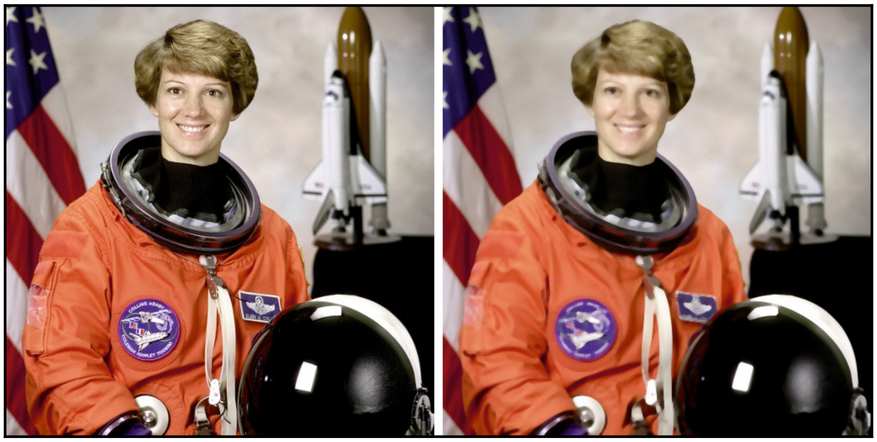


Figure 10: The original image is on the left and the output is on the right

Morphological operations

In this section, we see the implementation of the erosion and dilation using the OpenCV library that we learned about in `Chapter 2`, *Filters and Features*.

Erosion

As we saw in the earlier chapter, erosion requires a structuring element or kernel; therefore, the `erode()` function in OpenCV requires an image, kernel, and the number of times the erosion should be applied as input. Let's see an example code:

```
>>> import cv2
>>> import numpy as np
>>> img = cv2.imread("thresholding.jpg")
>>> ker = np.ones((5,5),np.uint8)
>>> new_img = cv2.erode(img,ker,iterations = 1)
>>> cv2.imwrite("erosion.jpg", new_img)
>>> cv2.imshow("erosion", new_img)
```

The following figure shows the output of the code:

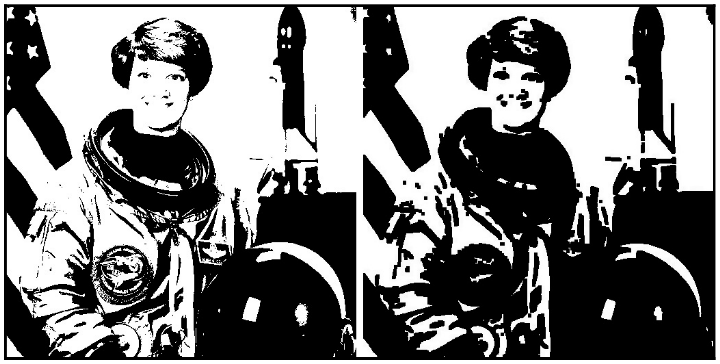


Figure 11: The input image is on the left and the image on the right is the output

Dilation

Similarly, let's see the code for dilation using the `dilate()` function:

```
>>> import cv2
>>> import numpy as np
>>> img = cv2.imread("thresholding.jpg")
>>> ker = np.ones((5,5),np.uint8)
>>> new_img = cv2.dilate(img,ker,iterations = 1)
>>> cv2.imwrite("dilation.jpg", new_img)
>>> cv2.imshow("dilation", new_img)
```

The following figure is the output of the preceding code:

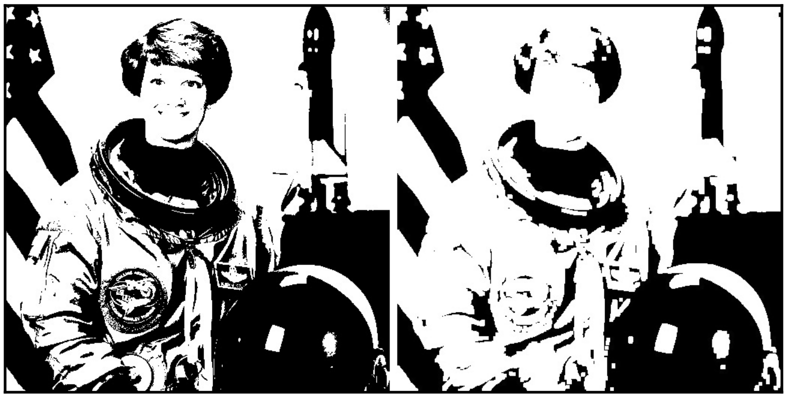


Figure 12: The input image is on the left and the image on the right is the output

Edge detection

In this section, we will see the implementation of edge detection algorithms using OpenCV that we studied in `Chapter 2`, *Filters and Features*. We will cover Sobel and Canny edge detection algorithms.

Sobel edge detection

The `Sobel()` function in OpenCV can be used to find the edges of an image. The `Sobel()` function takes the image, output image depth, order of derivative in *x* and *y* direction, and the size of the kernel as input. In the following code, we will look at both horizontal and vertical edges:

```
>>> import cv2
>>> img = cv2.imread("image.jpg")
>>> gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
>>> x_edges = cv2.Sobel(gray,-1,1,0,ksize=5)
>>> cv2.imwrite("sobel_edges_x.jpg", x_edges)
>>> y_edges = cv2.Sobel(gray,-1,0,1,ksize=5)
>>> cv2.imwrite("sobel_edges_y.jpg", y_edges)
>>> cv2.imshow("xedges", x_edges)
>>> cv2.imshow("yedges", y_edges)
```

In the code, we have taken the order of derivative in *x* direction equal to 1 and the order of derivative in *y* direction equal to 0 for Sobel in the *x* direction and the opposite for the *y* direction. Also, we have used -1 for the image depth, which means the image depth of the output will be the same as the input.

The following figure is the output Sobel in the *x* direction:

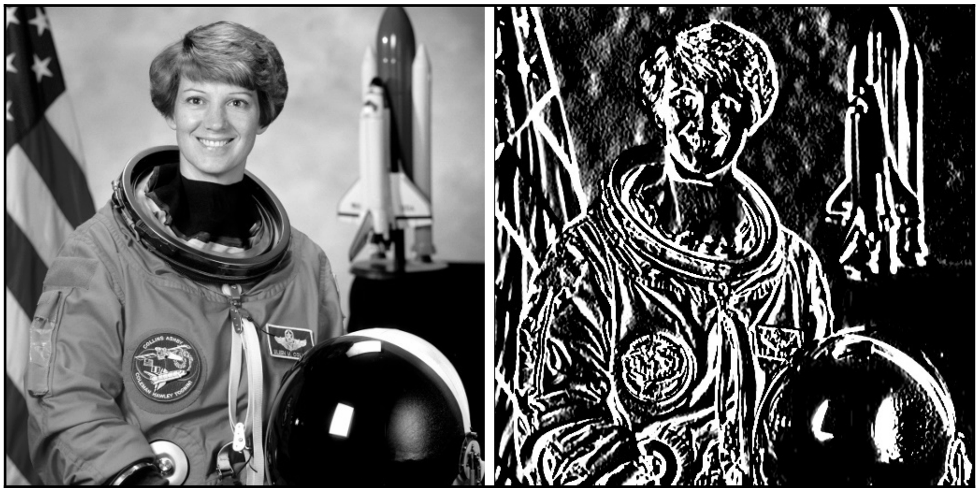


Figure 13: The image on the left is the input image and the image on the right is the output of Sobel in the x direction

The following figure is the output in the *y* direction:

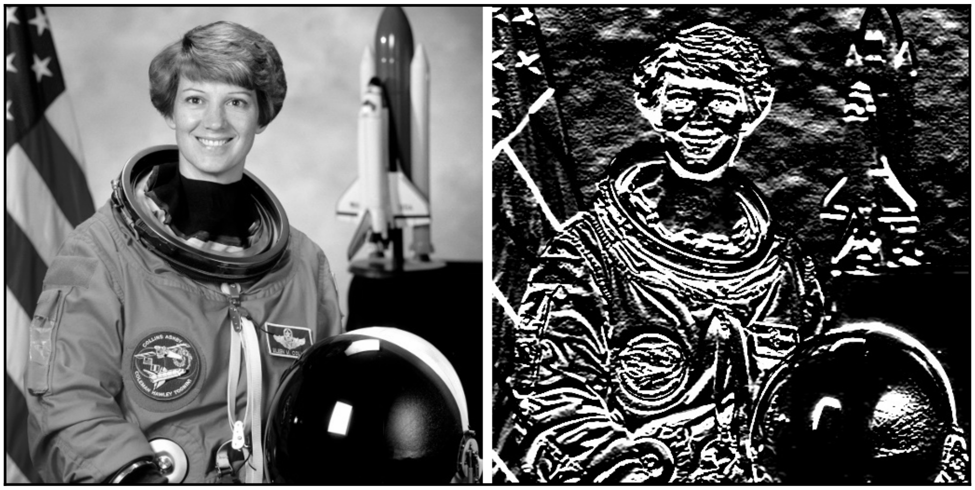


Figure 14: The image on the left is the input image and the image on the right is the output of Sobel in the y direction

Canny edge detector

Now, let's see the `Canny()` function in OpenCV. The `Canny()` function takes the image, min and max threshold values, and the size of the kernel as input. The following code shows how to use Canny edge detection in Python:

```
>>> import cv2
>>> img = cv2.imread("image.jpg")
>>> gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
>>> edges = cv2.Canny(gray, 100, 200, 3)
>>> cv2.imwrite("canny_edges.jpg", edges)
>>> cv2.imshow("canny_edges", edges)
```

Here is the output image of the preceding code:

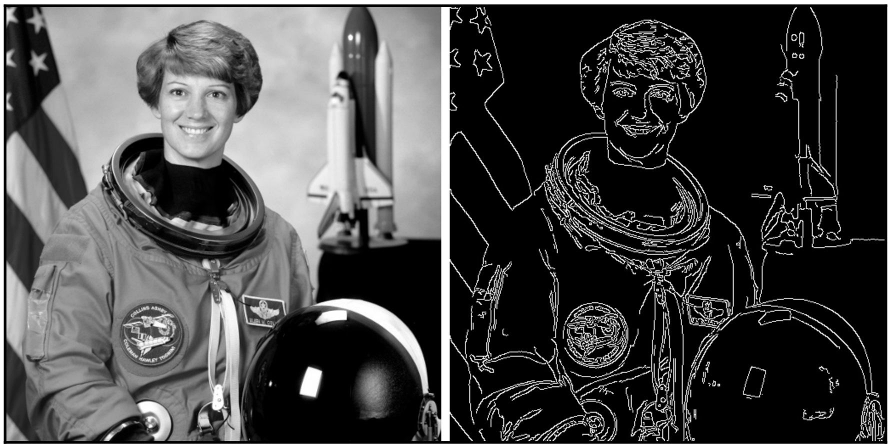


Figure 15: The image on the left is the input image and the image on the right is the output of the Canny edge detector

Contour detection

In this section, we will see how to use the `findContours()` function in OpenCV. First let's see what the function returns. It returns three arrays: first is the input image array, the second is the contours found in the image, and the third is the hierarchy array. The hierarchy array stores the relation between the contours: for example, if one contour is within another contour, then they will have a parent-child relationship and this is stored in the hierarchy array. The `findContours()` function takes three arguments: the source image, contour retrieval mode, and contour approximation method.

The contour retrieval mode tells us about the kind of hierarchy of contours: for example, in `RETR_LIST`, parents and child are considered equal and they are considered to be in the same level of hierarchy. Other types of retrieval modes are as follows:

- `RETR_EXTERNAL`
- `RETR_CCOMP`
- `RETR_TREE`

```
>>> import cv2
>>> img = cv2.imread('image.jpg')
>>> gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
>>> thresh_img = cv2.threshold(gray,127,255,0)
>>> im, contours, hierarchy =
cv2.findContours(thresh_img[1],cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
>>> cv2.drawContours(img, contours, -1, (255,0,0), 3)
>>> cv2.imwrite("contours.jpg", img)
>>> cv2.imshow("contours", img)
```

Figure 16 shows the output of the code:

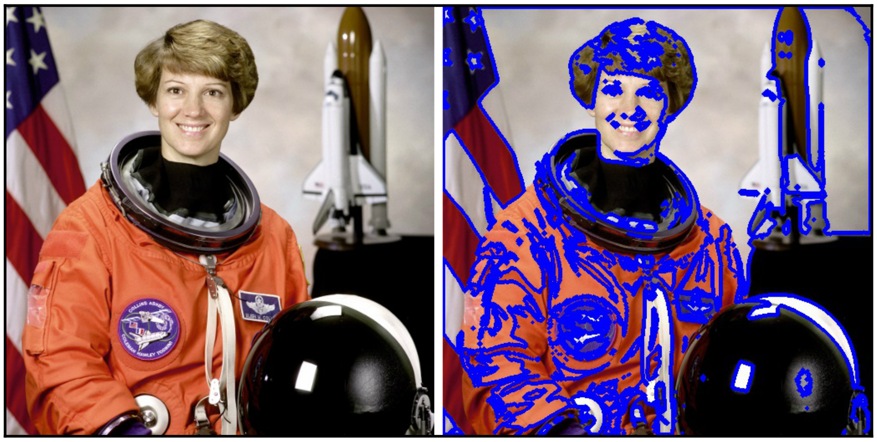


Figure 16: The image on the left is the input image and the image on the right is the output with contours shown in blue color

Template matching

In this section, we learn about how to locate a template image in an image using some OpenCV functions. A smaller image template matching will help get coordinates in a larger image that match with the template. Let us try to understand this with an example and also see how to write the code for it in Python. The following *Figure 17* is the template image we are going to use in our example:

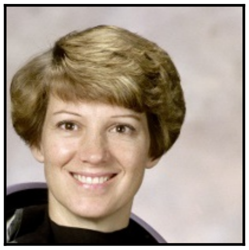


Figure 17: Template image that we wish to find in an image

The following *Figure 18* is the original image in which we have to locate the template image:

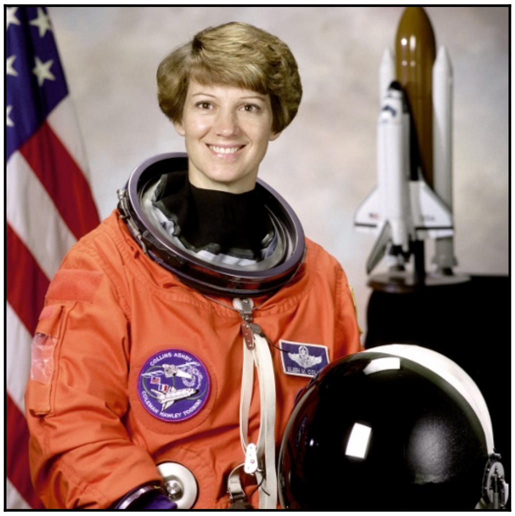


Figure 18 Image used to find the template shown in Figure 17

In the code for template matching, we are going to use `cv2.matchTemplate()` and `cv2.minMaxLoc()` functions. The `cv2.matchTemplate()` function iterates over the image and compares the input with template to find the match. The `cv2.minMaxLoc()` will give you the location of the best match. There are a few methods for finding the template in the image:

- `cv2.TM_CCOEFF`
- `cv2.TM_CCOEFF_NORMED`
- `cv2.TM_CCORR`
- `cv2.TM_CCORR_NORMED`
- `cv2.TM_SQDIFF`
- `cv2.TM_SQDIFF_NORMED`

The code for the template matching is as follows:

```
import cv2

img = cv2.imread("image.jpg")
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

img_temp = cv2.imread("template.jpg")
gray_temp = cv2.cvtColor(img_temp, cv2.COLOR_BGR2GRAY)

w, h = gray_temp.shape[::-1]

output = cv2.matchTemplate(gray,gray_temp,cv2.TM_CCOEFF_NORMED)
min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(output)

top = max_loc
bottom = (top[0] + w, top[1] + h)

cv2.rectangle(img,top, bottom, 255, 2)

cv2.imshow("image",img)
cv2.imwrite("img.jpg",img)
```

In the code, the `max_loc` variable will give us the coordinates of the top left corner of the rectangle. To find the bottom right coordinates, we add the width and height of the template image to the top left coordinates. *Figure 19* shows the output of the code:

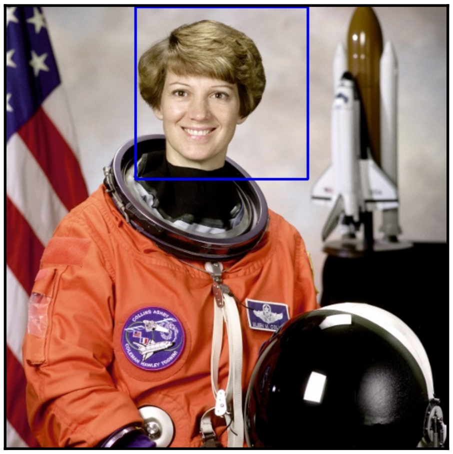


Figure 19: Output of the template matching. The blue box shows the part of the image where the template shown in Figure 17 matched with this image.

Summary

In this chapter, we revisited all the algorithms that we had seen so far in this book and implemented them using a new open source library, OpenCV. The topics discussed in this chapter lay the foundations of image processing using OpenCV and will equip you to build more sophisticated applications.

In the next chapter, we will look at some feature extraction algorithms provided by OpenCV.

8
Object Detection Using OpenCV

This chapter is a re-visit of `Chapter 3`, *Drilling Deeper into Features - Object Detection*, but with new algorithms. As you may be familiar with object detection from `Chapter 3`, *Drilling Deeper into Features - Object Detection*, in this chapter, we will cover some more feature extraction algorithms. Unlike `Chapter 3`, *Drilling Deeper into Features - Object Detection*, in this chapter, we will use OpenCV to implement all the algorithms.

The following topics will be covered in this chapter:

- Cascade classifier—Haar Cascades
- Scale Invariant Feature Transformation (SIFT)
- Speeded-up robust features (SURF)

Haar Cascades

Haar Cascades is a type of cascade classifier, where the system is made up of multiple chained classifiers (refer to `Chapter 3`, *Drilling Deeper into Features-Object Detection*, for details on cascade classifiers). Haar Cascades is one of the first feature extraction algorithms. Viola and Jones, in their face detection algorithm, first introduced a way of using Haar wavelets to extract features in an image. They proposed an algorithm for face detection using Haar Cascades. The main idea behind the algorithm was the inherent structure that is present in all the faces. For example, in every human face, the eye region is darker than the cheeks, and the nose bridge region is darker than the eyes. Using such characteristics of a human face, we learn the generic models of the face and then use these trained models to detect faces in images.

For Haar Cascades, we first train a model using some face images and then test it out on test images like we did in `Chapter 5`, *Integrating Machine Learning with Computer Vision* and `Chapter 6`, *Image Classification Using Neural Networks*. Initially, we feed a learning algorithm with positive images (images with faces) and negative images (images without faces) and learn the classifier. Then, we extract Haar features from the images using convolutional kernels (as shown in the following diagram). Feature values are obtained by subtracting the sum of white pixels under the white rectangle from the sum of pixels under the black rectangle. We slide these kernels (nothing but Haar features) over the entire image and calculate the feature values. If the value is above a certain user-defined threshold, we say that there is a match; otherwise, we reject that region. The following diagram shows some Haar features that are used for face detection. These features can also be used for other objects too and are not restricted to just faces:

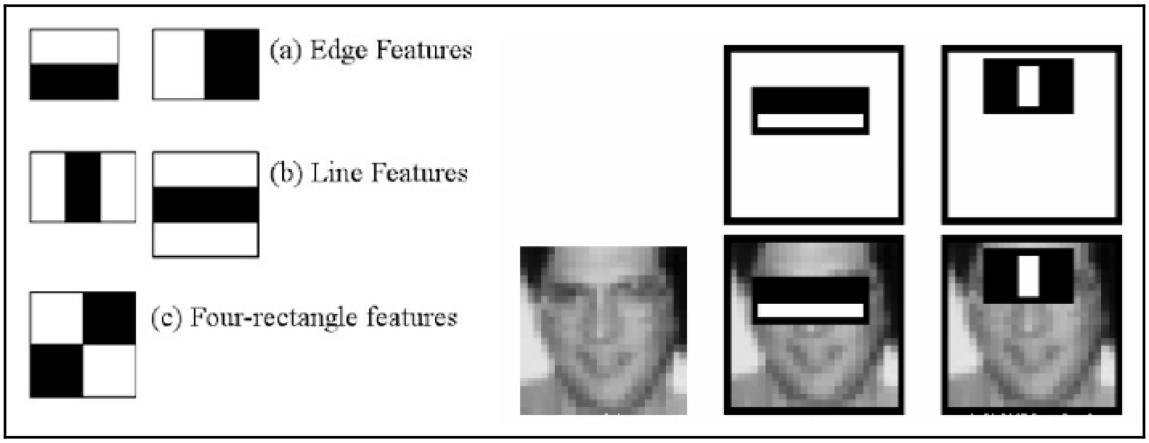


Figure 1: Different Haar features (left) and how they are used for face detection (right)

To reduce calculations while testing for Haar features in an image, we make use of integral images. Let's understand what integral images are.

Integral images

Integral images, also known as summed area tables, are another form of storing images. For integral images, we replace each pixel in the image with the sum of all the elements on the left and above that pixel.

The formula shown next is used to compute integral images:

$$\boxed{I_\Sigma(x,y) = \sum_{\substack{x' \le x \\ y' \le y}} i(x',y')}$$

Where, *i(x', y')* means the value of pixel at *(x', y')* in the image. *I* is the **integral image**.

If we closely look at the formula for calculating integral images, we will notice that there are a lot of redundant calculations that we make. For computing the value for pixel (5, 5), we compute the value for pixel (1, 1), (2, 2), and so forth. We can avoid these calculations if we use the precomputed values in the integral image. The formula shown next computes integral images in an efficient manner:

$$\boxed{I(x,y) = i(x,y) - I(x-1,y-1) + I(x,y-1) + I(x-1,y)}$$

Here, *i* is the original image and *I* is the integral image.

Coming back to Haar Cascades, let's now look at an implementation of Haar Cascades to detect faces in an image. But before we look at the code, we need to download pretrained Haar Cascade XML files from the OpenCV trunk. The link for downloading the files is: `https://github.com/opencv/opencv/tree/master/data/haarcascades`.

You can download any pretrained Haar Cascade that you want. But for the purpose of this example, we will work with `haarcascade_frontalface_default.xml` and `haarcascade_eye.xml`.

The code is as follows:

```
import cv2

face_cascade =
cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier('haarcascade_eye.xml')

img = cv2.imread('image.jpg')
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

faces = face_cascade.detectMultiScale(img_gray, 1.3, 5)

for (x,y,w,h) in faces:
```

```
            cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
            roi_gray = img_gray[y:y+h, x:x+w]
            roi_color = img[y:y+h, x:x+w]
            eyes = eye_cascade.detectMultiScale(roi_gray)
            for (ex,ey,ew,eh) in eyes:
                cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)

        cv2.imwrite('output.jpg',img)
```

The preceding code is very easy to follow. We first load the Haar Cascade classifier using the `cv2.CascadeClassifier()` function. We load two classifiers, one for the face and the other one for the eyes. The motive of this code is to first detect faces and then within the face region detect the eyes. So, after loading the XML files, we read an image on which we want to perform face detection. We first run the face detection over the entire image. After storing the results of face detection in the faces, we iterate over the list and run the eye detection classifier to find all the eyes in the images. To reduce computation cost, we run the eye detection only in the face region by extracting the **region of interest** (**ROI**).

The output for the preceding code is as follows:

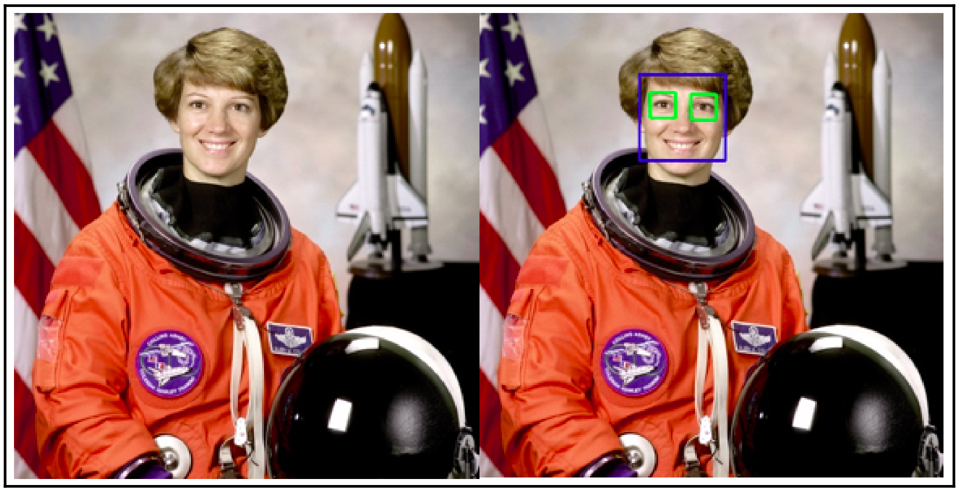


Figure 2: Output of face detection using Haar Cascades

Haar Cascades can be extended to work with other types of objects as well. You will have to train the classifier by providing positive and negative images for the object you want to detect. Training Haar Cascade takes a lot of time but it is fast when it comes to using the trained classifier over test images.

The next feature extraction algorithm that we will look at is SIFT.

Scale Invariant Feature Transformation (SIFT)

Scale Invariant Feature Transform (SIFT) is one of the most widely used feature extraction algorithms to date. Its scale, translation, and rotation invariance, its robustness to change in contrast, brightness, and other transformations, make it the go-to algorithm for feature extraction and object detection. It was proposed by David Lowe in 2004.

The original publish paper can be found at `http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf`.

Some important properties of SIFT are as follows:

- It is invariant to scaling and rotation changes in objects
- It is also partially invariant to 3D viewpoints and illumination changes
- A large number of keypoints (features) can be extracted from a single image

Let's see how SIFT is able to achieve all of the earlier mentioned properties. The next section explains the SIFT algorithm in detail.

Algorithm behind SIFT

The main motivation behind SIFT is to extract local features from an image that is robust. To achieve this, the algorithm is divided into the following four main stages:

- Scale-space extrema detection
- Keypoint localization
- Orientation assignment
- Keypoint descriptor

If you carefully read `Chapter 3`, *Drilling Deeper into Features-Object Detection,* you will realize that this sounds very similar to the ORB algorithm (SIFT was proposed before ORB). The takeaway from this point is that most of the feature-detection algorithms have the same motivation to extract robust features and hence they have a very similar approach. All the algorithms do something unique in each of these stages, which differentiates them from each other in terms of scenarios where one works better than the other.

Coming back to SIFT, let's explore each of those stages in detail.

Scale-space extrema detection

In the first stage of SIFT, we aim to achieve scale invariance by generating a multiple scale pyramid of the original image as shown in *Figure 3*:

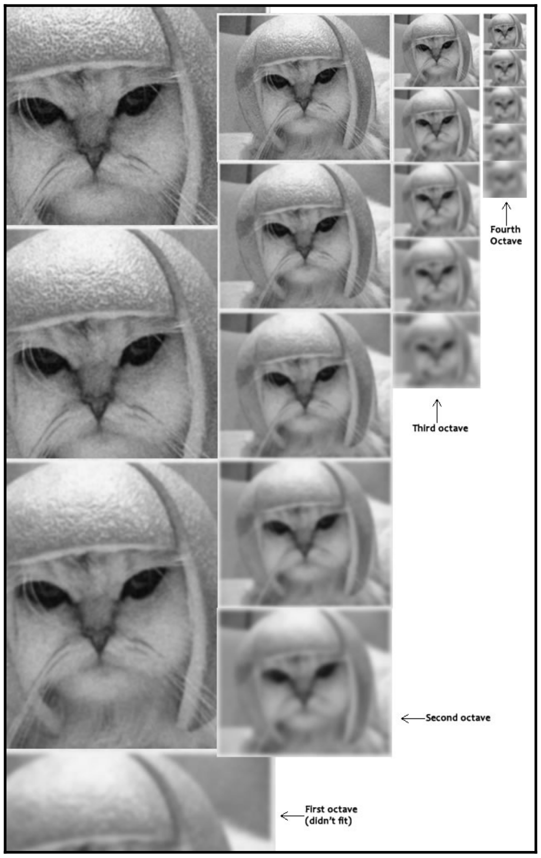


Figure 3: Original image blurred and resized to generate four octaves

What we are essentially doing here is getting rid of details that do not persist at different scales. By doing so, we are only left with information that is scale invariant. Now, to achieve the pyramid shown in the preceding figure, we apply Gaussian Blur to the image.

We first take the original image and apply Gaussian Blur five times with different sigma values. This results in five blurred images of the same size. In the context of SIFT, this is known as an octave. Next, we will resize the original image and create another octave with the resized image. We repeat this procedure four times to get four different octaves. In *Figure 1*, each vertical row represents an octave. A point to note here is that five blurred images and four octaves are hyper-parameters for SIFT. You are free to choose how many octaves and blurred images in each octave you want. After empirical testing, four octaves and five blurred images for each octave gave the best results.

After getting all the images in place, we will apply Laplacian of Gaussian, which is used to precisely detect edges in an image. In Laplacian of Gaussian, we calculate the second order derivative of the image, which locates all the edge and corner points in the image that will be used as potential keypoints of the image. Since the second order derivative is extremely sensitive to noise, Gaussian Blur helps in stabilizing the derivative. There is another challenge with second order derivatives—they are computationally expensive to calculate, which is not ideal if we want to use SIFT for real-time applications. To reduce the computational cost of calculating the second order derivative, we do an approximation. We approximate the second order derivative as a difference of Gaussian (refer to `Chapter 2`, *Filters and Features*, for more details). The following formula shows the approximation:

$$\boxed{\begin{aligned} D(x,y,\sigma) &= \left(G(x,y,k\sigma) - G(x,y,\sigma)\right) * I(x,y) \\ &= L(x,y,k\sigma) = L(\mathrm{x},\mathrm{y},\sigma) \end{aligned}}$$

Here, *D* represents the Difference of Gaussian, *G* represents the Gaussian filter, *L* is the Laplacian of Gaussian, *k* is the multiplicative constant that decides the amount of blurring in each image in the scale space. A scale space is defined as the set of images that have been either scaled-up or down for the purpose of computing keypoints. For example, *Figure 4* shows two sets of images; one set is the original set of five images that have been blurred with different blurring radius and the other is a set of scaled-down images:

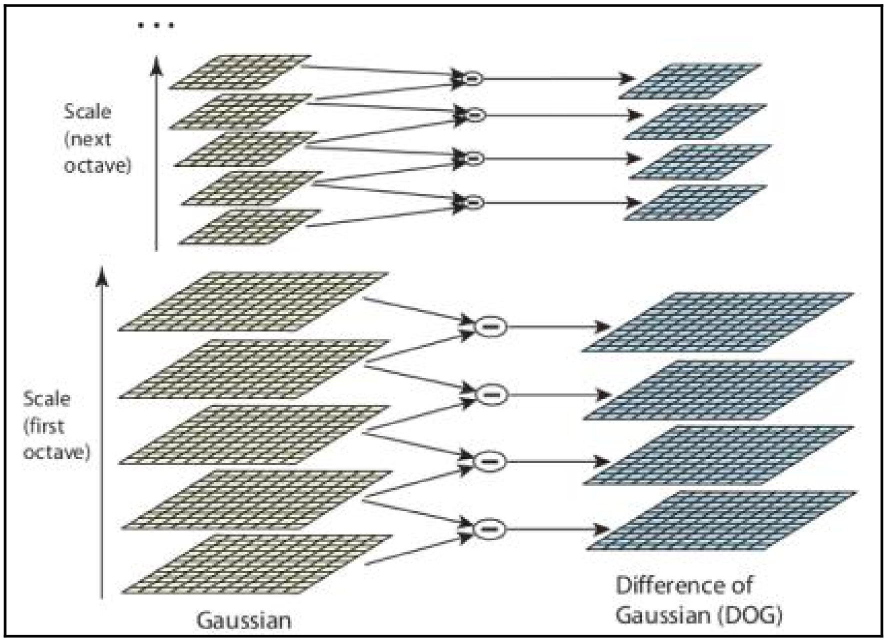


Figure 4: Two sets of images blurred five times for computing the Difference of Gaussian

To generate the Laplacian of Gaussian images, we calculate the difference between two consecutive images in an octave. This is called the **Difference of Gaussian** (**DoG**). These DoG images are approximately equal to the ones obtained by calculating the Laplacian of Gaussian. Using DoG also has an added benefit—the images obtained are also scale invariant.

After this step, we have successfully filtered all the nonimportant points that did not persist over different scales. In the next stage, we will further filter out points with stronger constraints.

Keypoint localization

In this stage, we will find points that are local extrema. This means, we need to identify points that are best representations of a region of the image (in other words, the neighborhood of the point) in different scales. To locate these keypoints, we iterate over each pixel and compare it with all its neighbors. Now, this is where things start to become interesting. Until now in the book, we always thought of neighbors as the eight pixels that are adjacent to a pixel, but for SIFT, we will not only look at these eight pixels but also at the nine pixels in the preceding images and below this image in the scale space or octave (look at *Figure 5*). Here we are comparing the pixel value to its 26 neighboring pixels. We select this point as a local extremum if it is the minimum or the maximum pixel among its neighbors. On an average, we rarely compare the pixel value to all the 26 values; it only takes a few comparisons to check:

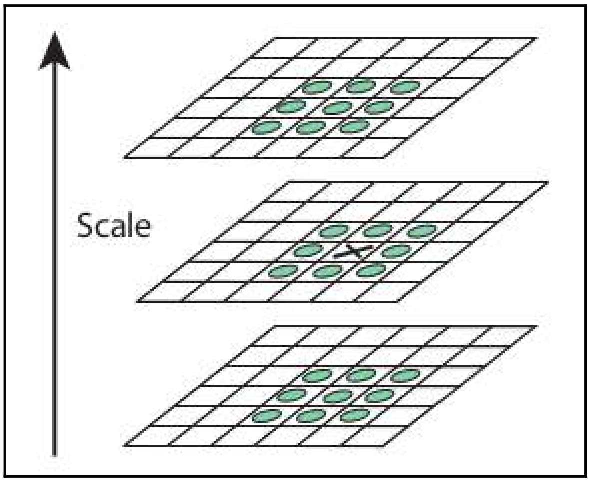


Figure 5: To determine whether a pixel marked x is an extremum, we compare it with all the neighbors marked in green

We do not calculate the keypoints in the uppermost and lowermost images in an octave because we do not have enough neighbors to identify the extrema.

To make the algorithm more efficient in finding perfect extrema, the author observed that the extrema are never located at the exact pixels. They may be present in between the pixels, but we have no way to access this information in an image. The keypoints located are just their average positions. We use the Taylor series expansion of the scale space function *D(x, y,σ)* (up to the quadratic term) shifted until the current point as the origin gives us:

$$\boxed{D(x) = D + \frac{\partial D^T}{\partial x} x + \frac{1}{2} x^T \frac{\partial^2 D}{\partial x^2} x}$$

Here, *D* and its derivatives are calculated at the point we are currently testing for extrema. Using this formula, by differentiating and equating the result to zero, we can easily add the subpixel keypoint locations.

SIFT recommends that you generate two such extrema images. Thus, to generate two extrema, we need four DoG images. To generate these four DoG images, we need five Gaussian blurred images. Thus, we need five images in a single octave. It has also been found that the optimal results are obtained when $\sigma = 1.6$ and $k = 2$.

So far we have been able to further filter out points we got from the first stage, but if you look at the actual number of points we still have, it is quite high. Some of these points lie on the edge or do not have enough contrast that they are actually useful to us. Remember we want an algorithm that is not sensitive to contrast or brightness change. Let's solve these problems one at a time. First, the points that lie on an edge. To filter these points out, we use an approach that is similar to the one used in the Harris corner detector (refer to `Chapter 3`, *Drilling Deeper into Features-Object Detection*, for more details).

To eliminate keypoints along the edges, we calculate two gradients at the keypoint, which are perpendicular to each other. The region around the keypoint can be one of the following three types:

- A region (both gradients will be small).
- An edge (here, the gradient parallel to the edge will be small, but the one perpendicular to it will be large).

- A corner (both gradients will be large) as we want only corners as our keypoints, we only accept those keypoints whose both gradient values are high. To calculate this, we use the **Hessian matrix**. This is similar to the Harris corner detector. In the Harris corner detector, we calculate two different eigenvalues, whereas, in SIFT, we save the computation by just calculating their ratios directly.

To tackle the contrast problem, we use a rather simple technique. We simply compare the intensity value of the current pixel to a preselected threshold value. If it is less than the threshold value, it is rejected. Because we have used subpixel keypoints, we again need to use the Taylor series expansion to get the intensity value at subpixel locations.

Once we have performed all the preceding operations, we have successfully filtered out all the points that are not important for describing the image and all the points that we are left with are the SIFT keypoints. But all is not done. Until now we have only fulfilled the scale-invariant property; next up, we will make these key points rotation invariant.

Orientation assignment

Until now, we have stable keypoints and we know the scales at which these were detected. So, we have scale invariance. Now we try to assign an orientation to each keypoint. This orientation helps us achieve rotation invariance. We try to compute the magnitude and direction of the Gaussian blurred images for each keypoint. The magnitudes and directions are calculated using these formula:

$$\boxed{\begin{aligned} m(x,y) &= \sqrt{\left(L(x+1,y)-L(x-1,y)\right)^2+\left(L(x,y+1)-L(x,y-1)\right)^2} \\ \theta(x,y) &= \tan^{-1}\left(\left(L(x,y+1)-L(x,y-1)\right)/\left(L(x+1,y)-L(x-1,y)\right)\right) \end{aligned}}$$

The magnitude and orientation are calculated for all pixels around the keypoint. We create a 36-bin histogram covering the 360-degree range of orientations. Each sample added to the histogram is weighted by its gradient magnitude and by a Gaussian-weighted circular window with σ, which is 1.5 times that of the scale of the keypoint. Suppose you get a histogram, as shown in the following figure:

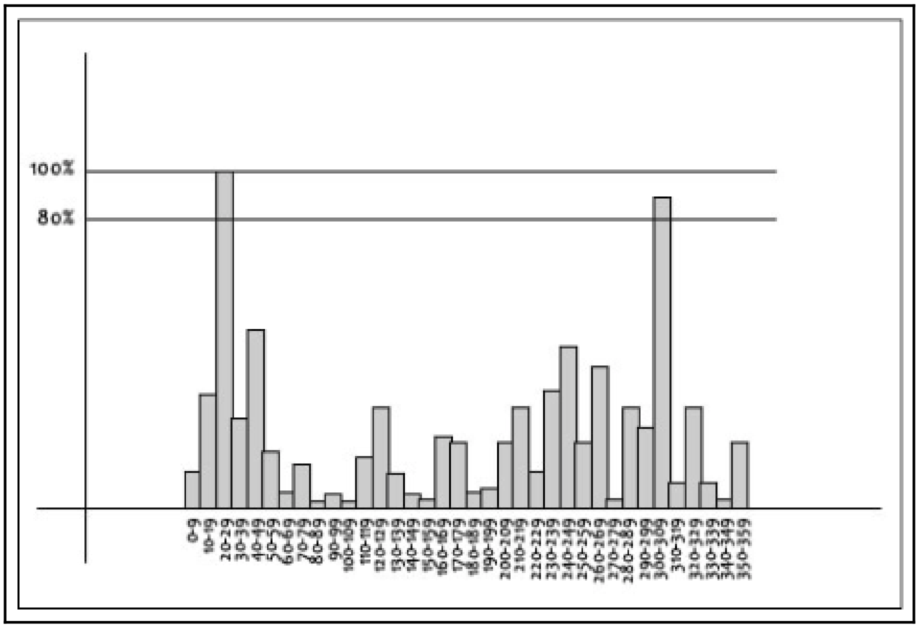


Figure 6: Histogram showing different orientation values

After this has been done for all the neighboring pixels of a particular keypoint, we will get a peak in the histogram. In the preceding figure, we can see that the histogram peaks in the region **20-29**. So, we assign this orientation to the keypoint. Also, any peaks above the **80%** value are also converted into keypoints. These new keypoints have the same location and scale as the original keypoint, but its orientation is assigned to the value corresponding to the new peak.

The fourth and the final stage is the keypoint descriptor.

Keypoint descriptor

Until now, we have achieved scale and rotation invariance. We now need to create a descriptor for various keypoints so as to be able to differentiate them from the other keypoints. To generate a descriptor, we take a 16 x 16 window around the keypoint and break it into 16 windows of size 4 x 4. This can be seen in the following figure:

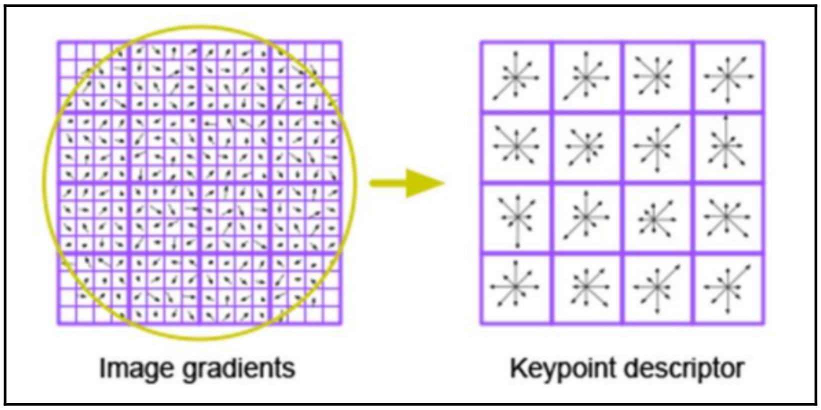


Figure 7: Image showing how the descriptor is computed

We do this in order to incorporate the fact that objects in two images are rarely never exactly the same. Hence, we try to lose some precision in our calculations. Within each 4 x 4 window, gradient magnitudes and orientations are calculated. These orientations are put in an 8-bin histogram. Each bin represents an orientation angle of 45 degrees.

Now that we have a large area to consider, we need to take the distance of the vectors from the keypoint into consideration. To achieve this, we use the Gaussian weighting function.

We put the 16 vectors into 8-bin histograms each, and doing this for each of the 4 x 4 windows, we get 4 * 4 * 8 = 128 numbers. Once we have all these 128 numbers, we normalize the numbers (by dividing each by the sum of their squares). This set of 128 normalized numbers forms the feature vector.

By the introduction of the feature vector, some unwanted dependencies arise, which are as follows:

- **Rotation dependence**: The feature vector uses gradient orientations. So, if we rotate the image, our feature vector changes and the gradient orientations are also affected. To achieve rotation independence, we subtract the keypoint's rotation from each orientation. Thus, each gradient orientation is now relative to the keypoint's orientation.
- **Illumination dependence**: Illumination independence can be achieved by thresholding large values in the feature vector. So any value greater than 0.2 is changed to 0.2 and the resultant feature vector is normalized again. We have now obtained an illumination independent feature vector.

That’s all that is there from the algorithm's point of view. But do we have to write code for all that we have read in the last three pages? Absolutely not. OpenCV provides off-the-shelf functions that can compute all of this internally without the developer having to deal with anything.

Let's see an example code for SIFT using OpenCV:

```
import cv2

image = cv2.imread('image.jpg')
gray= cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
sift_obj = cv2.xfeatures2d.SIFT_create()
keypoints = sift_obj.detect(gray,None)
img=cv2.drawKeypoints(gray,keypoints,image)
cv2.imwrite('sift_keypoints.jpg',image)
```

Only eight lines of code do everything that we discussed so far in this section. We first read the image for which we want to detect SIFT features. We then create `sift_obj` using `cv2.xfeatures2d.SIFT_create`. The point to note here is that SIFT is provided in the `opencv contrib` module only. Using `sift_obj`, we detect keypoints in the image and draw them on the image. The output of the preceding code is shown as follows:

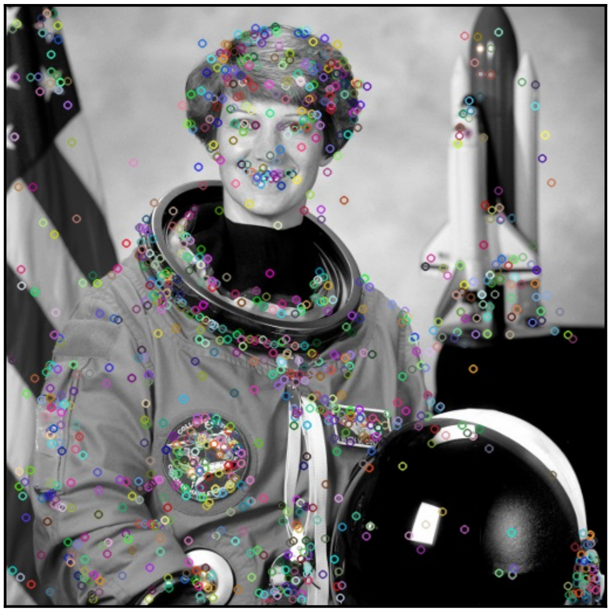


Figure 8: Image with keypoints drawn over the image

Finding keypoints is only half the job done. The final aim is to be able to match keypoints in different images. To be able to do that, we have to make slight changes to the code. The complete code for the same is shown as follows:

```
import cv2
import random

image = cv2.imread('image.jpg')
image_rot = cv2.imread('image_rot.jpg')
gray= cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
gray_rot = cv2.cvtColor(image_rot,cv2.COLOR_BGR2GRAY)

sift = cv2.xfeatures2d.SIFT_create()

kp, desc = sift.detectAndCompute(gray,None)
kp_rot, desc_rot = sift.detectAndCompute(gray_rot, None)
```

```
# BFMatcher with default params
bf = cv2.BFMatcher()
matches = bf.knnMatch(desc,desc_rot, k=2)

# Apply ratio test
good = []
for m,n in matches:
    if m.distance < 0.4*n.distance:
    good.append([m])

# Shuffle the matched keypoints
random.shuffle(good)
# cv2.drawMatchesKnn expects list of lists as matches.
image_match =
cv2.drawMatchesKnn(image,kp,image_rot,kp_rot,good[:10],flags=2,
outImg=None)

cv2.imwrite('sift_matches.jpg',image_match)
```

This is very similar to the previous code; it's just that we do the same thing for two images. After calculating keypoints for both the images, we use the brute force matcher to match the keypoints with each other. Just for the sake of simplicity, we randomly shuffle the matched points and pick 10 points from that to display. The following figure shows the output of the code:

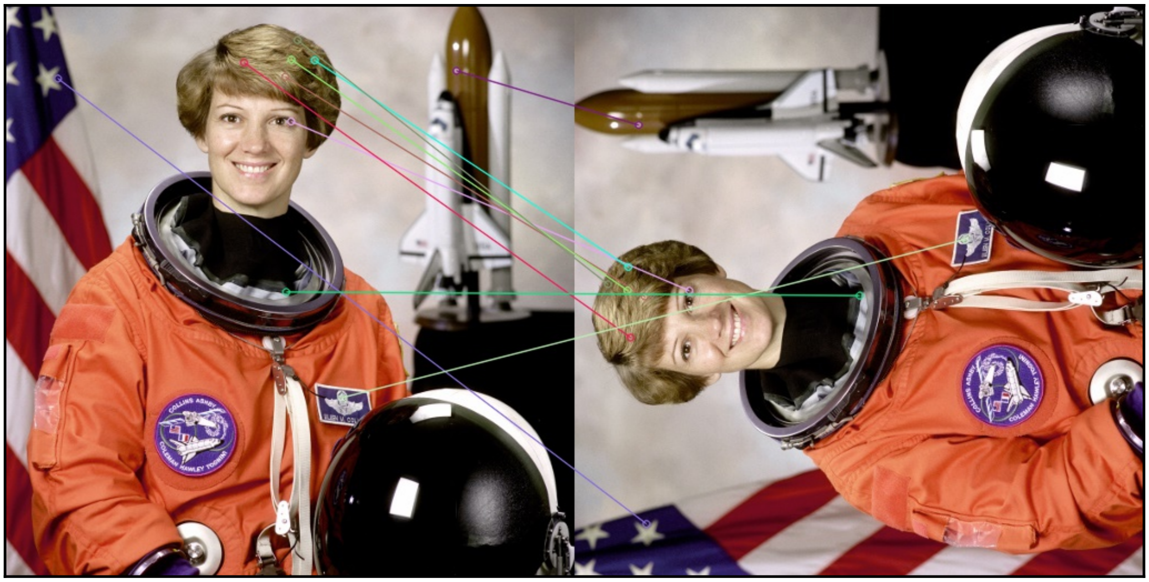


Figure 9: This shows keypoint matching between two images

Using the matching technique, we can find out if similar objects exist in two different images. This can be used to build simple image search applications.

Like SIFT, there is another algorithm called Speeded Up Robust features. This algorithm tries to fill in some gaps left by SIFT and is faster. We will learn more about this in the next chapter.

Speeded up robust features

Speeded up robust features (**SURF**) was proposed by Herbert Bay, Tinne Tuytelaars, and Luc Van Gool in 2006. Some of the drawbacks of SIFT are that it is slow and computationally expensive. To target this problem, SURF was conceived. Apart from the increase in speed, the other motivations behind SURF were as follows :

- Fast interest point detection
- Distinctive interest point description
- Speeded up descriptor matching

Just like we saw in SIFT, or even ORB for that matter, SURF also aims to achieve invariance in rotation, scale changes, illumination changes, and also any change in the viewpoint. In the coming sections, we will look at how SURF is able to achieve all of these invariances and at the same time be fast and efficient. We will also occasionally draw parallels between SIFT and SURF just to appreciate the difference and how optimization actually happens.

Detecting SURF keypoints

SURF keypoints are computed using concepts similar to Haar wavelets. Just like in SIFT, we did an approximation by using Difference of Gaussian instead of Laplacian of Gaussian. For SURF, we will use integral images (like for Haar Cascades) to speed up the keypoint detection step. SURF uses a technique called the fast Hessian detector that will be described next.

To select the location and scale of keypoints, SURF uses the determinant of the Hessian matrix. SURF proves that Gaussian is overrated as the property that no new structures can appear while going down to lower resolutions has only been proved in 1D, but does not apply to the 2D case. Given SIFT's success with the log approximation, SURF further approximates log using box filters. Box filters approximate Gaussians and can be calculated very quickly. The following figure shows an approximation of Gaussians as box filters:

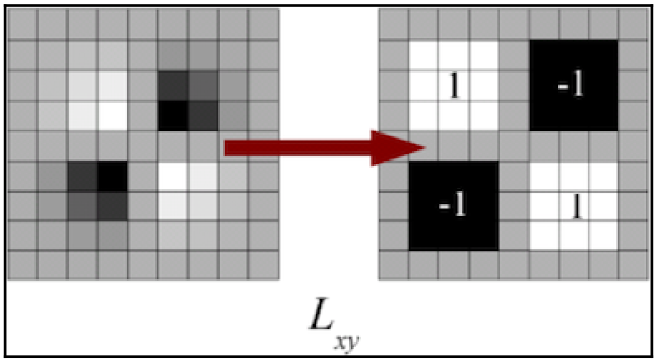


Figure 10: Box filters used for SURF

These are similar to the ones we saw in Haar Cascades for face detection.

Due to the use of box filters and integral images, we no longer have to perform repeated Gaussian smoothing like we did for SIFT. Instead of creating octaves by blurring and resizing the image, we apply box filters of different sizes directly to the integral image. Instead of iteratively down-scaling images, we up-scale the filter size. By doing so, the scale analysis is done using only a single image, making the algorithm faster and easy to implement. The output of the preceding 9 x 9 filter is considered as the initial scale layer. Other layers are obtained by filtering, using gradually bigger filters. Images of the first octave are obtained using filters of size 9 x 9, 15 x 15, 21 x 21, and 27 x 27. At larger scales, the step size between the filters should also scale accordingly. Hence, for each new octave, the filter size step is doubled (that is, from 6 to 12 to 24). In the next octave, the filter sizes are 39 x 39, 51 x 51, and so on.

In order to localize interest points in the image and over scales, a nonmaximum suppression in the 26 pixels neighborhood is applied, which is very similar to what we did in SIFT (refer to *Figure 5*). The maxima of the determinant of the Hessian matrix is then interpolated in scale and image space using the method proposed by Brown, and others. Scale space interpolation is especially important in our case, as the difference in scale between the first layers of every octave is relatively large.

After finding the keypoints in the image, we now want to generate descriptors for the keypoints that will help us in matching keypoints between images.

SURF keypoint descriptors

Now that we have localized the keypoints, we need to create a descriptor for each, so as to uniquely identify it from the other keypoints. SURF works on similar principles of SIFT, but with lesser complexity. Bay and others also proposed a variation of SURF that doesn't take rotation invariance into account, which is called U-SURF (upright SURF). In many applications, the camera orientation remains more or less constant. Hence, we can save a lot of computation by ignoring rotation invariance.

First, we need to fix a reproducible orientation based on the information obtained from a circular region centered about the keypoint. Then, we construct a square region that is rotated and aligned based on the selected orientation, and then we can extract the SURF descriptor from it.

Orientation assignment

In order to add rotation invariance, the orientation of the keypoints must be robust and reproducible. For this, SURF proposes calculating Haar wavelet responses in the x and y directions. The responses are calculated in a circular neighborhood of radius 6s around the keypoint, where s is the scale of the image (that is, the value of σ). To calculate the Haar wavelet responses, SURF proposes using a wavelet size of 4s after obtaining the wavelet responses and weighing them with a Gaussian kernel ($\sigma = 2.5s$) centered about the keypoint; the responses are represented as vectors. The vectors are represented as the response strength in the horizontal direction along the abscissa, and the response strength in the vertical direction along the ordinate.

All the responses within a sliding orientation window covering an angle of 60 degrees are then summed up. The longest vector calculated is set as the direction of the descriptor:

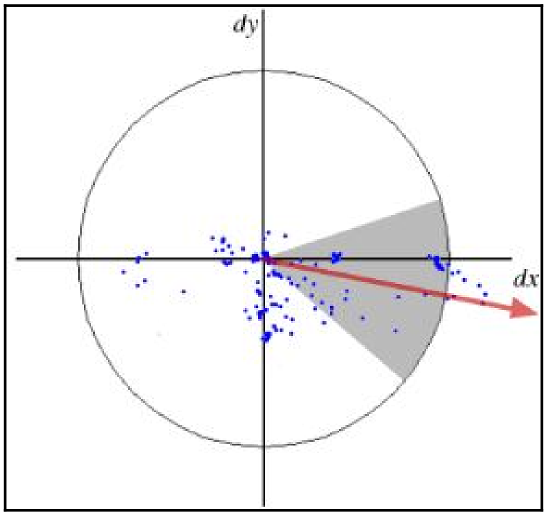


Figure 11: The orientation assignment for SURF keypoints

The size of the sliding window is taken as a parameter, which has to be calculated experimentally. Small window sizes result in single dominating wavelet responses, whereas, large window sizes result in maxima in vector lengths that are not descriptive enough. Both result in an unstable orientation of the interest region. This step is skipped for U-SURF, as it is doesn't require rotation invariance.

Descriptor based on Haar wavelet response

For the extraction of the descriptor, the first step consists of constructing a square region centered around the interest point and oriented along the orientation selected in the previous section. This is not required for U-SURF. The size of the window is 20. The following steps are taken to find the descriptors:

1. Split the interest region into 4 x 4 square subregions with 5 x 5 regularly spaced sample points inside.
2. Calculate Haar wavelet responses *dx* and *dy* (*dx* = Haar wavelet response in *x* direction; *dy* = Haar wavelet response in *y* direction. The filter size used is 2 s).
3. Weight the response with a Gaussian kernel centered at the interest point.

4. Sum the response over each subregion for *dx* and *dy* separately, to form a feature vector of length 32.
5. In order to bring in information about the polarity of the intensity changes, extract the sum of the absolute value of the responses, which is a feature vector of length 64.
6. Normalize the vector to the unit length.

The wavelet responses are invariant to a bias in illumination (offset). Invariance to contrast (a scale factor) is achieved by turning the descriptor into a unit vector (normalization).

Experimentally, Bay and others tested a variation of SURF that adds some more features (SURF-128). The sums of dx and $|dx|$ are computed separately for $dy < 0$ and $dy \geq 0$. Similarly, the sums of dy and $|dy|$ are split according to the sign of dx, thereby doubling the number of features. This version of SURF-128 outperforms SURF.

Let's implement all of this using OpenCV and Python. The following code computes SURF features in two images and matches the keypoints:

```
import cv2
import random

image = cv2.imread('image.jpg')
image_rot = cv2.imread('image_rot.jpg')
gray= cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
gray_rot = cv2.cvtColor(image_rot,cv2.COLOR_BGR2GRAY)

surf = cv2.xfeatures2d.SURF_create()

kp, desc = surf.detectAndCompute(gray,None)
kp_rot, desc_rot = surf.detectAndCompute(gray_rot, None)

# BFMatcher with default params
bf = cv2.BFMatcher()
matches = bf.knnMatch(desc,desc_rot, k=2)

# Apply ratio test
good = []
for m,n in matches:
    if m.distance < 0.4*n.distance:
    good.append([m])


random.shuffle(good)
# cv2.drawMatchesKnn expects list of lists as matches.
image_match =
cv2.drawMatchesKnn(image,kp,image_rot,kp_rot,good[:10],flags=2,
outImg=None)

cv2.imwrite('surf_matches.jpg',image_match)
```

OpenCV provides an exact same interface for both SIFT and SURF. As you can see, this code is exactly the same as SIFT from the last section with just one change. Instead of creating a SIFT object, we create a SURF object using `cv2.xfeatures2d.SURF_create()`. Apart from that, the way we match the keypoint between the images is exactly the same.

The following is the output of keypoint matching using SURF:

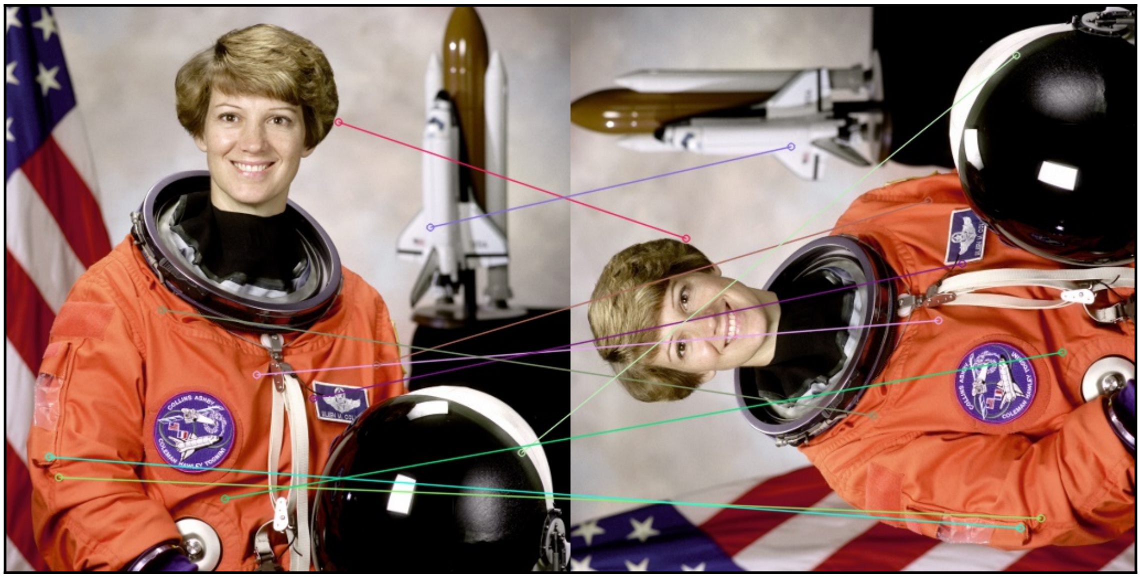


Figure 12: Output showing keypoint matching using SURF keypoints

These were two very famous feature-extraction algorithms. Going forward, more and more machine learning techniques are being used to extract and match features but to be able to fully appreciate them, it is imperative that we understand what traditional image features look like.

Summary

In this chapter, we first learned about different feature extraction algorithms that are useful in different situations. We started with looking at one of the first face detection algorithms that uses Haar Cascades. The concepts used in Haar Cascades, such as integral images and Haar wavelets, are fundamental concepts that are used in many other algorithms such as SURF. Going forward, we understood how SIFT and SURF work and looked at their OpenCV implementations.

In the next chapter, we will expand our OpenCV skills and look at how we can do video processing in real time.

9 Video Processing Using OpenCV

In the previous chapters, we learned how to process images. Let's take this a step further and see how to process videos using the OpenCV library. Videos are nothing but a sequence of images and hence dealing with videos is similar to how we dealt with images with a few exceptions of course. Most of the algorithms that we will see in this chapter will in the end apply on individual images of a video. There are a few algorithms in the field of motion study that deal with pairs of images in a video; for example, optical flow. In this chapter, we will try to understand concepts and techniques such as converting color spaces for videos, detecting objects of a specific color, and finally tracking objects in a video.

We will cover the following topics in this chapter:

- Reading/writing videos
- Operations such as resizing and color space conversion of videos
- Color tracking
- Object tracking

Reading/writing videos

As we read earlier, videos are nothing but a sequence of images taken at very short intervals. Doing this gives the viewer an illusion of continuity. In this section, we will write our own piece of code to record a video using a USB camera/in-built webcam on our laptops.

Reading a video

Reading a video using OpenCV is really simple. You do not have to worry about different formats (mp4, avi, and more) as OpenCV does all the heavy lifting for you. It provides different ways to read a video. You can read a live video using your webcam or use an external USB camera or read a saved video on your computer using an object of the `VideoCapture` module. The `VideoCapture()` constructor takes either an integer or the name of a file. The integer argument is the ID of the camera attached to the computer. Let's look at how to capture a video using a webcam. A webcam has the ID 0; therefore, to read a video from your webcam, pass 0 to `VideoCapture()`. All other USB cameras will have IDs starting from 1:

```
import cv2

cam = cv2.VideoCapture(0)

while (cam.isOpened()):

        ret, frame = cam.read()
        cv2.imshow('frame',frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
                break

cam.release()
cv2.destroyAllWindows()
```

In this code, we use the `read()` function, which returns a Boolean value and a frame of the video. The Boolean value is `true` if the frame was read successfully; otherwise, it is false.

To read a file, look at the following code:

```
import cv2

cam = cv2.VideoCapture("video.mp4")

while (cam.isOpened()):

        ret, frame = cam.read()
        cv2.imshow('frame',frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
                break

cam.release()
cv2.destroyAllWindows()
```

The `VideoCapture` module provides other functionalities such as setting the start point from where to read the video:

```
import cv2

cam = cv2.VideoCapture("video.mp4")

cam.set(cv2.CAP_PROP_POS_FRAMES, 1800) # This will set the start point to
frame 1800

while (cam.isOpened()):

         ret, frame = cam.read()
         cv2.imshow('frame',frame)

         if cv2.waitKey(1) & 0xFF == ord('q'):
                  break

cam.release()
cv2.destroyAllWindows()
```

Later in the chapter, we will look at how to work with images that are read using the preceding code.

Writing a video

To save a video, OpenCV provides the `VideoWriter` class. Let's see how to use this class to save a video by writing some code. For this example, we will first read images from the default webcam in computers and then save it to a file using the `VideoWriter` class.

The following is the code that does the same:

```
import cv2

cam = cv2.VideoCapture(0)
ret, frame = cam.read()

h, w = frame.shape[:2]
fourcc = cv2.VideoWriter_fourcc(*'DIVX')
video_write = cv2.VideoWriter(saved_out.avi', fourcc, 25.0, (w, h) )

while (cam.isOpened()):

         ret, frame = cam.read()
         video_write.write(frame)
         cv2.imshow('video',frame)
```

```
            if cv2.waitKey(1) & 0xFF == ord('q'):
                    break

    cam.release()
    video_write.release()
    cv2.destroyAllWindows()
```

In this code, we first create a `video_write` object of the `VideoWriter` class and the arguments passed to `VideoWriter` are the filename, fourcc, fps, and frame size. The `fourcc` is a four character code used to represent compression formats. After defining the `VideoWriter` object, we read frame by frame and write each frame using the `write()` function to the output file.

To read more about codes, you can follow this link: `https://en.wikipedia.org/wiki/Video_codec`.

Basic operations on videos

Now that we are able to read and write video files using OpenCV, let's look at different operations that can be performed on these videos.

Converting to grayscale

Converting a video to grayscale is very easy. We have already learned how to convert an image to grayscale in the previous chapter. We will use this knowledge to convert each frame to grayscale. Similar to converting to grayscale, you can perform any other operation such as edge detection/contour detection.

The following is the code for converting a video to grayscale:

```
    import cv2

    cam = cv2.VideoCapture(0)

    while (cam.isOpened()):

            ret, frame = cam.read()

            gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            cv2.imshow('gray_frame',gray_frame)
            cv2.imshow('original_frame',frame)

            if cv2.waitKey(1) & 0xFF == ord('q'):
                        break

    cam.release()
    cv2.destroyAllWindows()
```

Figure 1 is the output of the preceding code:

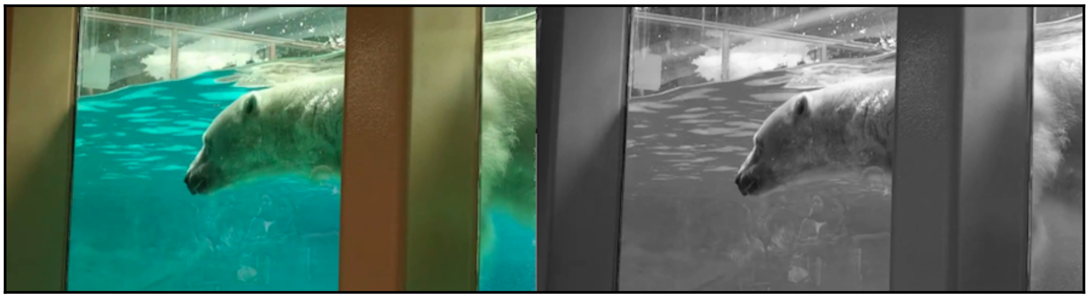


Figure 1: The image on the left shows the original frame and the image on the right is the output of the preceding code

Now it should also be easy to understand how to save a grayscale video (this is left as an exercise for you).

Color tracking

In this section, we will try to understand how to track a color in a video using OpenCV. The following is the code for tracking yellow color in a video:

```
import cv2
import numpy as np

def detect(img):

         lower_range = np.array([40,150,150], dtype = "uint8")
         upper_range = np.array([70,255,255], dtype = "uint8")

         img = cv2.inRange(img,lower_range,upper_range)
         cv2.imshow("Range",img)

         m=cv2.moments(img)
         if (m["m00"] != 0):
                  x = int(m["m10"]/m["m00"])
                  y = int(m["m01"]/m["m00"])
         else:
                  x = 0
```

```
                y = 0

        return (x, y)


cam = cv2.VideoCapture(0)

last_x = 0
last_y = 0

while (cam.isOpened()):

        ret, frame = cam.read()

        cur_x, cur_y = detect(frame)

        cv2.line(frame,(cur_x,cur_y),(last_x,last_y),(0,0,200),5);
        last_x = cur_x
        last_y = cur_y
        cv2.imshow('frame',frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
                break

cam.release()
cv2.destroyAllWindows()
```

In this code, we read the video from the webcam frame by frame and pass the frame to the `detect()` function, which finds the pixel values that lie in the range of yellow. So `lower_range` defines the lower bound to the color we want to detect and `upper_range` defines the upper bound. We use the `inRange()` function to find the pixels within the range of pixel values defined by `lower_range` and `upper_range`. It returns a threshold image, as shown in *Figure 2*, and we use that threshold image to calculate the coordinates of the detected region:

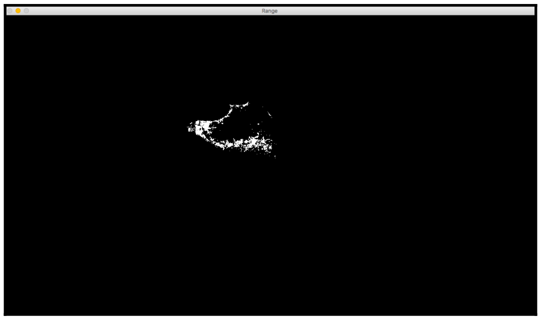


Figure 2: The output of the inRange() function used to find the region within the pixel value bounds

To find the coordinates, we use image moments. An image moment is defined as:

$$\boxed{m_{ji} = \sum_{x,y} \left(array(x,y) \cdot x^j \cdot y^i \right)}$$

Using this, we can calculate the *x* and *y* coordinates using the following formulas:

$$\boxed{x = \frac{m10}{m00}} \quad \boxed{y = \frac{m01}{m00}}$$

So now, after calculating the coordinates, the `detect()` function returns the coordinate and then we use these coordinates to draw a line using the `cv2.line()` function, which shows the path of the color in the video:

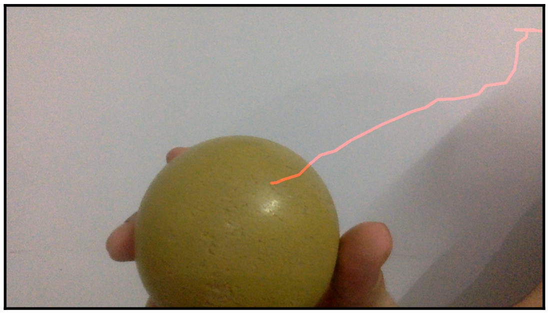


Figure 3: The output of the color tracking code we saw the red color is used to track the object

Tracking color is not always very accurate. It can perform poorly in dim or constantly changing lighting environments. To tackle this problem, let's look at more sophisticated methods of tracking objects that use the concept of motion to track objects.

Object tracking

In this section, we will see how to track an object in a video. The trackers available in OpenCV are:

- KCF
- Lucas Kanade Tracker
- MIL
- BOOSTING
- MEDIANFLOW
- TLD

In our code, we will use **Kernelized Correlation Filter** (**KCF**) to track an object. But you can use any of the aforementioned trackers. There are pros and cons to each of these. From experience, KCF works better for a general case. If you have a lot of occlusions, then you might be better off using the TLD tracker.

Kernelized Correlation Filter (KCF)

How does KCF work? Given the initial set of points, a tracker tries to calculate the motion of these points by looking at the direction of change in the next frame. In every consecutive frame, we try to look for the same set of points in the neighborhood. Once the new positions of these points are identified, we can move the bounding box over the new set of points. There is mathematics involved in making the search faster and more efficient, which is beyond the scope of this book:

```
import cv2

tracker = cv2.Tracker_create("KCF")

cam = cv2.VideoCapture(0)
for i in range(5):
    ret, frame = cam.read()

obj = cv2.selectROI("Tracking",frame)
```

```
ok = tracker.init(frame, obj)

while True:

    ret, frame = cam.read()

    upd, obj = tracker.update(frame)
    if upd:
        x1 = (int(obj[0]), int(obj[1]))
        x2 = (int(obj[0] + obj[2]), int(obj[1] + obj[3]))
        cv2.rectangle(frame[1], x1, x2, (255,0,0))
    cv2.imshow("Track object", frame)

    k = cv2.waitKey(1) & 0xff
    if k == 27 :
        break

cam.release()
cv2.destroyAllWindows()
```

In the preceding code, we have used the `Tracker` class. Using this class, we can use any tracking algorithm by passing it as an argument in the `Tracker_create()` function, which returns an object. Then, we will first read a frame and select the object in the frame that we need to track in the video. Selecting a region in a frame can be done by the `selectROI()` function. Just click and drag over the region to select and press the space key or enter the key as shown in *Figure 4*:

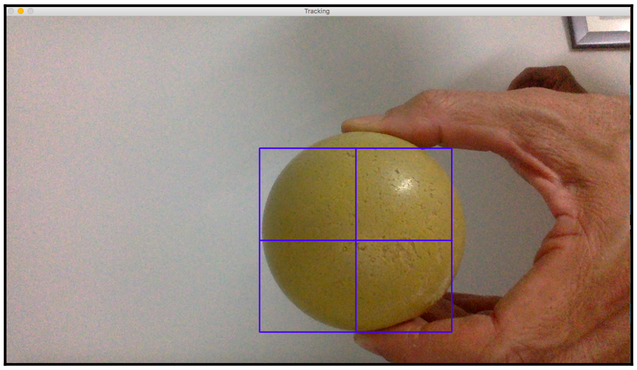


Figure 4: The initial frame where you can select the object you want to track

We store the coordinates of this region in the `obj` variable and initialize the `tracker` object with these object coordinates using the `tracker.init()` function. After initializing the tracker with the object of interest, we read frame by frame and update the position of the object in the new frame by calling the `tracker.update()` function, which returns the new coordinates of the object:

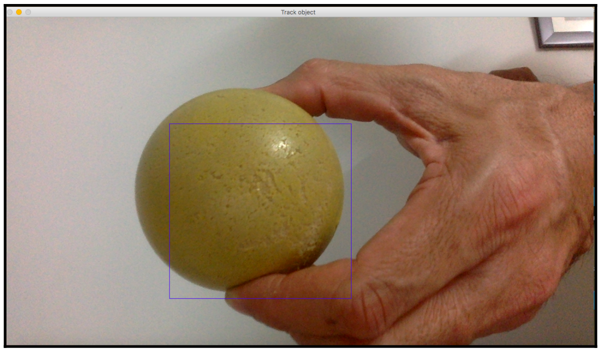


Figure 5 The updated bounding box over the ball that we are trying to track

TIP

The published research paper for Kernelized Correlation Filters is available at `https://arxiv.org/pdf/1404.7584.pdf`.

Let's look at another tracking algorithm, Lucas Kanade Tracker.

Lucas Kanade Tracker (LK Tracker)

The LK Tracker works on the principle that the motion of objects in two consecutive images is approximately constant relative to the given object. Unlike for the KCF Tracker, for the LK Tracker, we will select the points to follow by extracting key points from a given image and we will only follow these key points in the given sequence of images. The reason we do this is--first, it makes the computation faster as we only have to worry about fewer points in the image. Second, tracking the key points in an image is similar to tracking the entire object because of the rigidity of the object.

The following code is an implementation of the LK Tracker:

```
import numpy as np
import cv2

cap = cv2.VideoCapture(0)

# params for ShiTomasi corner detection
feature_params = dict( maxCorners = 1000,
 qualityLevel = 0.3,
 minDistance = 7,
 blockSize = 5,
 useHarrisDetector=1,
 k=0.04)
# Parameters for lucas kanade optical flow
lk_params = dict( winSize = (15,15),
 maxLevel = 2)

# Create some random colors
color = np.random.randint(0,255,(1000,3))

# Take first frame and find corners in it
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)

p0 = cv2.goodFeaturesToTrack(old_gray, mask = None, **feature_params)
# Create a mask image for drawing purposes
mask = np.zeros_like(old_frame)

count = 0 #To keep track of how many frames have been read

while(cam.isOpened()):
 ret, frame = cap.read()
 frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
 # calculate optical flow
 p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None,
**lk_params)
 # Select good points
 good_new = p1[st==1]
 good_old = p0[st==1]
 # draw the tracks
 for i,(new,old) in enumerate(zip(good_new,good_old)):
 a,b = new.ravel()
 c,d = old.ravel()
 mask = cv2.line(mask, (a,b),(c,d), color[i].tolist(), 2)
 frame = cv2.circle(frame,(a,b),5,color[i].tolist(),-1)
 img = cv2.add(frame,mask)
 cv2.imshow('frame',img)
 k = cv2.waitKey(30) & 0xff
 if k == 27:
 break
 # Now update the previous frame and previous points
 old_gray = frame_gray.copy()
 #Recompute the goodFeaturesToTrack as the scene may have changed
drastically
 count = count + 1
 if count % 100 == 0:
 p0 = cv2.goodFeaturesToTrack(old_gray, mask = None, **feature_params)
 else:
 p0 = good_new.reshape(-1,1,2)
cv2.destroyAllWindows()
cap.release()
```

This looks like a complex piece of code but let's try to break it down. We first calculate the feature points in the image using corner detection. We can use any other feature detection algorithm such as SIFT or SURF. The LK Tracker is independent of the feature extraction algorithm. Once we have the feature points ready, we can pass these points into the tracker for tracking. Now, in the while loop, we calculate the feature points again as there is a possibility that the scene might have changed significantly.

Summary

In this chapter, we looked at how to process videos. We started with performing basic operations on videos such as reading/writing. Then, we looked at color space conversion. Along the same lines, the reader can write programs to resize a video, make video with only the edges of a video, and more. Then, we finally looked at tracking objects in a video. First we looked at a very basic technique of tracking objects based on the color, then we looked at more sophisticated techniques such as the Kernelized Correlation Filter and Lucas-Kanade Tracker.

In the next chapter, we will look at how we can build a computer vision service that provides users with the ability to apply computer vision techniques over an image by not actually writing any code themselves. We will build a web interface that will enable users to consume such a service.

10
Computer Vision as a Service

In the recent past, there has been tremendous advancement in cloud computing, which has led to more and more services being provided over the internet. A classic example of this is the music industry. Earlier users had to store songs and music videos on their laptops, but now they are able to stream the same songs over the internet. The growth in the usage of cloud services only means that going forward it would be imperative for a developer to be able to build services that leverage the cloud infrastructure. In this chapter, we will look at how we can build a computer vision service that can take in an input from a user over the internet, process the image, and send the result back to the user in no time.

But why do we need such a service? Advancements in computer vision research has rendered the need for more powerful computers to process images. A lot of times, an algorithm is tweaked in a way that consumes less computational power but at the cost of the quality of the output (we get sub-optimal results). To overcome this hurdle, what if we are able to expose one powerful computer to a lot of users? This will have a two-fold advantage. One where each researcher/developer would not have to spend money to buy expensive hardware, and the second is where it will save the developers the time and effort to implement the algorithm again.

This chapter is broadly divided into the following sections:

- Architecture of the computer vision service
- Environment setup
- Developing a server-client model
- Adding computer vision services (computer vision engine)

Unlike other chapters, in this chapter, we will focus not only on computer vision but also on other aspects of software development such as networking and a server-client model. Let's begin with understanding the overview of the service that we are building.

Computer vision as a service – architecture overview

We are trying to build a service over the internet where a user can upload a picture, select an operation on the image that he wants to perform, and in turn will get the output from the server. The following figure shows an overview of the service:

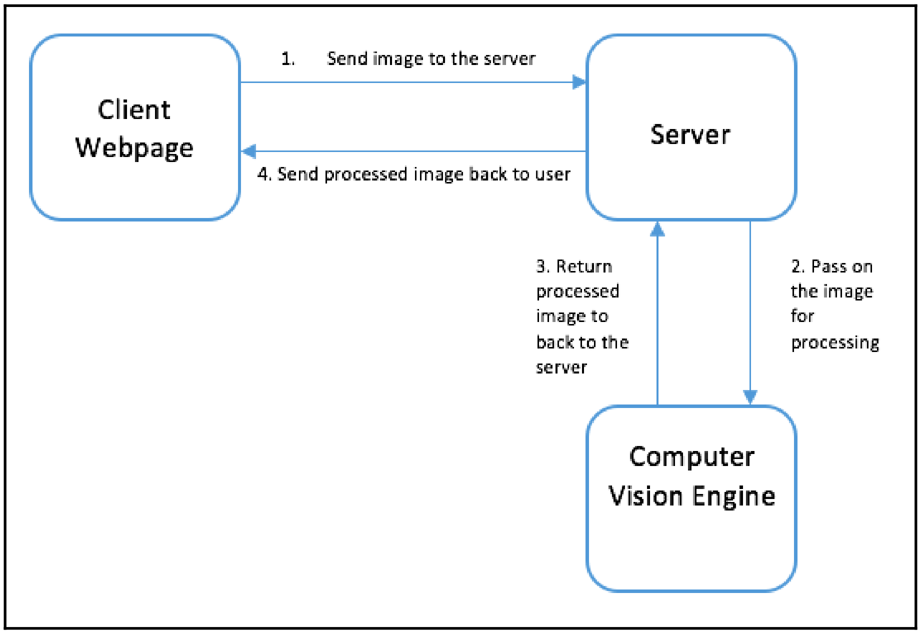

Client Webpage
1. Send image to the server
Server
4. Send processed image back to user
3. Return processed image to back to the server
2. Pass on the image for processing
Computer Vision Engine

Figure 1: The flow of information

In the preceding diagram, we see three boxes (the boxes are labeled clockwise starting with the **Client Webpage** box). The first box is the webpage that the user will interact with. The webpage will provide the functionality to upload an image, select an operation, and send it to the server. From there, the image goes to the **Server** (over the internet), which is represented by the second box. The server's work is to receive the image, determine what operation to perform, and pass on that information to the **Computer Vision Engine**, which is shown in the third box. The **Computer Vision Engine** performs the operation selected by the user on the image and returns the new image back to the **Server** (second box). The server then sends the image back to the **Client Webpage** (first box) and the image gets displayed on the webpage. Throughout this chapter, we will implement each of the the three boxes and in the end have a complete end-to-end system.

Let's set up our environment first before we actually start implementing.

Environment setup

As we saw in the previous section, there are three main parts of the service that we are building. To implement each of them, we need to install tools and libraries other than just OpenCV or scikit-image. The following is the list of libraries that we will be installing, with a brief explanation of why we need them:

- `http-server`: This will serve the web files for the client
- `virtualenv`: This will help in isolating your environment from other libraries on your computer
- `flask`: This will help you to build a server (second box, clockwise in *Figure 1*)
- `flask-cors`: This will help you to allow cross-origin requests to the server

http-server

This tool will start an `http-server` that will serve all the HTML, CSS, and Javascript files that we will build going forward in the chapter. To install, run the following command:

```
$: npm install http-server -g
```

virtualenv

The `virtualenv` is a tool that helps you to isolate your Python environment from other libraries or different versions installed on your computer. We will set up our own virtual environment for this chapter. The following command installs `virtualenv` on your computer:

```
$: pip3 install virtualenv
```

Once we have installed `virtualenv`, we can start making the folder structure for our service. Make a parent folder named `CVaaS` using `mkdir`. Once the folder is created, run the following command:

```
$: virtualenv -p python3 CVaaS
```

This will create a virtual environment in the `CVaaS` folder.

flask

Once we have the virtual environment setup, we will install `flask` within that environment.

Go to the `CVaaS` folder that we created in the last section. Inside the folder, run the following commands:

```
$: source bin/activate
$: pip3 install flask
$: pip3 install flask-cors
```

The first command will activate the virtual environment. Everything that you do going forward will stay within this folder. The next two commands will install `flask` and `flask-cors`. To test whether `flask` was installed properly, run the following commands. If it does not return any errors, the installation was successful:

```
$: python3 -c 'import flask'
$: python3 -c 'from flask_cors import CORS,
cross_origin'
```

We will use `flask` to build our RESTFUL APIs that will help the client to talk to the server.

After installing flask and flask-cors, let's create two more directories, one for the client-side code and one for the server-side code:

```
$: mkdir Client
$: mkdir Server
```

The final structure should look something like this, where the **bin**, **include** and **lib** folders were created by the virtualenv command, the flask folder was created after we installed flask, and the other folders were created by us:

```
.
├── Client
├── Server
├── bin
├── flask
├── include
├── lib
└── pip-selfcheck.json
```

Developing a server-client model

In this section, we will implement the server-side and client-side code and towards the end of the section bring them together. We will leave the computer vision engine as a placeholder and add it in the next section.

Let's begin with writing the client-side code.

Client

The purpose of the client-side code is to provide an interface for the user to upload an image and select an operation. To build this, we will use JavaScript and HTML. To get started, we will make two files, index.html and index.js, in the Client folder that we created earlier:

```
<html>
    <head>
        <script src="index.js"></script>
        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js"></sc
ript>
    </head>
    <body>
        <!-- Create upload image function -->
        <input id="img_src" type="file"/>
        <input id = "load_img_btn" type="button" onclick="loadImage()"
value="Load"/>
        <!-- Drop down menu to select image processing operation -->
        <select id="image_op">
            <option value="to_grayscale">Convert to Grayscale</option>
            <option value="get_edge_canny">Get Edges (Canny)</option>
            <option value="get_corners">Get Corners</option>
        </select>
        <input type="button" id="process_img" value="Process Image"
onclick="processImage()"/>
        </br>
        </br>
        <!-- This canvas is for the original image -->
        <canvas id="local_canvas"></canvas>
        <!-- This is for the processes image -->
        <canvas id='processed_canvas' src=""></canvas>
    </body>
</html>
```

Let's break down the code. We first include the `index.js` and `jquery` file. We will need the `jquery` file to send and receive data from the server. In the body of the webpage, there are two major parts. First is the top row where the user has the option to select a file from the computer and load it onto the webpage. To load the image, we provide a `Load` button next to the file selection option. Next we have a drop-down menu, which lists down all the available operations. Finally, we have the `Process Image` button, which sends the loaded image to the server and waits for a response from the server with the processed image. The following is a screenshot of this part of the webpage:

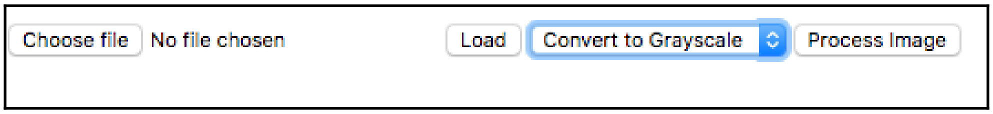


Figure 2: Screenshot of the top part of the webpage

In the next row of the webpage, we have two canvas elements. The first canvas element with the `local_canvas` ID is used to load the image that the user has selected from the computer. The other canvas with the `processed_canvas` ID is used to show the image that was sent back by the server.

For the two buttons that we have put in our webpage, we have called two function names—`loadImage()` and `processImage()`, which are triggered when the buttons are clicked. These functions are implemented in the `index.js` file that is shown next. In your `index.js` file, copy the code given here:

```
var fr; // Variable to store the file reader
var is_img_ready = false;

//Function to load the image from local path to img and canvas
function loadImage() {
    img_src = document.getElementById('img_src');
    if(!img_src.files[0]) {
        alert('Please select an Image first!')
        return;
    }
    fr = new FileReader();
    fr.onload = updateImage;
    fr.readAsDataURL(img_src.files[0])
}

function updateImage() {
    img = new Image();

    img.onload = function() {
        var canvas = document.getElementById("local_canvas")
        canvas.width = img.width;
        canvas.height = img.height;
        var ctx = canvas.getContext("2d");
        ctx.drawImage(img,0,0);
    };
    img.src = fr.result;
    is_img_ready = true;
}

function loadProcessedImage(data) {
    img = new Image();

    img.onload = function() {
        var processedCanvas = document.getElementById('processed_canvas');
        var localCanvas = document.getElementById('local_canvas');
        processedCanvas.width = localCanvas.width;
        processedCanvas.height = localCanvas.height;
        ctx = processedCanvas.getContext('2d');
        ctx.drawImage(img, 0, 0);
    };
    console.log(data);
    img.src = 'data:image/jpeg;base64,' + data;
}

function processImage() {
    if (is_img_ready == false) {
        alert('No image to process!');
        return;
    }

    //Send the image to the server and wait for a response
    canvas = document.getElementById('local_canvas');
    image_data = canvas.toDataURL('image/jpeg');
    img_op = document.getElementById('image_op');
    op = img_op.options[img_op.selectedIndex].value;

    $.ajax({
        url:"http://localhost:5000/process_image",
        method: "POST",
        contentType: 'application/json',
        crossDomain: true,
        data: JSON.stringify({
            image_data: image_data,
            msg: 'This is image data',
            operation: op
        }),
        success: function(data){
            loadProcessedImage(data['image_data']);
        },
        error: function(err) {
            console.log(err)
        }
    });
}
```

That looks like a lot of code. Let's break it down and understand what it does. We first declare two global variables; one is to store the `FileReader` object that we will use later in the code and the other variable `is_img_ready`, which is used to keep a check whether the image has been loaded by the user. This is used as a gatekeeper for the process image functionality. We do not want to send an empty image to the server for processing.

After that, the first function that we see is `loadImage()`. The work of this function is to get the file path set by the user and load the image from that path to the canvas (`local_canvas` ID). We use the `FileReader` object to get the file from the local filesystem of the user. In the same function, when the `FileReader` object is completed loading the image, we call the `updateImage` function. This is a helper function for the `loadImage()` function. We need to create a helper function because updating the canvas is an async task, which means we need to wait for the `FileReader` object to get the entire image from the filesystem and only then can we update the canvas. The `updateImage` function is where we actually update the canvas.

The next function that we will look at is the `processImage()` function. This function is responsible for sending the image to the server. Sending an image to the server is not trivial. We first convert the image to base64 encoding and send that encoded image over the network. Then, after receiving the image, the server selects the image and uses it. `image_data = canvas.toDataURL('image/jpeg');` is the line that returns the image in `base64` encoding. From the `select` tag, we extract the selected operation. Once we have the encoded image and the operation, we send the data over to the server using an AJAX POST request. We form the AJAX request by passing the `url`, `method`, `contentType`, and the actual `data`. The following is the snippet showing the AJAX request to the server. It is very important to follow the correct naming conventions between the client and the server while exchanging data. If the names are not consistent, nothing will work correctly:

```
$.ajax({
    url:"http://localhost:5000/process_image",
    method: "POST",
    contentType: 'application/json',
    crossDomain: true,
    data: JSON.stringify({
        image_data: image_data,
        msg: 'This is image data',
        operation: op
    }),
    success: function(data){
        loadProcessedImage(data);
    },
    error: function(err) {
        console.log(err)
    }
});
```

Writing a correct `ajax` request is very important. The first thing we pass is the `url` of the server we are trying to hit. As you will see in the next section, `http://localhost:5000/process_image` is the URL of the REST API that we will develop for our server. Next up, we set the method to `"POST"` (you will read more about this in the next section). Further, we set the `contentType` and `crossDomain` parameter values. It is important to set the `crossDomain` value to `true`; otherwise, you will not be able to hit the server's REST API because of the CORS security constraint. Finally, we write two callback functions for `success` and `error`. The `success` callback function receives the processed image from the server. In that function, we call the `loadProcessedImage()` function, which takes the image sent from the server and displays it on the second canvas (`processed_canvas` ID).

The following is the image of the entire webpage that you just built:

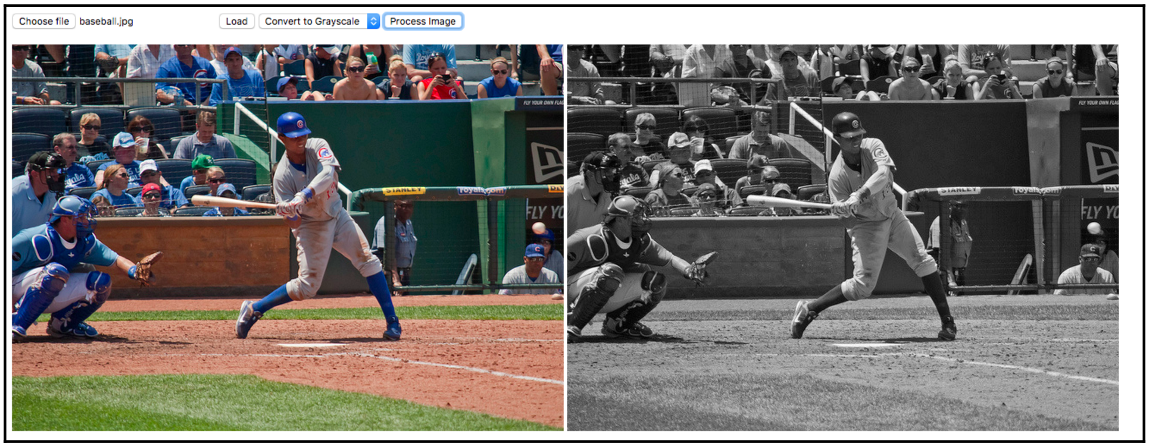


Figure3: The image to the left is the original image loaded by the user, while the image to the right is the one returned by the server

That is all for the client-side code. You can make the webpage look as fancy as you want, but for the purpose of this book, we will keep the webpage to the bare minimum when it comes to aesthetics.

Next is to get the server up and running.

Server

The server in our service is responsible for taking the image from the web client that we built in the previous section and passing it onto the computer vision engine (that we will build in the upcoming section). The computer vision engine will return the processed image back to the server and the server will then send the image back to the client. So to get this functionality working, we need to provide the client with a URL that it can use to send the image and the operation. The most common way of achieving this is through a REST API.

A little about REST APIs! These are HTTP endpoints that are exposed by an `http-server` as a way to contact the server. For example, say our `http-server` is running with the address `http://www.cvaas.com` (this is a hypothetical address and does not exist in real life), then we can build a REST API on that server with an address `http://www.cvaas.com/process_image`. What this means is that whenever a client hits this URL, we will perform the task of processing the image. Say, we build another REST API with an address `http://www.cvaas.com/is_server_running`. Whenever a client hits this URL, we will return irrespective of whether the server is running or not. The advantage of using REST APIs is that we can expose multiple functionalities from a server without affecting each other. This makes the architecture clean and easy to manage. In the AJAX request that we saw in the last section, the URL that was provided in the request was nothing but the REST API exposed by the server that we are going to build in this section.

It is now time for us to implement our own REST API using `flask`. The `flask` is a micro framework for building web applications using Python.

First, make an `app.py` file in the `Server` folder that we created at the beginning of the chapter. But before writing the final sever, let's write a sample server just to understand how `flask` works. The following is the sample code for a simple `'Hello World'` server:

```
from flask import Flask
from flask_cors import CORS

app = Flask('CVaaS')
CORS(app)

@app.route('/')
def index():
    return 'Hello World'

if __name__ == '__main__':
    app.run(debug=True)
```

Let's break down the code and understand what each function does. To begin with, we first `import Flask` and create a flask app using `app = Flask('CVaaS')` and name it `'CVaaS'`. The name of the application is not important for this example, but it is always good to have meaningful names. In the next line, allow cross-origin requests using `CORS(app)`. It is very important to set CORS, because you will not be able to hit the REST API from webpage that is hosted on a different server. If your `http-server` and the flask server are hosted within the same URL, setting CORS is not required. Once we have the flask application running, to test our server, we create a dummy `index()` function. Inside this function, we just return an `'Hello World'` string. If you take a look at it closely, we have a decorator before the `index()` function `@app.route('/')`. This is the most important part of our server. What this line does is that it tells the flask `app` to execute the `index()` function whenever someone hits a `'/'` from a webpage.

To start the server, run the following command in the Terminal:

```
$: python3 app.py
```

You should see something like this:

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 206-325-114
```

This means that the Flask server is running successfully on port `5000`. To test the server, open your favourite web browser and type `http://localhost:5000/`. You will see `Hello World` displayed on your browser. This means everything is working fine. Just to play around with this, in the decorator, change `/` to `/test` and hit `http://localhost:5000/test` from your web browser and see the output. You should see the same output that is 'Hello World'.

Let's go ahead and build the complete server. The following is the code for the complete server. Copy this code in the `app.py` file that we created at the beginning of the chapter:

```
from flask import Flask, request
from flask_cors import CORS, cross_origin
import base64
import cv2
import numpy as np

app = Flask('CVaaS')
CORS(app)

def cv_engine(img, operation):
    if operation == 'to_grayscale':
        return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return None

def read_image(image_data):
    image_data = base64.decodebytes(image_data)
    with open('temp_image.jpg', 'wb') as f:
        f.write(image_data)
        f.close()
    img = cv2.imread('temp_image.jpg')
    return img

def encode_image(img):
    ret, data = cv2.imencode('.jpg', img)
    return base64.b64encode(data)

# This is the server to handle requests and get images from client
@app.route('/process_image', methods=['POST'])
def process_image():
    if not request.json:
        return 'Server Error!', 500
    header_len = len('data:image/jpeg;base64,')
    image_data = request.json['image_data'][header_len:].encode()
    operation = request.json['operation']
    img = read_image(image_data)
    img_out = cv_engine(img, operation)
    image_data = encode_image(img_out)
    result = {'image_data': image_data, 'msg':'Operation Completed'}
    return result, 200

@app.route('/')
def index():
    return 'Hello World'

if __name__ == '__main__':
    app.run(debug=True)
```

The overall structure of the code stays the same; in the preceding code, we add new REST endpoints. As previously mentioned, REST APIs make it easier to add or remove REST APIs without having to change much in the overall structure of your code.

In the preceding code, we make a new function `process_image()` and set `@app.route('/process_image', methods=['POST'])` as its decorator. The decorator for this function specifies the URL, which is `/process_image` along with the method for the HTTP request. There are different methods that are used by HTTP. We choose `POST` for this example since we want to send large amounts of data (images) over the network. The `GET` method is useful when the amount of data that we want to transfer is not a lot. Once we have handled the logistics of the REST API, let's look at the actual implementation of the API. Inside the function, we first check whether the request that was sent by the client contains valid JSON data. We are designing the server to work over JSON data. The reader is free to use any other standard for exchanging data between the server and the client. After that we extract the image data and operation to perform from the JSON request. After extracting the image data from the request, we send the data to the `read_image()` function. Remember we had encoded the image to `base64` before sending it? Now we have to decode the image back. The `read_image()` function does exactly that. It takes in the data and decodes the data using `image_data = base64.decodebytes(image_data)`. After decoding the image, it creates a temporary image on the computer and reads it back using OpenCV so that we are able to use the OpenCV API to perform whatever operation we wish to. Once the `read_image` has successfully created an OpenCV image, it returns that to the `process_image()` function from where it was originally called. The `process_image()` function calls the `cv_engine()` function and passes the image and the operation that was requested from the client. We will look into this function later in the section. Assuming that the `cv_engine()` function returns the processed image, our job now is to send the processed image back to the client. We take the processed image and convert it to `base64` encoding using the `encode_image()` function. Finally, we create a `JSON` object with the encoded image and a success message, and send it to the client using the `return result, 200` line. `200` is the response code for a successful HTTP request.

To run the server, run the following command:

```
$: python3 app.py
```

This concludes the flask server needed to build our computer vision service. But we still have to write the computer vision engine that actually performs all the image manipulation and processing.

Computer vision engine

This is the final piece of the puzzle! In the last section, we briefly touched upon the computer vision engine and how it was being called using the `cv_engine()` function. In this section, we will look inside this function. For the sake of simplicity, in the last section, the `cv_engine()` function was capable of performing only one operation, which is to convert the image to a grayscale image. Let's add support for more operations for our service in this function.

First, we will add the capability to compute canny edges in images. The following is the code for this:

```
def cv_engine(img, operation):
    if operation == 'to_grayscale':
        return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    elif operation == 'get_edge_canny':
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        canny_edges = cv2.Canny(gray, 100, 200, 3)
        return canny_edges
    else:
        return None
```

Again, we keep the structure of the code the same and only add a new if condition that checks whether the operation passed by the client is `'get_edge_canny'`. An important thing to note here is that the name of the operation set in the HTML file that we created earlier should match with the operation that we are checking for in the `cv_engine()` function. Just to recall, the following is the code snippet where we had set the `operation` name:

```
<!-- Drop down menu to select image processing operation -->
<select id="image_op">
    <option value="to_grayscale">Convert to Grayscale</option>
    <option value="get_edge_canny">Get Edges (Canny)</option>
    <option value="get_corners">Get Corners</option>
</select>
```

Let's try to add a new operation, and compute Sobel edges. There are two places where we need to make changes. The first place is the HTML file for the client. We will add a new `option` tag in the preceding code snippet as follows:

```
<option value="get_edge_sobel">Get Edges (Sobel)</option>
```

The second place is where we have to add the `cv_engine()` function in the server code. We will add a new `elif` condition in the code. The addition will be as follows:

```
elif operation == 'get_edge_sobel':
    gray = cv2.cvtColor(cv2.COLOR_BGR2GRAY)
    x_edges = cv2.Sobel(gray, -1, 1, 0, ksize = 5)
    y_edges = cv2.Sobel(gray, -1, 0, 1, ksize = 5)
    edges = cv2.addWeighted(x_edges, 0.5, y_edges, 0.5, 0)
    return edges
```

By adding the preceding code, we have added Sobel edge detection capability.

The following is the output for the canny operation working end to end:

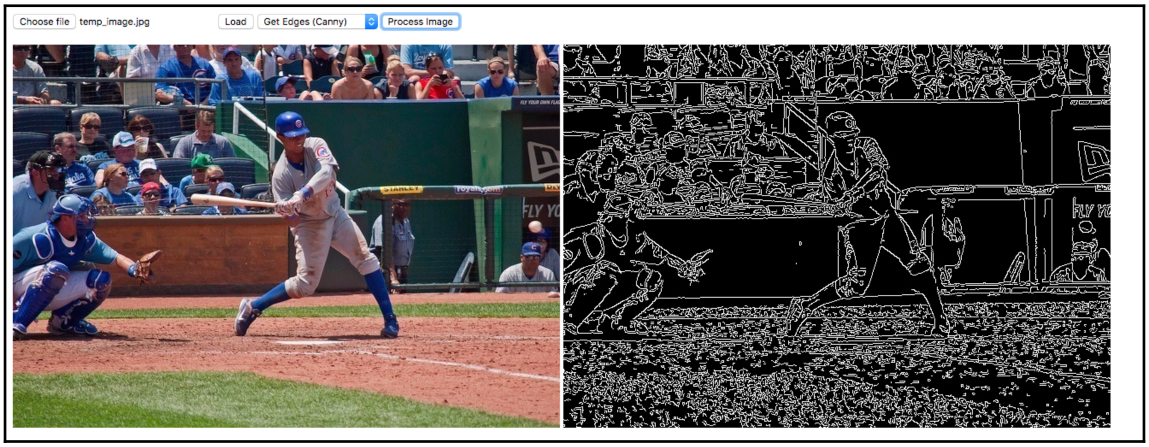


Figure 4: Example of a working canny edge detection

We can add as many computer vision capabilities as we want. We just have to add more conditions in the `cv_engine()` function and also to the HTML file. The modular structure of the code helps in adding and removing functions without disturbing any other function of the service. This is an important quality of a cloud-based service as it keeps the downtime of the service due to a buggy code minimum.

Let's implement a more complicated operation such as detecting corners. The overall structure of the code stays the same. We will add a new `elif` statement in the `cv_engine()` function and within the `elif` statement we will add the following lines of code. This code is similar to the code for Harris Corner that we saw earlier in `Chapter 3`, *Drilling Deeper into Features - Object Detection*, with one change towards the end.

After computing the corner pixels in the image, we plot the corner pixels over the image, unlike in `Chapter 3`, *Drilling Deeper into Features- Object Detection,* where we used `matplotlib` to superimpose corner pixels on the image while showing the image. We cannot do that here because we want to send all the information along with the image. So, we draw the corner points over the image:

```
#Compute the Harris corners in the image. This returns a corner measure
response for each pixel in the image
corners = corner_harris(image)

#Using the corner response image we calculate the actual corners in the
image
coords = corner_peaks(corners, min_distance=5)

# This function decides if the corner point is an edge point or an isolated
peak
coords_subpix = corner_subpix(image, coords, window_size=13)
image_corner = np.copy(image)

for corner in coords_subpix:
    if math.isnan(corner[0]) or math.isnan(corner[1]):
        continue
    corner = [int(x) for x in corner]
    rr, cc = circle(corner[0], corner[1], 5)
    image_corner[rr, cc] = 255 # Mark this point in the image with white
color!

print(image)
image = image * 255 + image_corner

return image
```

Another important line to notice here is `image = image * 255 + image_corner`. We multiple the image with `255` because we want to change the scale of the image from 0 to 1, to 0 to 255. After that we add the `image_corner` image, where we drew the corner points. And finally, we send this image back to the client, which has all the corner points marked in white.

The special thing in this example is that the result of the operation is additional data along with the image. We have the original image and then coordinates for the corners, which are not part of the image. Since we are only sending an image back to the client, we have to put all the information within the image. There are other examples of operations such as computing ORB features where we will have to do the same thing.

Another way of handling extra data along with the image is to also send that data back to the client. But this will increase the burden on the client and you will have to write different code to handle different outputs. Sending the information within the image is the easiest and the cleanest solution.

A point to note here is that we are implementing everything in just one function, but this is not advisable when the number of operations keep increasing. Once you realize that this function is getting burdened with a lot of work, it is time to make additional functions that take care of each operation and call those functions within the `cv_engine()` function.

Putting it all together

At this point, we have all the three components ready—the client web page, the flask server, and the computer vision engine. Let's see how they work together and take a look at the bigger picture. We will follow these steps to get everything up and running.

Client

Go to the client folder where we created the `index.html` and `index.js` files. In that folder, using the Terminal, run the following command:

```
$: http-server .
```

Don't forget the dot—this means to run the server using the files in the current folder. If you want to run the server from a different location, you can pass the appropriate path after `http-server`. The `http-server` will start on your localhost using port `8080`. To test the server, open your favorite browser and hit `http://localhost:8080/`. You should see a page similar to *Figure 1*.

Server

Go to the `Server` folder that we created earlier in the chapter. It should have just one file, `app.js`.

To run the Flask server, run the following command using the Terminal:

```
$: python3 app.py
```

Once you run that, you will see that the server is running again on localhost but on a different port (by default, it is `5000`). If you see no errors, it means that the server is running fine.

We have everything running now. You can go to the web page and start playing around with the page that you just built.

Summary

In this chapter, we looked at a very different topic than what we have seen so far in the book. We understood how it is possible to build a service over the internet that can provide computer vision capabilities to users who do not want to write their own code or just want to perform a computation only once (maybe for a research activity). We broke down the entire service into three major parts, the web client, a Flask server, and finally, the computer vision engine. Then, we tackled each part separately by implementing in a way that they have minimum dependencies with each other. Once all the parts were implemented, we brought them all together and because of the way the code was structured, bringing all of them together took no time.

In the coming years, as the cloud infrastructure gets more advanced, we will see more and more such services being provided to users and this chapter will help you get started with how such applications are built.

Index