Neural Network Computer Vision with OpenCV 5

Build computer vision solutions using Python and DNN module

Gopi Krishna Nuti

bpb

Neural Network Computer Vision with OpenCV 5

Build computer vision solutions using Python and DNN module

Gopi Krishna Nuti

Neural Network Computer Vision with OpenCV 5

Build computer vision solutions using Python and DNN module

Gopi Krishna Nuti

www.bpbonline.com

First published: 2024

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55516-961

www.bpbonline.com

Dedicated to

India, that is, Bharat

About the Authors

Gopi Krishna Nuti is an experienced professional with 21 years of experience in IT industry. He has done his B. Tech in Computer Science from Andhra University, M.S in Business Analytics from State University of New York at Buffalo and an Executive MBA from Amrita University, Bengaluru. He has worked extensively in analytics and software development projects and has delivered award winning products and solutions. He has authored a book about Machine Learning and has multiple patents and research papers against his name. He is a faculty at various training events and a guest faculty at various Engineering Colleges in AP and Telangana. He is a member of the board of studies for Geetanjali Institute of Science and Technology. He is currently working as a Data Science Manager at Autodesk, Bengaluru. He also volunteers for MUST Research and is committed to democratizing AI for al. An incorrigible foodie, he is a passionate teacher and is obsessed with demystifying AI for the next generation of Software developers.

About the Reviewer

Charan is a product manager at Microsoft, where he works on developing innovative solutions for various domains. He has a keen interest in computer vision, natural language processing, and large language models, and how they can be applied to solve real-world problems.

Before joining Microsoft, Charan was a product manager at two different startups, where he led the development of products that dealt with intelligent document recognition and data extraction, and with using computer vision for grading agricultural produce. He has a rich experience in managing cross-functional teams, conducting user research, and launching products in different markets.

Charan enjoys astrophotography, bike riding and cooking in his spare time.

Acknowledgement

First and foremost, I express my heartfelt gratitude to mother Gnanaprasunamba.

Next, I express my sincere thanks to my wife Padma Latha and my son Dheeraj for sacrificing their share of my time and encouraging me to keep writing this book. I owe them a lot and hope to be worthy of their affection for me.

I also thank my extended family for their planned and inadvertent influence on my growth.

I would also like to acknowledge the valuable contributions of my colleagues and coworkers in these past two decades who have graciously taught me much.

I am also thankful to the team of BPB Publications for their guidance and patience in dealing with my eccentricities.

Finally, I would like to thank you, my readers, for your support and feedback.

Preface

Welcome to your essential guide to unraveling the complexities of image processing. Whether you are a seasoned developer or a beginner exploring the world of Computer Vision, this book offers a comprehensive journey from the roots of Computer Vision to practical implementation. It goes beyond theory, offering professionals a practical roadmap for integrating Computer Vision into their projects. With detailed discussions, hands-on code examples, and a focus on applications such as face detection and object recognition, this guide is tailored for those aiming to excel in the dynamic landscape of computer vision applications.

Whether you are in machine learning, automation, or image analysis, this book equips you with the skills to revolutionize your approach to visual data. Each chapter provides practical insights and examples, fostering innovation and excellence in your endeavors. Stay ahead of the curve with "Computer Vision using OpenCV DNN".

Chapter 1: Introduction to Computer Vision - traces the historical roots and the fundamental concepts that underpin Computer Vision.

Chapter 2: Basics of Imaging - dives into the essentials of imaging, laying the foundation for understanding image processing techniques.

Chapter 3: Challenges in Computer Vision - Explores the challenges and complexities encountered in real-world Computer Vision applications.

Chapter 4: Classical Solutions - delves into classical solutions, gaining insights into traditional approaches to image processing.

Chapter 5: Deep Learning and CNNs - Uncovers the power of deep learning and Convolutional Neural Networks (CNNs) in the context of Computer Vision.

Chapter 6: OpenCV DNN Module - Navigates the OpenCV DNN module, mastering its functionalities for efficient deep learning-based image processing.

Chapter 7: Modern Solutions for Image Classification - Elevates your skills by implementing modern solutions for image classification using Python and OpenCV.

Chapter 8: Modern Solutions for Object Detection - Discusses cutting-edge techniques for object detection, enhancing your ability to identify and locate objects in images.

Chapter 9: Faces and Text - Delves into the fascinating realms of face detection and recognition, along with optical character recognition.

Chapter 10: Running the Code – Gives detailed instructions on how to setup the runtime environments needed to run the code provided in the book.

Chapter 11: End-to-end Demo - Concludes your journey with an end-to-end demonstration, bringing together the concepts learned throughout the book.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/ehreg50

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Neural-Network-Computer-Vision-with-OpenCV-5.**

In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about

our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

Chapter 1
Introduction to Computer Vision

Introduction

In a world where computers and cameras communicate seamlessly, the discipline of computer vision emerges as a profound domain. Envision a scenario where your computer assumes the role of an astute companion with the remarkable ability to comprehend visual data, much akin to your comprehension of textual content. Computer vision, in essence, imparts the capacity to perceive and comprehend the world through the lens of images and videos. It is akin to the endowment of sight and cognitive faculties to your computing machine.

Imagine presenting your computer with an image portraying an endearing feline creature. The computer, although lacking the faculty of perception akin to a human, possesses the competence to process the pixel-level data and decipher patterns and structures. It can discern, for instance, that the presence of pointed ears, fine whiskers, and a luxuriant tail coalesce to form the distinctive visage of a cat. The mechanism underpinning this comprehension is none other than image processing.

Image processing embodies the arsenal of tools with which the computer perfects and enhances the visual information at its disposal. It can effectuate alterations such as color correction, noise reduction, or the refinement of

edges, endowing the depicted cat with even greater clarity and visual appeal.

Computer vision extends its capabilities beyond the identification of cats. It engenders awe-inspiring feats, including enabling autonomous vehicles to navigate roads and evade obstacles. It is proficient at tallying the human presence in a crowd and deciphering handwritten textual content. Furthermore, it furnishes invaluable assistance to medical practitioners in the identification of ailments from radiographic imagery, such as X-rays.

A noteworthy aspect of computer vision is its capacity for continuous learning and adaptation. Analogous to how human cognition improves with exposure and experience, computer vision is enhanced by accumulating additional data and knowledge. This dynamic field, steeped in innovation, imparts augmented intelligence and utility to technology across diverse domains, be it in the realms of healthcare, security, entertainment, or myriad other spheres. Computer vision, in its essence, bestows upon computers the precious gift of vision and comprehension, ushering in a realm brimming with possibilities.

Structure

The chapter will cover the following topics:

- History of computer imaging
- Retrieving information from images
- Image processing
- Representation
- Manipulation
- Flexibility
- Reproducibility
- Digital image processing

Objectives

The objective of this chapter is to introduce the contents discussed in later chapters. This chapter starts with a history of computer imaging and walks

through image representation, processing, and manipulation. The chapter also introduces digital image processing and briefly explains the differences between digital and analog image processing.

History of computer imaging

The history of computer imaging is a fascinating journey that spans several decades. It has evolved from humble beginnings to become an integral part of our daily lives. Let us familiarize ourselves in detail with the history of computer imaging.

The roots of computer imaging can be traced back to the 1950s when computers were in their infancy. Researchers began exploring the idea of using computers to process and generate images. One of the earliest milestones was the development of the **Whirlwind** computer at **Massachusetts Institute of Technology** (**MIT**), which could display simple graphics on a screen. In the 1960s, efforts to digitize images started to gain momentum. Researchers devised methods to convert photographs and other analog images into digital form. NASA played a significant role in advancing computer imaging technology by using digital images in space exploration and remote sensing. The 1970s saw the emergence of early computer graphics. The development of devices like the framebuffer allowed computers to display images directly on screens. Companies like Xerox PARC and Atari contributed to the growth of computer graphics, leading to the development of the first video games and interactive **graphical user interfaces** (**GUI**s). In the medical field, computer imaging found applications in areas like **Computed Tomography** (**CT**) and **Magnetic Resonance Imaging** (**MRI**), revolutionizing diagnostics. These technologies enabled doctors to visualize the human body's internal structures in previously impossible ways.

The advent of personal computers in the 1980s brought about desktop publishing. Applications like Adobe Photoshop and Adobe Illustrator revolutionized image editing and design. The field of computer vision gained momentum during this period. Researchers focused on teaching computers to interpret and understand images, laying the groundwork for facial recognition, object detection, and more. The 1990s saw the rise of digital photography with the introduction of consumer digital cameras. This technology made it easier for individuals to capture and share digital

images. Advances in image sensors, image compression, and storage technologies played a pivotal role in the popularity of digital photography.

The entertainment industry embraced computer imaging for special effects in movies and the development of 3D animation in films like *Toy Story* by Pixar. Video games also evolved with increasingly realistic **computer-generated imagery** (**CGI**).

In recent years, deep learning and artificial intelligence have fueled significant advancements in computer imaging. **convolutional neural networks** (**CNNs**) have revolutionized image recognition and processing. Applications include self-driving cars, facial recognition, medical image analysis, and more.

Today, computer imaging is an integral part of numerous industries, from healthcare to entertainment, and it continues to evolve rapidly. With the growing influence of AI and machine learning, we can expect even more exciting developments in computer imaging in the years to come.

Retrieving information from images

The notion of data being stored in and extracted from images is a significant aspect of computer vision and image processing. Images have been used as carriers of hidden data for various purposes, a technique known as **steganography**. This process involves embedding information within an image so that it is imperceptible to the human eye. This hidden data could be text, files, or other forms of information. Several methods and algorithms like LSB substitution, Discrete Fourier Transform, Discrete Cosine Transform and so on, are used for hiding and retrieving data in images.

However, the field of computer vision is different from steganography. Steganography is used in the field of security and for maintaining secrets. Computer vision is for far more mundane and complex tasks like looking at an image and understanding it. For example, looking at *Figure 1.1*, humans can easily say it is the image of a flag. But how can a computer conclude that? That is the challenge addressed by computer vision. Techniques like statistical analysis, frequency domain analysis, and the like are used for this. Please refer to the following figure:

***Figure 1.1**: A flag on a beach*

Image processing

Image processing is a field of study and practice that involves manipulating and analyzing images to improve their quality, extract information, or make them suitable for various applications. It can be broadly divided into two domains: Analog and digital, each with its own set of techniques and signal processing algorithms.

Analog image processing primarily deals with continuous representations of images, typically in photographs, films, or other analog media. Digital image processing deals with images represented as discrete sets of numbers (pixels) and is the most common form of image processing in the digital age. It involves the use of algorithms to manipulate and analyze images. Let us explore the key differences between digital image processing and analog image processing with simple examples:

Representation

Digital images are represented as a grid of discrete pixels, with each pixel having a specific color or intensity value. These values are quantized, typically in 8-bit (0-255) for each color channel (red, green, blue). For

example, in a digital image, the color of a pixel may be represented as (128, 64, 255), where each number represents the intensity of a color channel. This topic is discussed thoroughly in *Chapter 2*.

Analog images are continuous representations of the scene, like photographs or film. There are no discrete pixels, and the image information is carried by continuous variations in properties like light intensity or color. For example, in an analog photograph, the color is captured by the varying chemical properties of the film emulsion.

Manipulation

In digital image processing, algorithms are applied to the discrete pixel values to enhance, modify, or analyze the image. For instance, you can use a digital filter to blur or sharpen an image, change its brightness, or remove noise. Analog image manipulation involves physical processes. For example, you can place a physical filter in front of a camera lens to change the color balance or use a darkroom technique to control exposure during photo development.

Flexibility

Digital images offer high flexibility because you can easily undo and redo processing steps. If you do not like the result, you can try a different algorithm or adjust parameters without harming the original image. Analog processes are less flexible. Once you apply a physical process to an analog image, it is challenging to revert to the original state. This lack of flexibility can be a limitation.

Reproducibility

Results are highly reproducible in digital processing because algorithms work with precise numerical values. If you apply the same algorithm to the same image multiple times, you will get identical results. However, in analog processing, results may vary due to factors like variations in chemicals or physical conditions. Reproducing the same analog image manipulation can be challenging.

Digital image processing

Digital image processing deals with images represented as discrete sets of numbers (pixels) and is the most common form of image processing in the digital age. It involves the use of algorithms to manipulate and analyze images. Here are some key aspects of digital image processing:

- **Pixel manipulation**: Digital images are composed of pixels, each with a specific color or intensity value. Algorithms can manipulate these values to enhance or modify the image, such as adjusting brightness and contrast.
- **Filtering**: Convolution-based filters are commonly used in digital image processing to perform operations like blurring, sharpening, edge detection, and noise reduction.
- **Transformations**: Techniques like the Fourier Transform and Discrete Cosine Transform are used to analyze the frequency components of an image, which is valuable for tasks like compression and feature extraction.
- **Image enhancement**: Histogram equalization, gamma correction, and contrast stretching are methods to improve the visual quality of an image.
- **Image restoration**: Digital techniques can be applied to restore degraded images by removing noise, deblurring, and correcting distortions.
- **Image compression**: Compression algorithms like JPEG and PNG are used to reduce the size of images for storage and transmission while maintaining acceptable image quality.

Digital image processing is a fundamental component of computer vision, where algorithms are used to interpret and understand images. Tasks include object recognition, face detection, and image segmentation. We shall discuss these in detail in *Chapter 3*.

Signal processing algorithms are the core of digital image processing. They involve mathematical operations on image data to achieve various goals. Some common signal processing algorithms used in image processing include:

- **Convolution**: Used for filtering and feature extraction by applying a convolution kernel to an image.
- **Fourier transform**: Decomposes an image into its frequency components, useful for tasks like image compression and analysis.
- **Wavelet transform**: Provides a multi-resolution representation of an image, which is valuable for image compression and denoising.
- **Histogram equalization**: Adjusts the distribution of pixel values to enhance contrast and improve image visibility.
- **Morphological operations**: Used for image analysis and processing tasks involving shape and structure.
- **Edge detection**: Algorithms like the Sobel and Canny operators identify edges and contours in images.
- **Hough transform**: Used for detecting lines and other geometric shapes in images.
- **Fast Fourier Transform**: A variant of the Fourier Transform optimized for efficiency in computing frequency components.

Conclusion

Digital image processing, with its discrete representation and algorithmic flexibility, has propelled us into a digital age where images can be effortlessly manipulated, shared, and analyzed. Its precision, reproducibility, and ease of storage make it the preferred choice for a wide range of applications, from medical imaging to computer vision. In the next chapter, we shall discuss imaging starting with the very basics and see some sample code for simple image processing algorithms.

Exercises

1. Open any of your favorite photographs in an image editor like MS Paint, Gimpel or Adobe Photoshop. Zoom into any part of the image to the maximum extent the software allows you. See if you can observe the individual pixels that make up the image. The effect will be more pronounced if you use a low-resolution image.

2. Start any image editing software on your machine and open your favorite photograph. Take a close look at the different image manipulation options provided in the software. Imagine how a computer program can be written to achieve that effect.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 2
Basics of Imaging

Introduction

Computer imaging involves the manipulation and analysis of digital images using computer algorithms and software. It encompasses various fields such as computer vision, image processing, and computer graphics. It enables tasks such as image filtering, restoration, enhancement, segmentation, and feature extraction. Image processing algorithms are used to extract information from images, detect objects, and recognize patterns. Computer imaging plays a crucial role in fields as diverse as medical imaging, surveillance, robotics, and entertainment. For instance, surveillance systems utilize computer imaging for object detection, tracking, and behavior analysis. Augmented reality and virtual reality heavily rely on computer imaging to create immersive visual experiences. Facial recognition systems use computer imaging to identify individuals based on facial features. It is utilized in autonomous vehicles for tasks like object detection, lane tracking, and traffic sign recognition. From cultural heritage preservation, remote sensing, manufacturing quality control to entertainment industry, new computer vision plays a crucial role.

Consequently, it becomes vitally important for AI developers to understand how computer imaging techniques can be used. Any such discussion will be incomplete without discussing OpenCV. OpenCV has played an enormous role in making image processing accessible to hobbyist developers. Owing

to its popularity, compatibility with a wide range of languages, and simplicity, OpenCV is becoming a go to approach, even for specialized applications.

Any developer wishing to work with computer images should understand the basic aspects of computer images and how it provides a foundation for working with and manipulating images using various software and programming tools.

Structure

The chapter contains the following topics:

- Pixels and image representation
    - Pixels
- Color spaces
    - Primary colors
    - Additive colors
    - Subtractive colors
    - Grayscale
    - Other color spaces
    - Pixels and color spaces
    - Examples
- Image filetypes
- Video files
    - Images and videos
- Programming for image data
    - A brief history of computer image programming
- OpenCV: History and overview
- Image processing code samples
    - Opening, viewing, and closing image files

- Programming with color spaces
    - Grayscale
    - RGB image

Objectives

The objective of this chapter is to provide a brief preliminary introduction to the basics of computer imaging. The chapter discusses topics like pixels, color spaces, imaging formats, and image processing at a shallow level. Many of these topics are vast and stand-alone, in their regard.

Note: Computer vision is different from computer graphics and image management. Computer vision focuses on developing algorithms and systems that enable computers to understand and interpret visual information. Computer graphics deals with the creation, rendering, and manipulation of images and visual content for display or simulation. Image compression techniques are employed to reduce the file size of images for storage and transmission.

Pixels and image representation

In this section, we will start with the basics of image processing. How is an image represented on a computer? How does a computer even manage a display screen? How are the different shapes and colors on a screen made possible? How can the different images move smoothly and leave the impression of live movements in the movies? We will discuss this briefly here. The domain of computer monitors is quite vast. However, we will touch upon the specific topics relevant to the scope of this book.

Pixels

Pixels are the smallest units in an image. It is a stylish union of the words **picture** and **element**. A pixel is a fundamental building block representing a single point of color and brightness. When combined with other pixels, it creates an image. A pixel is typically represented as a square or rectangular area on a display or image sensor. It holds information about the color and intensity of light that should be displayed or captured at that specific location. In the digital representation of an image, pixels are arranged in a

grid-like pattern, forming rows and columns. Each pixel in this grid has a specific location or coordinates, such as (x, y), which allows for its unique identification and manipulation.

Here is how pixels are used for representing images:

- **Color information**: Each pixel in a digital image stores color information. In the most common case of the RGB color model, a pixel is represented by three color channels: Red, green, and blue. The intensity or brightness of each channel is often represented by a numerical value ranging from 0 to 255 (in an 8-bit system) or a higher bit depth, determining the color of that pixel. By combining the color values of neighboring pixels, a complete image is formed.
- **Spatial arrangement**: Pixels are arranged in a regular grid pattern. The resolution of an image is determined by the number of pixels in the horizontal and vertical dimensions, such as 1920x1080 pixels for Full HD resolution. Higher resolutions generally mean more pixels, resulting in finer detail and sharper images.
- **Image formation**: When pixels are displayed on a screen or printed on paper, they collectively form the image that we perceive. Each pixel emits or reflects light according to its color and intensity values. When viewed together, they create the visual representation of the image.
- **Image manipulation**: Pixels are the basis for various image processing and manipulation techniques. Operations like resizing, cropping, rotating, adjusting brightness and contrast, applying filters, and more are performed on individual pixels or groups of pixels to modify the appearance or content of an image.
- **Image storage**: Digital images are stored as files, and the most common image file formats, such as JPEG, PNG, or BMP, store pixel information along with additional metadata and compression methods. These files store the color values of each pixel, allowing for the reconstruction of the image during display or further processing.

Understanding the concept of pixels is crucial for working with digital images, whether it is capturing, displaying, or manipulating them. Pixels

serve as the elemental units that collectively form the rich visual information we perceive in digital images.

Color spaces

Color spaces in image processing refer to different models or representations that define how colors are represented and encoded in an image. Each color space has its own set of rules and parameters for representing colors. These parameters are used to accurately capture, manipulate, and display colors in various applications, including photography, computer graphics, and image analysis.

Primary colors

We all studied in school that white light can be broken into seven components. These components are remembered using the acronym VIBGYOR (violet-indigo-blue-green-yellow-orange-red). Lord Isaac Newton identified these components and named the colors. However, even during Newton's time, there were questions regarding indigo and whether it was distinct enough to be recognized as a primary component. In later days, the theory of basic components of colors evolved into the trichromatic theory of vision. This theory states that the human eye has three types of color receptors (cones) responsible for color perception. These three types are sensitive to different wavelengths of light, and the combination of their responses allows us to perceive a wide array of colors.

In physics, *primary colors* refer to the set of colors from which all other colors can be derived through additive color mixing. In the physical world, the primary colors are red, green, and blue. All colors in nature can be achieved by mixing these three colors in various proportions. For example, when we see a lemon, we perceive it as yellow not because the lemon itself is yellow in color. The lemon absorbs all wavelengths of light except red and green. This reflected red and green light falls on our eyes and a combination of red and green receptors is stimulated. This combination makes our brain process the colors as yellow. Similarly, perceiving magenta involves the stimulation of both red and blue receptors. Given ahead is a brief physical description of the primary colors:

- **Red**: Red light has the longest wavelength among visible light and is associated with the lowest energy. It corresponds to wavelengths around 620-750 nanometers. When red light is perceived by our eyes, it stimulates the red-sensitive cones in the retina, leading to the perception of red color.
- **Green**: Green light has an intermediate wavelength and energy level, falling between red and blue. It corresponds to wavelengths around 495-570 nanometers. Green light stimulates the green-sensitive cones in the retina, resulting in the perception of green color.
- **Blue**: Blue light has the shortest wavelength and highest energy among visible light. It corresponds to wavelengths around 450-495 nanometers. Blue light stimulates the blue-sensitive cones in the retina, leading to the perception of blue color.

It is important to note that the primary colors in physics differ from the subtractive primary colors used in printing, as well as other color models used in different contexts. The primary colors in physics are specifically related to the trichromatic theory of vision and are used to explain the physiological basis of color perception. By understanding the properties of primary colors and how they combine to form other colors, physicists, optical scientists, and vision researchers can better comprehend the principles of color vision, light absorption, and the behavior of light in various material and systems.

Additive colors

The additive colors (RGB color space) is one of the most widely used color models in image processing, computer graphics, and display technologies. It represents colors by combining three primary colors: **Red** (**R**), **green** (**G**), and **blue** (**B**). In the RGB color space, each pixel in an image is represented by three color channels: Red, green, and blue. The intensity or brightness of each color channel is typically represented by an 8-bit value ranging from 0 to 255, where 0 represents no intensity (dark) and 255 represents maximum intensity (full brightness). By varying the intensities of these three channels, any color can be created. This color space is based on the additive color mixing principle which we will discuss shortly.

The RGB color space has a few important characteristics described as follows:

- **Additive color mixing**: In the RGB model, colors are created by adding different intensities of red, green, and blue light. When all three primary colors are set to their maximum intensity (255, 255, 255), the resulting color is white. Conversely, when all three primary colors are set to their minimum intensity (0, 0, 0), the resulting color is black.
- **Gamut**: The RGB color space defines a specific range of colors that can be represented. This range of colors is often referred to as the *gamut* of the color space. The gamut of the RGB color space depends on the color reproduction capabilities of the specific device or medium used, such as a display or printer.
- **Color mixing**: By combining different intensities of red, green, and blue, it is possible to create various secondary and tertiary colors. For example, equal intensities of red and green produce yellow, while equal intensities of red and blue produce magenta. Different combinations of intensities allow for a wide range of colors to be represented.
- **Color depth**: Color depth is the number of bits used to represent each color channel. For example, an 8-bit color depth allows 256 intensity levels per channel, resulting in a total of 16.7 million possible colors (256^3). Higher color depths, such as 16-bit or 24-bit, provide more precision and a larger number of available colors.

The RGB color space is used in a variety of applications, including digital photography, computer graphics, video processing, and display technologies. It serves as the basis for color representation in many image file formats, such as JPEG, PNG, and BMP, where pixel values are stored as RGB values.

Note that there are variations of the RGB color space, such as Adobe RGB and sRGB, which have slightly different gamut and color characteristics. These variations are designed to accommodate specific needs, such as accurate color reproduction for printing or standardized color spaces for web and digital content.

Subtractive colors

The CMYK color space, also known as the process color model, is primarily used in printing and color reproduction. CMYK stands for **cyan** (**C**), **magenta** (**M**), **yellow** (**Y**), and **key** (**K**), where *key* represents black. Unlike the RGB color model, which uses additive color mixing, the CMYK color model uses subtractive color mixing to achieve a wide range of colors.

In the CMYK color space, one starts with white background and creates the target color by subtracting different amounts of cyan, magenta, yellow, and black pigments. The presence of all four inks at full strength results in a dark, almost black color, while the absence of all inks produces white. By varying the percentages of these four ink colors, a wide range of colors can be reproduced for printing.

Here are some key points about the CMYK color space:

- **Subtractive color mixing**: Unlike the RGB model, where colors are created by adding light, the CMYK model starts with a white background (such as the color of the paper) and subtracts or absorbs certain wavelengths of light to create colors. When all four inks are applied at full strength, they absorb almost all light, resulting in a dark color.
- **Color gamut**: The CMYK color space has a smaller color gamut compared to the RGB color space. This is because the combination of subtractive inks cannot produce the same range of colors as additive light mixing. Consequently, some vibrant and highly saturated colors that can be represented in RGB may not be accurately reproduced in CMYK.
- **Black ink**: The *K* in CMYK represents black ink. It is added to the color model because the combination of cyan, magenta, and yellow inks does not produce a true black. Using black ink reduces the need for large amounts of ink to create a deep black color and helps to improve printing efficiency.
- **Color separation**: In the printing process, the CMYK color model is used to separate an image into four different color plates, one for each ink color. Each plate represents the intensity of the respective ink color for that particular area of the image. These plates are used

in combination during the printing process to recreate the original full-color image.

- **Conversion from RGB to CMYK**: When preparing images for printing, it is often necessary to convert RGB images to the CMYK color space. This conversion ensures that the colors in the image are properly adjusted to match the color capabilities of the printing process and the specific CMYK color profile used by the printer.

It is important to note that when converting from RGB to CMYK, there may be some loss of color accuracy or vibrancy due to the differences in color gamut between the two color spaces. Therefore, it is advisable to preview and make any necessary adjustments to the image before final printing.

The CMYK color space is widely used in various printing applications, including magazines, brochures, packaging, and other printed materials. By accurately representing colors, the CMYK model ensures that the intended colors are reproduced as closely as possible in the final printed output.

Grayscale

Grayscale images, also known as **black-and-white images**, consist of shades of gray ranging from black to white, with no colors in between. Unlike color images that have multiple color channels (such as RGB with red, green, and blue channels), grayscale images have only a single channel representing the brightness or intensity of each pixel. Here are some key points about grayscale images:

- **Single channel**: Grayscale images have a single channel, often referred to as the grayscale channel or luminance channel. Each pixel in the image is represented by a single value, which corresponds to the intensity of light at that particular point. This value typically ranges from 0 (black) to 255 (white) in an 8-bit grayscale image, with intermediate values representing different shades of gray.
- **Absence of color**: Grayscale images lack color information. Instead, they rely solely on the brightness levels to represent the image content. The absence of color simplifies the representation and

interpretation of the image, focusing primarily on the light and dark areas.

- **Simplicity and clarity**: Grayscale images often possess a sense of simplicity and clarity. Without the distraction of colors, they emphasize visual elements such as shapes, textures, and contrast. Grayscale images can be particularly effective in showcasing tonal variations, shadows, and highlights in a scene.
- **Conversion**: Grayscale images can be derived from color images through a process called **a color-to-grayscale conversion**. Various algorithms and techniques can be used to convert color images to grayscale. One common approach is to calculate the luminance of each pixel by considering the weighted average of the red, green, and blue color channels, as human perception is more sensitive to green light.
- **Storage and processing**: Grayscale images require less storage space and computational resources compared to color images. With only a single channel of intensity values, grayscale images occupy less memory and are generally faster to process. This can be advantageous in applications where color information is not necessary, or where storage and processing efficiency are important considerations.

Grayscale images have diverse applications in fields such as photography, printing, medical imaging, computer vision, and more. They can be used to represent depth maps, enhance image contrast, simplify image analysis algorithms, or create a classic black-and-white aesthetic in visual media.

It is worth noting that some grayscale images may also contain additional channels for auxiliary information, such as an alpha channel for transparency or a grayscale mask for image segmentation. However, the core characteristic of grayscale images is the representation of intensity values without color.

Other color spaces

In addition to the color spaces discussed above, there are other color spaces used in computer applications. We shall discuss them briefly here:

- **HSL**: **Hue, saturation, lightness** (**HSL**) color space defines colors based on three parameters: Hue, saturation, and lightness. Hue represents the color's position on the color wheel, saturation represents the intensity or purity of the color, and lightness represents the perceived brightness.
- **HSV**: **Hue, saturation, value** (**HSV**) is similar to HSL but replaces lightness with value. The value parameter represents the perceived brightness of the color, making it more suitable for certain image processing operations like adjusting brightness and contrast.
- **LAB**: **CIE Lab*** (**LAB**) is a device-independent color space that separates color information from brightness information. It consists of three channels: **L*** for lightness, and **a*** and **b*** for color information. LAB color space is often used for color transformations and image analysis tasks.

These are just a few examples of color spaces used in image processing. Each color space has its advantages and applications. Choosing the appropriate color space depends on the specific requirements of the image processing task at hand.

Pixels and color spaces

Pixels and color spaces work together to define the color representation of digital images. Color spaces define the range of colors that can be represented, while pixels store the color information for each specific point in an image. By combining the concepts of pixels and color spaces, it becomes possible to accurately represent, manipulate, and reproduce colors in digital images. Pixels store the color values based on the chosen color space, while the color space defines the available colors and their relationship within the image. Together, they form the foundation for color representation in digital imaging systems. Here is how pixels and color spaces work together:

- **Color representation**: Pixels store color information based on the color space used. In an RGB color space, each pixel holds color values for the red, green, and blue channels. These values determine the intensity or brightness of each primary color, and their combination creates the overall color appearance of the pixel.

Similarly, in a CMYK color space, pixels store color values for the cyan, magenta, yellow, and black channels.

- **Gamut**: Color spaces define the gamut, which represents the range of colors that can be represented within that specific color space. Gamut refers to the set of all possible colors that can be displayed or reproduced. Each pixel's color values are limited to the gamut of the chosen color space, meaning they can only represent colors within that defined range. The gamut is influenced by the properties and limitations of the device or medium used, such as a display, printer, or color profile.
- **Conversion**: When working with images, it may be necessary to convert between different color spaces. Conversion between color spaces involves mapping the color values of pixels from one color space to another while preserving the perceived color appearance as accurately as possible. This conversion ensures that colors are correctly represented when transferring images between devices or systems that use different color spaces.
- **Image processing**: Pixels and color spaces are also important in image processing tasks. When performing operations like color correction, adjusting brightness or contrast, applying filters, or any other image manipulation, the calculations are often performed on the color values of individual pixels within the specified color space. This allows for precise control and modification of the image's appearance based on the color space's characteristics.

Examples

Let us see an example of how pixels and color spaces together can be used to create images, refer to *Figure 2.1*:

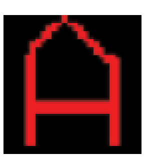
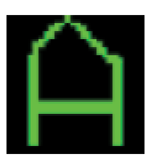
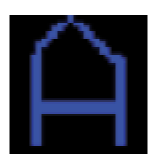
```
[0,0,0,0,0,0,0,0,0,0,0,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,255,255,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,255,255,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,255,255,0,0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,255,255,0,0,0,0,0,0,0,255,255,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0,0,0,0,0],
[0,0,0,0,0,255,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255,0,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255,0,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,0,0,0,0],
[0,0,0,0,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
```

***Figure 2.1**: 2-Dimensional matrix of numbers representing the alphabet A of size 26x26 matrix*

As shown in *Figures 2.1* and *Figure 2.2*, the 2D integer matrix of 26x26 can be rendered as the alphabet *A*:

***Figure 2.2**: The matrix in Figure 2.1 will be rendered above*

Each element in this matrix is a single number and hence the image will be a black and white image also called a grayscale image. Replacing the values of **255** with a vector of **(255,0,0)** will result in a red color **A**, **(0,255,0)** in green **A**, and **(0,0,255)** in blue **A**. In such a case, all **0 elements** shall be **[0,0,0]** and **255** value elements shall be as mentioned in *Table 2.1*:

Red	Green	Blue
(255,0,0)	(0,255,0)	(0,0,255)

***Table 2.1**: White pixels replaced with color channels in the RGB color space*

Image filetypes

So far, we have discussed how pixels and color spaces are used to represent images. Now let us briefly see how this information is stored. The imaging formats and filetypes available are extensive and it is out of scope for this book to discuss them all. Instead, we will only discuss the most popular formats used by regular ordinary computer users and limit ourselves to image processing on those file types.

There are several common file types used for storing image data. Each file type has its characteristics, compression methods, and supported features. The following are some of the most popular image file formats:

- **Joint Photographic Experts Group**: **Joint Photographic Experts Group** (**JPEG**) is a widely used lossy compression format suitable for storing photographs and natural images. It achieves high compression ratios by discarding some image details that are less perceptually significant. JPEG supports millions of colors and is used commonly for web images and digital photography. However, repeated editing or re-saving in JPEG format can result in quality degradation due to the lossy compression.
- **Portable Network Graphics**: **Portable Network Graphics** (**PNG**) is a lossless compression format that supports both full-color and indexed images. It is well suited for images with sharp edges, solid areas of color, or transparency. PNG files maintain a high level of detail and quality without the lossy compression artifacts found in JPEG files. They are commonly used for web graphics, logos, and images with transparency.
- **Graphics Interchange Format**: **Graphics Interchange Format** (**GIF**) is a lossless compression format that supports animated images and indexed color. It uses a limited color palette of up to 256 colors, making it suitable for simple graphics, icons, and animations.

GIF also supports transparency, allowing pixels to be fully transparent or fully opaque. However, GIF has a relatively low color depth, making it less suitable for complex or photographic images.

- **Tagged Image File Format**: **Tagged Image File Format** (**TIFF**) is a versatile file format that can support both lossless and lossy compression. It offers options for storing images with high color depth, multiple layers, and metadata. TIFF is commonly used in professional photography, graphic design, and printing industries. It provides flexibility and maintains image quality. However, it typically results in larger file sizes compared to formats like JPEG or PNG.
- **Bitmap**: **Bitmap** (**BMP**) is a simple and uncompressed file format that stores raw pixel data. It supports various color depths and can preserve high-quality images without compression artifacts. BMP files are typically larger in size, making them less suitable for web usage but often used in specific applications, such as some computer graphics software or as an intermediate format for image editing.
- **RAW**: RAW formats are proprietary file formats used by digital cameras to store minimally processed image data captured by the camera's sensor. RAW files retain the most information and allow for extensive post-processing adjustments. However, they tend to have larger file sizes and require specialized software for viewing and editing.

These are just a few examples of image file formats commonly used today. Each format has its strengths, considerations for compression, color support, and compatibility with different software applications and devices. The choice of file format depends on factors such as the intended use, desired image quality, level of compression, transparency needs, and compatibility requirements.

Video files

There are several common file types used for storing video data. Each file type has its characteristics, compression methods, and supported features. Here are some of the most popular video file formats:

- **MPEG-4**: **MPEG-4** (**MP4**) is a widely used video file format that employs the MPEG-4 video compression standard. It supports a variety of audio and video codecs, allowing for efficient compression and good video quality. MP4 files are compatible with most media players and devices, making them suitable for streaming, sharing, and storing videos. MP4 files can also include subtitles and metadata.
- **Audio Video Interleave**: **Audio Video Interleave** (**AVI**) is a video container format developed by Microsoft. It can store both audio and video data in a single file. AVI files can support various codecs, making them versatile, but they tend to be large in size and may not have as efficient compression as some newer formats. AVI files are commonly used in older video editing software and for local playback on Windows systems.
- **Matroska Video**: **Matroska Video** (**MKV**) is an open-source multimedia container format that can store multiple audios, videos, and subtitle streams in a single file. It supports a wide range of video codecs and can preserve high-quality video and audio. MKV files are often used for storing **High-Definition** (**HD**) and **Ultra-High-Definition** (**UHD**) video content. They are popular amongst video enthusiasts and media playback applications.
- **QuickTime Movie**: **QuickTime Movie** (**MOV**) is a file format developed by *Apple* for storing video, audio, and other media data. MOV files are commonly associated with QuickTime. They support various codecs and multiple tracks. MOV files are widely used in *Apple*'s ecosystem, including macOS and iOS devices. They can contain high-quality video and audio and are suitable for professional applications, video editing, and multimedia content distribution.
- **Windows Media Video**: **Windows Media Video** (**WMV**) is a video file format from *Microsoft*. It is primarily used for streaming and playback on Windows platforms. WMV files can support various codecs and provide good compression for efficient streaming and storage. While WMV files are well-suited for Windows-based

systems, they may have limited compatibility with other platforms and devices.

- **Flash Video**: **Flash Video** (**FLV**) is a video file format primarily associated with *Adobe Flash Player*. It supports streaming and playback of video content over the internet. FLV files use the **Sorenson Spark** or **VP6 codec** and can provide efficient video delivery. However, due to declining support for Flash technology, FLV has become less common and is being replaced by other formats like MP4 for web video playback.

These are just a few examples of video file formats commonly used today. Each format has its strengths, considerations for compression, compatibility, and features. The choice of file format depends on factors such as intended use, video quality requirements, target platforms, streaming capabilities, and compatibility with playback devices or editing software.

Images and videos

Video files and images are closely related, as videos are essentially a sequence of images played in rapid succession. Videos are a collection of frames or still images presented one after another to create the illusion of motion. Here are some noteworthy points about video files and images:

- **Frame-by-frame structure**: A video is composed of a series of frames, with each frame representing a single image. Each frame captures a snapshot of the scene at a specific moment in time. These frames are played back in sequence at a rapid rate (usually 24 to 30 frames per second) to create the perception of continuous motion.
- **Image compression**: Video files use various compression techniques to efficiently store and transmit the sequence of frames. Compression reduces the file size by encoding the differences between frames, removing redundancies, and optimizing the storage of pixel data. Different video file formats employ different compression algorithms to balance file size and video quality.
- **Keyframes**: In video compression, keyframes (also called **I-frames**) are complete and self-contained frames that can be decoded

independently. Keyframes serve as reference points in the video sequence, and subsequent frames (known as **P-frames** or **B-frames**) store only the changes or differences from the previous frames. This compression technique significantly reduces the file size by avoiding the need to store every pixel per frame.

- **Playback**: Video files can be played back on various devices and platforms. Media players, whether software or hardware-based, decode the frames of the video file and display them in rapid succession. The frames are reconstructed, and the illusion of motion is created by displaying the frames at the intended frame rate.
- **Editing and processing**: Video editing software allows the manipulation and processing of individual frames within a video file. Editors can extract frames from a video, apply filters or effects to specific frames, rearrange the order of frames, or even replace frames with different images. This flexibility enables precise control over the visual content of the video.
- **Exporting still images**: Video files can also be exported or saved as individual image files. By extracting frames from a video file, one can obtain still images at specific points in the video. This feature is useful for creating thumbnails, generating promotional material, capturing key moments, or analyzing individual frames for visual analysis or computer vision applications.

Programming for image data

Now that we are familiar with pixels, images, color spaces, and image files, let us understand how to programmatically work with images. In this section, we shall discuss how to open, close and view images using programming. After a brief introduction to the history of programming approaches for manipulating images, we shall start a conversation about OpenCV, the most popular programming library for image processing.

A brief history of computer image programming

Many developers initially tend to think that manipulating images is a unique software problem. This is not true. The necessity to manipulate image data predates computers by a long shot. Movies, having emerged as a

phenomenon in the late 19th and early 20th centuries, predated computers by a long shot. And with movies came the need to edit the images and frames for providing a better experience to the audience. Consequently, several image processing techniques were developed for manipulating parameters like brightness, contrast, and so on. All these techniques treated image data as an analog signal and performed signal manipulation using mathematical functions.

The notable difference with computerized image processing is that computer images are digital in nature. While it is certainly possible to convert this digital data into analog mode and perform the same image manipulations, it is far more desirable to manipulate data in the digital domain. Especially when the changes needed are detailed and precise, digital algorithms often outscore the analog algorithms.

Note: If you ever came across an old model television set, you will notice that it has knobs for changing brightness, contrast, sharpness, color and so on. Rotating the knobs adjusts the settings. But there is no computer or GPU sitting behind those knobs. The image is manipulated using analog algorithms to achieve the desired effect.

So, how are images manipulated programmatically? In the 1960s, researchers began exploring digital image processing techniques. Early efforts involved developing algorithms for basic image operations like filtering, edge detection, and noise reduction. However, computer resources were limited, and processing power was low. During the 1970s, advances in computer graphics led to the development of raster graphics systems. These systems used a grid of pixels to represent and display images. Early programming languages like FORTRAN and assembly language were used to write code for image manipulation, but the focus was primarily on graphics rendering rather than image processing. In the 1980s, computer vision and image analysis gained attention as subfields of computer science. Researchers began exploring techniques for understanding and interpreting images. Algorithms for feature extraction, object recognition, and image understanding were developed. Programming languages like C and C++ started gaining popularity for image processing tasks due to their efficiency and low-level control. In the 1990s, image processing libraries and APIs started emerging, providing developers with pre-built functions and

algorithms for image manipulation. Examples include *Intel*'s **image processing library** (**IPL**), the **vision interface library** (**VIL**), and libraries like **ImageMagick**. These libraries provided developers with tools for performing image operations, such as filtering, transformations, and color space conversions. However, the explosion of libraries also brought many challenges to the developers. The libraries were often incompatible with one another, too expensive for hobbyist programmers and were feature incomplete. These problems were addressed by OpenCV.

OpenCV: History and overview

Open-Source Computer Vision Library (**OpenCV**) is an open-source computer vision and image processing library that provides a wide range of functions and algorithms. It is designed to offer a comprehensive set of tools for developing real-time computer vision applications. OpenCV was initially developed by *Intel* in 2000 and later released as an open-source project. It is written in C and C++ and has interfaces for various programming languages, including Python, Java, and MATLAB/Octave. OpenCV is cross-platform and runs on *Windows*, *macOS*, *Linux*, *Android*, and *iOS*.

OpenCV offers a vast collection of functions and algorithms that cover various areas of computer vision and image processing. Some key features and functionalities provided by OpenCV include:

- **Image and video I/O**: OpenCV allows developers to read, write, and process images and video frames from files, cameras, and video streams.
- **Image processing**: OpenCV provides functions for common image processing operations like filtering, blurring, resizing, thresholding, morphological operations, and color space conversions.
- **Feature detection and description**: OpenCV supports feature detection algorithms such as Harris corner detection, FAST, SURF, ORB, and SIFT. It also provides methods for feature description and matching.
- **Object detection and tracking**: OpenCV includes pre-trained models and functions for object detection and tracking, such as Haar

cascades, **Histogram of Oriented Gradients** (**HOG**), and deep learning-based approaches.

- **Camera calibration**: OpenCV supports camera calibration to correct lens distortions and obtain accurate camera parameters for 3D reconstruction and augmented reality applications.
- **Machine learning**: OpenCV has a great integration with popular machine learning frameworks like **TensorFlow** and **PyTorch**. It provides tools for training and deploying machine learning models for tasks such as image classification, object recognition, and semantic segmentation.
- **Deep neural networks**: OpenCV has a module called **dnn** that enables working with pre-trained deep learning models, including popular architectures like **AlexNet**, **VGG**, **ResNet**, and **YOLO**.
- **GUI and visualization**: OpenCV includes functions for creating **graphical user interfaces** (**GUI**) and visualizing images, videos, and results using drawing tools, annotations, and overlays.
- **Robotics and embedded systems**: OpenCV is widely used in robotics and embedded systems, providing support for tasks like motion detection, gesture recognition, and autonomous navigation.

OpenCV has a vibrant and active community of developers and researchers. The community provides extensive documentation, tutorials, sample code, and a dedicated forum for discussing OpenCV-related topics. The library is continuously updated and improved, incorporating new algorithms and optimizations. OpenCV can also be integrated with other popular libraries and frameworks, such as **NumPy**, **SciPy**, **Matplotlib**, and **Robot Operating System** (**ROS**). This allows developers to leverage the capabilities of OpenCV alongside other tools and libraries for advanced image processing and analysis tasks.

OpenCV finds applications in various fields, including robotics, surveillance, augmented reality, medical imaging, video analysis, and more. It is used in both research and commercial projects due to its versatility, performance, and extensive feature set. Whether you are a beginner or an experienced computer vision developer, OpenCV provides a powerful and flexible toolkit for working with images and videos, enabling you to

implement a wide range of computer vision algorithms and build sophisticated applications with ease.

Image processing code samples

Having established a thorough theoretical base, it is now time to get into programming. In this chapter, we shall see some programs written in multiple programming languages to achieve some simple image operations. We will also explore basic activities like opening and showing images and so on, in C, C++ and Python using OpenCV. The remainder of the book shall use only Python for developing OpenCV based programs.

Opening, viewing and closing image files

Let us see some sample programs for opening and closing images using OpenCV.

CPP code

To compile and run the program, make sure you have OpenCV properly installed and configured. Explanation on installing OpenCV for C++ programming is out of scope for this book. Follow the given steps after the configuration. Compile the program using the following command:

g++ display_image.cpp -o display_image `pkg-config --cflags --libs opencv4`

Run the compiled program, providing the image path as a command-line argument:

./display_image ../BigBuckBunny_Frames/frame_0.jpg

When you run the program, it will open the specified JPG image file and display it in a window using OpenCV. The following program will wait for a key press before closing the window:

```
1. #include <stdio.h>
2. #include <opencv2/opencv.hpp>
3.
4. int main(int argc, char** argv) {
```

```
5. if (argc != 2) {
6. printf("Usage: ./display_image <image_path>\n");
7. return 1;
8. }
9.
10. // Read the image file
11. cv::Mat image = cv::imread(argv[1]);
12.
13. // Check if the image was read successfully
14. if (image.empty()) {
15. printf("Error opening image file: %s\n", argv[1]);
16. return 1;
17. }
18.
19. // Create a window to display the image
20. cv::namedWindow("Image", cv::WINDOW_AUTOSIZE);
21.
22. // Display the image
23. cv::imshow("Image", image);
24. cv::waitKey(0);
25. cv::destroyAllWindows();
```

```
26.
27. return 0;
28. }
```

We shall not discuss CPP any further in this book. We will confine our discussions to programming in Python.

Python code

This program defines a function **display_image** that takes the path of a JPG image file as input. It uses OpenCV's **imread** to read the image file and **imshow** to display it. When you run the program, it will open the specified JPG image file and display it in a window using OpenCV. The below Python program will wait for a key press before closing the window with **cv2.waitKey(0)**.

Note: The Python environment and required libraries need to be installed on the computer before running this code. The procedure to do that has been explained in Chapter 10 Running the Code. Readers unfamiliar with Python programming are advised to refer to that chapter before executing the following code:

```
1. import cv2
2.
3. def display_image(image_path):
4.     # Read the image file
5.     image = cv2.imread(image_path)
6.
7.     # Check if the image was read successfully
8.     if image is None:
```

```
        print(f"Error opening image file: {image_path}")
        return

    # Display the image
    cv2.imshow("Image", image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

# Example usage:
image_path = " ../BigBuckBunny_Frames/frame_0.jpg"

display_image(image_path)
```

Videos and frames

As discussed earlier, a video is a collection of individual images called frames. Let us see programmatically how each individual frames can be extracted from video files. This program defines a function **save_frames_as_jpg** that takes the input video path and an output directory as arguments. It uses OpenCV's **VideoCapture** to read the video file and **imwrite** to save each frame as a JPG file. When you run the program, it will read the video file, extract each frame, and save them as individual JPG files in the specified output directory. The program will print the number of frames saved once the process is complete.

```
import cv2

```

```
3. def save_frames_as_jpg(video_path, output_directory):
4.     # Open the video file
5.     video = cv2.VideoCapture(video_path)
6.
7.     # Check if the video file was opened successfully
8.     if not video.isOpened():
9.         print(f"Error opening video file: {video_path}")
10.        return
11.
12.    # Read and save each frame as a JPG file
13.    frame_count = 0
14.    while True:
15.        # Read the next frame
16.        success, frame = video.read()
17.
18.        # Check if the frame was read successfully
19.        if not success:
20.            break
21.
22.        # Save the frame as a JPG file
23.        output_path = f"{output_directory}/frame_{frame_count}.jpg"
```

```
24.         cv2.imwrite(output_path, frame)
25.
26.         # Increment the frame count
27.         frame_count += 1
28.
29.     # Release the video file
30.     video.release()
31.
32.     print(f"Frames saved: {frame_count}")
33.
34. # Example usage:
35. video_path = "../BigBuckBunny.mp4"
36. output_directory = "../BigBuckBunny_Frames"
37.
38. save_frames_as_jpg(video_path, output_directory)
```

How about viewing the video? Well, that is achieved by displaying each frame for a brief time. Let us see the code for that as well. Try playing with the input value to the **cv2.waitKey()** function on line 19. It is the number of milliseconds for showing each frame on the screen. Increasing this value will make the video viewing experience choppy.

```
1. import cv2
2.
3. def play_video(filename):
```

```
# Open the video file
video = cv2.VideoCapture(filename)

while True:
    # Read a frame from the video
    ret, frame = video.read()

    # If the frame was not read successfully, exit the loop
    if not ret:
        break

    # Display the frame
    cv2.imshow(‹Video›, frame)

    # Exit the loop if the 'q' key is pressed
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the video capture object and close the OpenCV windows
video.release()
cv2.destroyAllWindows()
```

```
25. if __name__=="__main__":
26.     # Provide the path to your MP4 file
27.     video_file = ‹../BigBuckBunny.mp4›
28.
29.     # Call the function to play the video
30.     play_video(video_file)
```

Programming with color spaces

Now let us look at color spaces and code for playing with them. Here we shall write the code for achieving the effect explained in *Figure 2.1* and *Table 2.1*.

Grayscale

In this example, we create a sample **list_of_lists** with pixel intensity values ranging from **0** to **255**. The **display_image** function is then called with this list as an argument. The resulting grayscale image is displayed using **cv2.imshow**, and the program waits for a key press before closing the window with **cv2.waitKey(0)**.

```
1. import cv2
2. import numpy as np
3.
4. def display_image(list_of_lists):
5.     # Convert the list of lists to a NumPy array
6.     array = np.array(list_of_lists, dtype=np.uint8)
7.
8.     # Create a grayscale image from the array
```

```
    image = cv2.cvtColor(array, cv2.COLOR_GRAY2BGR)

    # Display the image
    cv2.imshow("Grayscale Image", image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

list_of_lists = [
    [0,0,0,0,0,0,0,0,0,0,0,0,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,255,255,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,255,255,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,255,255,0,0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,255,255,0,0,0,0,0,0,0,255,255,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,255,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255,0,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0],
```

```
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
    [0,0,0,0,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,0,0,0,0],
    [0,0,0,0,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
    [0,0,0,0,255,255,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
]

display_image(list_of_lists)
```

RGB image

This program defines a function **display_image** that takes a **list_of_lists** as input and displays it as an RGB image using OpenCV. In the example, we create a sample **list_of_lists** with RGB pixel values for a 26x26 image.

Each inner list represents the RGB values of a pixel. The **display_image** function is then called with this list as an argument. The resulting RGB image is displayed using **cv2.imshow**, and the program waits for a key press before closing the window with **cv2.waitKey(0)**. The following code is for creating a red image seen in the left most cell for *Table 2.1*.

```
1. import cv2
2. import numpy as np
3.
4. def display_image(list_of_lists):
5.     # Convert the list of lists to a NumPy array
6.     array = np.array(list_of_lists, dtype=np.uint8)
7.
8.     # Create a RGB image from the array
9.     image = cv2.cvtColor(array, cv2.COLOR_RGB2BGR)
10.
11.    # Display the image
12.    cv2.imshow("RGB Image", image)
13.    cv2.waitKey(0)
14.    cv2.destroyAllWindows()
15.
16. list_of_lists = [
17.    [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,  0],[0,0,0],
[0,0,0],[0,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]],
```

18. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0, 0],[0,0,0],[0,0,0]],

19. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0, 0],[0,0,0],[0,0,0]],

20. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]],

21. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]],

22. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0, 0,0],[0,0,0],[0,0,0]],

23. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0, 0],[0,0,0],[0,0,0]],

24. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]],

25. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]],

26. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

27. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

28. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0, 0,0],[0,0,0],
[0,0,0]],

29. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

30. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

31. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

32. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

33. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[255,0,0],
[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],

[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],
[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]],

34. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[255,0,0],
[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],
[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],[255,0,0],
[255,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]],

35. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

36. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

37. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

38. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

39. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

40. [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[255,0,0],[255,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0]],

```
    [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]],

    [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],
[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]],

]


display_image(list_of_lists)


```

Conclusion

In this chapter, we discussed the topics of pixels, color spaces, how images are rendered on a computer using pixels and color spaces, relationship between images, video frames and videos. We discussed OpenCV's history and how it came about to be the de facto platform for programming with images. We have also seen coding examples for basic image operations.

Exercises

1. In the file for creating grayscale and colored images, change the values of the individual pixels and rerun the programs. Observe how your changes are rendered visually.
2. In the program for opening an image, open different images like the photos of your friends and family. Observe how the image dimensions like tall, wide, portrait, landscape, and so on, can be understood from the matrix dimensions.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

OceanofPDF.com

Chapter 3
Challenges in Computer Vision

Introduction

Computer vision is an interdisciplinary field at the intersection of computer science and artificial intelligence. The objective of practitioners in this field is to enable machines to perceive and interpret visual information like humans. This vibrant field confronts programmers with a diverse range of challenges, including the complexity of visual data, variability, occlusion, semantic understanding, image classification, object detection, semantic segmentation, 3D reconstruction, depth estimation, and video understanding. Machine learning algorithms, particularly deep learning approaches like **Convolutional Neural Networks** (**CNN**s), have become indispensable tools for addressing these challenges and enabling machines to comprehend and interpret visual information. As computer vision continues to advance, tackling these challenges will pave the way for groundbreaking applications in diverse domains such as autonomous vehicles, robotics, healthcare, and augmented reality. In this chapter, we will explore the challenges inherent in computer vision and delve into various categories of challenges that programmers encounter in this exciting field.

Structure

The chapter discusses the following topics:

- Topics in computer vision
- Complexity in image processing
- Image classification
- Object localization
- Image segmentation
- Character recognition

Objectives

This chapter aims to introduce the various topics in computer vision. By end of the chapter, you should be able to explain the most common challenges in computer vision, their similarities, and differences. You should also be able to identify the appropriate algorithm to be chosen when you encounter a computer vision use case. The chapter will discuss in a considerable detail about the topics and sub-topics of computer vision along with their relationships, similarities, and differences. We will also discuss various terminologies which are of relevance to this context.

Topics in computer vision

As described earlier, computer vision focuses on enabling machines to understand and interpret visual information from images or videos. It involves analyzing and extracting meaningful insights from visual data, mimicking human visual perception. Several key topics and algorithms form the foundation of computer vision. Some of the most popular topics are listed here.

- **Image classification**: As the name suggests, image classification classifies as an image based on its content. A label name is assigned to the image based on its content. Popular deep learning models for image classification include based models like CNNS, AlexNet, VGG, GoogLeNet (Inception), ResNet, and EfficientNet. Transfer learning leverages pre-trained CNN models like ImageNet and fine-tunes the models for specific tasks.
- **Object detection**: Object detection focuses on locating and identifying multiple objects within an image. Well-known algorithms

and models for object detection are **region-based convolutional neural networks** (**R-CNN**s), Fast R-CNN, and Faster R-CNN, **single-shot multibox detectors** (**SSD**) and **you only look once** (**YOLO**).

- **Object localization**: Object localization involves determining the precise location or bounding box coordinates of objects within an image. It is often an integral component of object detection algorithms.
- **Semantic segmentation**: Semantic segmentation aims to label each pixel in an image with a corresponding class label, allowing for fine-grained understanding of object boundaries. Prominent models for semantic segmentation include **fully convolutional networks** (**FCN**), U-Net and DeepLab.
- **Instance segmentation**: Instance segmentation combines object detection and semantic segmentation by identifying and delineating individual instances of objects within an image. Notable algorithms include Mask R-CNN and Panoptic Segmentation.
- **Optical character recognition**: **Optical character recognition** (**OCR**) focuses on extracting text from images or documents. Tesseract is a widely used open-source OCR engine. Additionally, deep learning models, including **recurrent neural networks** (**RNN**s) and attention-based models, have achieved impressive results in OCR tasks.
- **Pose estimation**: Pose estimation estimates the position and orientation of human beings. Notable algorithms include OpenPose, AlphaPose and Mask R-CNN with key points estimation.
- **Video analysis**: Video analysis encompasses tasks such as action recognition, object tracking, and activity detection. Various deep learning models can be used for video analysis, including **3D convolutional neural networks** (**3D CNN**s) and RNNs.
- **Other topics**: Additional topics in computer vision include image registration, depth estimation, image super-resolution, image denoising, and more. The list mentioned here does not include

anything from **Generative AI** (**Gen AI**) which is a complete topic itself.

It is useful to introduce the terms *things* and *stuff* at this point. In the context of computer vision, things and stuff are terms used to differentiate between different types of visual entities or regions within an image or a scene. things typically refer to objects or entities that are distinct and recognizable, such as specific objects, people, animals, or vehicles. Things are typically defined by their individual identities and have well-defined boundaries. On the other hand, stuff refers to regions or areas in an image that do not necessarily have well-defined boundaries or individual identities. Stuff refers to the overall appearance or texture of a scene and represents more amorphous and context-dependent regions.

The task of object detection and recognition in computer vision focuses on identifying and categorizing these things within an image or a video frame. For example, detecting and classifying different types of cars in a traffic scene or recognizing specific objects like a chair, a dog, or a cup in an image. Examples of stuff can include the sky, grass, water, road, walls, and other background elements that do not correspond to specific objects but contribute to the overall visual understanding of a scene.

Note: The distinction between things and stuff is not always rigid. There can be overlaps or ambiguous regions that fall into both categories.

Differentiation between things and stuff helps in understanding and modeling the visual content of images and scenes, enabling a more comprehensive analysis and interpretation of visual data.

Complexity in image processing

In the field of machine learning, images are considered unstructured data. This is because, there is no defined structure that can be assigned to image data. It is impossible to say that a pixel in the top-left corner or a pixel in 22^{nd} row and 34^{th} column shall uniquely identify the contents of the image. This inherent unstructured nature of the data makes image processing an extremely difficult task. Images contain a wealth of data which can be processed by humans quite intuitively. Humans have achieved this ability

over the millions of years of evolution. It is a humungous challenge to impart that same level of knowledge to a computer program.

The unstructured nature of image data gives raise to certain challenges unique to the domain of computer vision. Let us discuss them as follows:

- **Variability in image appearance**: One of the primary challenges in image classification arises from the immense variability in image appearance. Images can exhibit variations in lighting conditions, viewpoints, scales, rotations, occlusions, and background clutter. These variations make it difficult for algorithms to generalize effectively from limited training data and accurately classify unseen images.
- **High-dimensional data**: Images are represented by high-dimensional data, with each pixel contributing to the overall feature space. This high dimensionality poses a challenge for image classification algorithms, as it increases the computational complexity and demands substantial computational resources. Efficient feature extraction and dimensionality reduction techniques are crucial to handle this challenge effectively.
- **Overfitting and generalization**: Overfitting occurs when a model learns the training data very well but performs poorly on unseen data. Due to the complexity of image datasets, overfitting is a significant concern in image classification. Deep neural networks, with their large number of parameters, are particularly prone to overfitting. Robust regularization techniques, such as dropout and weight decay, are necessary to mitigate overfitting and enable better generalization.
- **Limited training data**: The availability of labeled training data plays a pivotal role in image classification. However, collecting and annotating a diverse and extensive dataset can be expensive and time-consuming. Limited training data poses challenges in capturing the full spectrum of variations in real-world images, leading to poor performance on novel examples. Techniques like data augmentation, transfer learning, and domain adaptation can help alleviate the impact of limited training data.

- **Class imbalance**: Class imbalance occurs when certain classes in the dataset have significantly fewer instances than others. This issue hampers the learning process as models tend to be biased towards the majority classes, leading to poor classification performance on minority classes. Strategies such as oversampling, undersampling, and class weighting are employed to address class imbalance and ensure fair representation of all classes during training.
- **Computational resources**: Deep learning-based image classification models often require substantial computational resources, including high-performance GPUs and large memory capacities. Training and fine-tuning complex models can be computationally expensive and time-consuming, limiting the accessibility of advanced image classification techniques to researchers and organizations with adequate resources.
- **Adversarial attacks**: Adversarial attacks aim to fool image classification models by introducing imperceptible perturbations to input images. These perturbations are carefully crafted to deceive the model into misclassifying the image. Adversarial attacks highlight the vulnerability of image classification models and raise concerns about their robustness in real-world scenarios. Developing models that are resilient to adversarial attacks is an ongoing research area in image classification.

Now that we know what image processing and computer vision is about, let us start discussing the specific topics and tasks in the field of computer vision. We shall discuss the most common challenges in computer vision without getting into the field of Generative AI.

Image classification

Image classification is a fundamental task in computer vision that involves categorizing images into different predefined classes or labels based on their content. It aims to teach machines to recognize and differentiate between various objects, scenes, or concepts depicted in images. To use the terms discussed earlier, image classification aims to identify the most prominent *thing* in the given image. No matter how much area is occupied

by *stuff*, image classification aims to focus and identify the **thing** in the image.

Image classification has witnessed tremendous advancements with the rise of deep learning. However, despite the impressive progress, it remains a challenging problem that researchers continue to tackle. A simple example is, if the image has two different things, then how should the image be classified? Also, how to determine what is **thing** and what is **stuff**?

As an example, refer to *Figures 3.1* and *3.2*. Truck is present in both images, but it is more a stuff in *Figure 3.1*. A layman explanation is saying that truck is far away from our point of view in *Figure 3.1*. While this is obvious to humans, how can the computer identify this difference from a 2D matrix of pixels? Please refer to the following figure:

Figure 3.1*: Truck, dog and human where truck is in the background "stuff"*

While in *Figure 3.1*, the human and the dog are things, it is not so obvious in *Figure 3.2*:

***Figure 3.2**: Truck, dog and human where all objects are in the foreground "things"*

Object localization

Object detection is the task of identifying and localizing multiple objects within an image or a video frame. It goes beyond image classification, which focuses on assigning a single label to an entire image. Object detection involves precisely delineating the boundaries of objects and providing information about their positions in the image. Object detection algorithms aim to answer two primary questions: *What objects are present in the image? Where are these objects located?* Object detection plays a crucial role in numerous applications, including autonomous driving, surveillance systems, object tracking, augmented reality, and robotics, where the ability to recognize and locate objects accurately is essential for making informed decisions and taking appropriate actions. To accomplish this, object detection algorithms typically follow a multi-step process:

1. **Region proposal**: Initially, potential regions in the image that might contain objects are identified. Various methods, such as selective search, edge boxes, or **region proposal networks** (**RPN**s), are employed to generate these region proposals.

2. **Feature extraction**: For each proposed region, a set of features is extracted to capture the discriminative characteristics of the underlying objects. Common approaches involve using pre-trained CNNs like VGG, ResNet, or EfficientNet to extract rich feature representations from the region.
3. **Classification**: The extracted features are then utilized to classify each proposed region into specific object categories. This step employs classification algorithms, such as **support vector machines** (**SVM**s), logistic regression, or more commonly, softmax-based classifiers, to assign a class label to each region.
4. **Localization**: Along with classification, object detection also involves accurately localizing the objects within the proposed regions. This typically entails predicting the coordinates of bounding boxes that tightly enclose the objects. These bounding boxes provide information about the object's position, size, and orientation.
5. **Non-maximum suppression**: Since multiple region proposals may overlap or cover the same object, a technique called **non-maximum suppression** (**NMS**) is applied to filter out redundant or overlapping detections. NMS ensures that only the most confident and non-overlapping bounding boxes are retained, resulting in a more accurate and compact set of object detections.

Object detection algorithms differ from image classification in the aspects of localization, ability to handle multiple objects in the image and in their fine-grained analysis. The differences between the two can be tabulated as follows in *Table 3.1*:

	Object detection	**Image classification**
Localization	Identifies the presence of objects and provides precise localization information by specifying bounding boxes around each object.	Focuses solely on assigning a single label to the entire image without localizing specific objects.

	Object detection	Image classification
Handling multiple objects	Handles scenarios where multiple objects of different classes may be present within a single image. Algorithms aim to detect and classify each individual object separately.	Deals with classifying the entire image into a single category.
Level of analysis	Fine-grained analysis is made possible by providing detailed information about the location, size, and shape of objects within an image. This level of granularity enables subsequent tasks like tracking, counting, or interaction analysis.	Algorithms provide only a coarse-grained understanding of the image content without precise object-level details.

***Table 3.1**: Comparison of image classification and object detection algorithms*

As an example, refer to *Figures 3.3* and *Figure 3.4*:

***Figure 3.3**: Classification algorithm (InceptionV3) classifies this as a cat*

Image classification algorithms would classify *Figure 3.3* as the cat, but they will fail to give a satisfactory classification for *Figure 3.4*. Neither the dog nor cat will provide an accurate classification.

***Figure 3.4**: Classification algorithm does not work well. Neither cat nor dog is complete description*

The same images when fed to an object detection algorithm reveal much more detailed information. See *Figures 3.5* and *Figure 3.6* for the additional

information that is revealed when object detection algorithms are run on them. Please refer to the following figure:

***Figure 3.5**: Bounding box information for single object*

Note the bounding boxes which provide the precise location of the aero plane, truck, dog and person. Please refer to the following figure:

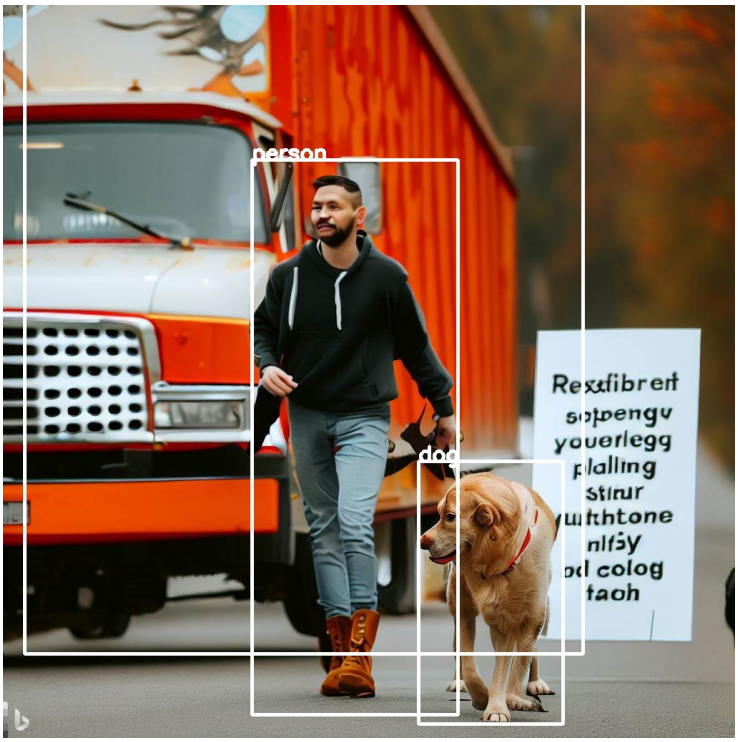


***Figure 3.6**: Bounding box information for multiple objects*

Object detection is crucial in applications that require identifying and localizing specific objects within an image, such as autonomous driving, surveillance systems, object tracking, and augmented reality. It enables

tasks like counting objects, monitoring, and making informed decisions based on the detected objects.

Image segmentation

Image segmentation is the task of partitioning an image into meaningful and semantically coherent regions or segments. It involves assigning a label or category to each pixel in the image, allowing for a fine-grained understanding of object boundaries and their context within the image. It allows for precise delineation of objects, accurate understanding of their spatial relationships, and enables subsequent analysis and decision-making based on the segmented regions.

Although image segmentation and object detection are related and share some similarities, they are two distinct tasks in computer vision. Image segmentation focuses on dividing an image into meaningful regions at the pixel level, while object detection aims to detect and localize specific objects with bounding boxes. Both tasks have distinct goals and applications, with segmentation providing detailed object boundaries and object detection offering object localization and class information. The task of semantic segmentation focuses on labeling and delineating the stuff regions within an image.

The key differences between image segmentation and object detection are listed in *Table 3.2*:

	Image segmentation	**Object detection**
Goal	Partition an image into semantically meaningful regions or segments, where each pixel is assigned a label or category. The focus is on understanding the internal structure and boundaries of objects within the image.	Identify and locate specific objects within an image. It involves recognizing the presence of objects and providing bounding box coordinates around each detected object. The emphasis is on both objec

		t classification and precise localization.
Output	A pixel-level mask or labeling, where each pixel is assigned a specific class or label. The result is a detailed understanding of the object boundaries and the relationships between different regions within the image.	A set of bounding boxes that tightly enclose each detected object along with the class label associated with each object.
Granularity	Fine-grained understanding of object boundaries and regions within an image.	Relatively coarser understanding of the objects in an image.

Table 3.2*: Comparison of image classification and object detection algorithms*

There are several methods and approaches for image segmentation, each with its own characteristics and trade-offs. Here, we will cover some of the commonly used techniques.

- **Thresholding**: Thresholding is a simple and intuitive segmentation technique that separates objects from the background based on pixel intensity values. A threshold value is selected, and pixels with intensity values above or below the threshold are classified as things or stuff, respectively.
- **Region-based segmentation**: Region-based techniques group pixels into regions based on certain criteria such as similarity in color, texture, or intensity. Popular algorithms in this category include Watershed Transform, Mean-Shift clustering, Graph cuts, Edge-based segmentation, Contour-based segmentation, and deep learning-based segmentation. We shall discuss these in detail in the upcoming chapters.

As an example, refer to *Figure 3.7*. Image classification algorithms would classify *Figure 3.3* as dog but they will fail to give a satisfactory classification for *Figure 3.7*. Neither dog nor cat will provide an accurate classification.

Figure 3.7*: Segmentation provides exact masks*

Image segmentation is useful in applications that require a detailed understanding of objects and their boundaries, such as medical image analysis, semantic image editing, and scene understanding. It enables precise analysis of object shapes, segmentation-based tracking, and content-aware image manipulation.

It is apt here to briefly discuss the two sub-topics of image segmentation viz. semantic segmentation and panoptic segmentation.

- **Semantic segmentation**: Semantic segmentation is the task of labeling each pixel in an image with a corresponding class or category label. It aims to partition the image into semantically meaningful regions based on object categories. Unlike traditional image segmentation methods that may assign different labels to different instances of the same object class, semantic segmentation treats all instances of a particular class equally. Semantic segmentation provides a pixel-level understanding of an image, allowing for a detailed analysis of object boundaries and their spatial relationships. It enables applications such as scene understanding, autonomous driving, and image-based reasoning.

- **Panoptic segmentation**: Panoptic segmentation aims to combine both instance-level and semantic segmentation into a unified framework. It provides a comprehensive understanding of an image by simultaneously detecting and segmenting objects at the instance level while also assigning semantic labels to regions that contain groups of objects. In panoptic segmentation, each pixel is assigned a class label for the semantic segmentation aspect, and unique object instances are identified and labeled separately. This means that in addition to providing detailed segmentation masks for individual objects, panoptic segmentation also labels regions where multiple objects are present. The output of panoptic segmentation consists of two components: instance masks for individual objects and semantic labels for groups of objects. These components together provide a complete understanding of the scene, including precise object boundaries and semantic category information. As can be imagined, panoptic segmentation is significantly more challenging compared to semantic segmentation.

Character recognition

Character recognition, also known as **optical character recognition** (**OCR**), is a branch of computer vision that focuses on the automatic extraction and recognition of text characters from images or scanned documents. The goal is to convert visual representations of characters into machine-readable text.

OCR systems typically follow a series of steps to perform character recognition:

1. **Preprocessing**: The input image or document is preprocessed to enhance the quality and clarity of the text. This may involve operations such as noise removal, image normalization, and binarization (converting the image to black and white).
2. **Text localization**: The regions containing text are identified and localized within the image. This step helps isolate the text from the rest of the image content, improving the efficiency of subsequent recognition processes.

3. **Text segmentation**: The individual characters are separated from each other, segmenting the text into its constituent parts. This step is crucial for recognizing each character independently.
4. **Feature extraction**: Relevant features of the segmented characters are extracted to create a representation suitable for recognition. These features can include aspects like shape, stroke width, contour, or texture.
5. **Classification**: Machine learning algorithms, particularly pattern recognition techniques, are employed to classify each character based on its extracted features. Commonly used algorithms include **support vector machines** (**SVM**s), **k-nearest neighbors** (**k-NNs**), and neural networks.
6. **Post-processing**: The recognized characters are further refined and processed to improve accuracy. Techniques such as language modeling, dictionary matching, and error correction algorithms are applied to enhance the accuracy of the recognized text.

As an example, refer to *Figure 3.8*. Both are alphabets. One is hand-written and the other is printed. The purpose of object recognition is to identify the characters that are represented in this figure:

***Figure 3.8**: Printed and hand-written characters*

It would be tempting to think of OCR as an application of image classification or object detection. But the processing is much more complicated than that. Image classification aims to classify entire images into predefined categories, whereas character recognition focuses on recognizing and extracting individual characters or textual elements. Image classification operates at a higher level and considers the overall content or context of an image, while character recognition deals with finer details by recognizing and analyzing individual characters. Unlike image

classification, character recognition provides a sequence of recognized characters or text.

Conclusion

In this chapter**,** we have discussed various topics in computer vision. We start with a discussion on the common challenges in computer vision and why they are challenging. We discussed the importance of things and stuff and how various algorithm families aim to identify them in a given image. We discussed the algorithm families for classifying images, detecting and locating objects in an image, getting granular information contained in an image by using segmentation techniques and finally discussed optical character recognition.

Exercises

1. Take any two images. Using traditional programming techniques (like logical statements, loops and so on) see if you can perform the jobs like detecting foreground, background, identifying the characters in the image, locating the objects in the image etc. Do not panic if you do not achieve good results.

Key terms

- **Image classification**: The task of assigning a label or category to an image based on its visual content.
- **Instance segmentation**: Process of not only identifying and categorizing objects in an image but also distinguishing and segmenting each individual instance of an object with pixel-level accuracy.
- **Object detection**: Process of identifying and localizing multiple objects within an image by drawing bounding boxes around them.
- **Object localization**: Another name for object detection.
- **OCR**: The process of extracting text from images or documents and converting it into machine-readable text.
- **Panoptic segmentation**: Combination of instance segmentation and semantic segmentation.

- **Pose estimation**: The task of estimating the 3D position and orientation of an object or a person in an image or video.
- **Semantic segmentation**: The task of assigning pixel-level labels to an image, classifying each pixel into specific categories to create a detailed understanding of the scene.

Chapter 4
Classical Solutions

Introduction

As discussed in earlier chapters, computer vision is a multidisciplinary field that aims to enable machines to understand and interpret visual information from images and videos. This chapter serves as an overview of the fundamental classical algorithms used in computer vision. These algorithms have stood the test of time and continue to be relevant, even with the advent of deep learning techniques. They provide valuable insights into image analysis, feature extraction, segmentation, motion estimation, and object detection.

While these classical algorithms have been extensively utilized in computer vision, it is important to note that recent advancements in deep learning, particularly **convolutional neural networks** (**CNN**s), have significantly improved the state-of-the-art performance across various vision tasks.

Structure

The following topics are covered in the chapter:

- Solutions for challenges in computer vision
    - Classical solutions
    - Modern solutions

- Algorithm families
    - Morphological operations
    - Thresholding
    - Detecting edges and corners
    - Image transformations
    - Region growing
    - Clustering
    - Template matching
    - Watershed algorithm
    - Foreground and background detection
    - Superpixels
    - Image pyramids
    - Convolution

Objectives

The objective of this chapter is to provide a comprehensive survey of classical computer vision algorithms. We shall highlight their key principles, applications, strengths, and limitations. We will also present a comparative analysis of different algorithms within specific domains. The chapter will demonstrate the practical applications of classical algorithms in real-world scenarios. We will present examples where these algorithms have been successfully applied, discuss the challenges, methodologies, and the outcomes.

Solutions for challenges in computer vision

Before we go into the details, let us quickly recapitulate the major challenges in computer vision. The challenges include image classification, object detection and localization, segmentation, character recognition, face detection, face recognition, depth perception and the like. Let us see the classical algorithms that we used to solve these problems.

Classical solutions

Classical computer vision algorithms are based on traditional techniques and heuristics, with a focus on explicit handcrafted features and rule-based methods. They offer well-defined steps and interpretability, making them suitable for tasks like edge detection, corner detection, and image filtering. However, these algorithms face challenges when dealing with complex and large-scale datasets, requiring extensive parameter tuning and lacking generalization capability. Their reliance on handcrafted features and explicit rule-based methods limits their adaptability to variations and different datasets. While classical algorithms provide a solid foundation and insights into computer vision tasks, they may fall short when confronted with complex real-world scenarios.

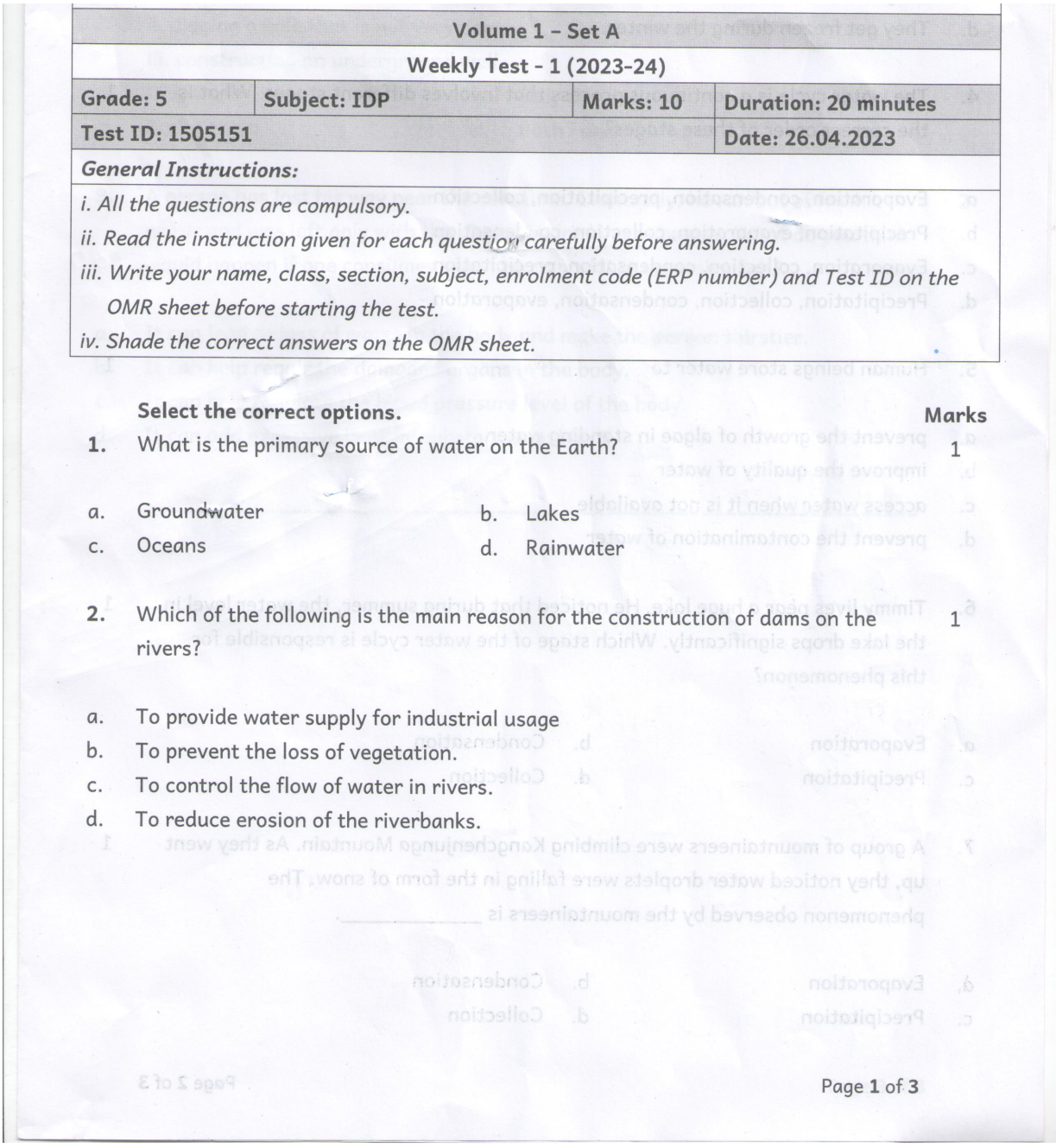
Modern solutions

Modern computer vision algorithms possess the ability to automatically learn features from data, leveraging the power of deep learning and neural networks. Algorithms like CNNs excel in tasks such as image classification, object detection, semantic segmentation, and image synthesis. By learning hierarchical representations from data, they can effectively handle complex patterns and achieve state-of-the-art performance. However, these algorithms demand large amounts of labeled data for training, involve computationally intensive training and inference processes, and can be challenging to interpret due to their black-box nature. Their capability to learn intricate and abstract features allows them to handle diverse data and generalize well to unseen examples. The deep learning capabilities of modern algorithms have revolutionized computer vision, yielding superior performance across various tasks, albeit with challenges related to data requirements and computational resources.

Algorithm families

In this section, we shall discuss some of the popular classical algorithm families and how they help in addressing some foundational tasks of computer vision. We shall use the image shown in *Figure 4.1* for most of our discussions in this chapter. It is the scan of an elementary school question paper. As can be seen from the image, the scan copy contains

varying color shades and some noise information as well. For example, the text printed on the obverse of this paper was dark enough to be captured while scanning the front of the paper.



Volume 1 – Set A			
Weekly Test - 1 (2023-24)			
Grade: 5	Subject: IDP	Marks: 10	Duration: 20 minutes
Test ID: 1505151			Date: 26.04.2023

General Instructions:

i. All the questions are compulsory.
ii. Read the instruction given for each question carefully before answering.
iii. Write your name, class, section, subject, enrolment code (ERP number) and Test ID on the OMR sheet before starting the test.
iv. Shade the correct answers on the OMR sheet.

Select the correct options. **Marks**

1. What is the primary source of water on the Earth? 1

a. Groundwater
b. Lakes
c. Oceans
d. Rainwater

2. Which of the following is the main reason for the construction of dams on the rivers? 1

a. To provide water supply for industrial usage
b. To prevent the loss of vegetation.
c. To control the flow of water in rivers.
d. To reduce erosion of the riverbanks.

Page 1 of 3

***Figure 4.1**: Noise and color shades are visible in the scanned image*

Morphological operations

Morphological operations are a set of image processing techniques used to analyze and manipulate the shape and structure of objects within an image. They are based on the principles of mathematical morphology and primarily operate on binary or grayscale images.

The two fundamental morphological operations are **dilation** and **erosion**. Dilation expands the shape of objects by adding pixels to their boundaries, resulting in larger and more connected regions. Erosion, on the other hand, shrinks the objects by removing pixels from their boundaries, causing objects to become smaller and disconnected. These basic operations can be combined to form more advanced morphological operations. Opening is the process of applying erosion followed by dilation, which can remove small objects and smooth boundaries. Closing, conversely, involves dilation followed by erosion and can fill in gaps and close small holes in objects.

Structuring elements play a crucial role in morphological operations. A structuring element is a small shape or kernel that defines the behavior of the operation. It determines the size, shape, and orientation of the neighborhood considered during the operation. By selecting different structuring elements, the effects of the morphological operations can be tailored to specific image features and objectives.

One important concept in morphological operations is the notion of connectivity. Connectivity defines how pixels or regions are considered connected or neighboring each other. It influences the behavior and outcome of morphological operations, especially when dealing with complex or irregular-shaped objects. Advanced morphological operations include morphological gradients, top-hat, and bottom-hat transformations. Morphological gradients highlight the boundaries of objects, while top-hat and bottom-hat transformations emphasize bright and dark regions, respectively, in relation to the background.

Erosion and dilation of images

Here, we show the code for performing erosion operation on *Figure 4.1*:

```
1. import cv2

2. import numpy as np
```

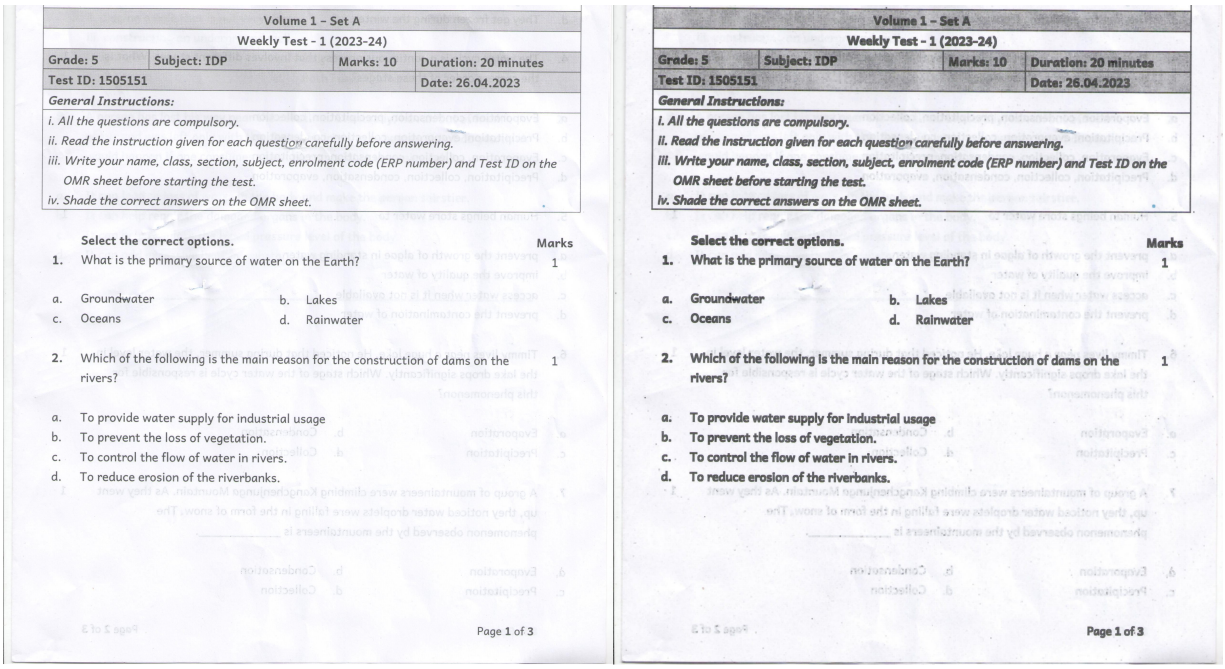
```
3.
4. # Read the input image
5. image = cv2.imread("input_images/4_ThresholdingImage.jpg")
6.
7. # Check if the image was successfully loaded
8. if image is None:
9.     print("Unable to load the image.")
10.     exit()
11.
12. # Define the structuring element for erosion
13. kernel_size = 5
14. structuring_element = cv2.getStructuringElement(cv2.MORPH_RECT, (kernel_size, kernel_size))
15.
16. # Perform image erosion
17. eroded_image = cv2.erode(image, structuring_element)
18.
19. stacked_results = np.hstack((image, eroded_image))
20. # Display the original image and the eroded image
21. cv2.imshow('Erosion', stacked_results)
22.
```

```
23. # Wait for key press and then close all windows
24. cv2.waitKey(0)
25. cv2.destroyAllWindows()
26. cv2.imwrite("output_images/erosion.jpg", stacked_results)
```

The program reads the input image in grayscale and defines a structuring element for erosion. In this example, a rectangular structuring element of size 5x5 is used, but you can adjust the **kernel_size** variable to change the size and shape of the structuring element. The program then applies the erosion operation using the **erode()** function from OpenCV, passing the input image and the structuring element as parameters. Finally, the program displays the original image and the eroded image using OpenCV's **imshow()** function. When you run the program, you will see the original image and the eroded image displayed side by side.

Note: Erosion removes pixels from the boundaries of objects, causing objects to become smaller and disconnected.

The success of erosion depends on the structuring element’s size, shape and its relationship to the objects in the image. This code produces the below output as shown in *Figure 4.2*:



Volume 1 – Set A

Weekly Test - 1 (2023-24)

Grade: 5	Subject: IDP	Marks: 10	Duration: 20 minutes
Test ID: 1505151			Date: 26.04.2023

General Instructions:

i. All the questions are compulsory.
ii. Read the instruction given for each question carefully before answering.
iii. Write your name, class, section, subject, enrolment code (ERP number) and Test ID on the OMR sheet before starting the test.
iv. Shade the correct answers on the OMR sheet.

Select the correct options. Marks

1. What is the primary source of water on the Earth? 1

a. Groundwater b. Lakes
c. Oceans d. Rainwater

2. Which of the following is the main reason for the construction of dams on the rivers? 1

a. To provide water supply for industrial usage
b. To prevent the loss of vegetation.
c. To control the flow of water in rivers.
d. To reduce erosion of the riverbanks.

Page 1 of 3

Volume 1 – Set A

Weekly Test - 1 (2023-24)

Grade: 5	Subject: IDP	Marks: 10	Duration: 20 minutes
Test ID: 1505151			Date: 26.04.2023

General Instructions:

i. All the questions are compulsory.
ii. Read the instruction given for each question carefully before answering.
iii. Write your name, class, section, subject, enrolment code (ERP number) and Test ID on the OMR sheet before starting the test.
iv. Shade the correct answers on the OMR sheet.

Select the correct options. Marks

1. What is the primary source of water on the Earth? 1

a. Groundwater b. Lakes
c. Oceans d. Rainwater

2. Which of the following is the main reason for the construction of dams on the rivers? 1

a. To provide water supply for industrial usage
b. To prevent the loss of vegetation.
c. To control the flow of water in rivers.
d. To reduce erosion of the riverbanks.

Page 1 of 3

Figure 4.2*: Left is original image and right is eroded image*

Let us now see the code for performing dilation operation on the same image:

```
1. import cv2
2. import numpy as np
3.
4. # Read the input image
5. image = cv2.imread("input_images/4_ThresholdingImage.jpg")
6.
7. # Check if the image was successfully loaded
8. if image is None:
9.     print("Unable to load the image.")
10.    exit()
```

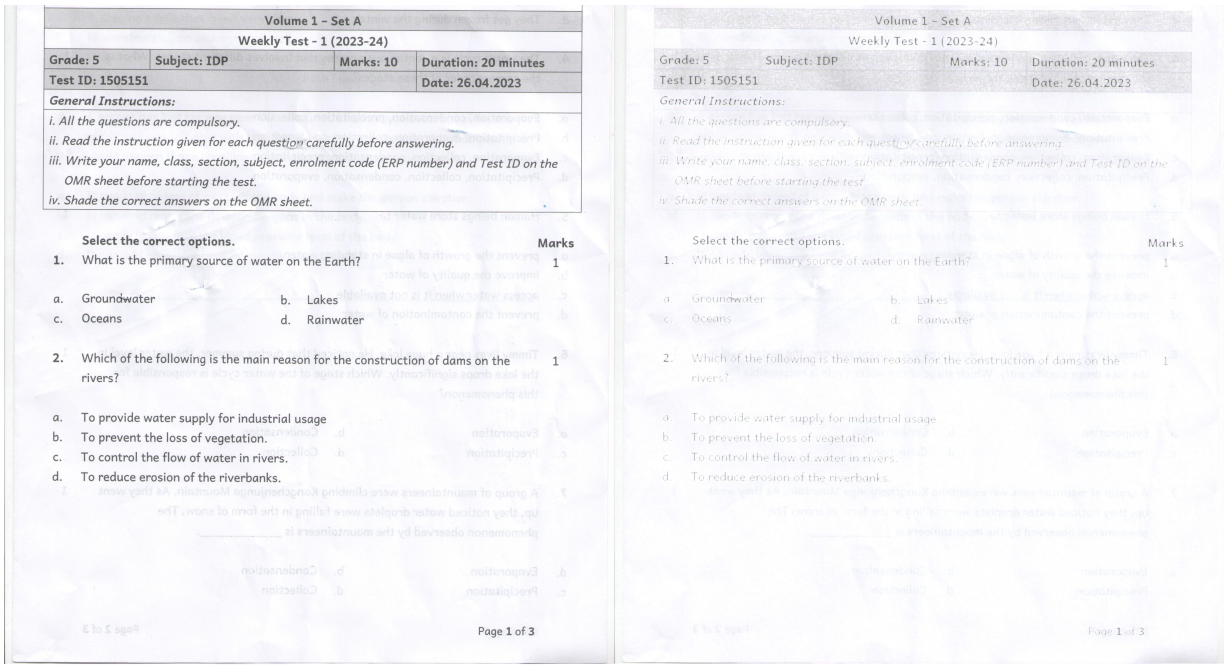
```
11.
12. # Define the structuring element for dilation
13. kernel_size = 5
14. structuring_element = cv2.getStructuringElement(cv2.MORPH_RECT, (kernel_size, kernel_size))
15.
16. # Perform image dilation
17. dilated_image = cv2.dilate(image, structuring_element)
18.
19. stacked_results = np.hstack((image, dilated_image))
20. # Display the original image and the dilated image
21. cv2.imshow('Dilation', stacked_results)
22.
23. # Wait for key press and then close all windows
24. cv2.waitKey(0)
25. cv2.destroyAllWindows()
26. cv2.imwrite("output_images/dilation.jpg", stacked_results)
```

The program reads the input image in grayscale and defines a structuring element for dilation. In this example, a rectangular structuring element of size 5x5 is used, but you can adjust the **kernel_size** variable to change the size and shape of the structuring element. The program then applies the dilation operation using the **dilate()** function from OpenCV, passing the input image and the structuring element as parameters. Finally, the program displays the original image and the dilated image using OpenCV's

imshow() function. When you run the program, you will see the original image and the dilated image displayed side by side.

Note: Dilation adds pixels to the boundaries of objects, causing objects to become larger and more connected.

The effectiveness of dilation depends on the size and shape of the structuring element and its relationship to the objects in the image. This code produces the below output as shown in *Figure 4.3*:



Volume 1 – Set A

Weekly Test - 1 (2023-24)

Grade: 5	Subject: IDP	Marks: 10	Duration: 20 minutes
Test ID: 1505151			Date: 26.04.2023

General Instructions:

i. All the questions are compulsory.
ii. Read the instruction given for each question carefully before answering.
iii. Write your name, class, section, subject, enrolment code (ERP number) and Test ID on the OMR sheet before starting the test.
iv. Shade the correct answers on the OMR sheet.

Select the correct options. Marks

1. What is the primary source of water on the Earth? 1

a. Groundwater b. Lakes
c. Oceans d. Rainwater

2. Which of the following is the main reason for the construction of dams on the rivers? 1

a. To provide water supply for industrial usage
b. To prevent the loss of vegetation.
c. To control the flow of water in rivers.
d. To reduce erosion of the riverbanks.

Page 1 of 3

Figure 4.3*: Left is original image and right is dilated image*

Closing and opening images

Opening operation happens when the erosion, dilation operations are performed in that sequence. This helps in eliminating the noise in the image. Please see the below code for opening an image:

```
1. import cv2
2. import numpy as np
3.
4. # Read the input image
```

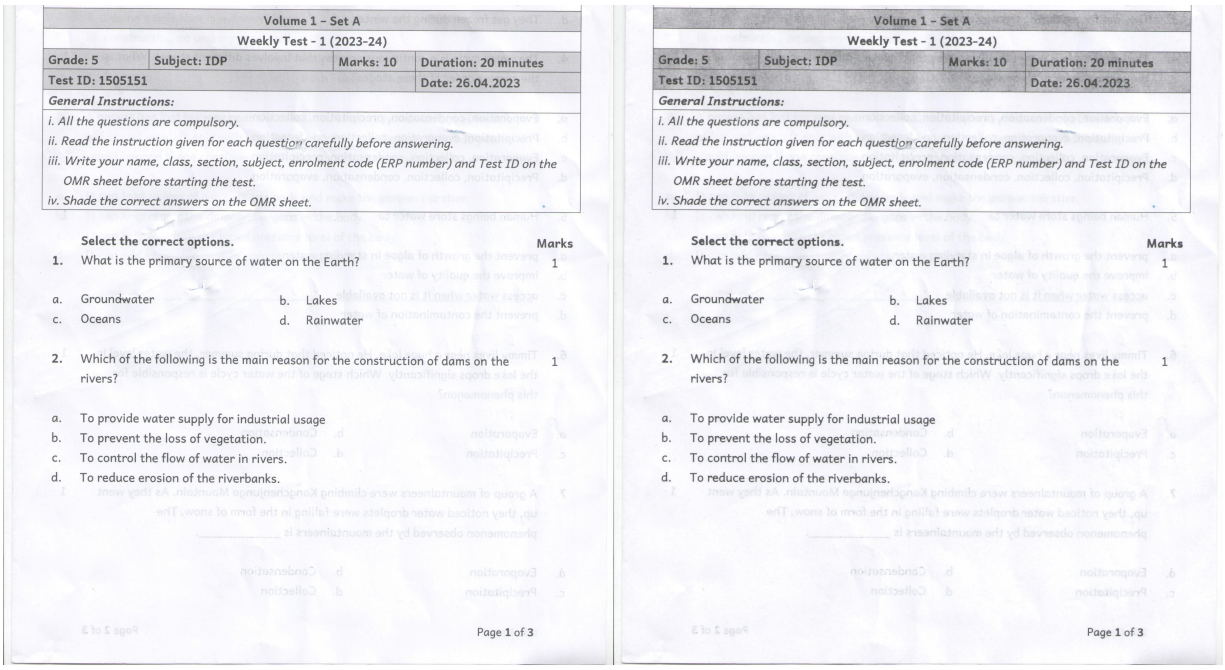
```
image = cv2.imread("input_images/4_ThresholdingImage.jpg")

# Check if the image was successfully loaded
if image is None:
    print("Unable to load the image.")
    exit()

# Define the structuring element for erosion
kernel_size = 5
structuring_element = cv2.getStructuringElement(cv2.MORPH_RECT, (kernel_size, kernel_size))

# Perform image erosion
eroded_image = cv2.erode(image, structuring_element)

# Perform image dilation
opened_image = cv2.dilate(eroded_image, structuring_element)

# Display the original image and the opened image
stacked_results = np.hstack((image, opened_image))
cv2.imshow('Opened Image', stacked_results)
```

```
25.
26. # Wait for key press and then close all windows
27. cv2.waitKey(0)
28. cv2.destroyAllWindows()
29. cv2.imwrite("output_images/opened.jpg", stacked_results)
```

The program reads the input image in grayscale and defines a structuring element for the closing operation. In this example, a rectangular structuring element of size 5x5 is used, but you can adjust the **kernel_size** variable to change the size and shape of the structuring element. The program then applies the closing operation by first eroding and the image and then dilating the eroded image. Finally, the program displays the original image and the closed image using OpenCV's **imshow()** function. When you run the program, you will see the original image and the closed image displayed side by side.

Note: The opening operation performs erosion and dilation to eliminate minor protrusions in the image.

It is useful for smoothing object boundaries and removing small, isolated regions in the image. The effectiveness of closing depends on the size and shape of the structuring element and its relationship to the objects in the image. This code produces the below output as shown in *Figure 4.4*.



Volume 1 – Set A

Weekly Test - 1 (2023-24)

Grade: 5	Subject: IDP	Marks: 10	Duration: 20 minutes
Test ID: 1505151			Date: 26.04.2023

General Instructions:

i. All the questions are compulsory.
ii. Read the instruction given for each question carefully before answering.
iii. Write your name, class, section, subject, enrolment code (ERP number) and Test ID on the OMR sheet before starting the test.
iv. Shade the correct answers on the OMR sheet.

Select the correct options. **Marks**

1. What is the primary source of water on the Earth? 1

a. Groundwater b. Lakes
c. Oceans d. Rainwater

2. Which of the following is the main reason for the construction of dams on the rivers? 1

a. To provide water supply for industrial usage
b. To prevent the loss of vegetation.
c. To control the flow of water in rivers.
d. To reduce erosion of the riverbanks.

Page 1 of 3

Volume 1 – Set A

Weekly Test - 1 (2023-24)

Grade: 5	Subject: IDP	Marks: 10	Duration: 20 minutes
Test ID: 1505151			Date: 26.04.2023

General Instructions:

i. All the questions are compulsory.
ii. Read the instruction given for each question carefully before answering.
iii. Write your name, class, section, subject, enrolment code (ERP number) and Test ID on the OMR sheet before starting the test.
iv. Shade the correct answers on the OMR sheet.

Select the correct options. **Marks**

1. What is the primary source of water on the Earth? 1

a. Groundwater b. Lakes
c. Oceans d. Rainwater

2. Which of the following is the main reason for the construction of dams on the rivers? 1

a. To provide water supply for industrial usage
b. To prevent the loss of vegetation.
c. To control the flow of water in rivers.
d. To reduce erosion of the riverbanks.

Page 1 of 3

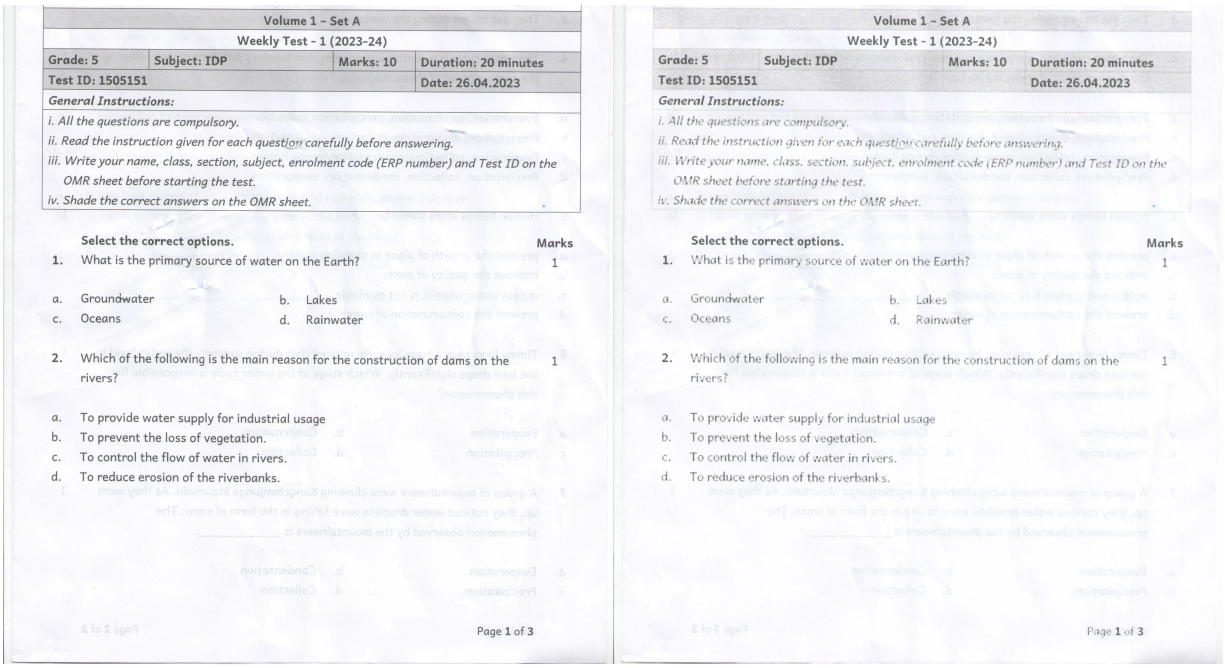
Figure 4.4*: Left is original image and the right one is opened image*

Closing is the reverse of opening operation. Here dilation happens before erosion. It can eliminate small holes in the foreground objects of the image. The code for closing is as follows:

The program reads the input image in grayscale and defines a structuring element for the closing operation. In this example, a rectangular structuring element of size 5x5 is used, but you can adjust the **kernel_size** variable to change the size and shape of the structuring element. The program then applies the dilation operation on the image and then erodes the resulting image. Finally, the program displays the original image and the closed image using OpenCV's **imshow()** function. When you run the program, you will see the original image and the closed image displayed side by side.

Note: The closing operation combines dilation and erosion to fill in gaps and close small holes in images.

It is useful for smoothing object boundaries and removing small, isolated regions in the image. The effectiveness of closing depends on the size and shape of the structuring element and its relationship to the objects in the image. This code produces the output as shown in *Figure 4.5*:



Volume 1 – Set A

Weekly Test - 1 (2023-24)

Grade: 5	Subject: IDP	Marks: 10	Duration: 20 minutes
Test ID: 1505151			Date: 26.04.2023

General Instructions:

i. All the questions are compulsory.
ii. Read the instruction given for each question carefully before answering.
iii. Write your name, class, section, subject, enrolment code (ERP number) and Test ID on the OMR sheet before starting the test.
iv. Shade the correct answers on the OMR sheet.

Select the correct options. Marks

1. What is the primary source of water on the Earth? 1

a. Groundwater b. Lakes
c. Oceans d. Rainwater

2. Which of the following is the main reason for the construction of dams on the rivers? 1

a. To provide water supply for industrial usage
b. To prevent the loss of vegetation.
c. To control the flow of water in rivers.
d. To reduce erosion of the riverbanks.

Page 1 of 3

Volume 1 – Set A

Weekly Test - 1 (2023-24)

Grade: 5 Subject: IDP Marks: 10 Duration: 20 minutes

Test ID: 1505151 Date: 26.04.2023

General Instructions:

i. All the questions are compulsory.
ii. Read the instruction given for each question carefully before answering.
iii. Write your name, class, section, subject, enrolment code (ERP number) and Test ID on the OMR sheet before starting the test.
iv. Shade the correct answers on the OMR sheet.

Select the correct options. Marks

1. What is the primary source of water on the Earth? 1

a. Groundwater b. Lakes
c. Oceans d. Rainwater

2. Which of the following is the main reason for the construction of dams on the rivers? 1

a. To provide water supply for industrial usage
b. To prevent the loss of vegetation.
c. To control the flow of water in rivers.
d. To reduce erosion of the riverbanks.

Page 1 of 3

Figure 4.5*: Left is original image and right is closed image*

Morphological operations are relatively simple, yet powerful, tools for shape analysis and manipulation in computer vision. Their versatility, ability to preserve object boundaries, and straightforward implementation make them essential in various image processing tasks, aiding in the extraction of meaningful information and enhancing the analysis and understanding of images.

```
1. import cv2
2. import numpy as np
3.
4. # Read the input image
5. image = cv2.imread("input_images/4_ThresholdingImage.jpg")
6.
7. # Check if the image was successfully loaded
8. if image is None:
```

```
    print("Unable to load the image.")
    exit()

# Define the structuring element for erosion
kernel_size = 5
structuring_element = cv2.getStructuringElement(cv2.MORPH_RECT, (kernel_size, kernel_size))

# Perform image dilation
dilated_image = cv2.dilate(image, structuring_element)

# Perform image erosion
closed_image = cv2.erode(dilated_image, structuring_element)



# Display the original image and the closed image
stacked_results = np.hstack((image, closed_image))
cv2.imshow('Closed Image', stacked_results)

# Wait for key press and then close all windows
```

```
29. cv2.waitKey(0)
30. cv2.destroyAllWindows()
31. cv2.imwrite("output_images/closed.jpg", stacked_results)
```

Thresholding

Thresholding is a fundamental technique in computer vision used to separate objects or regions of interest from an image based on their pixel intensities. It is a simple yet powerful method that relies on a specified threshold value to determine whether a pixel belongs to the foreground or background. In thresholding, each pixel in the image is compared to the threshold value. If the pixel intensity is above the threshold, it is assigned to the foreground; otherwise, it is assigned to the background. This process effectively creates a binary image where the foreground pixels represent the objects or regions of interest. Thresholding finds various applications in image processing tasks such as image segmentation, object detection, and feature extraction. It is particularly useful when the desired objects have distinct intensity characteristics compared to the background.

Choosing an appropriate threshold value is crucial for accurate results. Different thresholding techniques exist, including global thresholding, adaptive thresholding, and Otsu's thresholding. These methods automatically determine the threshold based on local or global image characteristics, improving the accuracy of segmentation. While thresholding is a simple technique, it has limitations. It assumes that the intensity distribution of foreground and background pixels is well-separated, which may not always be the case in complex images. Variations in illumination, noise, and uneven backgrounds can affect the thresholding process, leading to inaccurate results. To overcome these limitations, advanced techniques, such as multi-level thresholding and thresholding with multiple color channels, can be employed. These approaches leverage additional information to refine the segmentation results and handle more complex scenarios.

Let us see this in action with a simple code. Applying different thresholding algorithms on the image in *Figure 4.1* will have different results. The following code performs the thresholding on this image:

```
import cv2
import numpy as np
import sys

# Read the input image
image = cv2.imread("input_images/4_ThresholdingImage.jpg", 0)

# Check if the image was successfully loaded
if image is None:
    print("Unable to load the image.")
    sys.exit()

# Get the thresholding level from command line argument
threshold_level = int(sys.argv[1])

# Apply different thresholding algorithms
ret, thresh_binary = cv2.threshold(image, threshold_level, 255, cv2.THRESH_BINARY)
ret, thresh_binary_inv = cv2.threshold(image, threshold_level, 255, cv2.THRESH_BINARY_INV)
ret, thresh_trunc = cv2.threshold(image, threshold_level, 255, cv2.THRESH_TRUNC)
```

```
20. ret, thresh_tozero = cv2.threshold(image, threshold_level, 255, cv2.THRESH_TOZERO)
21. ret, thresh_tozero_inv = cv2.threshold(image, threshold_level, 255, cv2.THRESH_TOZERO_INV)
22. ret, thresh_otsu = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
23.
24. stacked_results = np.hstack((thresh_binary, thresh_binary_inv, thresh_trunc, thresh_tozero, thresh_tozero_inv, thresh_otsu))
25. # Create a window to display the thresholded images
26. cv2.namedWindow('Thresholding', cv2.WINDOW_NORMAL)
27. cv2.imshow('Thresholding', stacked_results)
28.
29. # Wait for a key press and then close the window
30. cv2.waitKey(0)
31. cv2.destroyAllWindows()
32.
33. cv2.imwrite("output_images/thresholding.jpg", stacked_results)
```

Let us see the results of the above code on various thresholding levels. The program applies different thresholding algorithms (binary, binary inverse, truncation, to zero, to zero inverse, and Otsu) on the input image with the specified threshold level. It then creates a composite image by stacking all the threshold images horizontally. See *Figure 4.6* for the thresholding results with 127 as the thresholding limit. The algorithms used are binary, inverse binary, truncated, threshold to zero, inverted threshold to zero and Otsu from left to right. Please refer to the following figure:

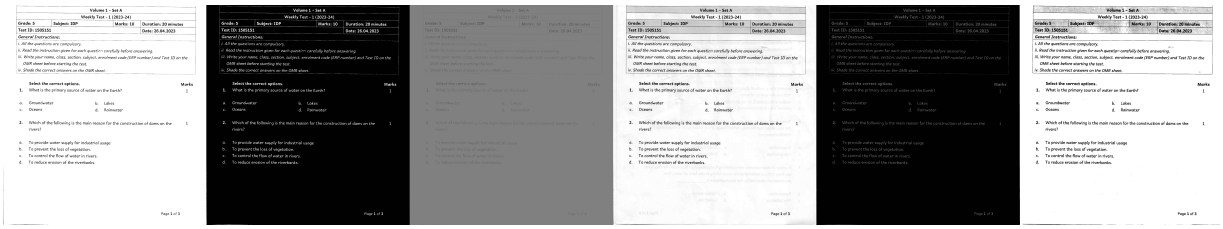


***Figure 4.6**: Results of thresholding with limit as 127*

See *Figure 4.7* for the thresholding results with 64 as the thresholding limit. The algorithms used are binary, inverse binary, truncated, threshold to zero, inverted threshold to zero and Otsu from left to right. Please refer to the following figure:

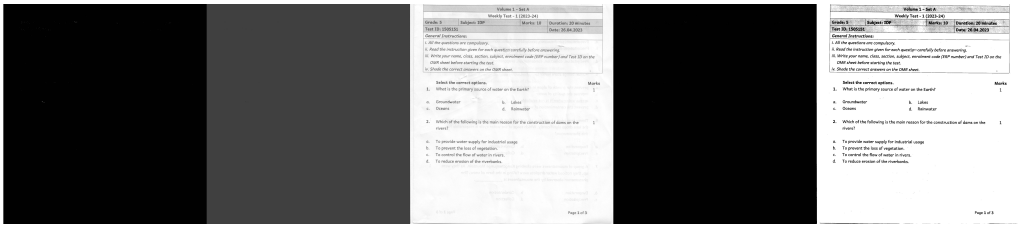


***Figure 4.7**: Results of thresholding with limit as 64*

Thresholding is a basic yet powerful technique in computer vision used to separate objects or regions of interest from an image based on their pixel intensities. It is widely applied in various image processing tasks and serves as a foundation for more advanced segmentation algorithms. However, it is important to consider the limitations of thresholding and explore advanced techniques when dealing with complex images or challenging conditions.

Detecting edges and corners

Edge and corner detection are the next fundamental techniques that we shall discuss. These techniques are used to identify and locate important features in an image. These features play a crucial role in various image processing tasks, such as object recognition, image stitching, and 3D reconstruction.

Edge detection aims to identify sudden changes in pixel intensity, which often correspond to object boundaries or significant image structures. It helps to extract the outline or silhouette of objects present in an image. Popular edge detection algorithms include the Canny edge detector, Sobel operator, and **Laplacian of Gaussian** (**LoG**) operator.

Corner detection, on the other hand, focuses on identifying sharp changes in image gradients, representing the intersection of edges or corners of objects. Corners are distinctive features that can be used for image registration, tracking, and matching. Well-known corner detection algorithms include the Harris corner detector, Shi-Tomasi corner detector, and **Features from Accelerated Segment Test** (**FAST**) corner detector.

While edge detection is more focused on capturing continuous boundaries, corner detection is designed to identify discrete, localized features. As a result, edge detection algorithms are often more sensitive to noise and may produce thicker edges, whereas corner detection algorithms are more robust to noise and can provide precise corner locations.

Both edge and corner detection algorithms rely on the analysis of image gradients and local image properties. They typically involve the computation of derivatives, convolutions, and thresholding operations to identify the desired features. They provide essential cues for subsequent processing steps, such as segmentation, object recognition, and tracking. However, they may struggle with complex scenes, occlusions, and varying lighting conditions. Additionally, parameter selection and tuning can significantly affect the detection results.

The below code employs Canny edge detection, Sobel edge detection and Laplacian edge detection algorithms on the image shown in *Figure 4.1*:

```
1. import cv2
2. import numpy as np
3. import sys
4.
5. # Read the input image
6. image = cv2.imread("input_images/4_ThresholdingImage.jpg")
7.
8. # Check if the image was successfully loaded
```

```
if image is None:
    print("Unable to load the image.")
    sys.exit()

# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply different edge detection algorithms
canny_edges = cv2.Canny(gray, 100, 200)
sobel_x = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
laplacian_edges = cv2.Laplacian(gray, cv2.CV_64F, ksize=3)

# Create a window to display the images
cv2.namedWindow('Edge Detection', cv2.WINDOW_NORMAL)

# Display the original image and edge detection results side by side
stacked_results = np.hstack((canny_edges, sobel_x, sobel_y, laplacian_edges))
cv2.imshow('Edge Detection', stacked_results)

```

```
29. # Wait for a key press and then close the window
30. cv2.waitKey(0)
31. cv2.destroyAllWindows()
32.
33. cv2.imwrite("output_images/edge_detect.jpg", stacked_results)
```

The program reads the input image, converts it to grayscale, and applies different edge detection algorithms such as Canny, Sobel (x and y gradients), and Laplacian. The detected edges are then overlaid on the original image using **addWeighted()** to combine the images. Finally, the result, showing the original image with overlaid edges, is displayed using OpenCV's **imshow()** function. *Figure 4.8* shows the results of this code. As noticed in the figure, the algorithms have different results on edge detection. The algorithms used are Canny, Sobel with derivatives in x-axis, Sobel with derivatives in y-axis, and Laplacian from left to right. Please refer to the following figure:

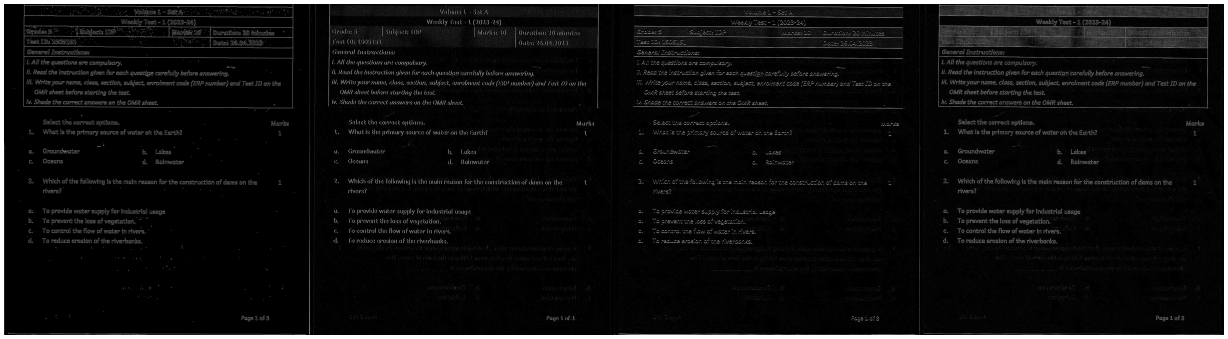


***Figure 4.8**: Results of edge detection*

Edge and corner detection are important techniques in computer vision for identifying significant image features. They form the foundation for numerous image processing applications and enable subsequent analysis and interpretation of images. Understanding these techniques and their strengths and limitations is essential for effectively working with images in computer vision tasks.

Image transformations

Image transformations are fundamental operations in computer vision and image processing that involve modifying the appearance or geometric properties of an image. These transformations can be applied to achieve various objectives, such as correcting distortions, changing the perspective, enhancing or manipulating the image content, and preparing images for further analysis.

Geometric transformations are commonly used to modify the spatial arrangement of pixels in an image. Some common geometric transformations include scaling (resizing), rotation, translation (shifting), and shearing. These transformations can be applied to modify the size, orientation, position, and shape of objects within an image. Affine transformations are a type of geometric transformation that preserve parallel lines and ratios of distances. They include translation, rotation, scaling, and shearing. Affine transformations are widely used in tasks like image alignment, registration, and perspective correction. Non-affine transformations, such as projective transformations, introduce perspective distortion and allow for more complex modifications of the image geometry. Projective transformations are particularly useful for tasks like image warping, 3D reconstruction, and virtual reality applications. Besides geometric transformations, there are various image enhancement and manipulation techniques, such as contrast adjustment, brightness correction, color manipulation, and filtering. These transformations can be used to improve the visual quality, enhance specific image features, or extract relevant information from the image.

Image transformations are implemented using mathematical operations and algorithms that manipulate the pixel values of the image. OpenCV provides built-in functions and methods to perform these transformations efficiently.

Region growing

Region growing is a technique used to segment an image based on pixel similarity. It aims to group adjacent pixels that exhibit similar characteristics, such as color or intensity, into meaningful regions or objects. The region growing algorithm starts with an initial seed pixel or set of seeds and iteratively adds neighboring pixels to the region based on a predefined similarity criterion. This process continues until no more pixels meet the similarity criteria or a stopping condition is reached. The algorithm

exploits the spatial connectivity of pixels to form coherent regions. It takes advantage of the fact that objects in an image often exhibit spatial continuity, where neighboring pixels tend to have similar properties.

Region growing can be used in various applications, including image segmentation, object extraction, and boundary detection. It enables the extraction of meaningful structures from images by grouping pixels with similar characteristics. The success of region growing depends on the choice of seed pixels and the definition of similarity criteria. Careful selection of seeds and appropriate similarity measures are crucial to achieve accurate and reliable results. Region growing algorithms can be adapted to handle different types of images, such as grayscale or color images. They can also incorporate additional constraints, such as gradient information or texture features, to enhance the segmentation process.

However, region growing algorithms may face challenges when dealing with complex scenes, noise, or weak boundaries. Selecting inappropriate seeds or similarity criteria can lead to under or over-segmentation. Their performance is influenced by various factors, including image resolution, object size, and the presence of occlusions, or overlapping objects. To improve the effectiveness of region growing, techniques like adaptive region growing and hybrid approaches combining multiple segmentation methods can be employed.

There are no built-in algorithms in OpenCV to support region growing. However, statistical approaches like k-means can be used to identify clusters of pixels which are part of a region.

Clustering

Clustering is a fundamental technique which groups similar data points together. It aims to discover patterns or structures within a dataset by partitioning it into distinct clusters based on their similarity. Clustering algorithms assign data points to clusters based on their proximity in a feature space. The choice of features, such as color, texture, or shape, depends on the specific application and the characteristics of the data. Clustering is an iterative process that aims to optimize a certain objective function, such as minimizing the intra-cluster distance or maximizing inter-

cluster dissimilarity. The convergence and stability of the clustering results are important considerations.

Popular clustering algorithms include k-means clustering, hierarchical clustering, and spectral clustering. These algorithms employ different approaches to define cluster similarity and assign data points accordingly. They can be unsupervised, meaning they do not rely on prior knowledge or labeled data, or they can be semi-supervised, leveraging a small amount of labeled data to guide the clustering process. The effectiveness of clustering algorithms depends on several factors, including the quality and representativeness of the features used, the choice of distance or similarity metrics, and the algorithm's parameters.

The number of clusters to be determined is a critical aspect of clustering. It can be predefined based on prior knowledge or determined automatically using techniques such as silhouette analysis or the elbow method.

The given code performs a k-means clustering on the image and shows the clusters as a color next to the original image:

```
1. import cv2
2. import numpy as np
3. import matplotlib.pyplot as plt
4. import sys
5.
6. # Read the input image
7. image = cv2.imread("input_images/4_Clustering.jpg")
8.
9. # Check if the image was successfully loaded
10. if image is None:
11.     print("Unable to load the image.")
```

```
    sys.exit()

# Reshape the image to a 2D array of pixels
pixels = image.reshape((-1, 3))

# Convert the pixel values to float
pixels = np.float32(pixels)

# Define the parameters for k-means clustering
num_clusters = 5
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
flags = cv2.KMEANS_RANDOM_CENTERS

# Apply k-means clustering
_, labels, centers = cv2.kmeans(pixels, num_clusters, None, criteria, 10, flags)

# Convert the centers to 8-bit values
centers = np.uint8(centers)

# Map each pixel to its corresponding cluster center
```

```
segmented_image = centers[labels.flatten()]
segmented_image = segmented_image.reshape(image.shape)

# Convert the segmented image to RGB for visualization
segmented_image_rgb = cv2.cvtColor(segmented_image, cv2.COLOR_BGR2RGB)

# Display the original image and segmented image side by side
stacked_results = np.hstack((image, segmented_image_rgb))
cv2.imshow('Clustering', stacked_results)

# Wait for a key press and then close the window
cv2.waitKey(0)
cv2.destroyAllWindows()

cv2.imwrite("output_images/clustering.jpg", stacked_results)
```

The program reads the input image and reshapes it to a 2D array of pixels. It then applies the algorithms k-means. For each algorithm, the resultant labels or centers are used to create an image with the clustered regions overlaid on the original image. Finally, the original image and clustering results are displayed side by side using OpenCV's **imshow()** function. *Figure 4.9* shows an original image and clustering performed on it with 3 clusters. Please refer to the following figure:

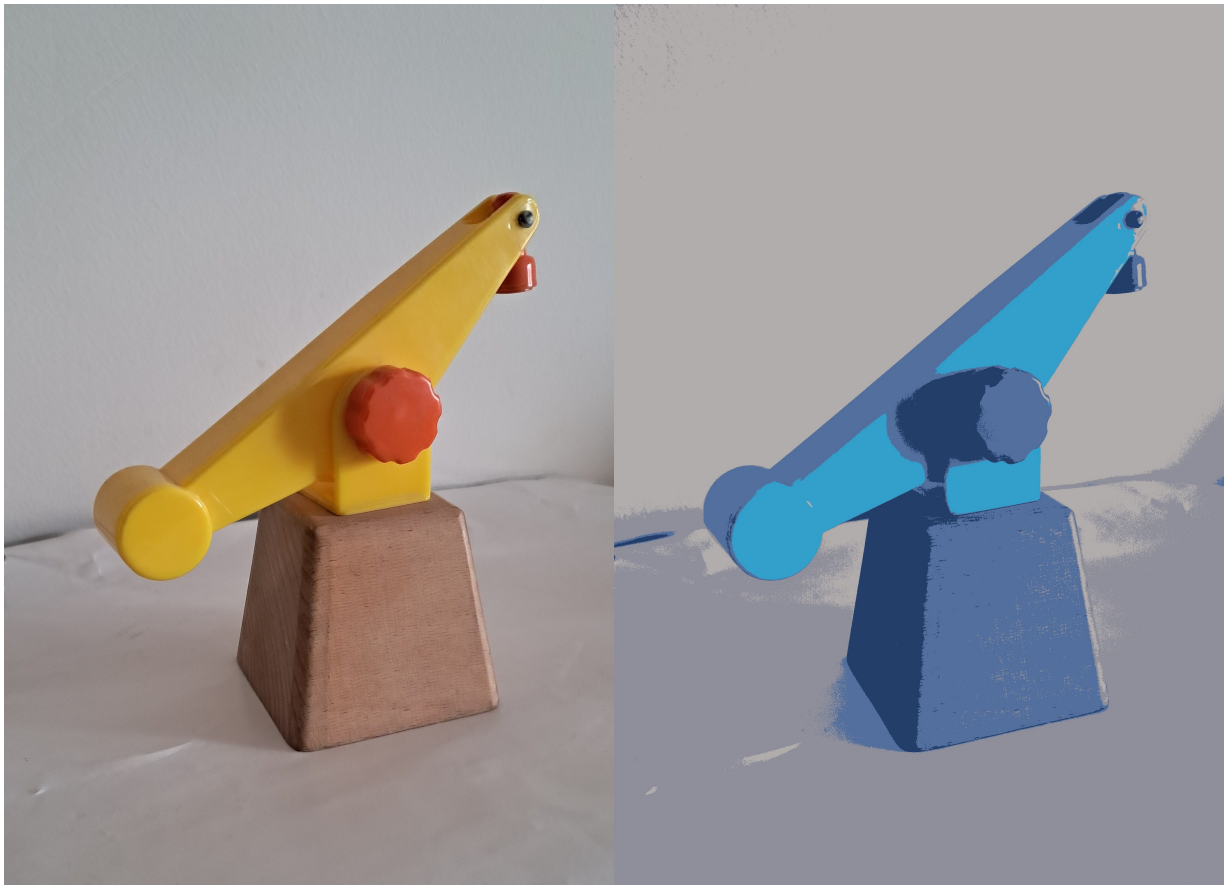


***Figure 4.9**: K-means clustering performed on the left image with 3 clusters has segmented the image as shown on the right*

Clustering is a powerful technique in computer vision for grouping similar data points and discovering meaningful patterns. It aids in tasks such as image segmentation and object recognition, allowing for effective analysis and interpretation of visual data. Clustering algorithms can handle a wide range of image data, but they may face challenges when dealing with noisy or ambiguous data, overlapping clusters, or non-convex shapes.

Template matching

Template matching is a technique in computer vision that involves finding a template image within a larger search image. It is commonly used to locate instances of a specific object or pattern in an image.

The template matching process involves comparing a template image, also known as a pattern or reference image, with different regions of the search image. The goal is to find the best match or similarity between the template

and the corresponding region in the search image. The matching process is typically based on measuring the similarity between the template and the image regions using various metrics, such as correlation, sum of squared differences, or normalized cross-correlation. These metrics provide a numerical score that represents the similarity between the template and each region in the search image. Template matching can be performed at different scales and orientations by rescaling or rotating the template image. This allows for the detection of objects that may vary in size or orientation within the search image. Template matching has several applications, including object recognition, object tracking, and image registration. It is widely used in tasks such as face recognition, character recognition, and document analysis.

Consider *Figure 4.10*, if we are to count the number of circles present here, we should use template matching algorithms:

Figure 4.10*: A search image having often repeating similar of same shapes or patterns*

This might look like a trivial scenario. However, it is often the case especially in manufacturing assembly lines. Imagine a scenario where a factory wants to monitor petty theft of shopping cart units between downloading and warehouse. It is extremely difficult to count the same items over and over. Pattern matching technique can be used here.

Let us see how we can use OpenCV for this purpose. By considering the image in Figure 4.11 as the template, we will code the template matching program as below:

***Figure 4.11**: Pattern or template to be counted*

```
1. import cv2
2. import numpy as np
3.
4. # Read the search image and the template image
5. search_image = cv2.imread('input_images/4_SearchImage.jpg', cv2.IMREAD_COLOR)
6. template_image = cv2.imread('input_images/4_Template.jpg', cv2.IMREAD_COLOR)
7.
8. # Check if the images were successfully loaded
```

```
if search_image is None or template_image is None:
    print("Unable to load the images.")
    exit()

# Convert the images to grayscale
search_gray = cv2.cvtColor(search_image, cv2.COLOR_BGR2GRAY)
template_gray = cv2.cvtColor(template_image, cv2.COLOR_BGR2GRAY)

# Perform template matching
result = cv2.matchTemplate(search_gray, template_gray, cv2.TM_CCOEFF_NORMED)

# Set a threshold for the match score
threshold = 0.6

# Find the locations where the match score is above the threshold
locations = np.where(result >= threshold)

# Draw rectangles around the matched regions
for pt in zip(*locations[::-1]):
```

```
28.     bottom_right = (pt[0] + template_gray.shape[1], pt[1] + template_g
        ray.shape[0])
29.     cv2.rectangle(search_image, pt, bottom_right, (0, 255, 0), 2)
30.
31. # Display the search image with the matched regions
32. cv2.imshow('Template Matching Result', search_image)
33.
34. # Wait for key press and then close the window
35. cv2.waitKey(0)
36. cv2.destroyAllWindows()
37. cv2.imwrite("output_images/template_matching.jpg", search_image)
```

The program reads the search image and the template image using **cv2.imread()**. It then converts the images to grayscale using **cv2.cvtColor()**. Next, the program performs template matching using **cv2.matchTemplate()**, passing the grayscale search image and template image as parameters. The result is a correlation map that represents the similarity between the template and different regions of the search image. A threshold is set to determine the matches based on the match score. In this example, a threshold of **0.6** is used, but you can adjust it according to your needs. The program finds the locations where the match score is above the threshold using **np.where()**. It then draws rectangles around the matched regions on the search image using **cv2.rectangle()**. Finally, the program displays the search image with the matched regions using **cv2.imshow()**.

When you run the program, you will see the search image with rectangles drawn around the matched regions. This code produces the result as shown in *Figure 4.12*. There are certainly some false negatives where the pattern was not spotted. This can be improved by modifying the threshold parameter in line 21. Please refer to the following figure:

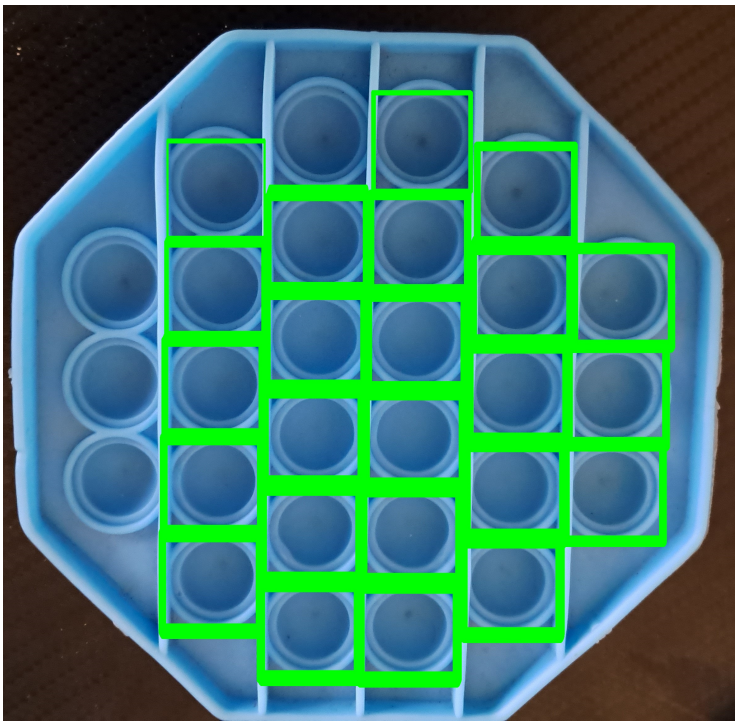


***Figure 4.12**: Results of template matching*

One limitation of template matching is that it is sensitive to variations in lighting conditions, noise, and occlusions. In complex scenes or with cluttered backgrounds, false positives or inaccurate matches can occur. To address these challenges, advanced techniques, such as multi-scale template matching, feature-based methods, and machine learning approaches, are often employed. These techniques aim to improve robustness, accuracy, and efficiency in template matching tasks.

Watershed algorithm

The watershed algorithm is a well-known technique for image segmentation that is based on the concept of watershed lines or boundaries. It mimics the behavior of water flowing in a topographic map, where the basins represent different regions or objects in the image.

The watershed algorithm starts by treating the grayscale or gradient image as a topographic map, where the intensity values represent the elevations. The algorithm then identifies the local minima in the image, which serves as the initial markers or seeds for the regions. Next, the algorithm performs a flooding process, where the basins are filled with water starting from the markers. As the water fills the basins, it naturally separates the adjacent

regions based on the watershed lines. These watershed lines represent the boundaries between different objects or regions in the image. It can be implemented using various approaches, including the flooding-based algorithm and the marker-controlled algorithm. The marker-controlled algorithm provides more control over the segmentation process by allowing manual or automatic placement of markers.

One advantage of the watershed algorithm is its ability to handle complex image structures, including objects with irregular shapes and overlapping boundaries. It is particularly useful in scenarios where the boundaries between objects are not well-defined or there are strong intensity gradients. However, the watershed algorithm can produce over-segmentation, where the boundaries are overly fragmented. To address this, post-processing steps such as merging or region merging techniques can be applied. Please see the below code for an implementation of the watershed algorithm:

```
1. import numpy as np
2. import cv2
3. from matplotlib import pyplot as plt
4.
5. image = cv2.imread("input_images/4_SearchImage.jpg")
6.
7. gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
8. _, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
9.
10. kernel_size = 5
11. kernel = np.ones((kernel_size,kernel_size),np.uint8)
12.
```

```
# Find area which is surely background
sure_bg = cv2.dilate(thresh,kernel,iterations=1)

# Find area which is surely foreground
dist_transform = cv2.distanceTransform(sure_bg,cv2.DIST_L2,3)
_, sure_fg = cv2.threshold(dist_transform,0.05*dist_transform.max(),255,0)
sure_fg = np.uint8(sure_fg)

# Find the region which is neither surely foreground nor surely background
unknown = cv2.subtract(sure_bg,sure_fg)

# Marker labelling
_, markers = cv2.connectedComponents(sure_fg)

# Add 1 to all labels to mark sure background
markers = markers+1

# Now, mark the region of unknown with zero
markers[unknown==255] = 0

```

```
33. markers = cv2.watershed(image,markers)
34. image[markers == -1] = [0,0,0]
35. 
36. 
37. 
38. # The next 3 steps are needed only for better visibilty in publishing.
39. structuring_element = cv2.getStructuringElement(cv2.MORPH_RECT, (kernel_size, kernel_size))
40. image = cv2.erode(image, structuring_element)
41. image = cv2.erode(image, structuring_element)
42. 
43. cv2.imshow('watershed', image)
44. cv2.waitKey(0)
45. cv2.destroyAllWindows()
46. cv2.imwrite("output_images/watershed.jpg",image)
```

The program reads an input image, using **cv2.imread()**. It converts the image to grayscale using **cv2.cvtColor()**. It applies Otsu's thresholding to the grayscale image using **cv2.threshold().** It then applies dilation to the thresholded image using **cv2.dilate()** with the defined kernel. This operation helps identify the background area. It applies distance transform to background image using **cv2.distanceTransform()**. Using the distance transform, the program applies thresholding to identify the foreground area using **cv2.threshold()**. The threshold value is set to 0.05 times the maximum distance transform value. This is interpreted as foreground. It subtracts foreground from background using **cv2.subtract()** to identify the region that is unknown. Marker labelling is performed on foreground using

cv2.connectedComponents() to assign labels to the connected components. After performing some mathematical operations to clearly differentiate foreground from background and unknown regions, the program applies the watershed algorithm to the input image using **cv2.watershed()**, passing the image and markers as parameters. The result is stored in markers. In the watershed result, regions labeled as **-1** indicate the boundary regions. The program sets the corresponding pixels in the input image to black **[0,0,0]** using boolean indexing (**markers == -1**) and assigns it to image.

For better visibility in publishing, the program performs erosion on the image twice using **cv2.erode()** and a structuring element defined by **cv2.getStructuringElement()**. This step is optional and can be modified or removed based on specific requirements. The program displays the resulting image using **cv2.imshow()**. Finally, the program saves the watershed result image as **watershed.jpg** using **cv2.imwrite()**. This code produces the result as shown in *Figure 4.13*:

***Figure 4.13**: Results of watershed algorithm*

Note: It is common to pre-process the image to enhance the boundaries or gradients using techniques such as morphological operations or gradient operators. This is done in order to obtain

accurate segmentation results. However, there is no guarantee that the same operations provide a similar performance on a different image.

Foreground and background detection

The task of separating the foreground and background of an image is a fundamental problem in computer vision. It involves segmenting an image to distinguish the objects or regions of interest (foreground) from the surrounding environment (background).

OpenCV provides an effective algorithm called **GrabCut** that allows for automatic foreground/background segmentation. GrabCut combines image data and user-provided guidance to iteratively refine the segmentation result. The GrabCut algorithm starts with an initial estimate of the foreground and background regions. This estimate can be provided by the user through a bounding box or a set of markers indicating the foreground and background regions. Based on this initial estimate, GrabCut models the image as a Markov random field and uses an energy minimization approach to iteratively update the foreground and background estimates. The algorithm adjusts the estimates based on color similarity, spatial proximity, and pixel connectivity within the image. Through multiple iterations, GrabCut refines the segmentation result by optimizing the energy function and adapting the segmentation boundaries. The algorithm converges when the segmentation result stabilizes, indicating a satisfactory separation between the foreground and background.

GrabCut offers several advantages for foreground/background segmentation. It can handle complex image structures, including objects with intricate boundaries and varying appearances. The algorithm is also able to adapt to different images and is relatively robust to variations in lighting conditions, colors, and textures. However, GrabCut's performance heavily relies on the initial estimates provided by the user. Accurate initialization of the foreground and background regions is crucial for obtaining satisfactory results. Incorrect initial estimates can lead to over-segmentation or under-segmentation, where parts of the foreground may be misclassified as background or vice versa.

Let us look at the implementation of Grabcut:

```
import cv2
import numpy as np

# Read the image
image = cv2.imread('input_images/test_image1.jpeg')

# Check if the image was successfully loaded
if image is None:
    print("Unable to load the image.")
    exit()

# Create a mask to indicate the areas of the image to be classified (foreground, background, etc.)
mask = np.zeros(image.shape[:2], np.uint8)

# Define the rectangle enclosing the foreground object (top left and bottom right coordinates)
rect = (225, 225, 850, 850)  # Adjust the coordinates based on the region of interest

# Perform the GrabCut algorithm
bgd_model = np.zeros((1, 65), np.float64)
```

```
20. fgd_model = np.zeros((1, 65), np.float64)
21. cv2.grabCut(image, mask, rect, bgd_model, fgd_model, 5, cv2.GC_INIT_WITH_RECT)
22.
23. # Create a mask where all probable foreground and foreground pixels are set to 1
24. foreground_mask = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
25.
26. # Apply the mask to the original image
27. segmented_image = image * foreground_mask[:, :, np.newaxis]
28.
29. # Display the original image and the segmented image side by side
30. image = cv2.rectangle(image, (rect[0],rect[1]), (rect[2],rect[3]), (0,0,0), 3)
31. combined_image = np.hstack((image, segmented_image))
32. cv2.imshow('Original vs Segmented', combined_image)
33. cv2.waitKey(0)
34. cv2.destroyAllWindows()
35. cv2.imwrite("output_images/grabcut.jpg", combined_image)
```

The program starts by reading the image using **cv2.imread()**. It then creates an initial mask, that will be used to indicate the areas of the image to be classified as foreground, background, and so on. Next, a rectangle is defined to enclose the foreground object. You can adjust the coordinates of **rect** in *line 16* based on the region of interest in the image. The GrabCut

algorithm is performed using **cv2.grabCut()**. It takes the image, mask, rectangle, background model, foreground model, number of iterations, and the initialization mode as parameters. The algorithm updates the mask to classify the pixels as probable foreground, probable background, and so on. A binary mask, **foreground_mask**, is created based on the updated mask. It sets the probable foreground and foreground pixels to **1**, while the probable background and background pixels are set to **0**. Finally, the mask is applied to the original image using element-wise multiplication, resulting in the segmented image. The original image and the segmented image are then displayed side by side using **cv2.imshow()**.

When you run the program, you will see the original image on the left and the segmented figure on the right, where the foreground object is highlighted. Also, in the left figure, a black rectangle is drawn to show the seed region mentioned in the *line 16.*

The results of this code are shown in *Figure 4.14*. Left is the original image with a rectangle showing the region of interest. Right side is the foreground detected by the algorithm:

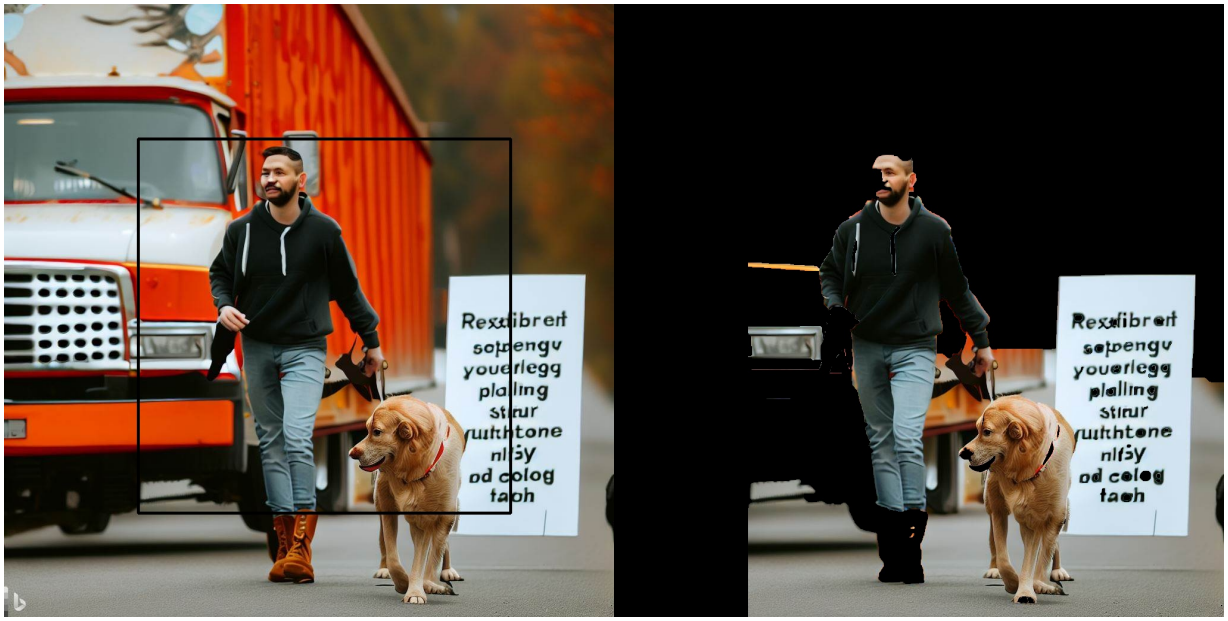


***Figure 4.14**: Grabcut algorithm*

Superpixels

Superpixels are used to group pixels into meaningful atomic regions. They aim to reduce the complexity of image processing tasks by providing a

higher-level representation of an image. They are formed by grouping pixels based on their spatial proximity and color similarity. This grouping process allows for the creation of compact and homogeneous regions that preserve important boundaries and structures in the image. By grouping pixels into super pixels, the number of images primitives to analyze is significantly reduced, making subsequent algorithms more robust and less sensitive to noise and small variations. Superpixels encapsulate contextual information and spatial relationships between pixels. They reduce the complexity of image processing tasks, preserve boundaries, and provide a higher-level representation of an image. This gives considerable advantages over traditional pixel-based approaches.

Superpixels can be generated using different algorithms, such as **Simple Linear Iterative Clustering** (**SLIC**), QuickShift, and Watershed. These algorithms employ various criteria, such as color, spatial proximity, and image gradient, to define superpixel boundaries. The size and shape of superpixels can be adjusted to strike a balance between capturing fine details and preserving the overall structure of the image. This flexibility allows for adaptability to different tasks and image characteristics. The choice of superpixel algorithm depends on the specific requirements of the task and the characteristics of the image with each algorithm having its own strengths and weaknesses.

Superpixels serve as building blocks for higher-level computer vision tasks. They can be used as input for feature extraction, region-based object detection, and image classification algorithms, providing more meaningful and informative representations. They are particularly useful in various applications, such as image segmentation, object tracking, and image enhancement. By providing a more efficient representation of an image, they enable faster processing and reduce computational costs.

Image pyramids

Image pyramids are a technique widely used in computer vision to create a series of images at different scales. They provide a multi-resolution representation of an image, allowing for efficient processing and analysis at different levels of detail. In an image pyramid, the original image is repeatedly downsampled or upsampled to create a series of images with decreasing or increasing resolutions, respectively. This is done by applying

a predefined scaling factor or using specific interpolation techniques. The pyramid structure allows algorithms to handle image variations caused by factors like scale changes, viewpoint changes, and occlusions. It provides a robust framework for matching and comparing features across different levels of detail. Pyramid levels can be accessed individually, allowing for selective processing based on the desired level of detail. This flexibility makes image pyramids suitable for adaptive algorithms that adapt their behavior to the available image resolution.

Image pyramids can be constructed using various algorithms, including hierarchical approaches and wavelet transforms. Pyramid-based techniques, such as Laplacian pyramids and Gaussian pyramids, are commonly used for image representation and processing. Laplacian pyramids decompose an image into a series of band-pass filtered layers, while Gaussian pyramids create a series of smoothed and downsampled images. The choice of pyramid size, scaling factors, and interpolation methods depends on the specific application and the characteristics of the input images. These parameters can be adjusted to strike a balance between preserving important image features and reducing computational overhead.

The code below creates a mock-up of an image pyramid. The code reads the input image and displays it as the original image. It then generates a pyramid by repeatedly downsampling the image using the **pyrDown()** function from OpenCV. Each level of the pyramid is stored in the **pyramid_images** list. The program iterates over the pyramid images and displays them with window titles indicating the pyramid level. The condition in the while loop controls the size of the pyramid. When you run the program, you will see the original image displayed along with a series of images representing different levels of the image pyramid. This code offers a simplified view of how image pyramids are built.

```
1. import cv2
2. import sys
3.
4. # Read the input image
```

```
image = cv2.imread("input_images/4_Clustering.jpg")

# Check if the image was successfully loaded
if image is None:
    print("Unable to load the image.")
    sys.exit()

# Display the original image
cv2.imshow('Original Image', image)

# Generate and display the image pyramid
pyramid_image = image.copy()
pyramid_images = [pyramid_image]

while pyramid_image.shape[0] > 100 and pyramid_image.shape[1] > 100:  # Adjust the condition to control the pyramid size
    pyramid_image = cv2.pyrDown(pyramid_image)
    pyramid_images.append(pyramid_image)

for i, image_level in enumerate(pyramid_images):
    cv2.imshow(f'Pyramid Level {i}', image_level)
```

```
25.
26. # Wait for key press and then close all windows
27. cv2.waitKey(0)
28. cv2.destroyAllWindows()
```

The program reads the input image and displays it as the original image. It then generates a pyramid by repeatedly downsampling the image using the **pyrDown()** function from OpenCV. Each level of the pyramid is stored in the **pyramid_images** list. The program iterates over the pyramid images and displays them with window titles indicating the pyramid level. The condition in the while loop controls the size of the pyramid, and you can adjust it as per your requirements. The program waits for a key press before closing all the windows.

When you run the program, you will see the original image displayed along with a series of images representing different levels of the image pyramid. This code generates the output shown in *Figure 4.15*:

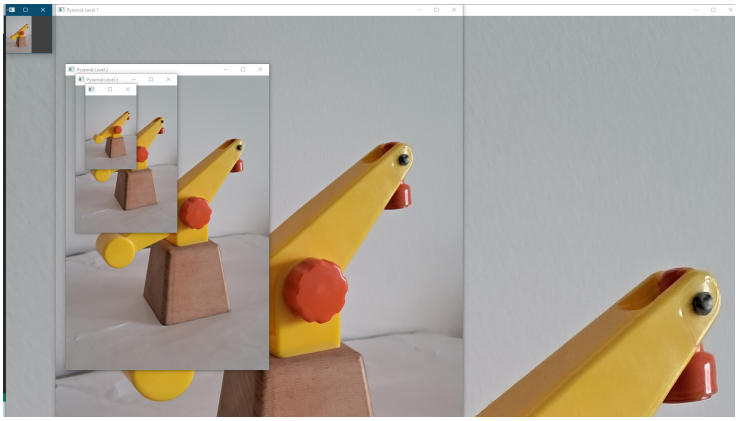


***Figure 4.15**: A mock image pyramid*

Image pyramids offer several benefits in computer vision tasks. They enable efficient object detection, recognition, and tracking by performing operations at different scales and capturing objects of varying sizes. They are particularly useful in applications such as image blending, image compression, and feature extraction. They facilitate seamless blending of images at different resolutions and enable efficient storage and transmission of large images.

Convolution

Remember we were talking about how older generation television sets were performing image processing without a processor? That was achieved using a simple, yet powerful technique called convolution. Convolution is a fundamental operation used to combine two analog signals into a new output signal. It involves a mathematical operation that measures the overlap between a fixed-sized window (kernel) and a segment of the input signal. The kernel is shifted across the input signal, and at each position, the corresponding values are multiplied and summed to produce an output value. It allows for the extraction of specific patterns or characteristics from signals, enabling analysis, enhancement, and manipulation of the original signal.

Convolution is so powerful that when images were digitized the process of convolution was also digitized. It involves a mathematical operation that combines a small matrix called a kernel or filter with an image to produce a new output image. The kernel is typically a small square or rectangular matrix with numerical values. The convolution operation is performed by sliding the kernel across the image, calculating the weighted sum of the pixel values in the kernel's neighborhood at each position. This weighted sum is then assigned to the corresponding pixel in the output image. Convolution allows for various image processing tasks such as smoothing, sharpening, edge detection, and feature extraction. By selecting different types of kernels, different image enhancements and transformations can be achieved. The process of convolution applies local operations to each pixel in the image, allowing for the extraction of local image features and the preservation of spatial relationships.

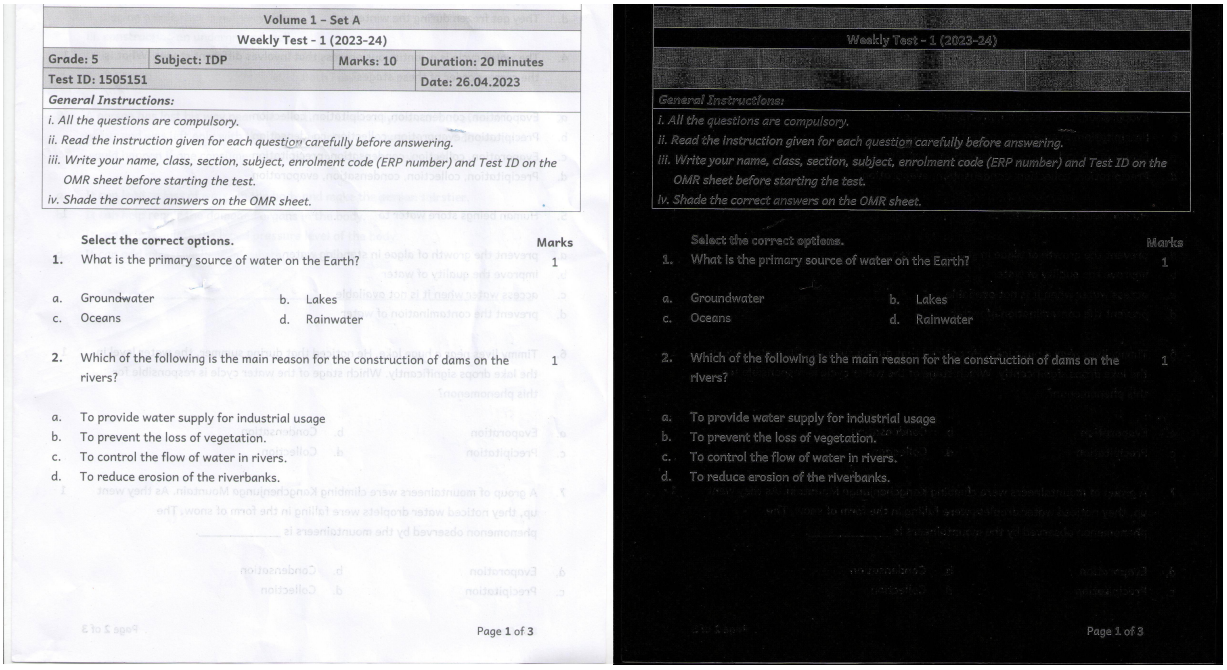
OpenCV has a built-in function for performing convolutions. We demonstrate a sample code as follows for performing a de-noising of image using convolution:

```
1. import cv2
2. import numpy as np
3.
```

```
# Read the input image
image = cv2.imread("input_images/4_ThresholdingImage.jpg")

# Check if the image was successfully loaded
if image is None:
    print("Unable to load the image.")
    exit()

# Define the kernel for de-noising convolution
kernel = np.array([[1, 1, 1],
                   [1, -8, 1],
                   [1, 1, 1]])

# Perform image convolution
convolved_image = cv2.filter2D(image, -1, kernel)


stacked_results = np.hstack((image, convolved_image))
# Display the original image and the convolved output image
cv2.imshow('Convolution', stacked_results)

```

25. *# Wait for key press and then close all windows*
26. cv2.waitKey(0)
27. cv2.destroyAllWindows()
28. cv2.imwrite("output_images/convolution.jpg", stacked_results)

The program reads the noisy input image and defines a denoising kernel for convolution. In this example, a kernel that enhances edges and suppresses noise is used. You can adjust the kernel values or use a different kernel to experiment with different denoising effects. The program then applies the denoising convolution operation using the **filter2D()** function from OpenCV. Finally, the program displays the noisy input image and the denoised output image using OpenCV's **imshow()** function. When you run the program, you will see the noisy input image and the denoised output image displayed side by side. Please note that the effectiveness of denoising depends on the characteristics of the noise and the specific kernel used. You may need to experiment with different kernels or denoising techniques to achieve optimal results for your particular noisy image. The previous code produces the results seen in *Figure 4.16*. As you can see, it looks similar to thresholding. But we have not used any thresholding formulae here. Please refer to the following figure:



Volume 1 – Set A

Weekly Test – 1 (2023-24)

Grade: 5	Subject: IDP	Marks: 10	Duration: 20 minutes
Test ID: 1505151			Date: 26.04.2023

General Instructions:

i. All the questions are compulsory.
ii. Read the instruction given for each question carefully before answering.
iii. Write your name, class, section, subject, enrolment code (ERP number) and Test ID on the OMR sheet before starting the test.
iv. Shade the correct answers on the OMR sheet.

Select the correct options. Marks

1. What is the primary source of water on the Earth? 1

a. Groundwater b. Lakes
c. Oceans d. Rainwater

2. Which of the following is the main reason for the construction of dams on the rivers? 1

a. To provide water supply for industrial usage
b. To prevent the loss of vegetation.
c. To control the flow of water in rivers.
d. To reduce erosion of the riverbanks.

Page 1 of 3

Weekly Test – 1 (2023-24)

General Instructions:

i. All the questions are compulsory.
ii. Read the instruction given for each question carefully before answering.
iii. Write your name, class, section, subject, enrolment code (ERP number) and Test ID on the OMR sheet before starting the test.
iv. Shade the correct answers on the OMR sheet.

Select the correct options. Marks

1. What is the primary source of water on the Earth? 1

a. Groundwater b. Lakes
c. Oceans d. Rainwater

2. Which of the following is the main reason for the construction of dams on the rivers? 1

a. To provide water supply for industrial usage
b. To prevent the loss of vegetation.
c. To control the flow of water in rivers.
d. To reduce erosion of the riverbanks.

Page 1 of 3

***Figure 4.16**: Convolution demonstration*

Convolution is a fundamental operation in deep learning-based image processing. Its abilities to extract features while preserving spatial relationships and providing translation invariance make it a key component for building powerful and effective deep learning models. It is not an exaggeration to say that the deep learning vision neural networks that we know today might not have been possible without convolution. We shall discuss this more in the next chapter.

Conclusion

Classical algorithms have played a pivotal role in shaping the field of computer vision. They provide a solid foundation for understanding and analyzing visual data, enabling machines to interpret and extract meaningful information from images and videos. These algorithms continue to be relevant and serve as important benchmarks for comparison as new techniques emerge.

Exercises

1. Implement an image filtering algorithm, such as a Gaussian blur or a median filter, from scratch using only basic image processing operations.
2. Create a program that applies image morphological operations, such as erosion and dilation, to enhance or modify specific features in an image. Experiment with different structuring elements and explore their effects.
3. Implement a program to perform image stitching, where multiple overlapping images are combined to create a panoramic view. Experiment with different algorithms, such as SIFT or SURF, for feature detection and matching.
4. Develop a program that performs image segmentation using a GraphCut algorithm. Experiment with different energy functions and explore their impact on the segmentation results.
5. Implement a program to track the motion of objects in a video sequence using optical flow algorithms, such as Lucas-Kanade or

Horn-Schunck. Test the program on different videos with varying object motions.

6. Create a program that performs image registration, aligning two or more images to a common coordinate system. Explore different registration techniques, such as feature-based or intensity-based registration.
7. Implement a program that performs image inpainting, filling in missing or damaged parts of an image using surrounding information. Experiment with different inpainting algorithms, such as exemplar-based or patch-based methods.
8. Create a program that performs image super-resolution, enhancing the resolution and quality of low-resolution images. Explore different super-resolution techniques, such as single-image or multi-image super-resolution.

Key terms

- **Binarization** :A process by which an image is modified to consist of exactly two colours. Usually, the colours chosen are black and white.
- **Canny algorithm**: Algorithm for detecting edges of objects in an image.
- **Closing**: A morphological operation to fill in gaps and close small holes in images.
- **Clustering**: A machine learning technique for grouping the pixels of an object together.
- **Convolution**: A mathematical operation where the shape of one function is modified by another function.
- **Convolutional neural network**: A deep learning neural network processing arrays of data. Very popular for image processing.
- **Denoising**: An image processing technique for reducing spots and discoloration with minimal loss of quality.
- **Dilation**: One of two fundamental morphological image processing.
- **Erosion**: One of two fundamental morphological image processing.

- **Image pyramids**: A type of image representation where in the original image repeatedly smoothened and subsampled.
- **K-means**: A statistical algorithm for achieving clustering.
- **Laplacian algorithm**: Algorithm for detecting edges of objects in an image.
- **Opening**: A morphological operation to eliminate minor protrusions in the image.
- **Otsu algorithm**: Algorithm for image binarization.
- **Region growing**: Algorithm for segmenting images.
- **Scale invariance**: An algorithm property where algorithms do not lose their effectiveness with changes to scale of image.
- **Search image**: The image in which we want to locate an object of interest.
- **Sobel algorithm**: Algorithm for detecting edges.
- **Super pixels**: A local grouping of pixels in an image which carry more semantic meaning than individual pixels.
- **Supervised learning**: A machine learning paradigm where the data consists of features and have a label associated to them.
- **Template**: The object of interest which we want to locate in a given image.
- **Thresholding**: The simplest algorithm for image segmentation.
- **Unsupervised learning**: A machine learning paradigm where the data consists of features but do not have a label associated to them.
- **Watershed algorithm**: An algorithm for image segmentation.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

OceanofPDF.com

Chapter 5
Deep Learning and CNNs

Introduction

Deep learning has revolutionized the field of artificial intelligence, enabling remarkable progress in areas such as computer vision, natural language processing, and machine translation. This chapter explores the multifaceted landscape of deep learning. Moreover, it investigates various architectural approaches, such as **convolutional neural networks (CNNs),** elucidating their mathematical foundations, strengths, and applications. Furthermore, the chapter introduces training and inference processes in deep learning, focusing on techniques for efficient and accurate predictions. It highlights the significance of optimization functions, activation functions, and model compression techniques in enhancing inference speed, reducing computational requirements, and ensuring robustness. This chapter aims to comprehensively examine these topics and introduce deep learning architectures and techniques that propel this rapidly evolving field.

Structure

The chapter discusses the following topics:

- History of deep learning
- Perceptron
- Shallow learning networks

- Deep learning networks
- Weights, biases, and activation functions
- Convolutional neural networks
- Deep learning process

Objectives

This chapter aims to familiarize the readers with the fundamental and relevant concepts of deep learning. By the end of the chapter, you should be able to understand the most commonly used terms in deep learning and write Python code for implementing deep learning programs. The dedicated libraries for deep learning, like Tensorflow, Keras, PyTorch, and so on, are not covered in this chapter.

History of deep learning

The history of deep learning and perceptrons dates back several decades. Let us take a journey through their key milestones:

- **Perceptrons and the perceptron rule (1950s-1960s)**: The concept of perceptrons was introduced by *Frank Rosenblatt* in the late 1950s. *Rosenblatt*'s work was influenced by the idea of simulating neural networks to mimic human cognition. The perceptron, a single-layer neural network, was designed to perform binary classification tasks. *Rosenblatt* developed the perceptron learning algorithm, also known as the **Perceptron rule**, which adjusted the weights and bias of the perceptron to learn from training examples.
- **Limitations and the Perceptron controversy (1960s)**: In 1969, *Marvin Minsky* and *Seymour Papert* published the book *Perceptrons*, highlighting the limitations of single-layer perceptrons. They demonstrated that perceptrons could not learn certain logical functions, such as the XOR function, which required more complex decision boundaries. This led to a decline in interest and funding for neural networks and perceptrons during this period.
- **Backpropagation and multilayer perceptrons (1970s-1980s)**: In the 1970s, the backpropagation algorithm was independently rediscovered by multiple researchers, including *Paul Werbos*, *David*

Rumelhart, and *Ronald Williams*. Backpropagation enabled the training of **multilayer perceptrons** (**MLP**s) with multiple hidden layers. MLPs could learn non-linear decision boundaries and overcome the limitations of single-layer perceptrons. This sparked renewed interest in neural networks and marked an important milestone in deep learning.

- **Neural network winter (1990s-early 2000s)**: Despite the advances in backpropagation and MLPs, neural networks faced challenges during this period. The complexity of training deep networks, the limited availability of computational resources, and the emergence of other machine learning algorithms, such as **support vector machines** (**SVM**s), led to a decline in interest and research in neural networks. This period became known as the **neural network winter.**
- **Resurgence of deep learning (mid-2000s onwards)**: Deep learning experienced a resurgence in the mid-2000s, driven by several factors. These included the availability of large-scale datasets, more powerful computational resources (for example, GPUs), and advances in optimization algorithms. Researchers began developing novel architectures, such as CNNs for image recognition and **recurrent neural networks** (**RNN**s) for sequential data processing. These deep learning architectures achieved breakthroughs in various domains, including computer vision, natural language processing, and speech recognition.
- **Deep learning revolution (2010s onwards)**: The 2010s witnessed an explosion of research and applications in deep learning. This period was marked by advancements in architecture design (for example, deep residual networks, generative adversarial networks), the development of frameworks and libraries (for example, TensorFlow, PyTorch), and the use of deep learning in numerous fields, including healthcare, finance, autonomous vehicles, and more. Deep learning models consistently achieved state-of-the-art performance on various complex tasks, cementing their position as a powerful tool in artificial intelligence.

Throughout this history, the perceptron remained an influential concept, serving as the foundation for more advanced neural network architectures.

The perceptron rule and the insights gained from studying its limitations paved the way for developing powerful deep learning techniques that we rely on today.

Perceptron

A perceptron is a fundamental building block of artificial neural networks inspired by the structure and functioning of biological neurons. It is a simple mathematical model used for binary classification tasks, meaning it can determine whether an input belongs to one class or another. A perceptron is a mathematical function that takes multiple inputs, applies weights to each input, sums them up, and then passes the sum through an activation function to produce an output. The activation function helps introduce non-linearity into the perceptron, allowing it to learn complex decision boundaries.

Mathematically, let us consider a perceptron with *n* inputs. Each input is associated with a weight, denoted by *w*, and a bias term, denoted by *b*. The bias term is an additional input with a fixed value of *1*, allowing the perceptron to adjust the decision boundary independently of the input values. The weights and bias are learnable parameters that the perceptron adjusts during training. Given the input vector $x = [x_1, x_2, ..., x]$, the weighted sum of the inputs is calculated as:

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

The activation function, typically a step function or a sigmoid function, takes the weighted sum **(z)** as input and produces the output of the perceptron, denoted by *y*. In the case of a step function, the output is binary:

$$y = step(z) = \{1 \text{ if } z \geq 0, 0 \text{ if } z < 0\}$$

In the case of a sigmoid function, the output is a continuous value between *0* and *1*:

$$y = sigmoid(z) = \frac{1}{1 + e^{-z}}$$

During training, the perceptron adjusts its weights and bias to minimize the error between its predicted output and the true output. This process is often

accomplished using gradient descent and backpropagation algorithms, where the error is propagated backward through the network to update the parameters. See *Figure 5.1* for how perceptron works.

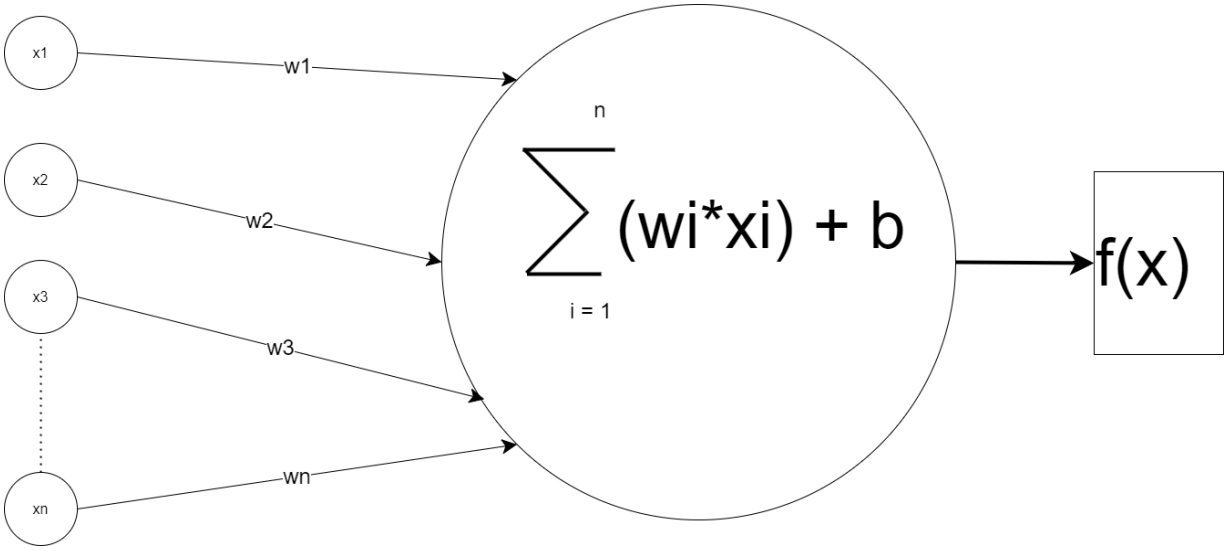


***Figure 5.1**: Perceptron*

The perceptron is the basis for more complex neural network architectures, such as MLPs, which stack multiple perceptron layers together to enable learning non-linear decision boundaries for more complex tasks. Please see the code as follows for creating a perceptron for performing data classification:

```
1. from sklearn.datasets import make_classification
2. from sklearn.linear_model import Perceptron
3. from sklearn.model_selection import train_test_split
4. from sklearn.metrics import accuracy_score
5.
6. # Generate a random binary classification dataset
7. X, y = make_classification(n_samples=100, n_features=20, n_inform
   ative=10, n_redundant=10, random_state=42)
8.
```

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a perceptron classifier
perceptron = Perceptron()

# Train the perceptron on the training data
perceptron.fit(X_train, y_train)

# Make predictions on the test data
y_pred = perceptron.predict(X_test)

# Calculate the accuracy of the perceptron
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Shallow learning networks

Shallow learning, traditional machine learning or shallow neural networks, refers to algorithms with only a few or no hidden layers in their architecture. These algorithms typically involve linear models, such as logistic regression, **support vector machines** (**SVM**s), or decision trees.

From a mathematical perspective, shallow learning algorithms aim to find a linear or non-linear function that maps input features (denoted as **x**) to output labels or predictions (denoted as **y**). The goal is to learn a set of

parameters (weights and biases) that minimize an objective function, such as the sum of squared errors or the hinge loss.

For example, in logistic regression, the model learns the **weights** (**w**) that minimize the logistic loss function, defined as the negative log-likelihood of the observed labels given the predicted probabilities. This optimization is typically achieved using techniques like gradient descent, where the gradients of the loss function with respect to the parameters are computed and used to update the weights iteratively.

Deep learning networks

Deep learning, on the other hand, refers to the class of algorithms that utilize deep neural networks with multiple hidden layers to learn intricate representations and hierarchical patterns in the data. Deep learning models consist of interconnected layers of artificial neurons (perceptrons) that progressively transform the input data to generate predictions.

Mathematically, deep learning involves composing multiple functions to form a complex computation graph. Each layer of the neural network applies linear transformations (matrix multiplications) to the inputs and passes the result through a non-linear activation function (for example, sigmoid, ReLU) to introduce non-linearity. This allows the network to learn complex, non-linear relationships between the input and output.

The training process in deep learning typically involves minimizing a loss function through backpropagation. This technique computes the gradients of the loss function with respect to the model parameters using the chain rule of calculus, allowing for efficient optimization of the weights and biases in each layer. Techniques like **stochastic gradient descent** (**SGD**) or its variants are commonly used to update the parameters iteratively.

One of the key advantages of deep learning is its ability to automatically learn hierarchical representations of the data, extracting meaningful features at different levels of abstraction. This hierarchical representation allows deep learning models to excel in tasks such as image recognition, natural language processing, and speech recognition, where the underlying data has intricate structures.

Shallow learning algorithms rely on simpler models with fewer or no hidden layers, whereas deep learning algorithms employ deep neural networks with multiple hidden layers. The mathematical foundations of shallow learning involve optimizing a simpler objective function using linear or non-linear models. In contrast, deep learning involves composing multiple layers of non-linear transformations to learn intricate representations.

Weights, biases, and activation functions

We have been discussing terms like weights, biases, and activation functions. But what exactly are they? Let us understand the intuition and math behind them. These are essential components of neural networks, including both shallow and deep learning models.

Weight

In a neural network, weights (often denoted by w) are parameters associated with the connections between neurons. Each connection between two neurons has an associated weight that determines the strength and impact of the input from one neuron on the other. Mathematically, the weighted sum of inputs is calculated by multiplying each input by its corresponding weight and summing them up.

Let us consider a neuron with n inputs. The inputs are represented as a vector $x = [x_1, x_2, ..., x_n]$, and the weights are represented as a vector $w = [w_1, w_2, ..., w_n]$. The weighted sum of inputs, denoted as z, is computed as follows:

$$z = w_1x_1 + w_2x_2 + ... + w_nx_n$$

The weights in a neural network are learnable parameters, adjusted during the training process. The learning algorithm, such as gradient descent, optimizes these weights to minimize errors between the network's predictions and the true values.

Bias

Biases (often denoted by b) are additional parameters associated with each neuron in a neural network. Biases provide additional freedom and allow

the network to adjust the decision boundary independently of the input values. Mathematically, biases are conceptually similar to weights but are associated with the neuron rather than the connections.

In a neuron with a bias term, the weighted sum of inputs z is adjusted by adding the bias term:

$$z = w_1x_1 + w_2x_2 + ... + w_nx_n + b$$

Biases allow the neural network to learn offset values and capture information that cannot be represented solely through the input features. Like weights, biases are also learnable parameters that are optimized during training.

Activation function

Activation functions introduce non-linearity to the output of a neuron or a layer in a neural network. They determine whether a neuron should be activated (fired), based on the input it receives. Activation functions help neural networks learn and represent complex, non-linear relationships in the data. There are various types of activation functions, but let us focus on two commonly used ones:

- **Step function**: The step function is a simple binary activation function that outputs either *0* or *1* based on a threshold. It can be represented mathematically as follows:

  $$y = step(z) = \{1 \text{ if } z \geq 0, 0 \text{ if } z < 0\}$$

  The step function is typically used in perceptrons and early neural networks.

- **Sigmoid function**: Sigmoid function was discussed earlier.

Activation functions serve as the **thresholding** mechanism of a neuron, determining whether the neuron should be activated based on the weighted sum of inputs. They introduce non-linearities to the neural network, enabling it to learn and represent complex relationships in the data.

Note: Intuitively, these concepts can be understood as below. Weights determine the strength and impact of inputs in a neural network, biases provide an additional level of freedom to adjust decision

boundaries, and activation functions introduce non-linearities to the output of neurons, allowing neural networks to learn complex patterns and relationships in the data.

Optimization function

What is an optimization function? Is it another name for the activation function? What is its purpose? It is common for beginners to need clarification on the objectives of the activation function and the optimization functions' objectives.

An optimization function, also known as a loss function or objective function, is a mathematical measure that quantifies how well a machine learning model performs on a given task. The purpose of an optimization function is to guide the learning process by providing a measure of the model's performance, allowing the model's parameters to be adjusted to minimize the defined objective.

The choice of an optimization function depends on the specific task and the nature of the problem being solved. The objective may vary, such as minimizing the error between predicted and true values (in regression tasks), maximizing the likelihood of observing the true labels (in classification tasks) or minimizing a combination of various factors (in multi-objective optimization).

Mathematically, let us denote the optimization function as j, which takes the model's predicted outputs $\hat{y}$ and the true outputs y as inputs. The optimization function computes a scalar value that represents the model's performance. The target is to find the combination of model parameters that minimize this function.

$$J(\hat{y}, y)$$

The optimization process typically involves an iterative algorithm, such as gradient descent, that updates the model's parameters in the direction that reduces the value of the optimization function. The gradients of the optimization function with respect to the model's parameters indicate the direction of steepest descent, allowing the parameters to be adjusted accordingly. However, optimization functions have certain limitations and challenges, mentioned as follows:

- **Local minima**: The optimization landscape can be complex, and there may be multiple local minima where the optimization algorithm can get stuck. This implies the algorithm may converge to a suboptimal solution instead of the global minimum.
- **Plateaus**: In some cases, the optimization function can have flat regions called plateaus, where the gradients become very small. This can significantly slow down the learning process, as the parameter updates become negligible.
- **Computational complexity**: Depending on the complexity of the optimization function and the size of the dataset, the optimization process can be computationally expensive and time-consuming. Deep learning models with millions of parameters can require substantial computational resources.
- **Sensitivity to hyperparameters**: The choice of hyperparameters, such as learning rate or regularization strength, can significantly affect the optimization process. Selecting appropriate hyperparameters often requires experimentation and tuning.
- **Overfitting**: Optimization functions typically consider the performance of the training data. However, the model's goal is to generalize well to unseen data. Over-optimizing the training data can lead to overfitting, where the model performs poorly on new data.

Addressing these limitations often requires careful algorithmic design and regularization techniques to avoid getting trapped in poor solutions or overfitting the training data. Researchers continually develop new optimization algorithms and techniques to improve the efficiency and effectiveness of training deep learning models.

An optimization function is a mathematical measure used to guide the learning process in machine learning models. It quantifies the model's performance, and the goal is to minimize this function by adjusting the model's parameters. However, optimization functions have limitations, including the possibility of local minima, plateaus, computational complexity, sensitivity to hyperparameters, and the risk of overfitting.

Here are some code samples to implement deep learning. The below code implements a multi-layer perceptron, that is, deep learning neural network,

for performing data classification:

```
from sklearn.datasets import make_classification
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Generate a random binary classification dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=10, n_redundant=10, random_state=42)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create an MLP classifier
mlp = MLPClassifier(hidden_layer_sizes=(5,), activation='relu', solver='adam', random_state=42)

# Train the MLP on the training data
mlp.fit(X_train, y_train)

# Make predictions on the test data
```

```
y_pred = mlp.predict(X_test)

# Calculate the accuracy of the MLP
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Now let us see some code that performs regression on some real-world data. Following is the code for predicting a continuous variable on the tips dataset:

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Load the tips dataset from Seaborn
tips_df = pd.read_csv('https://raw.githubusercontent.com/mwaskom/seaborn-data/master/tips.csv')

# Preprocess the data
X = tips_df[['total_bill', 'size']]
y = tips_df['tip']
```

```
# Standardize the input features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Create an MLP regressor
mlp = MLPRegressor(hidden_layer_sizes=(10,), activation='relu', solver='adam', random_state=42)

# Train the MLP on the training data
mlp.fit(X_train, y_train)

# Make predictions on the test data
y_pred = mlp.predict(X_test)

# Calculate the root mean squared error (RMSE) of the MLP
rmse = mean_squared_error(y_test, y_pred, squared=False)
print("RMSE:", rmse)
```

Convolutional neural networks

A **convolutional neural network** (**CNN**) is a specialized type of deep neural network commonly used for analyzing visual data, such as images. CNNs are designed to effectively capture and process spatial relationships within the data using convolutional layers. Let us explore the mathematical details of CNNs.

The fundamental building block of a CNN is the convolutional layer. This layer applies a set of learnable filters, also known as **convolutional kernels** or **filters**, to the input data. Each filter is a small matrix of weights. We discussed convolution on *Chapter 4*.

Mathematically, let us consider an input feature map (also known as **an activation map**) denoted as X. A single filter of size $f \times f$ is convolved (element-wise multiplication and summation) with a corresponding local receptive field in the input map. The output of this operation is a feature map, denoted as Y, which represents the presence or activation of certain patterns or features in the input. The convolution operation can be represented mathematically as follows:

$$Y(i, j) = f(X * W(i, j) + b(i, j))$$

$Y(i, j)$ represents the element at position (i, j) in the output feature map. X is the input feature map, $W(i, j)$ is the filter at position (i, j) with its corresponding weights, and $b(i, j)$ is the bias term associated with that filter.

- **Activation function**: Following the convolution operation, an activation function is applied element-wise to the output feature map to introduce non-linearity. Commonly used activation functions in CNNs include **rectified linear unit** (**ReLU**) and its variants. ReLU is defined mathematically as:

  $$f(x) = max(0, x)$$

  The activation function helps the CNN model to capture non-linear relationships and introduces sparsity by zeroing out negative values.

- **Pooling layer**: In CNNs, pooling layers are typically used to downsample the spatial dimensions of the feature maps, reducing the computational complexity and extracting higher-level features. Max pooling is a commonly used pooling operation where the maximum

value within a local neighborhood is selected. Mathematically, max pooling can be represented as follows:

$$Y(i, j) = max(X(i \times s : (i+1) \times s, j \times s : (j+1) \times s))$$

Here, $Y(i, j)$ represents the element at position (i, j) in the downsampled feature map, and $X(i \times s : (i+1) \times s, j \times s : (j+1) \times s)$ refers to the local neighborhood of size $s \times s$ in the input feature map.

- **Fully connected layers**: After the convolutional and pooling layers, the feature maps are typically flattened into a vector and passed through one or more fully connected layers, similar to those in traditional neural networks. Fully connected layers connect each neuron in one layer to every neuron in the following layer, allowing the model to learn complex combinations of features. Mathematically, the fully connected layers involve matrix multiplications and activation functions similar to those in shallow neural networks.

By combining multiple convolutional layers, pooling layers, and fully connected layers, CNNs can learn hierarchical representations of the input data, capturing low-level features in the earlier layers and higher-level abstractions in the deeper layers. The training of CNNs often involves gradient-based optimization techniques, such as backpropagation and stochastic gradient descent, to adjust the weights and biases throughout the network.

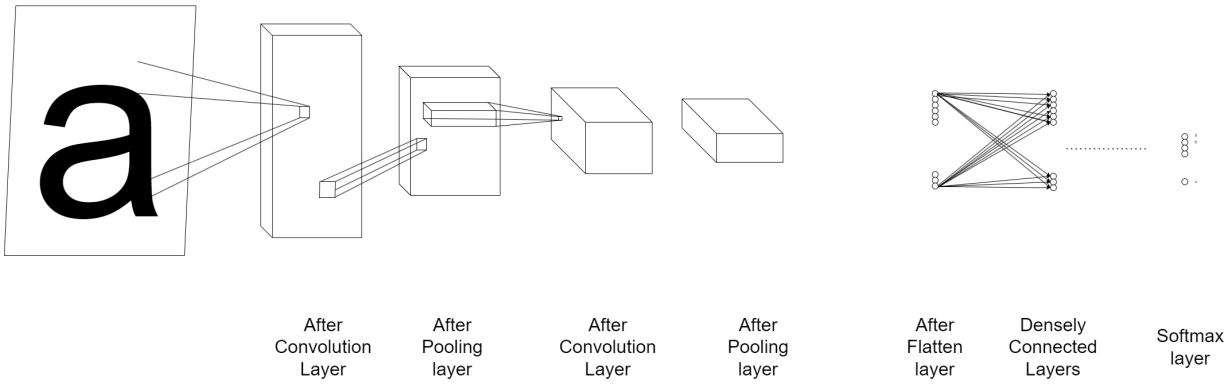


Figure 5.2*: Image processing in a deeply connected CNN*

CNN uses convolutional layers to perform local receptive field operations on the input data, followed by activation functions and pooling layers to extract features and down sample the spatial dimensions. Fully connected layers are then used to learn high-level representations. CNNs excel in analyzing visual data because they can effectively capture spatial relationships within the data. Refer to *5.2* for a visual depiction of how CNNs process an image for classification.

CNNs versus fully connected networks

The key differences between a CNN and a fully connected network lie in their architectural design and the operations performed on the data. Let us tabulate some of them here for better understanding here in *Table 5.1*:

	CNNs	**Fully connected networks**
Convolutional layers	Convolutional layers play a crucial role. These layers apply learnable filters to small local regions of the input data. The filters capture spatial patterns by performing convolutions, which involve element-wise multiplication and summation. The output of a convolutional layer is a feature map that represents the activation or presence of specific patterns in the input.	Do not have specific layers dedicated to spatial operations. Instead, every neuron in one layer is connected to every neuron in the next layer, without considering the spatial structure of the input.

Weight sharing	This is a notable aspect of CNNs. In convolutional layers, the same set of filters is applied across the entire input, regardless of location. This weight sharing allows the network to efficiently learn spatial hierarchies of features and significantly reduces the number of parameters in the model. By sharing weights, CNNs can learn to recognize certain patterns regardless of their location in the input.	Weight sharing does not exist in these networks. Each neuron in one layer has its own unique set of weights connecting it to every neuron in the subsequent layer. This lack of weight sharing results in a higher number of parameters compared to CNNs, making fully connected networks more computationally expensive and prone to overfitting, especially when dealing with high-dimensional inputs like images.
Pooling layers	Pooling layers are commonly used in CNNs to downsample the spatial dimensions of the feature maps, reducing computational complexity and extracting higher-level features. Max pooling is a widely used pooling operation, which selects the maximum value within a local neighborhood.	Fully connected networks do not incorporate pooling layers. They typically operate on the full-resolution feature maps without downsampling.

***Table 5.1**: Differences between CNNs and fully connected networks*

Here is some code for implementing a convolutional neural network-based deep learning in Python:

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load the digits dataset from scikit-learn
digits = load_digits()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size=0.2, random_state=42)

# Preprocessing - Scale the input features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
19.
20. # Create a pipeline for feature extraction and classification
21. pipeline = Pipeline([
22.     (‹classifier›, SVC())  # You can replace SVC with any
    other scikit-learn classifier
23. ])
24.
25. # Train the model on the training data
26. pipeline.fit(X_train_scaled, y_train)
27.
28. # Make predictions on the test data
29. y_pred = pipeline.predict(X_test_scaled)
30.
31. # Calculate the accuracy of the model
32. accuracy = accuracy_score(y_test, y_pred)
33. print("Accuracy:", accuracy)
```

In this program, we use the **load_digits** function from scikit-learn to load the digits dataset, which consists of images of handwritten digits. We split the dataset into training and testing sets using **train_test_split**, allocating 20% of the data for testing. Next, we preprocess the data by scaling the input features using **StandardScaler**. We create a pipeline using pipeline, which allows us to chain multiple steps together. In this case, we only have one step, which is the classifier (SVC in this example, but you can replace it with any other scikit-learn classifier). We train the model on the training data using the fit method of the pipeline. After training, we use the model to

make predictions on the test data with the predict method of the pipeline and store the predictions in **y_pred**. Finally, we calculate the accuracy of the model by comparing the predicted labels (**y_pred**) with the true labels (**y_test**) using the **accuracy_score** function from scikit-learn.

Please keep in mind that this approach using scikit-learn is a simple feature-based approach rather than a true CNN-based approach. For more complex image classification tasks, it is recommended to use dedicated deep learning libraries such as TensorFlow or PyTorch, which provide specialized tools and architectures for convolutional neural networks. We will discuss these libraries in the next chapter.

Deep learning process

Training and inference are two fundamental processes in deep learning. While they share certain similarities, they also have distinct characteristics. Let us explore each process in detail, along with their similarities, differences, and techniques/tricks commonly used:

Training

The training process involves training a deep learning model using labeled data to learn the underlying patterns and relationships within the data. It typically consists of the following steps:

1. **Forward pass**: During the forward pass, the input data is fed through the layers of the model, and the output or predictions are generated. The model's parameters (weights and biases) are used to transform the input data and produce the output.
2. **Loss calculation**: The loss function is used to quantify the discrepancy between the model's predictions and the true labels. It is a measure of how well the model is performing on the training data. The goal is to minimize this loss during training.
3. **Backward pass**: The backward (backpropagation) pass involves computing the gradients of the loss with respect to the model's parameters. This is done using the chain rule of calculus, allowing the gradients to flow backward through the layers of the model. The

gradients indicate the direction and magnitude of the parameter updates required to minimize the loss.

4. **Parameter update**: The model's parameters are updated using an optimization algorithm, such as gradient descent or variants. The updates are determined by multiplying the gradients with a learning rate, which controls the step size taken in the parameter space.
5. **Repetition or iteration**: Steps 1 to 4 are repeated for multiple iterations or epochs to refine the model's parameters and improve its performance. The model is exposed to the training data multiple times to gradually learn and adjust its internal representations.

Techniques in training

The purpose of techniques and tricks in training deep learning models is to improve their performance, address common challenges, and enhance their generalization capabilities. These techniques aim to overcome limitations such as overfitting, vanishing/exploding gradients, slow convergence, and poor optimization. By applying these techniques and tricks, deep learning models can achieve better performance, faster convergence, improved generalization, and enhanced efficiency. These techniques are crucial for addressing common challenges in training deep learning models and ensuring their effectiveness in real-world applications. The techniques are as follows:

- **Activation functions**: Choosing appropriate activation functions can affect the model's capacity to learn complex relationships.
- **Regularization**: Techniques like L1 or L2 regularization help prevent overfitting by adding penalty terms to the loss function.
- **Dropout**: Randomly dropping units during training helps to regularize the model and prevent co-adaptation of neurons.
- **Batch normalization**: Normalizing the input data within each mini-batch helps stabilize and speed up training.
- **Learning rate scheduling**: Adjusting the learning rate during training can help find a balance between convergence and avoiding overshooting.

Inference process

The inference process involves making predictions or generating outputs from the trained model on unseen or test data. It typically consists of the following steps:

1. **Forward pass**: Similar to the training process, the input data is fed through the layers of the model, and predictions or outputs are generated.
2. **Output generation**: The model's predictions or outputs are generated based on the forward pass. The specific output depends on the task, such as class probabilities for classification or continuous values for regression.

Techniques/tricks in inference

The purpose of techniques and tricks in inference for deep learning models is to optimize their performance, improve efficiency, and address specific challenges that arise during the deployment or utilization of the models. These techniques enhance the model's prediction accuracy, reduce computational requirements, increase speed, and ensure robustness. Some of the techniques used in inference are listed as follows:

- **Dropout inference**: During inference, dropout is typically turned off or modified to retain all units for more stable predictions.
- **Ensemble methods**: Combining predictions from multiple models, such as using model averaging or bagging, can improve performance.
- **Quantization**: Reducing the precision of the model's parameters can reduce memory requirements and increase inference speed.
- **Pruning**: Removing unnecessary connections or parameters in the model to reduce its size and improve inference efficiency.
- **Knowledge distillation**: Transferring knowledge from a larger, more complex model (teacher) to a smaller model (student) to improve its performance.

Training and inference in deep learning involve similar steps of forward pass and output generation but differ in terms of backpropagation, loss calculation, and parameter updates. Various techniques and tricks are employed during each process to improve performance, optimize model parameters, and manage computational resources effectively. Refer to *Table 5.2* for a quick view of differences:

	Training	Inference
Forward pass	Y	Y
Model architecture	Y	Y
Backward pass	Y	N
Loss calculation	Y	N
Parameter update	Y	N
Data usage	Y	N

***Table 5.2**: Comparison of sub-steps in training and inference*

Conclusion

In conclusion, this chapter has explored deep learning architectural approaches and essential techniques. The discussion has encompassed the architectural approaches of CNN. Training techniques for model optimization and inference strategies for efficient predictions have also been addressed. In the next chapter we shall discuss the inferencing architectures for computer vision and shall introduce the OpenCV DNN module.

Key terms

- **Perceptron:** A simple machine learning algorithm that can be used to classify data into two categories.
- **Deep learning:** A machine learning technique that uses artificial neural networks to learn from data.

- **Shallow learning:** A machine learning technique that uses simple algorithms to learn from data.
- **Weights:** Coefficients that determine how much each input feature contributes to the final output.
- **Biases:** A constant offset that is added to the weighted sum of the input features before it is passed to the activation function.
- **Activation functions:** Mathematical functions used to introduce non-linearity into artificial neural networks.
- **Optimization function:** Algorithms used to update the weights and biases of the network with the objective to minimize the loss.
- **Convolutional neural network (CNN):** A type of neural network that is specifically designed for processing and analysing visual imagery.
- **Tensorflow:** It is a deep learning library.
- **PyTorch:** It is a deep learning library.

Exercises

1. Implement a classification using perceptron on the iris dataset.
2. After training the deep learning model, store the model weights to disk. Then read them from a separate python program and perform inference on data.

Chapter 6
OpenCV DNN Module

Introduction

In this chapter, we will delve into the OpenCV **deep neural networks** (**DNNs**) module. It is a powerful tool that combines the best aspects of both computer vision and deep learning. Over the years, this module has seen significant development, expanding its support for multiple deep learning frameworks, pre-trained models, and hardware acceleration. With a rich history of integration and enhancement, the DNN module has become an indispensable asset for building cutting-edge computer vision applications.

With a comprehensive overview of the OpenCV DNN module's key aspects, including its history, supported and unsupported layers, and essential classes, this chapter shall explore the depths of the OpenCV DNN module and explain the building blocks to powerful computer vision solutions.

Structure

This chapter discusses the following topics:

- Deep learning frameworks
    - TensorFlow
    - PyTorch

    - Keras
- Inference for computer vision
    - Local inferencing
        - Local CPUs
        - Local GPUs
    - Cloud
    - Edge computing
- OpenCV DNN module
    - History
    - Features and limitations
        - Capabilities
        - Limitations
        - Considerations
    - Supported layers
    - Unsupported layers and operations
    - Important classes

Objectives

This chapter aims to explain the OpenCV DNN module without resorting to software coding. We will start with a cursory view of deep learning frameworks, and solution architectures for computer vision. We will then introduce the OpenCV DNN module. We will discuss its features, capabilities, and discuss deep learning layers and operations supported by DNN module. We shall also be aware of the limitations inherent in the module.

Deep learning frameworks

We begin by discussing about the deep learning frameworks available for programmers. As explained earlier, building large and complex neural networks using vanilla Python libraries like sklearn can be complex and

ineffective. Deep learning training and inference have specialized needs that cannot be addressed by libraries like scikit. The two popular deep learning frameworks are TensorFlow and PyTorch. Any detailed discussion of these libraries is beyond the scope of this book. We will only discuss them in a perfunctory way.

TensorFlow

TensorFlow is an open-source deep learning framework developed by Google Brain. It offers a comprehensive ecosystem for building and deploying machine learning models. TensorFlow provides a flexible and efficient way to define and train neural networks on various hardware platforms, including CPUs, GPUs, and even specialized accelerators like TPUs. Its computational graph abstraction allows for efficient execution and distributed computing. TensorFlow also offers tools for visualization, model deployment, and serving in production environments. With its wide adoption and extensive community support, TensorFlow is widely used in research and industrial applications.

PyTorch

PyTorch is another popular deep learning framework widely embraced by researchers and practitioners. Developed by Facebook's AI research lab, PyTorch is known for its dynamic computational graph, which provides a more intuitive and flexible approach to model development. It allows you to build neural networks using Pythonic syntax and provides automatic differentiation for efficient gradient computation during training. PyTorch's eager execution mode enables interactive experimentation and debugging. Additionally, PyTorch's seamless integration with the Python scientific computing ecosystem makes it easy to leverage powerful libraries like NumPy and SciPy. PyTorch has gained significant popularity in the deep learning community due to its user-friendly interface and strong support for research workflows.

Keras

Keras is an open-source high-level neural network API written in Python. It serves as a user-friendly and intuitive interface to build and experiment with deep learning models. One of the key strengths of Keras lies in its

simplicity and ease of use, making it a popular choice for beginners and researchers alike. Keras was a standalone library, but it became part of TensorFlow starting from TensorFlow version 2.0. As a result, it now works seamlessly with TensorFlow as its high-level API. However, one notable difference between Keras and TensorFlow is their level of abstraction. Keras abstracts away many of the low-level details of TensorFlow, providing a more streamlined development experience, whereas TensorFlow gives more control and flexibility, enabling developers to customize models and algorithms at a lower level. This distinction often makes Keras a preferred option for quick prototyping and building models efficiently, while TensorFlow caters to developers who require fine-grained control over their deep learning workflows.

Both TensorFlow and PyTorch offer rich libraries and APIs for common deep learning tasks such as image classification, object detection, natural language processing, and more. They provide extensive documentation, tutorials, and pre-trained models to accelerate the development process. Moreover, these frameworks support advanced features like distributed training, model quantization, and deployment on various platforms, ensuring scalability and efficiency.

Whether it is for academic research, prototyping models, or deploying large-scale production systems, these frameworks provide the tools and resources developers need to bring their ideas to life.

Inference for computer vision

Once you have trained a model for your use case, how will it run in production? Certainly not the way we have seen in code examples in *Chapter 5*. Those are simplistic cases meant to show how deep learning models are built. Solutioning a computer vision is more than simply calling **model.predict()**. There are several approaches to architecting solutions, each with a particular cost versus performance trade-off. These options include running on a PC and deciding between CPU and GPU utilization. Another approach involves centralizing video stream processing using a dedicated server or utilizing Cloud CPUs, Cloud GPUs, and Cloud VPUs, as well as exploring off-the-shelf solutions. Additionally, edge computing options like Intel® OpenVino and Qualcomm® SNPE present alternative

avenues for consideration. Choose the solution that best aligns with your specific requirements and objectives.

Consider the inference needs for image data processing. There are two main approaches: Offline image processing, where photos are uploaded to a cloud drive and processed later, and real-time processing, where video frames from a camera are continuously analyzed. Depending on your business requirements, it is essential to evaluate the cost implications to ensure a successful product delivery.

Deep learning techniques offer superior accuracy for models, but they can be slow, CPU-intensive, and memory-intensive. To address these challenges, various approaches have been adopted in the industry.

Local inferencing

In this scenario, the camera is directly connected to a machine where the code is executed. This setup offers great flexibility in controlling hardware and software. However, it is limited by the host machine's hardware capabilities, and practical constraints dictate that the camera cannot be placed too far from the host computer. For instance, running a camera wire for 200/300 meters to perform face recognition on passengers would be impractical. While streaming video frames over the internet can address this issue, it introduces additional complexities and does not scale efficiently when multiple camera feeds need analysis. The computational limits of the CPU are often reached relatively quickly in this setup.

Local CPUs

Running deep learning models on the same machine as the camera is suitable for scenarios where the image cannot leave the computer for legal or privacy reasons. However, scaling becomes a challenge. Adding more cameras can significantly increase costs due to CPU-heavy computations.

Local GPUs

GPUs are beneficial in reducing computational time for deep learning models and improving runtime performance. While they do not impact the algorithm's effectiveness, they enhance processing speed.

Cloud

The cloud provides a simplified alternative compared to local machines. However, it is important to remember that the cloud is not free; though it can be affordable, it might not always be cheap. To ensure optimal response times, careful design of lambda functions and other components is necessary. Programming a cloud-based CPU and GPU is similar to programming a local CPU or GPU. However, network latencies must be considered, especially for applications like driverless cars, where cameras are remote from the cloud server.

Edge computing

Special purpose devices like Intel Movidius (Neural Computing Sticks) and Qualcomm Snapdragon provide an alternative solution. Offloading inference processing to these devices frees up the CPU and offers edge computing capabilities. This approach is cost-effective, efficient, and suitable for sending only inference results to the main server instead of processing entire video frames.

While edge devices have limitations in supporting predefined neural networks like Inception or YOLO, transfer learning allows developers to train them for specific projects. Despite some compromises in creating custom networks, this field is rapidly evolving, with competition driving companies to innovate and progress in the computer vision domain.

When deciding between local inferencing and cloud-based solutions, it is crucial to weigh the advantages, limitations, and cost factors associated with each approach, ultimately aligning with your project's specific requirements and budget considerations.

OpenCV DNN module

Why did we discuss the concepts of cloud computing inference in the previous section? It was done to drive home a point.

OpenCV's **deep neural networks** (**DNN**s) module is a powerful tool for deploying pre-trained deep learning models and building custom deep learning models for various computer vision tasks. OpenCV's DNN module is a versatile tool for integrating deep learning models into computer vision

projects. Developers can effectively leverage this module to build powerful computer vision applications.

History

The OpenCV DNN module has undergone significant development and evolution over the years. The module was first introduced in OpenCV 3.1.0 in 2016. This initial version allowed users to deploy deep learning models from the Caffe framework. Caffe was one of the popular deep learning frameworks at the time and provided a wide range of pre-trained models for tasks like image classification and object detection. In subsequent releases, OpenCV's DNN module started supporting more deep learning frameworks, such as TensorFlow and Torch/PyTorch. This allowed users to utilize models trained in these frameworks and benefit from their respective architectures and performance optimizations. The integration of multiple frameworks expanded the range of tasks that the DNN module could handle. OpenCV's DNN module saw significant improvements in terms of performance and optimization. In OpenCV 3.4.0 (late 2017) and later versions, hardware acceleration through OpenCL and Intel's **Inference Engine** (**IE**) was introduced. This enabled users to leverage compatible hardware, such as GPUs and specialized accelerators, to accelerate the inference process, especially for computationally-intensive tasks. OpenCV continued to expand its collection of pre-trained models in the DNN module. More models were added for tasks like face recognition, pose estimation, semantic segmentation, and more. This allowed developers to easily access state-of-the-art models and deploy them in their computer vision applications. As deep learning continues to evolve, the OpenCV DNN module will likely integrate with new and emerging deep learning frameworks, allowing users to deploy the latest models and architectures in their computer vision projects.

The DNN module has been continuously improved and refined with each subsequent release of OpenCV. Developers have worked on enhancing the compatibility with different frameworks, improving model loading, optimizing performance, and addressing bugs reported by the community.

It is worth noting that OpenCV is an open-source project with an active community of developers, and the DNN module's history is closely tied to advancements in deep learning and the availability of new frameworks and

models. The module's continued growth and improvement reflect the ongoing efforts to provide a robust and flexible platform for deploying deep learning models in computer vision applications.

Features and limitations

Here is a detailed explanation of its features, capabilities, limitations, and aspects to consider before using this module for your project:

Capabilities

Here are some of the capabilities of the DNN module:

- **Deep learning framework integration**: OpenCV's DNN module supports several popular deep learning frameworks, including TensorFlow, Caffe, PyTorch, and **Open Neural Network Exchange** (**ONNX**). This enables seamless integration and direct use of models trained in these frameworks within OpenCV.
- **Pre-trained models**: The DNN module comes with a collection of pre-trained models, ranging from image classification to object detection, face recognition, pose estimation, semantic segmentation, and more. These models are trained on large datasets and offer state-of-the-art performance for specific tasks.
- **Custom model deployment**: Developers can deploy their own trained deep learning models by importing them from supported frameworks. This allows you to leverage the strengths of your custom architectures and trained weights directly within OpenCV.
- **Hardware acceleration**: OpenCV's DNN module supports hardware acceleration through OpenCL and Intel's IE. This enables faster inference on devices with compatible hardware.
- **Model optimization**: The module provides options for model optimization, such as model quantization, to reduce model size and improve inference speed on resource-constrained devices.
- **Multiple backends**: OpenCV DNN offers different backends for optimized inference, including CPU and GPU (via OpenCL and CUDA). This allows developers to choose the most suitable backend based on hardware availability and performance requirements.

Limitations

Here are some limitations of the DNN module:

- **Limited training support**: While OpenCV's DNN module allows deployment of custom-trained models, it does not offer training capabilities. Training deep learning models should be done using dedicated deep learning frameworks like TensorFlow, PyTorch, or Caffe.
- **Model compatibility**: Not all layers or architectures supported by deep learning frameworks are fully compatible with OpenCV's DNN module. Some complex layers or custom operations may not be supported, requiring model adjustments or custom implementations.
- **Memory consumption**: Deep learning models, especially those with many layers, can be memory-intensive. Running complex models on resource-limited devices might lead to out-of-memory issues.
- **Performance variability**: Inference performance can vary based on the backend, hardware, and model architecture. Benchmarking and testing on target devices are necessary to optimize performance.

Considerations

So, what should developers consider before choosing to work with OpenCV DNN? Here is a list of items you should consider before working with OpenCV DNN:

- **Task requirements**: Assess your project's computer vision requirements and select pre-trained models or frameworks that align with your specific task. Determine if you need to deploy pre-trained models or if custom model creation is necessary.
- **Hardware constraints**: Consider the hardware available for inference, as it will impact the choice of backend and overall performance. Hardware acceleration (for example, GPUs) may significantly speed up inference, but it might not be available on all devices.
- **Model size and complexity**: For resource-constrained devices, consider model size and complexity. Optimize the model through

techniques like quantization to achieve a balance between accuracy and memory usage.

- **Runtime environment**: If your project requires real-time or near real-time performance, choose the backend and model architecture that can meet those performance requirements.
- **Model compatibility**: Ensure that the models you plan to use are supported by OpenCV's DNN module and that any custom layers or operations are compatible or can be replaced with supported alternatives.
- **Benchmarking and optimization**: Thoroughly benchmark the chosen models and backends on target hardware to identify bottlenecks and optimize performance.

Supported layers

OpenCV's DNN module supports a wide range of deep learning layers and operations. However, not all layers or operations available in various deep learning frameworks are fully supported. The support status may change with newer versions of OpenCV as the module continues to evolve. Here is an overview of the commonly supported and unsupported deep learning layers and operations in OpenCV DNN:

- **Convolutional layers**: Standard convolutional layers, including 2D and 3D convolutions, depth-wise separable convolutions, dilated convolutions, and so on, are supported.
- **Pooling layers**: Max pooling and average pooling are typically supported.
- **Fully connected layers**: Also known as dense layers, these are widely supported.
- **Activation functions**: Common activation functions like **Rectified Linear Unit** (**ReLU**), Sigmoid, Tanh, and Leaky ReLU are supported.
- **Normalization layers**: Batch Normalization and Layer Normalization are often supported.

- **Concatenation and element-wise operations**: Operations like concatenation, element-wise addition, subtraction, multiplication, and division are usually supported.
- **Reduction layers**: Operations like **Global Average Pooling** (**GAP**) and Global Max Pooling are commonly supported.
- **Upsampling and interpolation**: Layers that perform upsampling or interpolation, such as bilinear or nearest-neighbor upsampling, are supported.
- **Detection and region proposal layers**: Operations like **Non-Maximum Suppression** (**NMS**) and region proposal layers are supported for object detection tasks.

Unsupported layers and operations

Here are some of the layers and operations unsupported by DNN module:

- **Custom layers**: Layers that are specific to a particular model or not commonly used in standard architectures may not be supported in OpenCV DNN. These custom layers may need to be replaced or re-implemented for compatibility.
- **Recurrent layers**: Recurrent layers like **Long Short-Term Memory** (**LSTM**) and **Gated Recurrent Unit** (**GRU**) were not fully supported. This means models with recurrent layers might not work as expected in OpenCV DNN.
- **Complex embeddings and transformations**: Some complex embedding layers or non-linear transformations may not be supported, depending on the specific framework and version of OpenCV.
- **Dynamic shapes**: Layers that require dynamic input shapes during inference may not be fully supported in OpenCV DNN. This limitation can affect models with variable input sizes.
- **Complex attention mechanisms**: Some advanced attention mechanisms, such as transformers, may not be fully supported in older versions of OpenCV.

It is essential to check the latest OpenCV documentation and release notes to get the most up-to-date information on supported and unsupported layers and operations in the DNN module. If your model contains unsupported layers, you might need to consider re-implementing them using supported operations or frameworks, or explore using other deep learning frameworks that provide full compatibility with your model's architecture. Additionally, newer versions of OpenCV may bring improvements and expanded support for various layers and operations, so keeping your OpenCV version up-to-date is advisable.

Important classes

OpenCV DNN module provides several important classes for building and working with deep learning models. These classes provide functionality for model loading, inference, and optimization. We will discuss the most common and most important classes here. A detailed discussion of each class is out of scope for the purpose of this book. Interested readers are advised to visit the official documentation from OpenCV at the given URL: **https://docs.opencv.org/4.x/d6/d0f/group__dnn.html**.

Here is an overview of some of the key classes in the OpenCV DNN module:

- **cv::dnn::Net**

  The **Net** class represents a deep learning model. It allows you to load pre-trained models from various deep learning frameworks (such as Caffe, TensorFlow, PyTorch, ONNX, and so on) or create custom models by stacking layers programmatically. You can use the **Net** class to load models from model files (for example, **.caffemodel**, **.pb, .t7, .onnx,** and so on) and configure the backend for inference (for example, CPU, GPU, OpenCL, and so on). The class provides methods for forwarding input data through the network to obtain output predictions.

- **cv::dnn::Layer**

  The **Layer** class is the base class for all layers in the OpenCV DNN module. It provides methods for setting layer parameters,

connecting layers together, and performing forward propagation during inference.

- **cv::dnn::LayerParams**

  LayerParams is a structure that holds parameters for a specific layer when constructing a custom network using the **Net** class. It includes properties like the layer type, input and output blob names, activation function, kernel size, stride, and so on.

- **cv::dnn::Blob**

  The **Blob** class represents a tensor or multi-dimensional array used as input or output in the deep learning model. It allows you to access and manipulate the data stored in the blob during inference.

- **cv::dnn::BackendNode**

  The **BackendNode** class provides backend-specific information about nodes in the network, such as memory allocation, storage formats, and optimized computation.

- **cv::dnn::NMSBoxes**

  NMSBoxes is a utility class for performing **Non-Maximum Suppression** (**NMS**) on bounding box detections to filter out overlapping detections and retain only the most relevant ones.

- **cv::dnn::DetectionModel**

  DetectionModel is a specialized class for object detection tasks. It provides methods to perform detection on input images and retrieve bounding boxes, confidence scores, and class labels for detected objects.

- **cv::dnn::TextDetectionModel**

  TextDetectionModel is a specific class for text detection tasks, enabling the detection of text regions in images.

These are some of the essential classes in the OpenCV DNN module. Using these classes effectively, you can load pre-trained models, create custom architectures, perform inference on images or videos, and work with different types of deep learning models for various computer vision tasks,

such as image classification, object detection, semantic segmentation, and more.

Conclusion

OpenCV DNN module offers a powerful and flexible platform for integrating deep learning models into computer vision projects. With its support for various frameworks, pre-trained models, and hardware acceleration, developers can tackle a wide range of tasks with ease. While being aware of the module's limitations, the understanding of supported layers and essential classes equips us to make informed decisions and optimize performance. Armed with this knowledge, we are well-positioned to harness the potential of deep learning and computer vision, paving the way for innovative and impactful applications in the field.

Exercises

1. Please visit the official OpenCV documentation and gain an understanding of the classes and terms described in this chapter.

Chapter 7
Modern Solutions for Image Classification

Introduction

This chapter delves into the realm of computer vision and deep learning, focusing on the powerful technique of image classification. Image classification involves the categorization of images into distinct classes using advanced machine learning models. Throughout this chapter, we will explore renowned image classification architectures, including ResNet, InceptionV3, MobileNetV2, and more. By understanding these architectures and their implementation, we will equip ourselves with the tools to analyze, preprocess, and classify images, showcasing the intersection of cutting-edge technology and visual understanding.

Structure

The chapter discusses the following topics:

- CNNS for classification
- Inception V3
- ResNet
- MobileNetV2

- Comparison of models
- Parameters for blobFromImage()

Objectives

The objective of this chapter is to evaluate the deep learning solutions for image classification. This chapter aims to explain image classification using the Keras framework and OpenCV's DNN module. We will also learn about the architecture and salient points of the major classification models. This chapter does not attempt to explain the training or transfer learning processes of deep learning.

CNNS for classification

As discussed in *Chapter 5, Deep Learning and CNNs* are a class of deep learning models widely used for image classification tasks. They are designed to learn spatial hierarchies of features automatically and adaptively from input images. The general CNN architecture for image classification consists of the following key components:

- **Input layer**: The first layer of the CNN takes the raw input image as a multi-dimensional array of pixel values. Typically, images are represented as 3D arrays with dimensions (height, width, channels). The number of channels corresponds to the color channels in the image (for example, RGB images have 3 channels, grayscale images have 1 channel).
- **Convolutional layers**: These layers are the core building blocks of a CNN. Each convolutional layer applies a set of learnable filters (also known as kernels) to the input image. These filters slide over the image in small strides and perform element-wise multiplications and summations to produce feature maps. Each feature map represents a particular feature learned by the filter, such as edges, textures, or higher-level patterns.
- **Activation function**: After the convolution operation, an activation function (for example, ReLU - Rectified Linear Unit) is applied element-wise to introduce non-linearity into the network. This allows

CNN to learn complex relationships and improve its ability to represent more intricate features in the data.

- **Pooling layers**: These layers help reduce the spatial dimensions of the feature maps while retaining important information. Pooling is typically done using operations like MaxPooling, where the maximum value within a local window is taken, or AveragePooling, where the average value is computed. Pooling helps reduce the computational complexity of the network and makes it more robust to spatial translations.
- **Fully connected layers**: After several convolutional and pooling layers, the final output is usually flattened into a 1D vector. This vector is then passed through one or more fully connected layers (also known as dense layers) to perform high-level reasoning and decision-making based on the extracted features. These layers contain a large number of neurons, and each neuron is connected to every output of the previous layer.
- **Output layer**: The last fully connected layer typically consists of neurons equal to the number of classes in the classification task. It provides the final classification scores or probabilities using an appropriate activation function (for example, softmax for multi-class classification).
- **Loss function**: During training, CNN computes a loss or cost function based on the difference between its predictions and the true labels of the training data. The goal of training is to minimize this loss function, typically achieved using optimization algorithms like **stochastic gradient descent** (**SGD**) or its variants.

The overall process of training a CNN involves passing the input image through the layers, computing the loss, and then using backpropagation to update the weights of the network to improve its performance. Once trained, the CNN can be used to make predictions on new, unseen images for image classification tasks.

We shall test the different deep learning models using the Keras framework and OpenCV's DNN module using the image shown in *Figure 7.1*:

***Figure 7.1**: Image used for testing the classification models*

Inception-v3

Inception-v3 is a deep learning architecture developed by Google's research team as part of the Inception series of models. It is designed for image classification and other computer vision tasks. Inception-v3 is an evolution of the earlier Inception-v1 and Inception-v2 models. It introduces several improvements to achieve higher accuracy and efficiency. The key innovation of Inception-v3 is the use of Inception modules, which are designed to capture multi-scale patterns and hierarchies of features. These modules use a combination of different-sized convolutional filters in parallel to capture information at different scales. The main idea is to create a network that can effectively learn both local and global features, allowing it to recognize patterns at various levels of abstraction.

Here is an overview of the Inception-v3 architecture:

- **Input layer**: The input to the network is an image represented as a 3D array with dimensions (height, width, channels). Images are typically resized to a fixed size before feeding them into the network.

- **Initial convolution and pooling layers**: The input image passes through an initial set of convolutional layers, which perform feature extraction. This is followed by max-pooling layers to reduce spatial dimensions and control the computation.
- **Inception modules**: These are the core building blocks of Inception-v3. Each Inception module consists of multiple parallel convolutional layers with different filter sizes and a pooling layer. Specifically, each Inception module includes:
    - **1x1 convolution**: This captures linear combinations of features from the previous layer.
    - **3x3 convolution**: This captures more spatial information and detects mid-level patterns.
    - **5x5 convolution**: This captures larger spatial patterns and detects higher-level features.
    - **3x3 max pooling**: This captures dominant features in different regions.

  The outputs of all these operations are then concatenated along the depth dimension to form the output of the Inception module. The use of different filter sizes in parallel allows the network to learn features at multiple scales and helps prevent overfitting.
- **Auxiliary classifiers**: Inception-v3 includes auxiliary classifiers at intermediate stages of the network. These classifiers are added to provide additional gradient flow during training and act as regularization. They encourage the network to learn more useful and discriminative features.
- **Global average pooling**: Inception-v3 uses global average pooling to reduce spatial dimensions to 1x1 before the final classification.
- **Output layer**: The final fully connected layer or softmax layer is used to produce the final classification scores or probabilities.

See *Figure 7.2* for the diagrammatic representation of inception architecture. This image has been taken from the inception-v3 paper at **https://paperswithcode.com/method/inception-v3**. **Please refer to the following figure:**

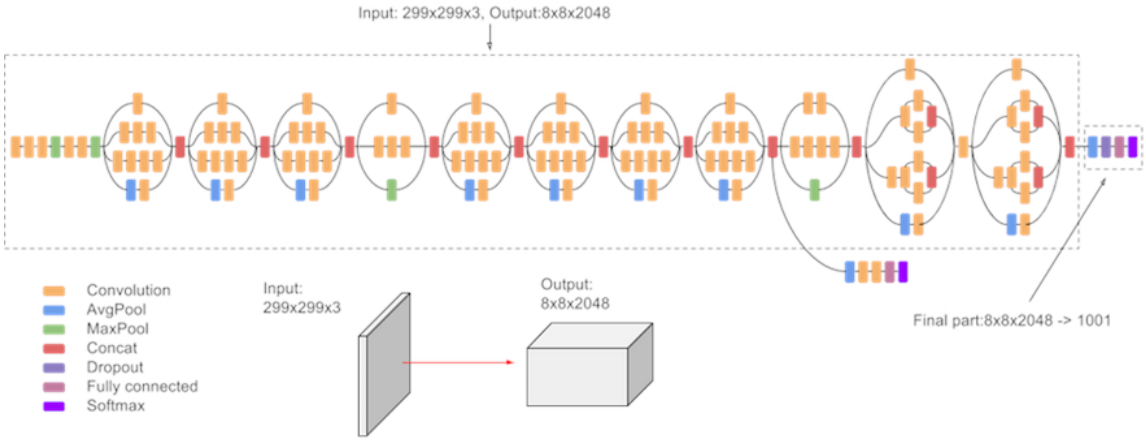


***Figure 7.2**: Inception-v3 architecture*

Inception-v3 achieved significant improvements in accuracy over its predecessors and has been widely used for various computer vision tasks, such as image classification, object detection, and image segmentation. It has been a popular choice for transfer learning and is available in many deep learning frameworks, making it accessible to developers for various applications.

Note: Keras provides functionality to print a detailed view of the model's nodes and their connectivity. It can be used to visualize and debug the model architecture. The plot_model() function in tensorflow.keras.utils does this. However, as the model becomes more complex, it becomes difficult to fit the output into a book publishing scenario. Figure 7.3 intentionally presents a distorted image to accommodate publishing challenges. Throughout the remainder of the book, we will refrain from utilizing this particular output. Nevertheless, the readers can refer to the code bundle for the image in its true resolution. The accompanying code for achieving this visualization is provided below.

```
1. import tensorflow as tf
2. from tensorflow.keras.applications.inception_v3 import InceptionV3
3. from tensorflow.keras.utils import plot_model
```

4.
5.
6. model = InceptionV3(weights='imagenet')
7. plot_model(model, to_file="inveptionv3.png", show_shapes=True)

***Figure 7.3**: Tensorflow output of plotmodel() function for Inception-v3*

Let us now see the code for using Inception-v3 model. We shall use the model in two different ways. One is by using the Keras framework and the second is by using the DNN module of OpenCV.

Keras

Below is the code implementation to use Inception-v3 with Keras library:

```
1. import os
2. import numpy as np
3. from keras.applications import InceptionV3
4. from keras.applications.inception_v3 import preprocess_input, decod
   e_predictions
5. from keras.preprocessing import image as image_utils
6. from keras import utils
7.
8.
9. def preprocess_image(im):
10.     # Load the image and resize it to the target size of (299, 299).
11.     img = utils.load_img(im, target_size=(299, 299))
12.
13.     # Convert the loaded image to a NumPy array.
14.     img = utils.img_to_array(img)
15.
```

```
    # Add an additional dimension to the array to represent batch size (1).
    img = np.expand_dims(img, axis=0)

    # Preprocess the image data for the specific deep learning model.
    img = preprocess_input(img)

    # Return the preprocessed image.
    return img


def classify_image_using_tensorflow(imagepath):
    # Create an empty list to store the predicted labels.
    predicted_labels = []

    # Load the pre-trained InceptionV3 model with weights from the ImageNet dataset.
    model = InceptionV3(include_top=True, weights='imagenet')

    # Preprocess the input image using the preprocess_image function (not shown here).
```

```
    preprocessed_image = preprocess_image(imagepath)

    # Make a prediction using the pre-trained model on the preprocessed image.
    pred = model.predict(preprocessed_image)

    # Decode and process the prediction to get the top predicted labels.
    for prediction in decode_predictions(pred, top=7)[0]:
        predicted_labels.append(prediction)

    # Return the list of predicted labels.
    return predicted_labels


if __name__ == "__main__":
    # This block of code will be executed only if this script is run directly as the main program.

    # Call the classify_image_using_tensorflow function with the specified image file path.
    labels = classify_image_using_tensorflow("../input_images/aeroplane.jpg")
```

```
53.
54.     # Iterate over the list of predicted labels and print each
        label.
55.     for l in labels:
56.         print(l)
57.
```

The **preprocess_image** function takes an image file path, loads and resizes the image, converts it to a NumPy array with an added batch dimension. Additionally, it preprocesses the image according to the requirements of a specific deep learning model and returns the preprocessed image ready for use with the model. **classify_image_using_tensorflow** function takes an image file path, uses a pre-trained **InceptionV3** model to classify the image, and returns a list of the top predicted labels along with their associated probabilities. The function utilizes TensorFlow's capabilities for model loading, prediction, and label decoding to achieve this classification task. When this script is run directly, it loads the image of an airplane, uses the **classify_image_using_tensorflow** function to predict its labels, and then prints out the top predicted labels along with their associated probabilities. This can be useful for quickly testing and verifying the classification performance of the model on a specific image.

Executing this code produces the given output:

```
1. D:\bpb\995\7>python classify_inception_keras.py
2. 2023-08-
   11 17:44:11.904259: I tensorflow/core/platform/cpu_feature_guard.cc
   :193] This TensorFlow binary is optimized with oneAPI Deep Neural
   Network Library (oneDNN) to use the following CPU instructions in
   performance-critical operations:  AVX AVX2
3. To enable them in other operations, rebuild TensorFlow
   with the appropriate compiler flags.
4. 1/1 [==============================] - 1s 1s/step
```

5. ('n02690373', 'airliner', 0.92636836)
6. ('n04592741', 'wing', 0.018255161)
7. ('n04552348', 'warplane', 0.0010660181)
8. ('n02423022', 'gazelle', 0.00072263)
9. ('n04266014', 'space_shuttle', 0.0006761261)
10. ('n02483362', 'gibbon', 0.0006553345)
11. ('n02641379', 'gar', 0.00035934924)

As you can see, an airliner is the class with the highest confidence value of 92.63%. In fact, the class with the second highest confidence value is the wing with a mere 1.8%. So, we can very confidently conclude that the given image is that of an airplane.

OpenCV DNN module

Now let us run the same model in OpenCV using DNN module. This script demonstrates using OpenCV's DNN module to load a pre-trained Inception-v3 model, preprocess an image, classify it, and print the top predicted labels with their confidences. The script is structured to be run as a standalone program:

1. import os
2. import numpy as np
3. import cv2
4. *# File paths to the InceptionV3 model weights and class names*
5. imagenet_classes_filepath = "../weights/7/ILSVRC2012.txt"
6. inceptionv3_weights_filepath = "../weights/7/inceptionv3/inceptionv3.pb"
7.

```
# Shape of the input image expected by the InceptionV3 model
inceptionv3_shape = (299, 299)

# Function to decode and format the predicted labels
def decode_predictions(predictions, class_names, top=5):
    results = []
    top_indices = predictions[0].argsort()[-top:][::-1]
    for i in top_indices:
        result = class_names[i] + ": " + str(predictions[0][i])
        results.append(result)
    return results

# Function to classify an image using OpenCV's dnn module and InceptionV3
def classify_image_using_opencvdnn(imagepath):
    # Read the class names from the provided file
    imagenet_class_names = None
    with open(imagenet_classes_filepath, 'rt') as f:
        imagenet_class_names = f.read().rstrip('\n').split('\n')

    # Load the InceptionV3 model from disk
```

```
    model = cv2.dnn.readNet(inceptionv3_weights_filepath)

    # Read and preprocess the input image
    im = cv2.imread(imagepath)
    resized_image = cv2.resize(im, inceptionv3_shape)
    image_blob = cv2.dnn.blobFromImage(resized_image, 1/127.5, inceptionv3_shape, [127.5, 127.5, 127.5])

    # Set the input blob for the model and perform forward pass
    model.setInput(image_blob)
    predictions = model.forward()

    # Return the decoded predictions using the decode_predictions function
    return decode_predictions(predictions, imagenet_class_names, 7)

# Main script execution
if __name__ == "__main__":
    # Call the classify_image_using_opencvdnn function with the specified image file path
    labels_and_confidences = classify_image_using_opencvdnn("../input_images/aeroplane.jpg")

```

```
    # Iterate over the list of labels and confidences and print each label with confidence
    for label in labels_and_confidences:
        print(label)
```

The code defines file paths for the Inception-v3 model's class names and weights, as well as the expected shape of the input image. The **decode_predictions** function takes the model's predictions, class names, and a top parameter, and returns a formatted list of the top predicted labels along with their associated confidences. The **classify_image_using_opencvdnn** function uses OpenCV's DNN module to classify an image. It reads the class names from a file, loads the Inception-v3 model, preprocesses the input image, performs a forward pass through the model, and returns the decoded predictions.

Executing this script gives the below output:

```
D:\bpb\995\7>python classify_inception_dnn.py
airliner: 0.9326485
wing: 0.015929082
warplane, military plane: 0.00069996953
gazelle: 0.00067140284
space shuttle: 0.0005667596
gibbon, Hylobates lar: 0.00053297967
screen, CRT screen: 0.0003772492
```

As you can see the DNN module also gave **airliner** as its most confident prediction. The most often asked question at this point is – how to interpret the inputs given to **cv2.dnn.blobFromImage()** function. Readers are requested to refer to *Chapter 6* for the details of this function. An explanation of the values is provided at the end of this chapter after covering ResNet and MobileNetV2.

ResNet

Residual Network (**ResNet**) is a deep learning architecture introduced by *Kaiming He et al*. in their 2015 paper, *Deep Residual Learning for Image Recognition*. It addresses the vanishing gradient problem that can occur in very deep neural networks by introducing skip connections or shortcuts.

The key idea behind ResNet is the introduction of *residual blocks*, which allow the network to learn residual functions instead of attempting to learn the actual mapping. This is done by reformulating the learning process to learn the difference (residual) between the input and output rather than learning the output directly. These residual blocks make it easier to train deep neural networks and have been instrumental in enabling the construction of deeper CNNs.

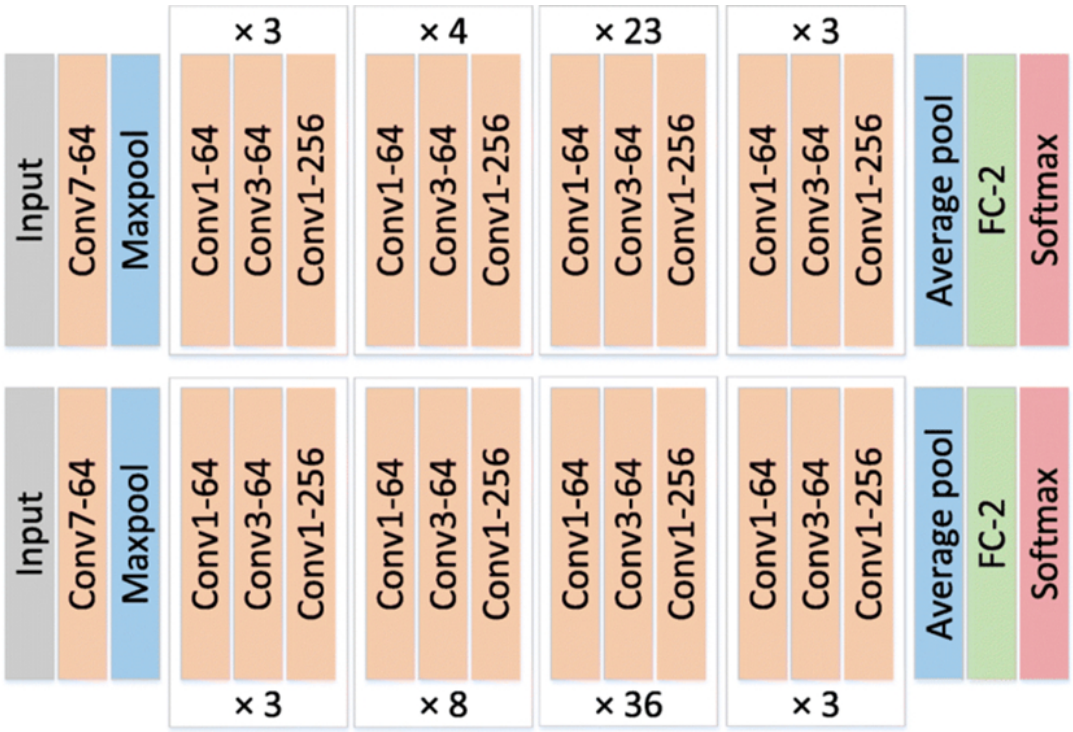
The general architecture of a ResNet can be summarized as follows:

- **Input layer**: The input to the network is an image or a feature map from a previous layer.
- **Convolutional and pooling layers**: The initial layers may consist of traditional convolutional and pooling layers that help in feature extraction and dimensionality reduction.
- **Residual blocks**: The main building blocks of ResNet are the residual blocks. Each residual block contains multiple convolutional layers and shortcut connections. A typical residual block has the following components:
    - **Convolutional layer**: Applies a set of filters to the input feature map to extract new features.
    - **Batch normalization**: Normalizes the output of the convolutional layer to accelerate training and improve generalization.
    - **Activation function**: Typically, a ReLU activation function is applied to introduce non-linearity.
    - **Convolutional layer**: Another set of filters is applied to the previous output.
    - **Batch normalization**: Normalizes the output again.

    - **Shortcut connection**: A skip connection, which adds the original input of the residual block to the output of the second convolutional layer. If the shapes of the inputs do not match, a *1x1* convolution may be used to adjust the dimensions accordingly.
- **Global average pooling**: Instead of fully connected layers, ResNet employs global average pooling. This reduces the spatial dimensions of the feature maps to *1x1* and helps reduce the number of parameters in the network.
- **Output layer**: A final fully connected layer or softmax layer is used to produce the final classification scores or probabilities based on the learned features.

The main advantage of ResNet is that it allows the creation of much deeper networks (for example, ResNet-50, ResNet-101, ResNet-152) without sacrificing performance. By introducing shortcut connections, gradients can flow more easily during training, enabling the successful training of very deep networks. ResNet has become a foundational architecture for various computer vision tasks and has been widely adopted and adapted for many state-of-the-art applications.

For the discussion in this chapter, we shall use ResNet152. This is one of the most advanced models in the ResNet family. Refer to *Figure 7.4* for a diagrammatic representation of ResNet models:



Network structures of ResNet101 (top) and ResNet152 (bottom)

***Figure 7.4**: Diagrammatic representation of ResNet architecture*[1]

Keras implementation

The following code uses Keras library for image classification using MobileNet architecture:

```
1. import numpy as np
2. from keras import utils
3. from keras.applications.resnet import ResNet152, preprocess_input
4. from keras.applications.imagenet_utils import decode_predictions
5.
6. def preprocess_image(im):
7.     # Load the image and resize it to the target size of (224, 224).
```

```
    img = utils.load_img(im, target_size=(224, 224))

    # Convert the loaded image to a NumPy array.
    img = utils.img_to_array(img)

    # Add an additional dimension to the array to represent batch size (1).
    img = np.expand_dims(img, axis=0)

    # Preprocess the image data for the specific deep learning model.
    img = preprocess_input(img)

    # Return the preprocessed image.
    return img


def classify_image_using_tensorflow(imagepath):
    # Create an empty list to store the predicted labels.
    predicted_labels = []
```

```
    # Load the pre-trained ResNet-152 model with weights from the ImageNet dataset.
    model = ResNet152(include_top=True, weights='imagenet')

    # Preprocess the input image using the preprocess_image function (not shown here).
    preprocessed_image = preprocess_image(imagepath)

    # Make a prediction using the pre-trained model on the preprocessed image.
    pred = model.predict(preprocessed_image)

    # Decode and process the prediction to get the top predicted labels.
    for prediction in decode_predictions(pred)[0]:
        predicted_labels.append(prediction)

    # Return the list of predicted labels.
    return predicted_labels

if __name__ == "__main__":
    # This block of code will be executed only if this script is run directly as the main program.
```

```
    # Call the classify_image_using_tensorflow function with the specified image file path.
    labels = classify_image_using_tensorflow("../input_images/aeroplane.jpg")

    # Iterate over the list of predicted labels and print each label.
    for l in labels:
        print(l)
```

The **classify_image_using_tensorflow** function takes an image file path, uses a pre-trained ResNet-152 model to classify the image, and returns a list of the top predicted labels and their associated probabilities. The function utilizes TensorFlow's capabilities for model loading, prediction, and label decoding to achieve this classification task. When this script is run directly, it uses the **classify_image_using_tensorflow** function to classify an input image of an airplane, then iterates over the predicted labels and prints them out. This script provides a convenient way to quickly see the top predicted labels for a specific image using the classification function. When executed, this script provides the following output:

```
D:\bpb\995\7>python classify_resnet_keras.py
2023-08-11 18:47:43.499226: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
1/1 [==============================] - 2s 2s/step
```

5. ('n02690373', 'airliner', 0.7484503)
6. ('n04592741', 'wing', 0.14459641)
7. ('n04552348', 'warplane', 0.09993937)
8. ('n04266014', 'space_shuttle', 0.0051170834)
9. ('n02687172', 'aircraft_carrier', 0.0017884057)

OpenCV DNN implementation

The below script demonstrates how to use OpenCV's DNN module to load a pre-trained ResNet-152 model, preprocess an image, classify it, and print the top predicted labels with their confidences. The script is structured to be run as a standalone program:

1. *# Import necessary libraries*
2. import os
3. import numpy as np
4. import cv2
5.
6. *# File paths to the ResNet-152 model weights and class names*
7. imagenet_classes_filepath = "../weights/7/ILSVRC2012.txt"
8. resnet152_weights_filepath = "../weights/7/resnet/resnet152.pb"
9.
10. *# Shape of the input image expected by the ResNet-152 model*
11. resnet152_shape = (224, 224)
12.

```
# Function to decode and format the predicted labels
def decode_predictions(predictions, class_names, top=5):
    results = []
    top_indices = predictions[0].argsort()[-top:][::-1]
    for i in top_indices:
        result = class_names[i] + ": " + str(predictions[0][i])
        results.append(result)
    return results

# Function to classify an image using OpenCV's dnn module and Res Net-152
def classify_image_using_opencvdnn(imagepath):
    # Read the class names from the provided file
    imagenet_class_names = None
    with open(imagenet_classes_filepath, 'rt') as f:
        imagenet_class_names = f.read().rstrip('\n').split('\n')

    # Load the ResNet-152 model from disk
    model = cv2.dnn.readNet(resnet152_weights_filepath)

    # Read and preprocess the input image
```

```
    im = cv2.imread(imagepath)
    resized_image = cv2.resize(im, resnet152_shape)
    image_blob = cv2.dnn.blobFromImage(resized_image, 1.0, resnet152_shape, [103.939, 116.779, 123.68])

    # Set the input blob for the model and perform forward pass
    model.setInput(image_blob)
    predictions = model.forward()

    # Return the decoded predictions using the decode_predictions function
    return decode_predictions(predictions, imagenet_class_names, 7)

# Main script execution
if __name__ == "__main__":
    # Call the classify_image_using_opencvdnn function with the specified image file path
    labels_and_confidences = classify_image_using_opencvdnn("../input_images/aeroplane.jpg")

    # Iterate over the list of labels and
confidences and print each label with confidence
    for label in labels_and_confidences:
```

```
print(label)
```

Executing this script provides the following output:

```
D:\bpb\995\7>python classify_resnet_dnn.py
airliner: 0.78563213
wing: 0.15230058
warplane, military plane: 0.05804099
space shuttle: 0.0029098869
aircraft carrier, carrier, flattop, attack aircraft
carrier: 0.0010048238
missile: 2.588793e-05
projectile, missile: 2.4692781e-05
```

MobileNetV2

MobileNetV2 is a deep learning architecture designed for efficient on-device inference, particularly for mobile and embedded devices. It is an evolution of the original MobileNetV1, developed by Google's research team. MobileNetV2 achieves a good balance between model size and accuracy, making it well-suited for applications where computational resources and memory are limited. The key features of MobileNetV2 include the use of depth-wise separable convolutions and linear bottlenecks, which significantly reduce the number of parameters and operations required while maintaining good performance.

Here is an overview of the MobileNetV2 architecture:

- **Input layer:** The input to the network is an image represented as a 3D array with dimensions (height, width, channels). Images are typically resized to a fixed size before being fed into the network.
- **Initial convolution and bottleneck layers**: The input image passes through an initial convolutional layer with a larger number of filters,

followed by batch normalization and ReLU activation. This is referred to as the **stem** of the network. It is worth noting that MobileNetV2 uses *3x3* depth-wise separable convolutions for most of its convolutions, which involves applying a depth-wise convolution followed by a point-wise convolution (*1x1* convolution).

- **Inverted residual blocks**: The core building blocks of MobileNetV2 are the *inverted residual blocks*, which consist of three main operations:
    - **Depth-wise convolution**: A depth-wise convolution with a small filter size (for example, *3x3*) is applied to the input feature map, reducing the number of parameters.
    - **Point-wise Convolution** (**Expansion**): A *1x1* point-wise convolution is applied to expand the number of channels, allowing the network to learn more complex representations.
    - **Point-wise convolution** (**Projection**): Another *1x1* point-wise convolution is applied to project the expanded feature map back to a lower-dimensional space.

  The use of point-wise convolutions after depth-wise convolutions helps maintain representational power while keeping the computational cost low.
- **Bottleneck width multiplier**: MobileNetV2 introduces a hyperparameter called the *bottleneck width multiplier*, denoted as α. This multiplier scales the number of channels in each layer, effectively reducing or increasing the network's width. A smaller α reduces the number of parameters and operations, making the model more lightweight.
- **Final layers**: MobileNetV2 often includes a few additional convolutional layers and global average pooling to reduce spatial dimensions before the final fully connected layer or softmax layer, which produces the classification scores or probabilities.

MobileNetV2 has been widely used in mobile and embedded applications, such as real-time object detection and image classification on mobile devices. Its efficient design allows it to run with low memory and computational requirements while achieving respectable accuracy. It serves

as an excellent choice when resource constraints are a significant consideration for the deployment of deep learning models. Refer to *Figure 7.5* for the diagrammatic representation of MobileNet V2's architecture:

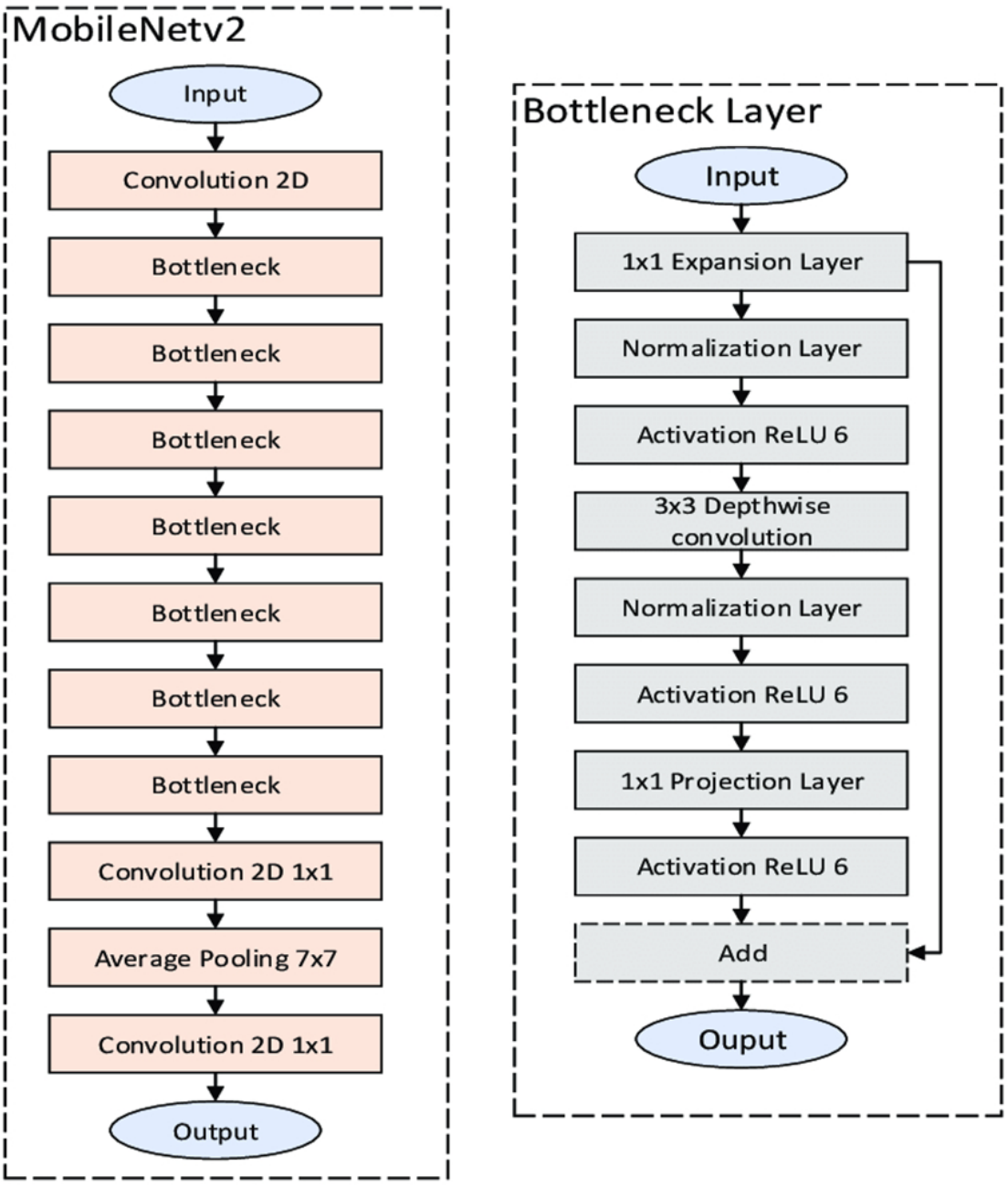


***Figure 7.5**: MobileNetV2 architecture*[2]

Keras implementation

The below script demonstrates how to use Keras to load a pre-trained MobileNetV2 model, preprocess an image, classify it, and print the top predicted labels. The script is structured to be run as a standalone program:

```
# Import necessary libraries
import os
import numpy as np
from keras.applications.mobilenet_v2 import MobileNetV2, preprocess_input, decode_predictions
from keras.preprocessing import image as image_utils
from keras import utils

# Function to preprocess an image for MobileNetV2
def preprocess_image(im):
    img = utils.load_img(im, target_size=(224, 224))
    img = utils.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = preprocess_input(img)
    return img

# Function to classify an image using MobileNetV2
def classify_image_using_tensorflow(imagepath):
    predicted_labels = []
```

```
19.
20.     # Load the pre-trained MobileNetV2
    model with weights from the ImageNet dataset
21.     model = MobileNetV2(include_top=True, weights='imagenet')
22.
23.     # Preprocess the input image using the preprocess_image function
24.     preprocessed_image = preprocess_image(imagepath)
25.
26.                 # Make a prediction using the pre-
    trained model on the preprocessed image
27.     pred = model.predict(preprocessed_image)
28.
29.     # Decode and process the prediction to get the top predicted
    labels
30.     for prediction in decode_predictions(pred, top=7)[0]:
31.         predicted_labels.append(prediction)
32.
33.     # Return the list of predicted labels
34.     return predicted_labels
35.
36. # Main script execution
37. if __name__ == "__main__":
```

```
38.     # Call the classify_image_using_tensorflow function with the specif
        ied image file path
39.     labels = classify_image_using_tensorflow("../input_images/aeropla
        ne.jpg")
40.
41.     # Iterate over the list of predicted labels and print each label
42.     for l in labels:
43.         print(l)
```

The **preprocess_image** function takes an image file path, loads and preprocesses the image using Keras utilities, and returns the preprocessed image in a format suitable for MobileNetV2. The **classify_image_using_tensorflow** function uses Keras to classify an image. It loads a pre-trained MobileNetV2 model, preprocesses the input image using the **preprocess_image** function, makes a prediction using the model, and decodes the predictions to get the top predicted labels. It calls the **classify_image_using_tensorflow** function to classify an input image of an airplane, then iterates over the predicted labels and prints them.

Executing this scripts generates the following output:

```
1. D:\bpb\995\7>python classify_mobilenet_keras.py
2. 2023-08-
   12 09:55:53.770728: I tensorflow/core/platform/cpu_feature_guard.cc
   :193] This TensorFlow binary is optimized with oneAPI Deep Neural
   Network Library (oneDNN) to use the following CPU instructions in
   performance-critical operations:  AVX AVX2
3. To enable them in other operations, rebuild TensorFlow
   with the appropriate compiler flags.
4. 1/1 [==============================] - 1s 961ms/step
5. ('n02690373', 'airliner', 0.7056889)
```

6. ('n04552348', 'warplane', 0.16613244)
7. ('n04592741', 'wing', 0.034881666)
8. ('n02687172', 'aircraft_carrier', 0.016458299)
9. ('n04266014', 'space_shuttle', 0.0023486638)
10. ('n03126707', 'crane', 0.0014191336)
11. ('n03355925', 'flagpole', 0.0012728848)

OpenCV DNN implementation

Below code uses OpenCV DNN for image classification using MobileNet architecture:

```
# Import necessary libraries
import os
import numpy as np
import cv2

# File paths to the MobileNetV2 model weights and class names
imagenet_classes_filepath = "../weights/7/ILSVRC2012.txt"
mobilenetv2_weights_filepath = "../weights/7/mobilenet/mobilenetv2.pb"

# Shape of the input image expected by the MobileNetV2 model
mobilenetv2_shape = (224, 224)

```

```
# Function to decode and format the predicted labels
def decode_predictions(predictions, class_names, top=5):
    results = []
    top_indices = predictions[0].argsort()[-top:][::-1]
    for i in top_indices:
        result = class_names[i] + ": " + str(predictions[0][i])
        results.append(result)
    return results

# Function to classify an image using OpenCV's dnn module and MobileNetV2
def classify_image_using_opencvdnn(imagepath):
    # Read the class names from the provided file
    imagenet_class_names = None
    with open(imagenet_classes_filepath, 'rt') as f:
        imagenet_class_names = f.read().rstrip('\n').split('\n')

    # Load the MobileNetV2 model from disk
    model = cv2.dnn.readNet(mobilenetv2_weights_filepath)

    # Read and preprocess the input image
```

```
    im = cv2.imread(imagepath)
    resized_image = cv2.resize(im, mobilenetv2_shape)
    image_blob = cv2.dnn.blobFromImage(resized_image, 1/127.5, mobilenetv2_shape, [127.5, 127.5, 127.5])

    # Set the input blob for the model and perform forward pass
    model.setInput(image_blob)
    predictions = model.forward()

    # Return the decoded predictions using the decode_predictions function
    return decode_predictions(predictions, imagenet_class_names, 7)

# Main script execution
if __name__ == "__main__":
    # Call the classify_image_using_opencvdnn function with the specified image file path
    labels_and_confidences = classify_image_using_opencvdnn("../input_images/aeroplane.jpg")

    # Iterate over the list of labels and confidences and print each label with confidence
    for label in labels_and_confidences:
```

```
print(label)
```

The **decode_predictions** function is similar to the previous examples and decodes the model's predictions into human-readable labels. The **classify_image_using_opencvdnn** function uses OpenCV's DNN module to classify an image. It reads the class names from a file, loads the MobileNetV2 model, preprocesses the input image, performs a forward pass through the model, and returns the decoded predictions. It calls the **classify_image_using_opencvdnn** function to classify an input image of an airplane, then iterates over the predicted labels and prints them. This code generates the following output:

```
D:\bpb\995\7>python classify_mobilenet_dnn.py
airliner: 0.8213743
warplane, military plane: 0.048832133
wing: 0.03108493
aircraft carrier, carrier, flattop, attack aircraft carrier: 0.0054382924
space shuttle: 0.0034200125
projectile, missile: 0.003020886
reel: 0.0024129024
```

Comparison of models

Let us compare and contrast the three models: ResNet, Inception-v3, and MobileNetV2, based on several key aspects. Refer to *Table 7.1*:

Basis	ResNet	Inception-v3	MobileNetV2
Processing image size	224 x 224 pixels	299 x 299 pixels	224 x 224 pixels
Architecture and building bl	ResNet uses residual blocks	Inception-v3 uses inception	MobileNetV2 employs inver

Basis	ResNet	Inception-v3	MobileNetV2
ocks	with shortcut connections to address the vanishing gradient problem and enable training of deep networks.	modules with multiple parallel convolutional layers of different sizes to capture multi-scale patterns and hierarchies of features.	ted residual blocks with depth-wise separable convolutions and linear bottlenecks to reduce the number of parameters and operations for efficient on-device inference.
Model size and complexity	ResNet is deeper compared to the other two models and has a higher number of parameters, making it relatively more complex.	Inception-v3 is deeper than MobileNetV2 but generally lighter than ResNet in terms of model size and complexity.	MobileNetV2 is designed to be lightweight and has the smallest number of parameters among the three models, making it highly efficient for on-device inference.
Accuracy	ResNet is known for its excellent accuracy and has achieved state-of-the-art results in many image classification benchmarks.	Inception-v3 also achieves high accuracy and performs well in image classification tasks, though it may be slightly outperformed by more	MobileNetV2 provides good accuracy considering its small size, but it might not match the top-tier performance of larger and more comple

Basis	ResNet	Inception-v3	MobileNetV2
		recent models.	x models like ResNet and Inception-v3.
Computational efficiency	While ResNet achieves high accuracy, it requires more computational resources and memory due to its depth and parameter count.	Inception-v3 is more computationally efficient compared to ResNet, but it still demands more resources than MobileNetV2.	MobileNetV2 is specifically designed for efficiency, making it highly suitable for on-device inference, particularly on mobile and embedded devices.
Use cases	ResNet is well-suited for applications where high accuracy is the primary concern and computational resources are not a limiting factor.	Inception-v3 is a good choice when accuracy is crucial, and there is a moderate availability of computational resources.	MobileNetV2 is ideal for resource-constrained environments, such as mobile phones and embedded devices, where efficient inference is essential.

***Table 7.1**: Comparison of models for image classification*

ResNet, Inception-v3, and MobileNetV2 are all powerful deep learning models, each with its unique strengths and use cases. ResNet excels in accuracy but may be computationally expensive for resource-limited environments. Inception-v3 achieves high accuracy and provides a good balance between accuracy and computational efficiency. MobileNetV2 prioritizes efficiency and is specifically designed for on-device inference, making it an excellent choice for mobile and embedded applications where

resource constraints are a significant consideration. The choice of model depends on the specific requirements and constraints of the application at hand.

Parameters for blobFromImage()

Discerning readers might have noticed that the parameters used in the call to **cv2.dnn.blobFromImage()** are different for the three models. It naturally follows that there should be some significance to them. We shall discuss the parameters here briefly.

```
cv2.dnn.blobFromImage(image, scalefactor=1.0, size, mean, swapRB=True)
```

- **image**: This parameter represents the input image that we intend to preprocess before feeding it into our deep neural network for classification.
- **scalefactor**: After applying mean subtraction, we have the option to scale our images by a certain factor. The default value is 1.0, indicating no scaling. It is note worthy that the scale factor should be $1 / \sigma$.
- **size**: Here, we provide the spatial dimensions that the convolutional neural network expects. Most modern state-of-the-art neural networks commonly use sizes like *224x224*, *227x227*, or *299x299*.
- **mean**: This parameter corresponds to the values we subtract during mean subtraction. It can be a 3-tuple representing the RGB means, or a single value that is subtracted from every channel of the image. When performing mean subtraction, it is crucial to provide the 3-tuple in the order (R, G, B), especially if the **swapRB** parameter is set to True (which is its default behavior).
- **swapRB**: OpenCV assumes that images are in the BGR channel order, while the mean parameter assumes RGB order. To reconcile this difference, we can swap the R and B channels in the image by setting this parameter to true. By default, OpenCV handles this channel swapping for us.

Conclusion

This chapter has illuminated the captivating world of image classification, demonstrating the application of state-of-the-art deep learning models to categorize images with remarkable accuracy. We unraveled the intricacies of prominent architectures like ResNet, Inception-v3, and MobileNetV2, uncovering their respective strengths and trade-offs. Armed with this knowledge, we can confidently navigate image classification tasks, from preprocessing and model selection to decoding predictions and interpreting results.

Exercises

1. Try running the models on other images provided in the code base.
2. Calculate the time taken by the DNN module to process the image and compare it with the time taken by Keras to process the same image.
3. Try playing with the parameter values to the inputs of **blobFromImage()** and analyze the effects on the result.

---

1 **https://www.researchgate.net/figure/Network-structures-of-ResNet101-top-and-ResNet152-bottom_fig11_343137490**

2 **https://www.researchgate.net/figure/The-architecture-of-MobileNetV2-DNN_fig1_361260658**

Chapter 8
Modern Solutions for Object Detection

Introduction

In this chapter, we delve into object detection, a fundamental task in computer vision that involves identifying and localizing objects within images. We explore key object detection architectures and techniques, from foundational methods like R-CNN and YOLO to advanced models like SSD and Faster R-CNN. By understanding the principles behind these architectures, you will gain insights into how modern computer vision systems can efficiently and accurately detect objects in real-world scenarios. Moreover, we unravel the intricacies of object detection and its evolution in deep learning.

Structure

The chapter discusses the following topics:

- Convolutional neural networks architecture for object detection
- Faster region convolutional neural network
- Single shot multibox detector
- You only look once

Convolutional neural networks architecture for object detection

Convolutional Neural Networks (**CNNs**) are a type of deep learning architecture widely used for object detection tasks in computer vision. Object detection involves identifying and localizing objects within an image. CNNs are particularly effective for this task due to their ability to learn hierarchical features from images automatically. The general architecture of a CNN for object detection consists of several key components:

- **Convolutional layers**: These layers perform convolutional operations on the input image. Convolutional operations involve sliding a small filter (also known as a kernel) over the image and computing element-wise multiplications and summations to extract features. These layers capture low-level features like edges, textures, and patterns.
- **Activation functions**: Activation functions, such as **rectified linear activation** (**ReLU**), are applied after convolutional layers to introduce non-linearity into the network. This enables the network to capture complex relationships between features.
- **Pooling layers**: Pooling layers (for example, max pooling) downsample the spatial dimensions of the feature maps, reducing the computational load and increasing the network's ability to capture larger-scale patterns and features.
- **Convolutional blocks**: These blocks often consist of a combination of convolutional layers, activation functions, and pooling layers. They help capture increasingly abstract features as the network goes deeper.
- **Fully connected (FC) layers**: After several convolutional blocks, fully connected layers are added to the network. These layers combine features from different image regions and are used for classification and localization tasks.
- **Localization and classification heads**: Object detection CNNs typically have two main branches at the end: One for object

localization and another for object classification. The localization branch predicts bounding boxes that enclose the objects along with associated confidence scores. The classification branch assigns class labels to these detected objects.

- **Anchor boxes**: Many modern object detection architectures use anchor boxes or default boxes to predict bounding boxes of various sizes and aspect ratios. These anchor boxes serve as reference templates for the network to predict the final bounding box coordinates.
- **Loss functions**: The network is trained using loss functions that measure the difference between predicted bounding boxes and ground truth boxes, as well as the difference in predicted class probabilities and actual class labels. Common loss functions include the Smooth L1 loss for bounding box regression and the Cross-Entropy loss for classification.
- **Backpropagation and optimization**: During training, the network's weights are adjusted using optimization algorithms such as **stochastic gradient descent** (**SGD**) or Adam, and backpropagation is used to compute gradients and update the weights.
- **Post-processing**: After inference, a post-processing step is performed to filter and refine the detected bounding boxes based on confidence scores and **non-maximum suppression** (**NMS**) to eliminate duplicate detections.

Notable CNN architectures for object detection include Faster R-CNN, **you only look once** (**YOLO**), **single shot multibox detector** (**SSD**), and RetinaNet, each with its own variations and improvements to handle different aspects of object detection efficiently. These architectures have significantly advanced the field of object detection in computer vision.

Faster region convolutional neural network

Faster **region convolutional neural network** (**R-CNN**) is a widely used and influential object detection architecture that significantly improved the efficiency of region proposal generation compared to its predecessors like R-CNN. It combines a **region proposal network** (**RPN**) with a Fast R-

CNN detection framework to create a unified and end-to-end trainable object detection system. Here is an overview of the Faster R-CNN architecture and its object detection capabilities:

- **Region Proposal Network**: The key innovation of Faster R-CNN is the introduction of the region proposal network, which learns to generate region proposals directly from the convolutional feature maps of the input image. The RPN operates on sliding windows of different sizes, termed *anchors*, which serve as potential object bounding box candidates. For each anchor, the RPN predicts two outputs: Objectness score (probability of containing an object) and bounding box regression parameters (adjustments to the anchor to fit the object). The RPN uses these predictions to rank and select high-quality region proposals, filtering out redundant and irrelevant proposals.
- **Feature Pyramid Network**: Many Faster R-CNN implementations use a **feature pyramid network** (**FPN**) as a backbone architecture. FPN enhances feature extraction by aggregating features from different scales and levels of abstraction, enabling the network to detect objects of varying sizes.
- **RoI align and RoI pooling**: After region proposals are generated by the RPN, they are passed to the Fast R-CNN detection framework for further processing. **region of interest** (**RoI**) Align or RoI pooling is used to extract fixed-size feature maps from the convolutional feature maps for each proposed region. RoI align, in particular, overcomes the quantization issues of RoI pooling by introducing a more accurate sampling technique.
- **Classification and regression head**: The Fast R-CNN part of Faster R-CNN consists of a classification head and a bounding box regression head. The classification head predicts class probabilities for the proposed objects, and the regression head predicts adjustments to the bounding box coordinates.
- **Loss functions and training**: Faster R-CNN is trained using a combination of classification and regression loss functions, similar to other object detection architectures. The RPN is trained to minimize the objectness classification loss and bounding box

regression loss. The Fast R-CNN detection head is trained using the Smooth L1 loss for bounding box regression and the Cross-Entropy loss for classification.

- **Inference and post-processing**: During inference, the final set of bounding boxes is obtained by applying NMS to the proposed regions based on their scores and overlap thresholds.

Faster R-CNN's integration of the RPN for efficient region proposal generation, combined with the Fast R-CNN detection framework, resulted in significant speed improvements and better accuracy than previous methods. It became a foundational architecture for modern object detection models and paved the way for subsequent developments in the field.

Faster R-CNN can be used in OpenCV DNN module for object detection as per the code shown below:

```
1. import cv2
2.
3. def detect_coco80objects_using_opencvdnn(image_path, confidence_threshold):
4.     ssd_size=(300, 300)
5.
6.     image = cv2.imread(image_path)
7.     height, width, channels = image.shape
8.
9.     # Load Faster R-CNN model
10.     net = cv2.dnn.readNetFromTensorflow("../weights/8/faster_rcnn/faster_rcnn_resnet50_coco_2018_01_28.pb", "../weights/8/faster_rcnn/faster_rcnn_resnet50_coco_2018_01_28.pbtxt")
11.
```

```
12.     # Read the COCO class names
13.     with open("../weights/8/faster_rcnn/coco.names", 'r') as file:
14.         lines = file.readlines()
15.         classes = [line.strip() for line in lines]
16.
17.
18.      blob = cv2.dnn.blobFromImage(image, size=ssd_size, swapRB=Tr
    ue, crop=False)
19.     net.setInput(blob)
20.     results = net.forward()
21.     # print(results.shape)
22.
23.
24.     objects_and_locations = []
25.     for one_detection in results[0,0,:,:]:
26.         # Each detection is a  1d array. The contents are as explained be
    low
27.         # 2nd position (index 1 for Python) - classid of the object. In cas
    e of this program, it is the COCO80 dataset
28.         # 3rd position (index 2 for Python) - Confidence
    level for detected class.
29.         # 4th position       3           - Left most point of the bounding box
30.         # 5th position       4           - Top most point of the bounding box
```

```
        # 6th position       5           - Right most point of the bounding box
        # 7th position       6           - Bottom most point of the bounding box

        confidence_score = float(one_detection[2])
        if confidence_score <= confidence_threshold:
            continue

        left = int(one_detection[3] * width)
        top = int(one_detection[4] * height)
        right = int(one_detection[5] * width)
        bottom = int(one_detection[6] * height)

        class_ids = int(one_detection[1])
        class_label = classes[class_ids]
        one_object = {}
        one_object["class"] = class_label
        one_object["top_left"] = (left, top)
        one_object["bottom_right"] = (right, bottom)
        one_object["confidence"] = confidence_score
        objects_and_locations.append(one_object)
```

```
51.
52.     return objects_and_locations
53.
54. def draw_outlines_around_detections(impath, objects_and_locations):
55.     image = cv2.imread(impath)
56.     for one_object in objects_and_locations:
57.         cv2.rectangle(image, one_object["top_left"], one_object["bottom
    _right"], (255,255,255), 3)
58.         cv2.putText(image, one_object["class"], one_object["top_left"],
    cv2.FONT_HERSHEY_SIMPLEX, 1, (255,255,255), 3)
59.
60.     return image
61.
62. # Main script execution
63. if __name__ == "__main__":
64.     image_path = "../input_images/aeroplane.jpg"
65.     confidence_threshold=0.5
66.
67.     objects_and_locations = detect_coco80objects_using_opencvdnn(i
    mage_path, confidence_threshold)
68.
69.
```

```
70.     image = draw_outlines_around_detections(image_path, objects_and_locations)
71.     cv2.namedWindow("object detections", cv2.WINDOW_FULLSCREEN)
72.     cv2.imshow("object detections", image)
73.     cv2.waitKey(0)
```

This code demonstrates how to perform object detection using the Faster R-CNN model with OpenCV's **deep neural network** (**DNN**) module. The code detects objects in an image using the COCO80 dataset classes and overlays bounding box outlines and class labels on the detected objects. This is a complete pipeline for object detection. It loads a pre-trained model, detects objects in an image, and visualizes the results by overlaying bounding boxes and class labels on the original image.

Here is a breakdown of the code:

- Define a function named **detect_coco80objects_using_opencvdnn** that takes an image path and a confidence threshold as arguments and returns a list of detected objects and their locations:
    - Set the desired size for the input image (**ssd_size**) of (*300,300*) pixels.
    - Read the input image and get its dimensions.
    - Load the pre-trained Faster R-CNN model using **cv2.dnn.readNetFromTensorflow** with the paths to the model's **.pb** file and **.pbtxt** configuration file.
    - Read the class names from the COCO80 dataset.
    - Preprocess the input image by creating a blob and setting it as the input to the network.
    - Forward pass through the network to obtain detection results.
    - Process the detection results, filter out low-confidence detections, and store the detected object details in a list.
    - Return the list of detected objects and their locations.

- Define a function named **draw_outlines_around_detections** that takes an image path and the list of detected objects and their locations as arguments and returns the input image with bounding box outlines and class labels drawn around the detected objects:
    - Read the input image.
    - Iterate through each detected object and draw a bounding box around it using **cv2.rectangle**.
    - Draw the class label on the image using **cv2.putText**.
- In the main script execution:
    - Set the input image path and confidence threshold.
    - Call the **detect_coco80objects_using_opencvdnn** function to detect objects and store the results in **objects_and_locations**.
    - Call the **draw_outlines_around_detections** function to draw bounding box outlines and class labels on the input image.
    - Display the modified image using **cv2.imshow** and wait for a key press using **cv2.waitKey**.

Executing this program generates the *Figure 8.1*:

```
D:\bpb\995\8>python detect_fasterrcnn_dnn.py
```

***Figure 8.1**: Object detection with Faster R-CNN*

Single shot multibox detector

Single Shot Multibox Detector (**SSD**) is another popular object detection architecture in computer vision. SSD is known for its speed and accuracy in detecting objects of various sizes within an image. It is designed to perform real-time object detection by predicting object bounding boxes and class probabilities directly from a single pass through the network. Here is an overview of the SSD architecture and its object detection capabilities:

- **Base convolutional layers**: Similar to other convolutional neural networks, SSD begins with a base network (often VGG or a similar architecture) that extracts features from the input image. These features are used for object detection.
- **Multi-scale feature maps**: One of the key features of SSD is its use of multiple convolutional layers to capture features at different scales. These layers have varying receptive field sizes, allowing the network to detect objects of different sizes. Each of these layers generates a set of feature maps representing different scales of the input image.

- **Convolutional predictors**: On top of each feature map, SSD attaches a set of convolutional layers for making predictions. These prediction layers are responsible for detecting objects within specific size ranges. They predict bounding box coordinates (offsets from default boxes) and class scores for each position in the feature map.
- **Default boxes**: SSD employs a set of default boxes, also known as **anchor boxes**, at each position in the feature map. These default boxes have different aspect ratios and scales, enabling the model to handle a wide range of object sizes and shapes. The network then predicts adjustments to these default boxes to accurately localize objects.
- **Multi-scale predictions**: By having multiple prediction layers attached to feature maps of different scales, SSD generates predictions for objects of various sizes simultaneously. This is crucial for detecting small and large objects in the same pass.
- **Loss function**: SSD uses a combination of localization loss (often the Smooth L1 loss) and classification loss (typically the Softmax Cross-Entropy loss) to train the network. The localization loss measures the accuracy of bounding box predictions, while the classification loss evaluates the correctness of predicted class probabilities.
- **Hard negative mining**: To handle class imbalance (where background class is dominant), SSD often employs hard negative mining. This involves selecting a subset of background predictions with high confidence scores for training, helping the model focus on challenging examples.
- **Non-Maximum Suppression**: During inference, SSD applies NMS to filter out redundant and overlapping bounding boxes based on confidence scores and overlap thresholds.

The key strengths of SSD lie in its real-time detection capabilities, efficient single-pass inference, and ability to handle objects of various scales within a single network architecture. Over the years, SSD has inspired various improvements and variations, such as SSD with MobileNet, SSD with

ResNet, and more, to enhance its performance and adaptability to different applications.

The architecture diagram of SSD is shown in *Figure 8.2*. This image has been taken from **https://pytorch.org/assets/images/ssd_diagram.png**.

Liu *et al.*

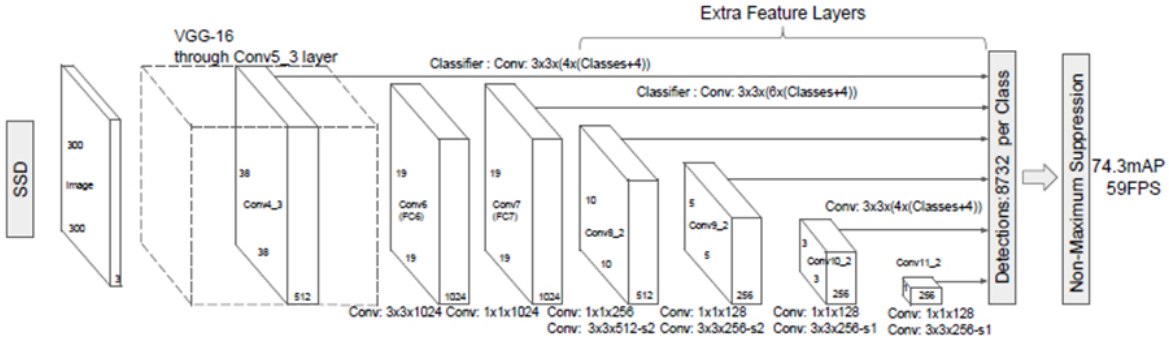


***Figure 8.2**: Architecture diagram of SSD*

The code for implementing object detection using SSD300 network can be seen as follows:

```
1. import cv2
2.
3. def  detect_coco80objects_using_opencvdnn(image_path, confidence_threshold):
4.     ssd_size=(300, 300)
5.
6.     image = cv2.imread(image_path)
7.     height, width, channels = image.shape
8.
9.     # Load SSD300
```

```
10.     net = cv2.dnn.readNetFromTensorflow("../weights/8/ssd300/ssd_mobilenet_v2_coco_2018_03_29.pb",  "../weights/8/ssd300/ssd_mobilenet_v2_coco_2018_03_29.pbtxt")
11.
12.     # Read the COCO class names
13.     with open("../weights/8/ssd300/coco.names", 'r') as file:
14.         lines = file.readlines()
15.         classes = [line.strip() for line in lines]
16.
17.
18.     blob = cv2.dnn.blobFromImage(image, size=ssd_size, swapRB=True, crop=False)
19.     net.setInput(blob)
20.     results = net.forward()
21.     # print(results.shape)
22.
23.
24.     objects_and_locations = []
25.     for one_detection in results[0,0,:,:]:
26.         # Each detection is a  1d array. The contents are as explained below
27.         # 2nd position (index 1 for Python) - classid of the object. In case of this program, it is the COCO80 dataset
```

```
        # 3rd position (index 2 for Python) - Confidence level for detected class.
        # 4th position  3  - Left most point of the bounding box
        # 5th position  4  - Top most point of the bounding box
        # 6th position  5  - Right most point of the bounding box
        # 7th position  6  - Bottom most point of the bounding box

        confidence_score = float(one_detection[2])
        if confidence_score <= confidence_threshold:
            continue

        left = int(one_detection[3] * width)
        top = int(one_detection[4] * height)
        right = int(one_detection[5] * width)
        bottom = int(one_detection[6] * height)

        class_ids = int(one_detection[1])
        class_label = classes[class_ids]
        one_object = {}
        one_object["class"] = class_label
```

```
47.         one_object["top_left"] = (left, top)
48.         one_object["bottom_right"] = (right, bottom)
49.         one_object["confidence"] = confidence_score
50.         objects_and_locations.append(one_object)
51.
52.     return objects_and_locations
53.
54. def draw_outlines_around_detections(impath, objects_and_locations):
55.     image = cv2.imread(impath)
56.     for one_object in objects_and_locations:
57.         cv2.rectangle(image, one_object["top_left"], one_object["bottom
    _right"], (255,255,255), 3)
58.         cv2.putText(image, one_object["class"], one_object["top_left"],
    cv2.FONT_HERSHEY_SIMPLEX, 1, (255,255,255), 3)
59.
60.     return image
61.
62. # Main script execution
63. if __name__ == "__main__":
64.     image_path = "../input_images/aeroplane.jpg"
65.     confidence_threshold=0.5
66.
```

```
    objects_and_locations = detect_coco80objects_using_opencvdnn(image_path, confidence_threshold)


    image = draw_outlines_around_detections(image_path, objects_and_locations)
    cv2.namedWindow("object detections", cv2.WINDOW_FULLSCREEN)
    cv2.imshow("object detections", image)
    cv2.waitKey(0)
```

This code demonstrates how to perform object detection using the MobileNetV2-based SSD model with OpenCV's DNN module. The code detects objects in an image using the COCO80 dataset classes and overlays bounding box outlines and class labels on the detected objects. Here is a breakdown of the code:

1. Define a function named **detect_coco80objects_using_opencvdnn** that takes an image path and a confidence threshold as arguments and returns a list of detected objects and their locations:
    a. Set the desired size for the input image (**ssd_size**).
    b. Read the input image and get its dimensions.
    c. Load the pre-trained SSD300 model based on MobileNetV2 using **cv2.dnn.readNetFromTensorflow** with the paths to the model's **.pb** file and **.pbtxt** configuration file.
    d. Read the class names from the COCO80 dataset.
    e. Preprocess the input image by creating a blob and setting it as the input to the network.
    f. Forward pass through the network to obtain detection results.

g. Process the detection results, filter out low-confidence detections, and store the detected object details in a list.

h. Return the list of detected objects and their locations.

2. Define a function named **draw_outlines_around_detections** that takes an image path and the list of detected objects and their locations as arguments and returns the input image with bounding box outlines and class labels drawn around the detected objects:

   a. Read the input image.

   b. Iterate through each detected object and draw a bounding box around it using **cv2.rectangle**.

   c. Draw the class label on the image using **cv2.putText**.

3. In the main script execution:

   a. Set the input image path and confidence threshold.

   b. Call the **detect_coco80objects_using_opencvdnn** function to detect objects and store the results in **objects_and_locations**.

   c. Call the **draw_outlines_around_detections** function to draw bounding box outlines and class labels on the input image.

   d. Display the modified image using **cv2.imshow** and wait for a key press using **cv2.waitKey**.

Executing this code generates the output shown in *Figure 8.3*:

```
1. D:\bpb\995\8>python detect_ssd_dnn.py
```

***Figure 8.3**: Object detection with SSD300 with MobilenetV2*

Note: Discerning reader will observe that the code for using SSD and Faster R-CNN networks is exactly the same. The two programs are identical except for using different model networks, weights, and location of coco.names file. This, however, is the most crucial difference. SSD300 contains an additional class called None compared to Faster R-CNN. Comparing the coco.names files will make this difference much more obvious. The coco.names file, pb and pbtxt file should always remain in sync. Mismatches in this will throw off the detection. OpenCV expects the developers to be responsible for keeping these files in synch.

You only look once

You Only Look Once (**YOLO**) is a revolutionary object detection framework in computer vision that has greatly influenced the field of deep learning and image understanding. YOLO's unique approach to object detection, focusing on real-time speed and accuracy, has made it one of the most popular and impactful object detection architectures. YOLO aims to simultaneously predict object bounding boxes and class probabilities within

a single forward pass of a neural network. Unlike traditional object detection methods that involve multiple stages and complex pipelines, YOLO approaches object detection as a regression problem, directly predicting object attributes from raw image data.

The original YOLO model was introduced by *Joseph Redmon et al*. in the paper *You Only Look Once: Unified, Real-Time Object Detection*. It presented a groundbreaking approach to real-time object detection. YOLO divided the input image into a grid and made predictions for bounding boxes and class probabilities at each grid cell, allowing for rapid inference. YOLOv2, also known as **YOLO9000,** brought improvements in accuracy and versatility. It introduced anchor boxes and a multi-scale approach for predicting objects of different sizes. YOLOv2 also expanded its detection capability to a wide range of object categories, extending beyond the initial 20 classes of the COCO dataset.

YOLOv3, released in 2018, enhanced detection accuracy and introduced a **feature pyramid network** (**FPN**) to capture multi-scale information. It utilized multiple scales of anchor boxes and introduced the concept of *darknet-53*, a deeper neural network backbone, to improve feature extraction. Although not an official release, YOLOv4 gained significant attention as an unofficial continuation of the YOLO series. YOLOv4 introduced numerous optimizations, architectural enhancements, and advanced training techniques, resulting in remarkable accuracy improvements and competitive performance.

YOLOv5 was developed as an independent project by the Ultralytics team in 2020. It aimed to simplify the YOLO architecture while maintaining or even improving accuracy. YOLOv5 introduced variants (s, m, l, and x) to provide a balance between speed and accuracy, along with efficient training and deployment. We will discuss YOLOv5 later in the chapter.

YOLOv3

YOLOv3 is a popular and powerful object detection architecture that can detect and localize objects in an image with high speed and accuracy. It builds upon the YOLO concept of treating object detection as a regression problem, predicting bounding box coordinates and class probabilities directly from a single pass through the network.

Here is an overview of the YOLOv3 architecture:

- **Input processing**: YOLOv3 takes an input image and divides it into a grid. Each grid cell is responsible for detecting objects that fall within its region.
- **Darknet-53 backbone**: The architecture starts with a backbone network called Darknet-53, which is a deep convolutional neural network with 53 layers. It extracts hierarchical features from the input image and provides a rich representation that captures both low-level and high-level features.
- **Detection at different scales**: YOLOv3 performs detection at three different scales using feature maps from different layers of the Darknet-53 backbone. Each scale is associated with anchor boxes of different sizes to handle objects of various scales and aspect ratios.
- **Detection head**: The detection head consists of convolutional layers that predict bounding box coordinates (center x, center y, width, height) and class probabilities for each anchor box. This is done for all grid cells and anchor boxes at each scale.
- **Anchors and predictions**: YOLOv3 employs a set of predefined anchor boxes at each scale. The predicted bounding box coordinates are adjusted relative to these anchor boxes. For each grid cell, multiple anchor boxes are used, and the detection head predicts offsets to adjust the anchor box dimensions.
- **Multi-scale detection**: To capture objects of varying sizes, YOLOv3 uses feature maps from different layers of the Darknet-53 backbone. The detection head at each scale predicts object detections, and these predictions are then consolidated into a final set of detections.
- **Non-Maximum Suppression**: After predictions are made, a post-processing step involves applying NMS to remove redundant and overlapping bounding boxes based on their confidence scores and the degree of overlap.
- **Output**: The final output of YOLOv3 is a set of bounding boxes, each associated with a class label and a confidence score.

- **Architecture variants**: YOLOv3 has different configurations: YOLOv3-tiny (a smaller and faster version), and YOLOv3-SPP (spatial pyramid pooling) which incorporates spatial pyramid pooling to capture context information.
- **Training**: YOLOv3 is trained using labeled training data, optimizing for a combination of localization loss (often using the Smooth L1 loss) and classification loss (typically using the cross-entropy loss). The network is trained end-to-end using backpropagation and optimization algorithms like SGD or Adam.

The key strengths of YOLOv3 are its real-time inference speed and ability to detect objects of different scales and aspect ratios. It is widely used in various applications, including autonomous vehicles, surveillance, and robotics, where fast and accurate object detection is crucial. Please see *Figure 8.4* for YOLOv3 architecture. This image is taken from **https://viso.ai/deep-learning/yolov3-overview/**.

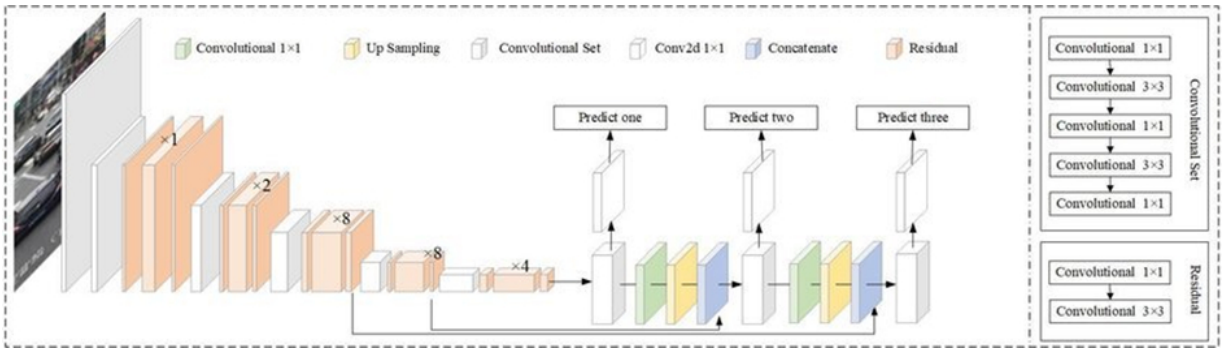


***Figure 8.4**: Architecture of YOLOv3*

Here we shall see code for using YOLOv3 from OpenCV DNN module:

```
1. import cv2
2. import numpy as np
3.
4. def process_detection(image, detection):
5.     # YOLO returns values between 0 and 1. This value has to be scale
   d to suit the image size.
```

```
    # This is the reverse of standardizing the data.
    height, width, channels = image.shape
    center_x = int(detection[0] * width)
    center_y = int(detection[1] * height)
    object_width = int(detection[2] * width)
    object_height = int(detection[3] * height)

    # Rectangle coordinates
    topleft_x = int(center_x - object_width/2)
    topleft_y = int(center_y - object_height/2)

    return (topleft_x, topleft_y, object_width, object_height)



def detect_coco80objects_using_opencvdnn(impath, confidence_threshold = 0.5):

    scaling_factor = 1/255
    yolo_shape = (416,416)

```

```
    # Load Yolo
    model = cv2.dnn.readNet("../weights/8/yolo/yolov3.weights", "../weights/8/yolo/yolov3.cfg")

    # Read the COCO class names
    with open("../weights/8/yolo/coco.names", 'r') as file:
        lines = file.readlines()
        classes = [line.strip() for line in lines]

    image = cv2.imread(impath)
    blob = cv2.dnn.blobFromImage(image, scaling_factor, yolo_shape, (0, 0, 0))


    # get all the layer names of the model
    layer_names = model.getLayerNames()
    # filter and choose only the output layers
    output_layers = [layer_names[i - 1] for i in model.getUnconnectedOutLayers()]

    # Detecting objects
    model.setInput(blob)
```

```
    results = model.forward(output_layers)

    # results is a tuple. It›s length is equal to the number of output layers in the model.
    object_classes = []
    object_confidences = []
    object_coordinates = []


    for one_layer in results:
        # each layer can have multiple detections. Cycle through them all.
        for one_detection in one_layer:
            # Each detection is a  1d array. The contents are as explained below
                # 1st position (index 0 for Python) - x-coordinate of the bounding box›s centroid of the detected object
                # 2nd position (index 1 for Python) - y-coordinate of the bounding box›s centroid of the detected object
            # 3rd position - width of the bounding box
            # 4th position - height of the bounding box
            # 5th till end - Confidence level for each class of detected object. In case of this program,
```

```
62.         #                       it is the COCO80 dataset
63.         confidence_scores_for_classes = one_detection[5:]
64.         classid_with_highest_confidence = np.argmax(confidence_sco
    res_for_classes)
65.         class_confidence = confidence_scores_for_classes[classid_wit
    h_highest_confidence]
66.
67.         if class_confidence > confidence_threshold:
68.             object_location = process_detection(image, one_detection)
69.             object_coordinates.append(object_location)
70.             object_confidences.append(float(class_confidence))
71.              object_classes.append(classes[classid_with_highest_confid
    ence])
72.
73.
74.     # Now we are left with only objects that meet the confidence thresh
    old. However, there can still be multiple detections
75.     # for the same object with overlapping areas.
76.     # So, we need to de-duplicate them. We shall do so
    using the Non Maximum Suppression algorithm.
77.     indexes = cv2.dnn.NMSBoxes(object_coordinates, object_confiden
    ces, 0.4, 0.3)
78.
```

```
79.
80.     # indexes contains the objects which are of
interest to us. Cycle through the indexes and calculate the coordinates
 in a way
81.     # that OpenCV can understand them.
82.     objects_and_locations = []
83.     for inx in indexes:
84.         class_label = object_classes[inx]
85.         (x,y,width,height) = object_coordinates[inx]
86.         top_left_coordinate = (x, y)
87.         bottom_right_coordinate = (x + width, y + height)
88.
89.         one_object = {}
90.         one_object["class"] = class_label
91.         one_object["top_left"] = top_left_coordinate
92.         one_object["bottom_right"] = bottom_right_coordinate
93.         objects_and_locations.append(one_object)
94.
95.
96.     return objects_and_locations
97.
98.
```

```
99.
100. def draw_outlines_around_detections(impath, objects_and_locations):
101.     image = cv2.imread(impath)
102.     for one_object in objects_and_locations:
103.         cv2.rectangle(image, one_object["top_left"], one_object["bottom
_right"], (255,255,255), 3)
104.         cv2.putText(image, one_object["class"], one_object["top_left"],
cv2.FONT_HERSHEY_SIMPLEX, 1, (255,255,255), 3)
105.
106.     return image
107.
108. # Main script execution
109. if __name__ == "__main__":
110.     image_path = "../input_images/test_image1.jpeg"
111.
112.     objects_and_locations = detect_coco80objects_using_opencvdnn(i
mage_path)
113.
114.     image = draw_outlines_around_detections(image_path, objects_an
d_locations)
115.     cv2.namedWindow("object detections", cv2.WINDOW_FULLSCR
EEN)
116.     cv2.imshow("object detections", image)
```

```
117.    cv2.waitKey(0)
118.
119.
```

We will understand the function **detect_coco80objects_using_opencvdnn** here. This function differs significantly from the implementations we have seen for SSD and Faster R-CNN and bears a more elaborate explanation. Let us go through the function step by step:

1. Set scaling and shape parameters:
    a. **scaling_factor = 1/255**: This variable scales the pixel values of the input image to the range [0, 1].
    b. **yolo_shape = (416, 416)**: This specifies the desired shape of the input image that YOLO expects (*416x416* pixels).
2. Load YOLO model:
    a. **model = cv2.dnn.readNet(...)**: Loads the YOLO model using its configuration file **(yolov3.cfg)** and pre-trained weights **(yolov3.weights)**.
3. Read COCO class names:
    a. The COCO class names are read from the file **coco.names** and stored in the classes list. Note that this **coco.names** matches the **coco80** dataset.
4. Load and preprocess the input image:
    a. **image = cv2.imread(impath)**: Reads the input image.
    b. **blob = cv2.dnn.blobFromImage(...)**: Preprocesses the image by converting it into a blob that the YOLO model can process. It applies scaling, resizing, and normalization to the image.
5. Get output layer names:
    a. **layer_names = model.getLayerNames()**: Gets the names of all layers in the YOLO model.

b. **output_layers = [...]**: Filters and selects the output layers from the YOLO model using **model.getUnconnectedOutLayers()**.

6. Detect objects:
   a. **model.setInput(blob)**: Sets the preprocessed image as input to the model.
   b. **results = model.forward(output_layers)**: Performs a forward pass through the YOLO model and obtains detection results.
7. Process detection results:
   a. Iterates through each layer's detection results.
   b. For each detection in a layer, it extracts the *x*, *y*, width, and height of the bounding box, along with confidence scores for each class.
   c. Selects the class with the highest confidence score for the detection.
   d. If the class confidence is above the given **confidence_threshold**, it processes the detection to obtain the bounding box coordinates using the **process_detection** function.
   e. The processed detection information (class label, coordinates, and confidence) is added to respective lists (**object_classes**, **object_coordinates**, and **object_confidences**).
8. Non-maximum suppression:
   a. After processing all detections, NMS is applied to remove redundant and overlapping detections.
   b. **cv2.dnn.NMSBoxes(...)** returns indexes of the selected detections that pass NMS.
9. Finalize objects and locations: The selected detections are used to populate the **objects_and_locations** list, containing dictionaries with class labels, top-left, and bottom-right coordinates of the bounding boxes.
10. Return detected objects and locations.

The biggest difference here is the use of OpenCV DNN module's **NMSBoxes()** function. This activity is performed by SSD and Faster R-CNN automatically. For YOLOv3, the developers need to call it explicitly. Executing this program generates the output shown in *Figure 8.5*:

```
D:\bpb\995\8>python detect_yolov3_dnn.py
```

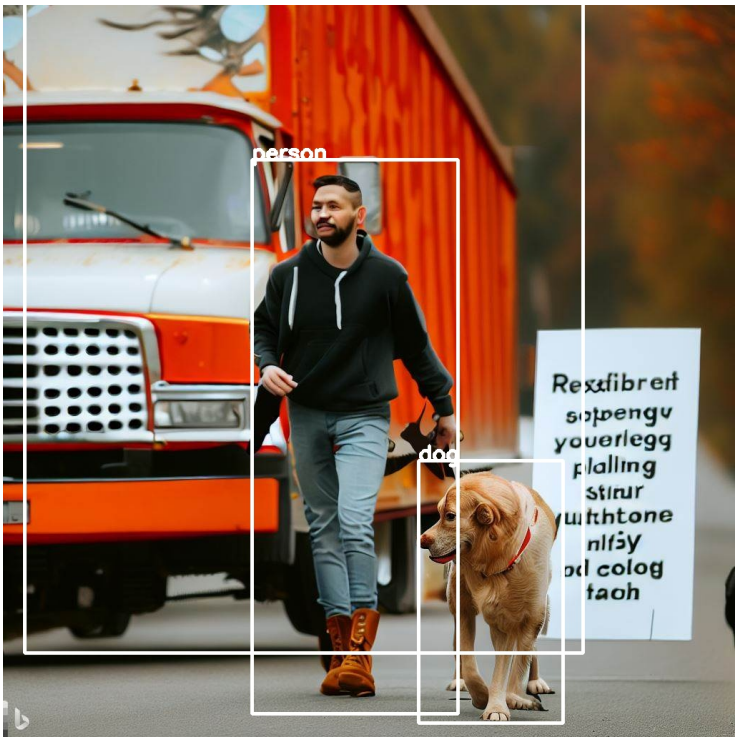


***Figure 8.5**: Object detection with YOLOv3*

Overview of NMSBoxes() API

The NMSBoxes function in OpenCV's DNN module is used for applying NMS to a list of bounding box detections. NMS is a post-processing step commonly used in object detection tasks to eliminate duplicate and overlapping detections, ensuring that each object is detected only once. NMS helps to improve the quality and reliability of the final detection results.

The input parameters are listed here:

- **bboxes**: A list of bounding boxes, each represented as a tuple of (x, y, width, height).

- **scores**: A list of confidence scores associated with the bounding boxes.
- **score_threshold:** A threshold value for confidence scores. Bounding boxes with scores below this threshold will be discarded.
- **nms_threshold**: A threshold value for NMS. Bounding boxes with an **intersection-over-union** (**IoU**) overlap greater than or equal this threshold will be suppressed.

NMS is applied to the bounding boxes with confidence scores higher than the **score_threshold**. It involves comparing pairs of bounding boxes and suppressing the one with lower confidence if their intersection over union is above the **nms_threshold**. This prevents redundant and overlapping detections from being included in the final result.

The function returns a list of indexes that correspond to the selected bounding boxes after NMS. These indexes can retrieve the final set of detected objects that have passed both the confidence threshold and NMS.

YOLOv5

YOLOv5 is an evolution of the YOLO series of object detection models. YOLOv5 aims to improve upon its predecessors with a focus on simplicity, efficiency, and performance. It was developed by the **Ultralytics** team and has gained popularity for its ease of use, speed, and strong detection capabilities. Here is an overview of the YOLOv5 architecture:

- **Backbone**: YOLOv5 uses a CSPDarknet53 backbone, which is a modified version of the Darknet backbone used in YOLOv3. CSPDarknet53 employs a cross-stage hierarchy, allowing for better feature reuse across different scales.
- **Neck**: YOLOv5 introduces PANet (path aggregation network) as a neck architecture. PANet helps to aggregate features from different scales and enhances the representation capabilities of the network.
- **Detection head**: The detection head consists of a series of convolutional and upsampling layers. This head predicts bounding box coordinates, class probabilities, and objectness scores for each anchor box at multiple scales.

- **Anchor boxes**: YOLOv5 employs anchor boxes of different sizes and aspect ratios, similar to previous YOLO versions. The network predicts offsets and scales to adjust these anchor boxes for accurate object localization.
- **Feature pyramid**: YOLOv5 uses a feature pyramid approach to capture multi-scale information. Features from different scales are fused to provide a more comprehensive representation of the input image.
- **Multi-scale training**: YOLOv5 introduces a multi-scale training strategy, where the network is trained on images of varying resolutions during different stages of training. This improves the model's robustness to different object scales.
- **Loss function**: The loss function used in YOLOv5 combines objectness loss, localization loss (usually Smooth L1 loss), and classification loss (Cross-Entropy loss). It also employs techniques like focal loss to prioritize challenging samples and improve convergence.
- **Data augmentation**: YOLOv5 uses extensive data augmentation during training, including random scaling, cropping, rotation, and color adjustments. This helps the model generalize better to different scenarios.
- **Inference**: During inference, YOLOv5 employs an NMS post-processing step to remove redundant and overlapping bounding boxes based on their confidence scores and overlap thresholds.
- **Model variants**: YOLOv5 comes in different sizes, labeled as YOLOv5s, YOLOv5m, YOLOv5l, and YOLOv5x, with varying numbers of layers and parameters. These variants provide a trade-off between speed and accuracy, allowing users to choose the model that best suits their needs.
- **Efficiency and portability**: YOLOv5 is designed for efficiency and can be easily deployed on various platforms, including edge devices and embedded systems.

Overall, YOLOv5 builds upon the YOLO concept of single-pass object detection while introducing improvements in terms of backbone

architecture, feature aggregation, and training strategies. It has become a popular choice for real-time object detection tasks due to its performance and user-friendly design.

YOLOv5 comes in different model variants, each with varying numbers of layers and parameters. These variants allow you to choose a trade-off between speed and accuracy based on your specific application requirements.

- **YOLOv5s (Small)**: YOLOv5s is the smallest variant in the YOLOv5 series. It has fewer layers and parameters, making it the fastest but least accurate option among the variants. YOLOv5s is suitable for scenarios where real-time or near-real-time inference speed is crucial, and a moderate level of accuracy is sufficient.
- **YOLOv5m (Medium)**: YOLOv5m is a medium-sized variant that balances speed and accuracy. It offers better detection performance compared to YOLOv5s while still maintaining relatively fast inference times. YOLOv5m is a good choice when you need a trade-off between speed and accuracy and want a versatile model for various applications.
- **YOLOv5l (Large)**: YOLOv5l is a larger variant with more layers and parameters, providing improved accuracy at the cost of slightly slower inference speed compared to the smaller variants. It is suitable for applications where accuracy is a higher priority and real-time performance is not as critical.
- **YOLOv5x (Extra large)**: YOLOv5x is the largest and most powerful variant in the YOLOv5 series. It has the most layers and parameters, resulting in the highest accuracy but slower inference times. YOLOv5x is ideal for tasks where achieving the highest possible accuracy is crucial and real-time performance can be sacrificed to some extent.

Overall, the choice between YOLOv5s, YOLOv5m, YOLOv5l, and YOLOv5x depends on your specific project requirements, including the balance between accuracy and speed that you need to achieve. It is recommended to consider factors such as the available hardware, the

desired level of accuracy, and the real-time constraints of your application when selecting the appropriate YOLOv5 variant.

Differences between YOLOv3 and v5

YOLOv3 and YOLOv5 are members of the YOLO family of object detection models. While they share some similarities, they also have significant differences in architecture, design philosophy, and performance. YOLOv5 introduces several architectural and training improvements aimed at simplifying the training process, enhancing feature aggregation, and providing a range of model variants for different applications. Refer *Table 8.1* for a comparison of YOLOv3 and YOLOv5:

	YOLOv3	**YOLOv5**
Backbone architecture	Uses a Darknet-53 backbone with 53 convolutional layers to extract features from the input image. It focuses on capturing both low-level and high-level features to improve detection accuracy.	Uses a CSPDarknet53 backbone, which is a modified version of Darknet-53. It incorporates a cross-stage hierarchy to enhance feature reuse across different scales and improve representation capabilities.
Neck architecture	Does not have a dedicated neck architecture for feature aggregation between different scales.	Introduces **Path Aggregation Network** as a neck architecture. PANet helps aggregate features from multiple scales, enhancing the network's ability to detect objects of varying sizes.

	YOLOv3	YOLOv5
Detection head	Has a detection head that predicts bounding box coordinates, class probabilities, and object scores for anchor boxes at three different scales.	Has a similar detection head but introduces some improvements.
Multi-scale training	Uses a fixed input image size during training.	Employs a multi-scale training strategy where the model is trained on images of varying resolutions. This helps improve the model's robustness to different object scales.
Model variants	Comes in a single full variant with a specific backbone architecture and detection head design.	Offers multiple variants (s, m, l, and x) with varying numbers of layers and parameters, allowing users to choose a trade-off between speed and accuracy.
Training Techniques	Uses focal loss and binary cross-entropy loss for classification and object prediction. It also incorporates anchor box design and other tech	Uses similar loss functions but introduces some additional training strategies and data augmentation techniques to improve performance.

	YOLOv3	YOLOv5
	niques for better det ection.	
Ease of use and de ployment	Widely known and used, but its configu ration and training p rocess can be moder ately complex.	Designed with a foc us on simplicity, eas e of use, and efficie nt deployment. It pr ovides a more strea mlined training proc ess and offers pre-tr ained models for qu ick implementation.

***Table 8.1**: Comparison of YOLO v3 and v5*

There have been subsequent releases to YOLO by Ultralytics team. As of the time of writing this book, the latest model is YOLOv8. However, from an OpenCV programmer's perspective, code for using v5 and v8 is almost similar.

The YOLOv5 model is provided as a Pytorch file. OpenCV DNN requires models to be written in ONNX format. The Pytorch model should be converted to ONNX and can then be used in the following code. Steps for this have been provided later in the chapter.

```
1. import cv2
2. import numpy as np
3.
4.
5. def process_detection(image, yolo_shape, detection):
6.     height, width, channels = image.shape
7.     x_scaling = width/yolo_shape[0]
```

```
    y_scaling = height/yolo_shape[1]

    center_x = detection[0]
    center_y = detection[1]
    object_width = detection[2]
    object_height = detection[3]

    # Rectangle coordinates
    topleft_x = int(x_scaling * (center_x - object_width/2))
    topleft_y = int(y_scaling * (center_y - object_height/2))

    object_width = int(x_scaling * object_width)
    object_height = int(y_scaling * object_height)

    return (topleft_x, topleft_y, object_width, object_height)


def detect_coco80objects_using_opencvdnn(impath, confidence_threshold = 0.5):

```

```
28.     scaling_factor = 1/255
29.     nms_threshold = 0.1
30.     yolo_shape = (640,640)
31.
32.     # Load Yolo
33.     model = cv2.dnn.readNet("../weights/8/yolo/YOLOv5s.onnx")
34.
35.     # Read the COCO class names
36.     with open("../weights/8/yolo/coco.names", 'r') as file:
37.         lines = file.readlines()
38.         classes = [line.strip() for line in lines]
39.
40.     image = cv2.imread(impath)
41.     blob = cv2.dnn.blobFromImage(image, scaling_factor, yolo_shape,
    (0, 0, 0), 1,crop=False)
42.
43.
44.     # get all the layer names of the model
45.     layer_names = model.getLayerNames()
46.     # filter and choose only the output layers
47.     output_layers = [layer_names[i - 1] for i in model.getUnconnected
    OutLayers()]
```

```
48.
49.     # Detecting objects
50.     model.setInput(blob)
51.     results = model.forward(output_layers)
52.
53.     # results is a tuple. It›s length is equal to the number of output layer
        s in the model.
54.     object_classes = []
55.     object_confidences = []
56.     object_coordinates = []
57.
58.
59.     number_of_detections = results[0].shape[1]
60.     for inx in range(number_of_detections):
61.         one_detection = results[0][0][inx]
62.
63.         # Each detection is a  1d array. The contents are as explained bel
        ow
64.                         # 1st position (index 0 for Python) - x-
        coordinate of the bounding box's centroid of the detected object
65.                         # 2nd position (index 1 for Python) - y-
        coordinate of the bounding box's centroid of the detected object
66.         # 3rd position              - width of the bounding box
```

```
67.         # 4th position              - height of the bounding box
68.         # 5th till end                    - Confidence level for each class of de
    tected object. In case of this program,
69.         #               it is the COCO80 dataset
70.
71.         confidence_scores_for_classes = one_detection[5:]
72.          classid_with_highest_confidence = np.argmax(confidence_score
    s_for_classes)
73.         class_confidence = confidence_scores_for_classes[classid_with_
    highest_confidence]
74.
75.         if class_confidence > confidence_threshold:
76.              object_location = process_detection(image, yolo_shape, one_
    detection)
77.             object_coordinates.append(object_location)
78.             object_confidences.append(float(class_confidence))
79.             object_classes.append(classes[classid_with_highest_confidenc
    e])
80.
81.
82.
83.     # Now we are left with only objects that meet the confidence thresh
    old. However, there can still be multiple detections
```

```
84.     # for the same object with overlapping areas.
85.     # So, we need to de-duplicate them. We shall do so using the Non Maximum Suppression a lgorithm.
86.     indexes = cv2.dnn.NMSBoxes(object_coordinates, object_confidences, confidence_threshold, nms_threshold)
87.
88.
89.     # indexes contains the objects which are of interest to us. Cycle through the indexes and calculate the coordinates in a way
90.     # that OpenCV can understand them.
91.     objects_and_locations = []
92.     for inx in indexes:
93.         class_label = object_classes[inx]
94.         (x,y,width,height) = object_coordinates[inx]
95.         top_left_coordinate = (x, y)
96.         bottom_right_coordinate = (x + width, y + height)
97.
98.         one_object = {}
99.         one_object["class"] = class_label
100.        one_object["top_left"] = top_left_coordinate
101.        one_object["bottom_right"] = bottom_right_coordinate
102.        one_object["confidence"] = object_confidences[inx]
```

```
103.         objects_and_locations.append(one_object)
104.
105.
106.     return objects_and_locations
107.
108.
109. def draw_outlines_around_detections(impath, objects_and_locations):
110.     image = cv2.imread(impath)
111.     for one_object in objects_and_locations:
112.         cv2.rectangle(image, one_object["top_left"], one_object["bottom
_right"], (255,255,255), 3)
113.         cv2.putText(image, one_object["class"], one_object["top_left"],
cv2.FONT_HERSHEY_SIMPLEX, 1, (255,255,255), 3)
114.
115.     return image
116.
117. # Main script execution
118. if __name__ == "__main__":
119.     image_path = "../input_images/aeroplane.jpg"
120.     confidence_threshold=0.99
121.
```

```
122.     objects_and_locations = detect_coco80objects_using_opencvdnn(image_path, confidence_threshold)
123.
124.     image = draw_outlines_around_detections(image_path, objects_and_locations)
125.     cv2.namedWindow("object detections", cv2.WINDOW_FULLSCREEN)
126.     cv2.imshow("object detections", image)
127.     cv2.waitKey(0)
```

We will focus here only on the differences in the code for v3 and v5 models inference.

- **process_detection()** function
    - YOLOv3 works with fixed size images. So, there is no need to apply scaling on the coordinates returned by the model. Whereas v5 can work with images of multiple sizes depending on the model chosen. So, a scaling factor has to be applied for the v5 code.
    - The size of the image is dependent on the Yolo model you choose. The P5 series models v5n (nano), v5s (small), v5m (medium), v5l (large), v5x (extra large) use image size of *640*640* pixels. P6 series models YOLOv5n6, YOLOv5s6, YOLOv5m6, YOLOv5l6, YOLOv5x6 use images of size (*1280,1280*). This value is represented by **yolo_size** parameter in this function.
- **detect_coco80objects_using_opencvdnn()** function
    - v3 code has loaded darknet compatible CFG and weights file.
    - v5 code has loaded the YOLOv5s model in ONNX format.
    - The return value of model processing differs in the object and data formatting. These have been addressed in the for loop.

V3 model needs a nested for loop. V5 model does not.

Executing the above code generates the below output shown in *Figure 8.6*:

```
1. D:\bpb\995\8>python detect_yolov5_dnn.py
```

***Figure 8.6**: Object detection using YOLOv5*

Obtaining v5 model ONNX file

In the code GitHub repository for this book, all the models and weights have been provided. However, the YOLOv5 and later models have slightly different licensing. So, the weights are not being provided with this book's repository. Detailed steps are provided here to download the weights. Readers are encouraged to analyze the licensing terms and download the models per the steps mentioned here. It is recommended to run these steps in a separate Python virtual environment created using either venv or Anaconda. The below script uses Anaconda:

```
1. conda create --name pyt2onnx
2. git clone https://github.com/ultralytics/YOLOv5
3. cd YOLOv5
```

4. pip install -r requirements.txt
5. pip install onnx
6. wget https://github.com/ultralytics/YOLOv5/releases/download/v6.1/YOLOv5s.pt
7. python export.py --weights YOLOv5s.pt --include onnx

This sequence of commands shall result in a **YOLOv5s.onnx** file getting created in the local folder. You can move it to the desired location and use it for the code shown earlier.

Working with v6, v7 and v8

The code for using YOLO v6, v7, and v8 is by-and-large the same as the one shown for using v5. The difference is in the model weights and **coco.names** file. You can download them from Ultralytics official Github repository at **https://github.com/ultralytics/ultralytics**. Alternatively, the files for v7 can be downloaded from **https://github.com/WongKinYiu/yolov7/releases/download/v0.1/yolov7-tiny.pt.** The files for v8 can be downloaded from **https://github.com/JustasBart/yolov8_CPP_Inference_OpenCV_ONNX/blob/minimalistic/source/models/yolov8s.onnx** and **https://github.com/lagadic/visp/blob/master/tutorial/detection/dnn/coco_classes.txt**.

Conclusion

Object detection stands as a cornerstone in the landscape of computer vision, enabling machines to perceive and understand the visual world. From the pioneering days of R-CNN to the real-time capabilities of YOLO and the versatility of SSD, we have witnessed a remarkable evolution in object detection methodologies. As technology advances, the boundaries of detection accuracy and speed continue to expand, fostering innovations in fields ranging from autonomous vehicles to surveillance and beyond. This chapter has offered a glimpse into the foundations and advancements of object detection, serving as a starting point for further exploration and application in the ever-evolving field of computer vision. The next chapter

shall deep dive into another exciting computer vision use case which is recognizing text and faces.

Exercises

1. Perform object detection on input images of varying sizes for each model. Observe the behavior differences.
2. Perform object detection of the same image with various models. Observe the differences in results.
3. Calculate the time taken to perform detection for each model. Observe the accuracy versus speed trade-off.

Chapter 9
Faces and Text

Introduction

In the realm of computer vision, few tasks are as captivating and essential as face and text recognition. In this chapter, we embark on an exploration of the fascinating world of deciphering human faces and text scripts through the lens of technology. We will uncover the intricacies of locating faces and text within images, distinguishing their features, and, ultimately, identifying the individuals they belong to. From fundamental principles to cutting-edge techniques, let us understand how machines perceive faces and alphabets**.**

Structure

The chapter covers the following topics:

- Face detection
- Face recognition
- Text recognition
- OpenCV Model Zoo

Objectives

The objective of this chapter is to understand how to perform face recognition and text recognition, which is also called **optical character**

recognition. We shall also explore the OpenCV Model Zoo and understand how to use the models made available for users.

Face detection

The history of face detection in computer vision spans several decades and has seen significant advancements. Here is a brief overview:

- **Early approaches (1960s-1990s)**: The earliest efforts in face detection involved rule-based methods that relied on simple heuristics and geometric features. These methods often struggled with variations in lighting, pose, and expression. In the 1980s and 1990s, researchers started using techniques like template matching and correlation filters for face detection.
- **Viola-Jones algorithm (2001)**: One of the most significant breakthroughs came with the Viola-Jones algorithm in 2001. *Paul Viola* and *Michael Jones* introduced a real-time face detection algorithm that used Haar-like features and a boosted cascade of simple classifiers. This method was highly efficient and helped pave the way for practical face-detection applications.
- **Feature-based approaches (2000s)**: As computational power increased, researchers began to explore more sophisticated feature-based approaches, such as using **local binary patterns** (**LBP**) and **Histogram of Oriented Gradients** (**HOG**) for detecting faces. These methods improved accuracy and robustness in various conditions.
- **Deep learning revolution (2010s)**: The introduction of deep learning, particularly **convolutional neural networks** (**CNN**s), revolutionized face detection. Convolutional neural networks demonstrated exceptional performance in various computer vision tasks, including face detection. The availability of large labeled datasets like FDDB and WIDER FACE facilitated the training of deep learning models.
- **Region proposal networks and Faster R-CNN (2015):** The Faster R-CNN architecture, introduced by *Shaoqing Ren*, *Kaiming He*, et al., combined **region proposal networks** (**RPN**s) with CNNs,

allowing for accurate and efficient object detection, including faces. This architecture marked a shift from single-shot detectors to more accurate region-based methods.

- **Single shot multibox detector and you inly look once (2016):** SSD and YOLO are two popular single-shot object detection methods that can be applied to face detection. These approaches are known for their speed and ability to detect faces in real-time applications.
- **Cascade CNNs and two-stage detectors (2017-present):** Researchers continued to refine face detection models, introducing cascaded CNN architectures that improved accuracy while maintaining real-time performance. Two-stage detectors like RetinaNet and Mask R-CNN also found applications in face detection, allowing for better localization and segmentation.

The field of face detection in computer vision continues to evolve, driven by advancements in deep learning, hardware, and real-world applications. Researchers are increasingly focused on creating models that are not only accurate but also ethical and respectful of privacy considerations. Ongoing research aims to improve the robustness of face detection algorithms to variations in pose, lighting, occlusion, and facial expressions. Techniques such as data augmentation, domain adaptation, and adversarial training contribute to making face detection systems more reliable in diverse real-world scenarios. With the proliferation of face detection technology, concerns about privacy, surveillance, and bias have emerged. Researchers and practitioners are actively to address these issues by developing fair and privacy-aware face detection methods.

Haar cascades

The Haar cascades algorithm is an integral component of the Viola-Jones face detection method. It is a machine learning-based approach that uses a cascade of simple classifiers to efficiently detect objects, particularly faces, in images. This algorithm significantly speeds up the detection process by focusing on potential positive regions while quickly rejecting non-object regions. Here is how the Haar cascades algorithm works within the context of the Viola-Jones algorithm:

- **Haar-like features**: Haar-like features are simple rectangular filters that compute the difference between the sum of pixel intensities in the white and black regions of the filter. These features capture basic patterns of light and dark areas in an image. Examples of Haar-like features include edge features, line features, and corner features.
- **Integral image**: To efficiently compute the Haar-like features over regions of an image, the concept of an integral image is used. The integral image is a transformed representation of the original image where each pixel stores the sum of all pixels above and to the left of it. This pre-computed information allows for rapid computation of Haar-like features in constant time, regardless of the feature size.
- **Adaptive Boosting (AdaBoost)**: The Viola-Jones algorithm employs AdaBoost, a machine learning technique, to select a small set of highly discriminative Haar-like features. AdaBoost focuses on the most informative features that best separate positive and negative training samples (that is, regions with faces and regions without faces).
- **Cascade of classifiers**: The cascade structure consists of multiple stages, each containing a set of weak classifiers. A weak classifier is a simple decision rule based on the evaluation of a single Haar-like feature. During training, AdaBoost assigns weights to the training samples and iteratively trains weak classifiers. The cascade structure helps reject non-face regions early in the detection process, reducing the number of regions that need to be evaluated by more complex classifiers.
- **Stage-wise classification**: In each stage, the weak classifiers are organized into a cascade, where each classifier produces a decision regarding the presence of a face in the region. The cascade structure allows for rapid rejection of negative regions and focuses computational effort on regions that have a higher likelihood of containing a face.
- **Final decision**: If a region passes through all stages without rejection, it is classified as a face region. The combination of multiple stages and the AdaBoost-trained weak classifiers contributes to accuracy and efficiency in detecting faces.

The Haar cascades algorithm is particularly well-suited for real-time applications due to its ability to quickly eliminate non-object regions and focus computational resources on potential object regions. While it was initially developed for face detection, this approach has also been adapted for detecting other objects in computer vision tasks.

OpenCV provides an excellent implementation to process Haar cascades for detecting objects. It even provides built-in Haar implementations for faces, eyes, lips, and so on. The below code demonstrates how to use these implementation.

```
1. import cv2
2.
3. def detect_face_using_haar(img):
4.
5.     face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
6.
7.     # convert to gray scale of each frames
8.     im_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
9.
10.    # Detects faces of different sizes in the input image
11.    front_faces = face_cascade.detectMultiScale(im_gray, 1.3, 5)
12.
13.    # To draw a rectangle in a face
14.    for (x,y,w,h) in front_faces:
15.        cv2.rectangle(img,(x,y),(x+w,y+h),(255,255,255),5)
```

```
16.
17.     return img
18.
19.
20.
21. # Main script execution
22. if __name__ == "__main__":
23.     image_path = "../input_images/crowd.pxhere.com.jpg"
24.
25.     img = cv2.imread(image_path)
26.     detected_faces = detect_face_using_haar(img)
27.     cv2.namedWindow("face detections", cv2.WINDOW_NORMAL)
28.     cv2.imshow("face detections", detected_faces)
29.     cv2.waitKey(0)
```

The key ingredient here is in the **haarcascade_frontalface_default.xml** file. This file is available as opensource implementation and is installed when OpenCV libraries are installed on the machine. It is located in the Python **venv** location where the **opencv** libraries are installed. A number of other Haar based classifiers are also available in the same location for different uses. This code gives the below output as shown in *Figure 9.1*:

***Figure 9.1**: Frontal face detection using Haar cascades*

As can be seen in the image, not all faces were detected. This is because of the limitations of the Haar cascades. Better solutioning requires deep learning approaches, which we shall see next.

Deep learning approaches: YuNet

YuNet is a tiny, millisecond-level face detector that is based on the MobileNetV2 architecture. It is designed to be fast and efficient while still maintaining a high accuracy. The YuNet face detector has been shown to be more accurate than other state-of-the-art face detectors, such as the YOLOv4 detector.

Here are some of the key features of YuNet:

- It is a tiny model with only 75,856 parameters.
- It is fast and efficient, with an inference speed of 1.6 milliseconds per frame on an Intel i7-12700K CPU. It is accurate, with a mAP of 81.1% on the WIDER FACE validation hard set.

- It is versatile and can be used on a variety of devices, including mobile phones, embedded systems, and edge devices.

YuNet is a powerful tool for face detection. It is ideal for applications such as real-time face recognition, video surveillance, and augmented reality, where speed and accuracy are important. It can be used for applications such as access control, attendance tracking, video surveillance, crowd monitoring, and law enforcement. It can also be used for applications such as virtual try-ons and face filters. YuNet is a good option for a fast, accurate, versatile face detector. Let us see some code for face detection using YuNet:

```
import numpy as np
import cv2

def visualize(input, faces):
    if faces is None:
        return

    if faces[1] is None:
        return

    thickness = 5
    facebox_color = (255, 255, 255)
    eyeline_color = (255, 0, 0)
    nosetip_color = (0, 255, 0)
    mouthline_color = (0, 0, 255)
```

```
16.
17.     # Cycle through the faces and landmarks
18.     for _, face in enumerate(faces[1]):
19.         # Convert the returned coordinates to integers
20.         coordinates = face[:-1].astype(np.int32)
21.
22.         face_box_topleft_x = coordinates[0]
23.         face_box_topleft_y = coordinates[1]
24.         face_box_width = coordinates[2]
25.         foce_box_height = coordinates[3]
26.         face_box_bottomright_x = coordinates[0] + face_box_width
27.         face_box_bottomright_y = coordinates[1] + foce_box_height
28.
29.         righteye_x = coordinates[4]
30.         righteye_y = coordinates[5]
31.
32.         lefteye_x = coordinates[6]
33.         lefteye_y = coordinates[7]
34.
35.         nosetip_x = coordinates[8]
36.         nosetip_y = coordinates[9]
```

```
37.
38.         mouth_rightcorner_x = coordinates[10]
39.         mouth_rightcorner_y = coordinates[11]
40.
41.         mouth_leftcorner_x = coordinates[12]
42.         mouth_leftcorner_y = coordinates[13]
43.
44.         # Draw rectangles, lines, and circles on the input image
45.         cv2.rectangle(input, (face_box_topleft_x, face_box_topleft_y), (face_box_bottomright_x, face_box_bottomright_y), facebox_color, thickness)
46.          cv2.line(input, (righteye_x, righteye_y), (lefteye_x, lefteye_y), eyeline_color, thickness, lineType=cv2.FILLED)  # Eye line
47.         cv2.line(input, (mouth_leftcorner_x, mouth_leftcorner_y), (mouth_rightcorner_x,  mouth_rightcorner_y),  mouthline_color,  thickness, lineType=cv2.LINE_4)  # Mouth line
48.         cv2.circle(input, (nosetip_x, nosetip_y), 1, nosetip_color, thickness)  # Nosetip
49.
50. if __name__ == '__main__':
51.     imgpath = "../input_images/crowd.pxhere.com.jpg"
52.
53.     # Initialize FaceDetectorYN with parameters
```

```
    nms_threshold = 0.3
    score_threshold = 0.5
    yunet_shape = (320, 320)
    topk = 500

    detector = cv2.FaceDetectorYN.create("../weights/9/face_detection_yunet_2023mar.onnx", "", yunet_shape, score_threshold, nms_threshold, topk)

    img = cv2.imread(imgpath)
    detector.setInputSize((img.shape[1], img.shape[0]))
    faces_and_landmarks = detector.detect(img)

    # Call the visualize function to draw on the image
    visualize(img, faces_and_landmarks)

    cv2.namedWindow("face detections", cv2.WINDOW_NORMAL)
    cv2.imshow("face detections", img)
    cv2.waitKey(0)
```

Let us execute this code on the same image used to test Haar cascades-based detector. This code gives the output as shown in *Figure 9.2*:

***Figure 9.2**: Face detection using YuNet*

As can be seen here, YuNet is considerably more successful in detecting faces compared Haar cascades. The anti-aliased line and filled lines seen in the image are additional information that YuNet can detect. They are the eye and lips detected in the face. These features are called landmarks and are crucial in the next challenge which is face recognition.

Face recognition

Face detection and face recognition are two distinct tasks in the field of computer vision, and serve different purposes. Face detection is about finding and localizing faces within an image, while face recognition goes a step further by determining the identity of the detected faces. Face detection is typically the first step in many face recognition systems because you need to locate the faces before you can recognize them. Understanding these distinctions is important when working on applications that involve faces, whether it is for security, entertainment, or other purposes.

Face detection versus recognition

Refer to *Table 9.1* to understand the differences between face detection and face recognition:

	Face detection	Face recognition
Task	The process of identifying the presence of faces within an image.	The process of identifying or verifying a person's identity based on their facial features.
Objective	Determine whether there is a face in the input data and, if so, provide the coordinates or bounding box around the detected face(s).	Determine who the person is by comparing their facial features to a database of known individuals.
Output	A bounding box that surrounds the detected face(s).	The identity of the detected face(s) or a determination of whether the detected face matches a known person.
Use cases	Used in various applications like autofocus in digital cameras, counting the number of people in a crowd, tracking faces in video for security purposes, and more.	Used in applications like unlocking smartphones with facial recognition, identity verification at airports, surveillance systems for identifying persons of interest, and more.
Complexity	It is a relatively simpler task compared to face recognition.	More complex task as it involves creating a feature represe

		ntation of the face a nd then comparing i t to a database of kn own faces to make a n identification.

***Table 9.1**: Difference between face detection and face recognition*

Face recognition using landmarks

Landmarks, also referred to as facial landmarks or facial keypoints, are specific points on a person's face that correspond to distinct anatomical features. These landmarks are often used as key reference points for various computer vision tasks, including face recognition. Here is how landmarks are used in the context of face recognition:

- **Localization and alignment**: Landmarks help locate and align faces within images. By identifying key points such as the corners of the eyes, nose, mouth, and chin, the face can be accurately positioned and oriented. This is particularly useful in scenarios where faces may be tilted, rotated, or vary in scale.
- **Feature extraction**: Once the landmarks are detected, they can be used to define specific **regions of interest** (**ROI**s) on the face. These regions can encompass important facial components like the eyes, nose, and mouth. Extracting features from these ROIs can improve the robustness of face recognition algorithms by focusing on the most informative parts of the face.
- **Normalization**: Facial landmarks enable the normalization of faces across different images. By aligning faces based on landmarks, variations in pose, scale, and rotation can be minimized, creating a more consistent representation for recognition.
- **Data augmentation**: Landmarks facilitate data augmentation techniques. By perturbing the positions of landmarks while keeping the overall facial structure intact, new training examples can be generated. This helps prevent overfitting and improves the generalization of face recognition models.

- **Verification and alignment**: In face verification tasks (determining if two images belong to the same person), landmarks are crucial in aligning faces before comparison. Aligning faces based on landmarks ensures that corresponding facial features are consistently positioned, making the comparison more accurate.
- **Pose estimation**: Facial landmarks can also aid in estimating the pose of a face, such as the yaw, pitch, and roll angles. This information can help adapt face recognition algorithms to different poses and viewpoints.
- **Expression analysis**: Landmarks can provide insights into facial expressions. Changes in the positions of landmarks can be indicative of different facial expressions, which may be considered when recognizing faces under varying emotional states.
- **3D reconstruction**: Some face recognition systems use 3D facial landmarks to assist in creating accurate 3D reconstructions of faces. This can enhance recognition accuracy by considering depth information.

Landmarks are typically detected using techniques such as facial landmark detection models, shape regression methods, or deep learning approaches. They offer a structured and informative representation of facial geometry that can significantly improve the performance and robustness of face recognition systems, especially in challenging and unconstrained scenarios.

OpenCV supports a wide variety of face recognition algorithms. In fact, OpenCV has provided an abstract class named **FaceRecognizer** from which all **FaceRecognizer** classes are inherited. This provides tremendous flexibility to support a wide variety of face recognizer algorithms. We shall briefly cover the **FaceRecogizer** module in the next section.

Face recognizer module

All face recognition models in OpenCV are derived from the abstract base class **FaceRecognizer**. Each **FaceRecognizer** is treated as an algorithm, allowing access to its internal workings through easy **get**/**set** operations if permitted by the implementation. Algorithm, a relatively new concept

introduced in OpenCV since the 2.4 release, offers the following features for all derived classes:

- A virtual constructor, enabling each Algorithm derivative to be registered at the program start. This registration allows you to obtain a list of registered algorithms and create instances of specific algorithms by name (refer to **Algorithm::create**). For those considering adding custom algorithms, it is advisable to use unique prefixes to distinguish them from other algorithms.
- Parameter management through name-based setting/retrieval. This approach resembles the **SetCaptureProperty** and **GetCaptureProperty** functions used in OpenCV's video capturing functionality. The algorithm provides similar methods where parameter names are specified as text strings instead of integer IDs.
- Serialization of parameters to and from XML or YAML files. Every algorithm derivative can store its parameters and retrieve them later, eliminating the need for reimplementing this functionality each time.

Additionally, every **FaceRecognizer** supports the following:

- Training with **FaceRecognizer::train** on a set of images, typically a face database.
- Prediction of a given sample image, usually a face, provided as a **Mat**.
- Loading/saving the model state to/from XML or YAML files.
- Setting/getting label information stored as a string. String labels are useful for associating names with recognized individuals.

Here is the class **inheritence** information of **FaceRecognizer** shown in *Figure 9.3:*

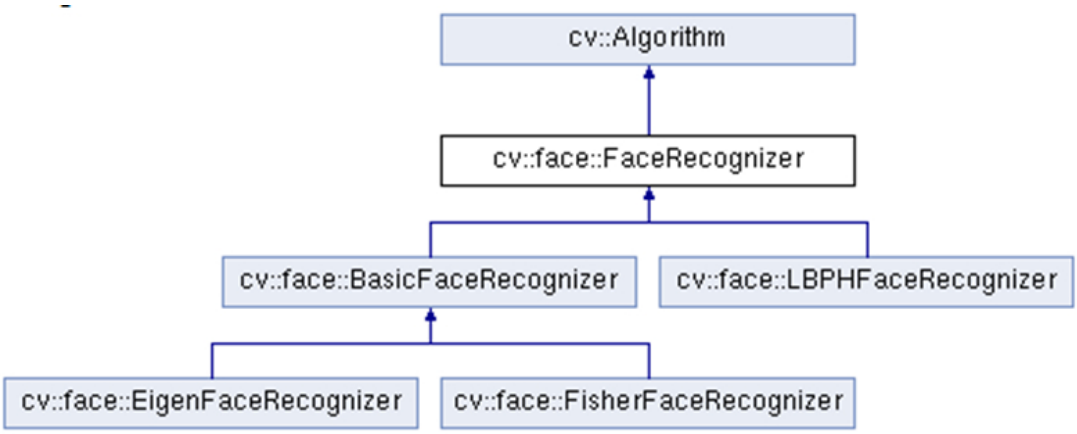


***Figure 9.3**: Inheritance diagram of FaceRecognier class*[1]

It is worth noting that when using the **FaceRecognizer** interface in conjunction with Python, it is advisable to use Python 2 as some underlying scripts like **create_csv** may not work with other versions, such as Python 3.

The **FaceRecognizer** module is a deep learning-based face recognition module that uses a pre-trained ONNX model to detect and recognize faces in images and videos. The module is available in OpenCV 3.3 and later versions. The **FaceRecognizer** module consists of two parts:

- **A face detector:** This part uses the pre-trained Caffe model to detect faces in an image or video.
- **A face recognizer**: This part uses the detected faces to train a face recognition model.

The face detector is a **Single Shot Multibox Detector** (**SSD**) model that is based on the ResNet-10 architecture. The face recognizer is a linear **Support Vector Machine** (**SVM**) model. To use the DNN **FaceRecognizer** module, developers must first download the pre-trained Caffe model from the OpenCV Model Zoo site. Details of downloading this model are available at the end of this chapter. If a high-accuracy face recognition system is needed, then the DNN **FaceRecognizer** module is a good option. It comes with distinct advantages like accuracy and ease of use. However, it is more computationally expensive as well.

For the rest of this chapter, we shall extensively use the **FaceRecognizerSF** module and the **Labeled Faces in the Wild** (**LFW**) dataset. Let us briefly look at both of these modules here.

Labeled Faces in the Wild dataset

The LFW dataset is a widely used benchmark dataset in the field of face recognition. It was created to evaluate and compare the performance of various face recognition algorithms in unconstrained, real-world scenarios. The dataset was compiled by researchers at the *University of Massachusetts, Amherst*. It has been a crucial resource for advancing the state of the art in face recognition. The dataset contains more than 13,000 images of faces of 5000 unique individuals collected from the internet. These images cover a wide range of identities, poses, lighting conditions, and facial expressions with variations in ethnicity, age, and gender. They reflect the diversity of real-world scenarios, making it more challenging for algorithms to recognize faces accurately.

The LFW dataset has been instrumental in advancing the field of face recognition by providing a realistic and challenging testbed for evaluating algorithms' ability to handle unconstrained real-world scenarios. Researchers often use this dataset to showcase advancements and improvements in face recognition algorithms and to understand their limitations. It is important to note that while the LFW dataset has played a significant role, there are now larger and more diverse datasets available to further challenge face recognition algorithms in even more complex scenarios.

The dataset can be downloaded freely from **https://vis-www.cs.umass.edu/lfw/**. Readers are encouraged to visit this site and download the images from here.

FaceRecognizerSF class

The **FaceRecognizerSF** module is a deep learning-based face recognition module that uses a pre-trained ONNX model to detect and recognize faces in images and videos. The class implements the *Sigmoid-Constrained Hypersphere Loss for Robust Face Recognition* algorithm, commonly called **SFace**. SFace is a loss function that is designed to improve the robustness of face recognition models. It does this by minimizing the similarity distance between images of the same person and maximizing it between images of different people.

The SFace loss function has been shown to be effective in improving the robustness of face recognition models to factors such as illumination variations, pose variations, and occlusions. It has been shown to outperform other loss functions, such as the cross-entropy loss function, on various face recognition benchmarks. However, it is also more computationally expensive than other loss functions and requires more training data.

Comparing faces

Let us now use this knowledge to recognize faces. This chapter uses two different photographs of the great Hollywood actor *Clint Eastwood*. Both these photographs are taken from LFW dataset.

```
import numpy as np
import cv2

if __name__ == '__main__':
    # Paths to the input images
    img1path = "../lfw/Clint_Eastwood/Clint_Eastwood_0001.jpg"
    img2path = "../lfw/Clint_Eastwood/Clint_Eastwood_0005.jpg"

    # Initialize FaceDetectorYN with parameters
    nms_threshold = 0.3
    score_threshold = 0.5
    yunet_shape = (320, 320)
    topk = 500

```

```
15.     detector = cv2.FaceDetectorYN.create("../weights/9/face_detection_yunet_2023mar.onnx", "", yunet_shape, score_threshold, nms_threshold, topk)
16.
17.     # Load and detect faces in the first image
18.     img1 = cv2.imread(img1path)
19.     detector.setInputSize((img1.shape[1], img1.shape[0]))
20.     faces_and_landmarks1 = detector.detect(img1)
21.
22.     # Load and detect faces in the second image
23.     img2 = cv2.imread(img2path)
24.     detector.setInputSize((img2.shape[1], img2.shape[0]))
25.     faces_and_landmarks2 = detector.detect(img2)
26.
27.     # Initialize FaceRecognizerSF
28.     recognizer = cv2.FaceRecognizerSF.create("../weights/9/face_recognition_sface_2021dec.onnx", "")
29.
30.     # Align and crop faces from the images
31.     face1_align = recognizer.alignCrop(img1, faces_and_landmarks1[1][0])
32.     face2_align = recognizer.alignCrop(img2, faces_and_landmarks2[1][0])
```

```
33.
34.     # Extract features from the aligned faces
35.     facial_features1 = recognizer.feature(face1_align)
36.     facial_features2 = recognizer.feature(face2_align)
37.
38.     # Set similarity score thresholds
39.     cosine_similarity_threshold = 0.363
40.     l2_similarity_threshold = 1.128
41.
42.     # Calculate cosine and L2 similarity scores
43.     cosine_score = recognizer.match(facial_features1, facial_features2, cv2.FaceRecognizerSF_FR_COSINE)
44.     l2_score = recognizer.match(facial_features1, facial_features2, cv2.FaceRecognizerSF_FR_NORM_L2)
45.
46.     # Determine if the images belong to the same person based on similarity scores
47.     if (cosine_score >= cosine_similarity_threshold) or (l2_score <= l2_similarity_threshold):
48.         print("Images are of the same person")
49.     else:
50.         print("Images do not belong to the same person")
```

Figure 9.4 shows the two figures used. As can be imagined, the above program identifies the two images to belong to the same person.

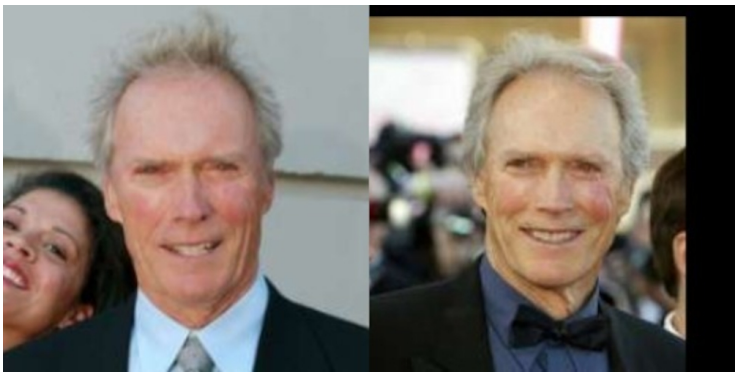


***Figure 9.4**: Images of Clint Eastwood from LFW dataset*

What is interesting is, the algorithm is just as capable of detecting the faces of people who are not celebrities. For example, *Figure 9.5* are images of a person captured nearly two years apart. However, the algorithm still identifies them to belong to the same person. This is incredibly useful for automated face recognition for security and identification uses.

***Figure 9.5**: Images detected to belong to the same person*

Text recognition

Text detection and recognition are essential components of computer vision with various applications, including document analysis, scene understanding, and augmented reality. Text detection involves locating and identifying textual content within images or scenes. The primary goal is to pinpoint the regions or bounding boxes where text is present. Text detection is a crucial preprocessing step for text recognition. Text detection methods range from traditional computer vision algorithms like edge detection and connected component analysis to more advanced deep learning-based approaches using CNNs and RPNs. Challenges in text detection include handling multi-oriented text, detecting text in complex backgrounds, and differentiating between text and non-text regions. Text detection is used in document scanning, license plate recognition, augmented reality, and more.

Text recognition, also known as **Optical Character Recognition** (**OCR**), is the process of converting detected text within an image into machine-readable and editable text. The objective is to understand and transcribe the textual content accurately. OCR can be performed using traditional techniques, such as template matching and pattern recognition, but modern OCR systems often rely on deep learning models, including **recurrent neural networks** (**RNN**s) and **convolutional-recurrent neural networks** (**CRNN**s). Challenges in text recognition include handling variations in fonts, sizes, languages, and distortions due to perspective, lighting, or noise. Text recognition finds applications in digitizing printed documents, extracting information from images, enabling text-to-speech synthesis, and automating data entry.

Combining text detection and recognition allows computer vision systems to locate and decipher text within images, making textual information accessible for further processing and analysis. These technologies are crucial in many industries, from finance and healthcare to automotive and robotics, where understanding and extracting information from visual content is essential.

Text detection

Let us now see some code for detecting text from an image:

```
import numpy as np
import cv2

def visualize(image, results):
    box_color=(255, 255, 255)
    isClosed=True
    thickness=10
```

```
    pts = np.array(results[0])
    return cv2.polylines(image, pts, isClosed, box_color, thickness)


def initialize_model(model_shape):
    backend_id = cv2.dnn.DNN_BACKEND_OPENCV
    target_id = cv2.dnn.DNN_TARGET_CPU

    binary_threshold = 0.3
    polygon_threshold = 0.5
    max_candidates = 200
    unclip_ratio = 2.0

    model_path = "../weights/9/text_detection_DB_TD500_resnet18_2021sep.onnx"
    model = cv2.dnn_TextDetectionModel_DB(cv2.dnn.readNet(model_path))

    model.setPreferableBackend(backend_id)
    model.setPreferableTarget(target_id)

    model.setBinaryThreshold(binary_threshold)
```

```
    model.setPolygonThreshold(polygon_threshold)
    model.setUnclipRatio(unclip_ratio)
    model.setMaxCandidates(max_candidates)

    model.setInputParams(1.0/255.0, model_shape, (122.67891434, 116.66876762, 104.00698793))
    return model

if __name__ == '__main__':

    image_path = "../input_images/4_EdgesCorners.jpg"
    model_shape = (736, 736)  # w, h

    model = initialize_model(model_shape)

    original_image = cv2.imread(image_path)
    original_h, original_w, _  = original_image.shape
    scaleHeight = original_h / model_shape[1]
    scaleWidth = original_w / model_shape[0]
    image = cv2.resize(original_image, model_shape)

```

```
    # Inference
    results = model.detect(image)

    # Scale the results bounding box
    for i in range(len(results[0])):
        for j in range(4):
            box = results[0][i][j]
            results[0][i][j][0] = box[0] * scaleWidth
            results[0][i][j][1] = box[1] * scaleHeight

    # Draw results on the input image
    original_image = visualize(original_image, results)

    cv2.namedWindow("input", cv2.WINDOW_NORMAL)
    cv2.imshow("input", original_image)
    cv2.waitKey(0)
```

This code generates the following output:

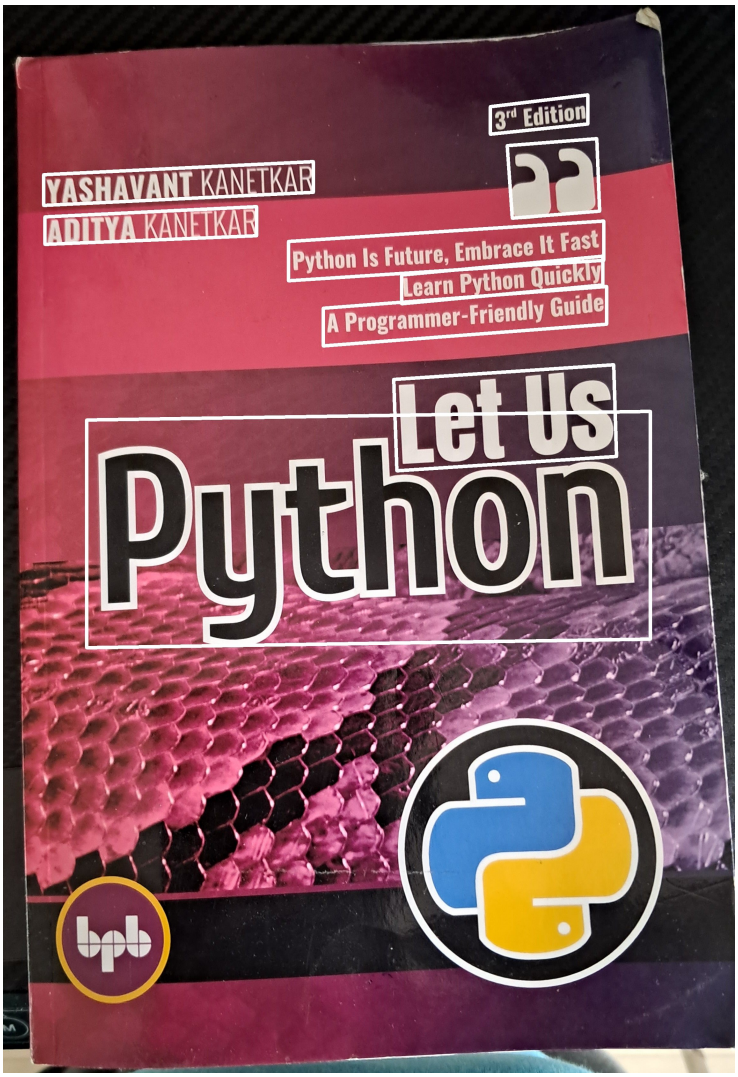


***Figure 9.6**: Detected text is highlighted in white rectangle*

Text recognition

Let us now go one step further and try identifying the text in the image. We start exactly where we left. We take the specific locations in the image which have been identified as text and then detect them.

However, we now need to pre-determine the language of the text. This is a tricky requirement for situations like driverless cars. Other than this constraint, the implementation works reasonably well. In this chapter, we are limiting ourselves to English language. So, we shall use the **text_recognition_CRNN_EN_2021sep.onnx** model from OpenCV Model Zoo in the following code:

```
import numpy as np
import cv2

backend_id = cv2.dnn.DNN_BACKEND_OPENCV
target_id = cv2.dnn.DNN_TARGET_CPU


def initialize_textdetector_model(model_shape):
    binary_threshold = 0.3
    polygon_threshold = 0.5
    max_candidates = 200
    unclip_ratio = 2.0

    model_path = "../weights/9/text_detection_DB_TD500_resnet18_2021sep.onnx"
    model = cv2.dnn_TextDetectionModel_DB(cv2.dnn.readNet(model_path))

    model.setPreferableBackend(backend_id)
    model.setPreferableTarget(target_id)

    model.setBinaryThreshold(binary_threshold)
```

```
    model.setPolygonThreshold(polygon_threshold)
    model.setUnclipRatio(unclip_ratio)
    model.setMaxCandidates(max_candidates)

    model.setInputParams(1.0/255.0, model_shape, (122.67891434, 116.66876762, 104.00698793))
    return model

def initialize_english_textrecognition_model():
    model_path = "../weights/9/text_recognition_CRNN_EN_2021sep.onnx"
    model = cv2.dnn.readNet(model_path)
    model.setPreferableBackend(backend_id)
    model.setPreferableTarget(target_id)
    character_set = '0123456789abcdefghijklmnopqrstuvwxyz'
    character_size = (100, 32)  # This must not be changed and must be in sync with next line
    vertex_coordinates = np.array([
                        [0, 31],
                        [0, 0],
                        [99, 0],
                        [99, 31]
```

```
                    ],
                    dtype=np.float32)
    return model, character_set, character_size, vertex_coordinates

def visualize(image, boxes, texts, color=(0, 255, 0), isClosed=True, thickness=2):
    pts = np.array(boxes[0])
    output = cv2.polylines(image, pts, isClosed, color, thickness)
    for box, text in zip(boxes[0], texts):
        print(text)
        cv2.putText(output, text, (box[1].astype(np.int32)), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255))
    return output

def recognize_text(model, character_set, character_size, vertex_coordinates, image, boxshape):
    # Preprocess the image

    # Remove conf, reshape and ensure all is np.float32
    vertices = boxshape.reshape((4, 2)).astype(np.float32)
    rotationMatrix = cv2.getPerspectiveTransform(vertices, vertex_coordinates)
```

```
58.     cropped_image = cv2.warpPerspective(image, rotationMatrix, character_size)
59.     cropped_image = cv2.cvtColor(cropped_image, cv2.COLOR_BGR2GRAY)
60.     text_blob = cv2.dnn.blobFromImage(cropped_image, size=character_size, mean=127.5, scalefactor=1 / 127.5)
61.
62.
63.
64.     # Forward
65.     model.setInput(text_blob)
66.     output_blob = model.forward()
67.
68.     # Postprocess
69.     text = ''
70.     for i in range(output_blob.shape[0]):
71.         c = np.argmax(output_blob[i][0])
72.         if c != 0:
73.             text += character_set[c - 1]
74.         else:
75.             text += '-'
76.
```

```
    # return text
    # adjacent same letters as well as background text must be removed to get the final output
    char_list = []
    for i in range(len(text)):
        if text[i] != '-' and (not (i > 0 and text[i] == text[i - 1])):
            char_list.append(text[i])

    return ''.join(char_list)

if __name__ == '__main__':
    image_path = "../input_images/4_EdgesCorners.jpg"
    model_shape = (736, 736)  # w, h

    # initialize text detection model
    detector = initialize_textdetector_model(model_shape)



    # initialize CRNN for text recognition
    recognizer, character_set, character_size, vertex_coordinates = initialize_english_textrecognition_model()
```

```
97.
98.     original_image = cv2.imread(image_path)
99.     original_h, original_w, _  = original_image.shape
100.    scaleHeight = original_h / model_shape[1]
101.    scaleWidth = original_w / model_shape[0]
102.    image = cv2.resize(original_image, model_shape)
103.
104.    # Detect the locations of text
105.    results = detector.detect(image)
106.
107.
108.    # Recognize text in the detected locations
109.    texts = []
110.    for box, score in zip(results[0], results[1]):
111.        text = recognize_text(recognizer, character_set, character_size, v
ertex_coordinates, image, box.reshape(8))
112.        texts.append(text)
113.
114.    # Scale the results bounding box
115.    for i in range(len(results[0])):
116.        for j in range(4):
```

```
            box = results[0][i][j]
            results[0][i][j][0] = box[0] * scaleWidth
            results[0][i][j][1] = box[1] * scaleHeight

    # Draw results on the input image
    original_image = visualize(original_image, results, texts)

    # Visualize results in a new window
    cv2.namedWindow("input", cv2.WINDOW_NORMAL)
    cv2.imshow("input", original_image)
    cv2.waitKey(0)
```

Executing this code generates the following output:

```
D:\bpb\995\9>python text_recognition.py
python
letus
pgcrihiaialls
leampythonguickly
ciomistimibinectisd
adityakanetkar
washanvantcanetar
```

9. 22
10. 3rdedition

As can be seen, the output is reasonably good, although not perfect. Obtaining a perfect output requires tweaking the model parameters to suit the individual usecases.

OpenCV Model Zoo

The OpenCV Model Zoo is a valuable resource for computer vision practitioners and researchers. It is a collection of pre-trained models and model weights that can be used with OpenCV. Model Zoo contains a wide range of models designed for different computer vision tasks, including object detection, image classification, face recognition, text detection, and more. These models are trained on large datasets and are often state-of-the-art in terms of accuracy and performance. Many of the models in the OpenCV Model Zoo are compatible with popular deep learning frameworks like TensorFlow, PyTorch, and Caffe.

Using pre-trained models from the Model Zoo is straightforward. OpenCV provides convenient APIs for loading these models and applying them to images or videos. Model Zoo is a collaborative effort, and contributions from the computer vision community are welcome. Developers can find models trained on various datasets and for specific use cases, which can be incredibly valuable for their projects. By leveraging pre-trained models from the Model Zoo, developers can significantly reduce the time and computational resources required for training deep learning models from scratch. Researchers and developers often use these pre-trained models as a starting point for their own experiments or projects. It allows for rapid prototyping and experimentation before committing to training custom models. Whether object recognition, image segmentation, or any other vision-related task, the Model Zoo likely has a pre-trained model that can jumpstart the work.

All the models used in this chapter have been downloaded from Model Zoo. Readers are encouraged to visit the Model Zoo at **https://github.com/opencv/opencv_zoo** and download the models from

there. Being opensource contributions, the models are likely to be upgraded to state-of-the-art implementations frequently.

Conclusion

In this chapter we have visited the foundational concepts of face detection to the intricacies of face recognition. We explored the mechanisms that underpin these essential computer vision tasks. We have also seen the techniques for detecting text from images. In a world increasingly reliant on facial biometrics and human-machine interaction, the knowledge gained here serves as a vital foundation for unlocking countless applications in security, entertainment, and beyond.

Exercises

1. Visit the OpenCV Model Zoo at **https://github.com/opencv/opencv_zoo/tree/main** and try using the other ONNX models provided for face recognition and character recognition.

Key terms

- Face detection: A computer vision use case to determine presence of human faces in an image.
- Haar cascades: An algorithm for rapid detection of objects in an image.
- Viola Jones algorithm: Algorithm that uses Haar cascades for face detection.
- YuNet: A deep learning algorithm built using Mobilenet for face detection.
- Facial Landmarks: Key points of a human face used for various computer vision tasks.
- LFW dataset: Labeled Faces in the Wild dataset containing famces of celebrities under various lighting conditions.
- Text detection: A computer vision use case to determine presence of text in an image.

- Text recognition: A computer vision use case to identify the text alphabets, numbers, special symbols in the image.
- OCR: Optical Character Recognition. Another name for text recognition.
- OpenCV Model Zoo: An opensource repository of popular models for solving computer vision usecases.

---

1 Source: OpenCV’s official documentation page

Chapter 10
Running the Code

Introduction

So far, all the chapters have referred extensively to code. All the provided code has multiple dependencies. This chapter aims to help the readers run this code on their machines. All the code given in this book so far have certain dependencies and need some setup to be done. For example, Python runtime environment is a prerequisite. Libraries like OpenCV, numpy and the like are required to run the program. Moreover, there is a question of fetching the code itself. All these are explained in a complete detail in this chapter.

Structure

The chapter includes the following topics:

- Sequence of steps
- Setting up Anaconda
- Installing Git
- Setting up Python environment
- Fetching the code
- Installing the libraries

- Running the code

Objectives

This chapter aims to enable the readers to set up the Python environment, fetch the code mentioned in the book, and execute it on their machines. Step-by-step instructions are provided for each activity, along with an explanation for the same. The instructions are applicable for both Windows and Linux environments. They have been tested on Windows 10 and on Ubuntu Linux 18.04. If the steps involved have to differ based on OS, then such distinctions have been clearly stated.

Sequence of steps

This guide is provided for people who are not familiar with Python and the options available for installing Python. There are multiple ways to do so. Experienced developers can skip the initial steps and proceed to *Fetching the code* step. Inexperienced developers are recommended to follow the below sequence of steps. An internet connection is required for executing all these steps:

1. Install Anaconda
2. Install Git
3. Install Python virtual environment
4. Fetch the code
5. Installing the libraries
6. Running the code

Among the above, *steps 1-5* are a one-time activity on any given machine. Only *steps 1* and *2* vary between Windows and Linux.

Setting up Anaconda

Anaconda is a popular open-source platform that is widely used in the fields of data science, machine learning, and scientific computing. It comes with a powerful package manager called **Conda**, which makes it easy to install, update, and manage libraries and dependencies. Conda handles not only Python packages but also packages for other programming languages,

making it versatile. Anaconda is available for Windows, macOS, and Linux, ensuring that data scientists and developers can work seamlessly across different operating systems. It includes a Python distribution that is optimized for data science and scientific computing. It comes with many pre-installed packages commonly used in these fields.

A big advantage of Anaconda is that it allows developers to create isolated Python environments, making it easy to manage different project dependencies and avoid conflicts between packages. This is crucial when working on multiple projects with different requirements. Anaconda also integrates seamlessly with Jupyter Notebook, a popular web-based interactive computing environment. Jupyter Notebooks are widely used for data exploration, analysis, and visualization. Additionally, Anaconda supports various IDEs, including Anaconda Navigator (a graphical interface for managing environments and packages), JupyterLab, and integration with popular text editors like Visual Studio Code.

Anaconda makes it easier to share and reproduce data science projects. Developers can export environment configurations to ensure that collaborators or others can recreate their environment precisely. This is the main reason for choosing Anaconda for this book.

Installing Anaconda on Windows

Administrator access is required for these steps. Please follow the instructions in the same sequence:

1. **Download Anaconda:** Visit the Anaconda website. (**https://www.anaconda.com/products/distribution**) and download the Anaconda Distribution for Windows. Choose the version that matches your system architecture (32-bit or 64-bit).
2. **Run the installer**: Locate the downloaded Anaconda installer executable (usually named something like **Anaconda3-<version>-Windows-x86_64.exe**) and double-click it to run the installer.
3. **Read and accept the license agreement**: Carefully read the license agreement and click **I Agree** if you accept the terms.
4. **Select installation type**: Choose **Just Me** (recommended) or **All Users** to specify who can use Anaconda.

5. **Choose installation location**: Select the directory where you want Anaconda to be installed. By default, it is installed in your **user profile** directory.
6. **Add Anaconda to PATH**: To make Anaconda's Python and other utilities accessible from the Command Prompt, check the box that says **Add Anaconda to my PATH environment variable**.
7. **Install Anaconda**: Click the **Install** button to start the installation process. This may take a few minutes.
8. **Installation complete**: Once the installation is finished, you will see a **Successful Installation** message. Click **Next**.
9. **Start Anaconda Navigator**: To launch Anaconda Navigator, a graphical interface for managing packages and environments, select the **Anaconda Navigator** option and click **Finish**.
10. **Test your installation**: Open a Command Prompt and type the following command. If Anaconda is installed correctly, it will display the version number:

```
conda --version
```

Installing Anaconda on Ubuntu Linux

Administrator access is required for these steps. Please follow the instructions in the same sequence:

1. **Download Anaconda:** Open your web browser and visit the Anaconda website (**https://www.anaconda.com/products/distribution**). Download the Anaconda Distribution for Linux. Choose the version that matches your system architecture (usually 64-bit).
2. **Open Terminal**: Open a Terminal window on your Linux machine. You can usually do this by pressing *Ctrl + Alt + T* or searching for **Terminal** in your application launcher.
3. **Navigate to the Downloads directory**: Use the **cd** command to navigate to the directory where you downloaded the Anaconda

installer. For example, if it is in the **Downloads** directory, you can use **cd ~/Downloads**.

4. **Run the installer script**: Use the following command to make the Anaconda installer script executable. Replace **<version>** with the version number you downloaded.

```
chmod +x Anaconda3-<version>-Linux-x86_64.sh
```

Execute the installer script using the following command:

```
./Anaconda3-<version>-Linux-x86_64.sh
```

5. **Follow the installation prompts**: Read the license agreement, type **yes** to accept, and follow the on-screen prompts to choose the installation location (usually in your **Home** directory) and whether to add Anaconda to your **PATH**. It is highly recommended to add Anaconda to **PATH** variable.
6. **Initialize Anaconda**: After installation, you may need to initialize Anaconda by running:

```
source ~/.bashrc
```

7. **Test your installation**: In the Terminal, type **conda --version**. If Anaconda is installed correctly, it will display the version number.

Installing Git

GitHub is a web-based platform for version control and collaborative software development. It allows individuals and teams to manage and track changes to their code repositories. Git, on the other hand, is a distributed version control system that GitHub is built upon. Git enables developers to track changes, collaborate on projects, and maintain a history of their codebase. Users can commit changes to their local Git repositories, create branches for parallel development, and merge changes seamlessly. GitHub extends Git's functionality by providing hosting services for Git repositories, issue tracking, code review, and project management tools. It is widely used for open-source and private software development, making it a fundamental platform for collaborative coding and version control. All the code provided in this book is available in the official BPB GitHub

repository **https://github.com/username/repository.git**. Git software can be used to download this code. If Git is not installed on your computer, developers need to install it.

Installing Git on Windows

Here are step-by-step instructions to install Git on a Windows 10 computer:

1. **Download Git for Windows**: Open your web browser and go to the official Git website: **https://git-scm.com/download/win**.

   This link will automatically detect your system and provide the appropriate download option for Windows.

2. **Download the installer**: Click on the **Download** button to start downloading the Git for Windows installer. The installer should be named something like **Git-2.x.x-64-bit.exe**, where **x.x** represents the version number.

3. **Run the installer**: Locate the downloaded installer file and double-click it to run it.

4. **Welcome screen**: You will see a welcome screen. Click the **Next** button to proceed.

5. **Select destination location**: Choose the destination where Git will be installed. The default location is usually fine for most users. Click **Next** to continue.

6. **Select components**: On this screen, you can choose additional components to install. The default selections are typically sufficient. Click **Next** to proceed.

7. **Choosing an editor (Optional)**: You may be asked to choose a default text editor for Git (for example, Notepad or Visual Studio Code). You can leave the default editor selected or choose a different one. Click **Next**.

8. **Adjusting PATH environment**: On this screen, select the option **Use Git from the Windows Command Prompt** to make Git accessible from the Command Prompt and PowerShell. This is recommended for ease of use. Click **Next**.

9. **Choosing HTTPS transport backend**: Select the option **Use the OpenSSL library** (the default choice) for secure HTTPS connections. Click **Next**.
10. **Configuring line endings**: Choose how you want Git to handle line endings. The default option, **Checkout Windows-style, commit Unix-style line endings**, is suitable for most projects. Click **Next**.
11. **Configuring the terminal emulator**: Choose your preferred terminal emulator. The default option, **Use the Windows' default console window**, is recommended. Click **Next**.
12. **Installing Git**: Click the **Install** button to start the installation process. Git will be installed on your system.
13. **Completing the Git**: Setup After the installation is complete, you will see a **Completing the Git Setup** screen. Ensure the **Launch Git Bash** option is selected and click **Finish**.
14. **Verify Git installation**: Open the **Git Bash** application from your **Start** menu or by searching for **Git Bash** in the Windows search bar. In the Git Bash terminal, you can verify that Git is installed by running the following command:

```
git --version
```

You should see the installed Git version displayed.

Installing Git on Ubuntu

Here are step-by-step instructions to install Git on an Ubuntu-based Linux distribution:

1. **Open Terminal**: Open a Terminal window on your Ubuntu system. You can do this by pressing *Ctrl + Alt + T* or searching for **Terminal** in your application launcher.
2. **Update package lists**: Before installing Git, it is a good practice to update your system's package lists to ensure you are installing the latest available version of Git. Run the following command:

```
sudo apt update
```

3. **Install Git**: To install Git, use the following command:

```
sudo apt install git
```

4. **Confirm installation**: After entering the installation command, the terminal will ask for your confirmation. Type **Y** and press *Enter* to proceed with the installation.
5. **Verify Git installation**: To verify that Git has been successfully installed, you can run the following command to check the installed version:

```
git --version
```

This should display the installed Git version.

Setting up Python environment

Here is a step-by-step guide to creating a **conda** environment with Python 3.8.16. These steps are same for both Windows and Ubuntu:

1. **Open Terminal or Command Prompt**: Open your **Terminal** or **Command Prompt** on your computer.
2. **Create a new conda environment**: Use the following command to create a new **conda** environment and specify Python 3.8.16 as the version:

```
conda create -n bpb995 python=3.8.16
```

   This will create a virtual environment with the name **bpb995** on your machine.

3. **Activate the new environment**: To use the newly created environment, activate it using the following command:

```
conda activate bpb995
```

4. **Verify Python version**: To ensure that Python 3.8.16 is installed in the activated environment, you can check the Python version using:

```
python --version
```

It should display Python 3.8.16.

Now, you have a **conda** environment with Python 3.8.16 installed. You can use this environment for all the Python code in this book.

Fetching the code

Fetching the code in this book is a two step process. First step is to download the programming code and the second step is to download the model weights files used.

Downloading the code

To fetch code from a GitHub repository, you need to clone it to your local machine. Run the below commands at the terminal. Use the **git clone** command followed by the repository's URL. Replace the text **<githuburl>** with the Github URL of the code mentioned earlier in the book.

1. conda activate bpb995
2. git clone **<githuburl>**

A progress bar is displayed to indicate the download in progress. At the end, a folder named **githuburl** is created in the folder where the command was executed.

Fetch the weights

To fetch the model weights file, please visit the Google drive URL mentioned earlier in the book and download the zip file to your local machine. Please unzip the file and move the contents along with the subdirectories to the weights folder created in the code fetched in the step above. This will set up the code ready for execution.

Installing the libraries

Use the **cd** command to navigate into the directory created when you cloned the repository. You can install the Python library dependencies listed in the **requirements.txt** file using the **pip** package manager:

1. cd repository
2. pip install -r requirements.txt

This command instructs **pip** to read the **requirements.txt** file and install the specified libraries and their versions into your virtual environment. You can verify that the dependencies were installed successfully by running:

```
pip list
```

This will display a list of installed packages, including the ones from the **requirements.txt** file.

Running the code

This step has to be executed every time the code has to be run. The virtual environment has to be activated every time a new terminal window is launched to run the code. IDEs like Visual Code have some additional help for launching the virtual environment automatically. However, discussing the IDE functionalities is beyond the scope of this book:

1. conda activate bpb995
2. cd repository

Now, the code can be executed at the terminal. As an example, the file **pixels_and_colour_images.py** from *Chapter 2* is executed here.

1. conda activate bpb995
2. cd repository
3. python 2\pixels_and_colour_images.py

If you can see the output as described in *Chapter 2* then your setup procedure is successful.

Notes:

- **It is recommended to run the code either from code editors like PyCharm, VS Code, Spyder and the like. The code should be run from a terminal command prompt as shown in the book or by appropriate configuration of the above-mentioned code editors.**

- **Configuring the code editors for running the code is beyond the scope of this book.**
- **It is not recommended to the run the code in Jupyter Notebooks as-is. Code statements like cv2.imshow() do not perform satisfactorily in Jupyter Notebooks and can result in system freeze and similar other problems.**

Conclusion

In this chapter, we have seen how to install the software required for running the Python code provided in this book. We have seen step-by-step instructions on installing the software and running the code. Following these steps in the exact sequence is highly recommended for novice developers. Experienced developers can follow different approaches which they are familiar and comfortable. Being able to run the code in this chapter is vitally important to achieve the objectives outlined in the beginning of the book. In the next chapter, we shall see an end-to-end code implementation for a computer vision use case. Starting from a simplified UI to using a combination of computer vision models, we will peruse code for performing automatic number plate recognition.

Exercises

1. Move the model weights files to a different folder. Try modifying the code in the book to read the models from the new locations.
2. Try the code with different images and see how the results change.

Chapter 11
End-to-end Demo

Introduction

So far, we discussed different use cases in computer vision and the modern deep learning and AI-based solutions for addressing them. In this chapter, we will combine them and build a single end-to-end use case. The different models discussed in the book so far will all be combined into a program to address one business requirement, which is recognizing number plates of vehicles.

Structure

The chapter covers the following topics:

- Code
- Running the code
- Application design
- Notes about codes

Objectives

This chapter provides code for running the usecase. The objective is to show how different computer vision models can be harnessed together to achieve a business goal using OpenCV DNN. The code in this chapter

provides a simple user interface that captures the video frames from a camera attached to the computer and executes the models. The program can detect COCO objects and also detects text and number plates in the video frame.

Code

The code is split into multiple files. The files are written in a pythonic manner with complete documentation and can be executed from the command line. The screenshots of the UI are provided as a reference.

main_app.py

The below code is for the main entry point to the code:

```
import tkinter as tk
import video_app_ui as mainui
import image_processor as mainproc

def main():
    # Create a Tkinter root window
    root = tk.Tk()

    # Initialize the ImageProcessor object with required filenames and parameters
    mainprocessor = mainproc.ImageProcessor(
        "weights/8/yolo/YOLOv5s.onnx",  # Object detection model file
        "weights/8/yolo/coco.names",   # Object detection class labels
```

```
        "weights/9/text_detection_DB_TD500_resnet18_2021sep.onnx",  # Text detection model file
        "weights/9/text_recognition_CRNN_EN_2021sep.onnx",  # Text recognition model file
        confidence_threshold=0.8  # Confidence threshold for object detection
    )

    # Create the VideoAppUI using the Tkinter root and ImageProcessor instance
    app = mainui.VideoAppUI(root, mainprocessor)

    # Start the Tkinter main event loop
    root.mainloop()

if __name__ == "__main__":
    main()
```

video_app_ui.py

Below code contains the Tkinter based GUI components:

```
import tkinter as tk
from tkinter import messagebox
import tkinter.font as tkFont
import cv2
```

```
import numpy as np
import threading
import time

class VideoAppUI:
    def __init__(self, root, improcessor):
        self.__initialize_processing_widgets(root)
        self.__initialize_processing_variables(improcessor)

    def __initialize_processing_widgets(self, root):
        self.__root = root

        # Setting title
        root.title("OpenCV DNN End-to-end Demo")

        # Setting window size
        width = 320
        height = 240
        screenwidth = root.winfo_screenwidth()
        screenheight = root.winfo_screenheight()
```

```
25.         alignstr = '%dx%d+%d+%d' % (width, height,
    (screenwidth - width) / 2, (screenheight - height) / 2)
26.         root.geometry(alignstr)
27.         root.resizable(width=False, height=False)
28.
29.         # Create Start Video button
30.         self.__btn_start_video = tk.Button(root)
31.         self.__btn_start_video["anchor"] = "w"
32.         self.__btn_start_video["bg"] = "#f0f0f0"
33.         ft = tkFont.Font(family='Times', size=10)
34.         self.__btn_start_video["font"] = ft
35.         self.__btn_start_video["fg"] = "#000000"
36.         self.__btn_start_video["justify"] = "center"
37.         self.__btn_start_video["text"] = "Start Video"
38.         self.__btn_start_video.place(x=30, y=50, width=70, height=25)
39.         self.__btn_start_video["command"] = self.__start_video
40.
41.         # Create Stop Video button
42.         self.__btn_stop_video = tk.Button(root)
43.         self.__btn_stop_video["bg"] = "#f0f0f0"
44.         ft = tkFont.Font(family='Times', size=10)
```

```
45.         self.__btn_stop_video["font"] = ft
46.         self.__btn_stop_video["fg"] = "#000000"
47.         self.__btn_stop_video["justify"] = "center"
48.         self.__btn_stop_video["text"] = "Stop Video"
49.         self.__btn_stop_video.place(x=30, y=100, width=70, height=25)
50.         self.__btn_stop_video["command"] = self.__stop_video
51.         self.__btn_stop_video["state"] = tk.DISABLED
52.
53.         # Create Process Video checkbox
54.         self.__process_video_var = tk.BooleanVar()
55.         self.__chk_process_video = tk.Checkbutton(root, variable=self.__process_video_var)
56.         self.__chk_process_video["anchor"] = "w"
57.         ft = tkFont.Font(family='Times', size=10)
58.         self.__chk_process_video["font"] = ft
59.         self.__chk_process_video["fg"] = "#333333"
60.         self.__chk_process_video["justify"] = "center"
61.         self.__chk_process_video["text"] = "Process video"
62.         self.__chk_process_video.place(x=0, y=170, width=150, height=30)
63.         self.__chk_process_video["offvalue"] = False
64.         self.__chk_process_video["onvalue"] = True
```

```
65.         self.__chk_process_video["command"] = self.__chk_process_video_command
66.
67.         # Create Detect Objects checkbox
68.         self.__detect_objects_var = tk.BooleanVar()
69.         self.__chk_detect_objects = tk.Checkbutton(root, variable=self.__detect_objects_var)
70.         ft = tkFont.Font(family='Times', size=10)
71.         self.__chk_detect_objects["font"] = ft
72.         self.__chk_detect_objects["fg"] = "#333333"
73.         self.__chk_detect_objects["justify"] = "center"
74.         self.__chk_detect_objects["text"] = "Detect objects"
75.         self.__chk_detect_objects.place(x=10, y=210, width=135, height=30)
76.         self.__chk_detect_objects["offvalue"] = False
77.         self.__chk_detect_objects["onvalue"] = True
78.         self.__chk_detect_objects["state"] = tk.DISABLED
79.
80.         # Create Detect Number Plate checkbox
81.         self.__detect_number_plates_var = tk.BooleanVar()
82.         self.__chk_detect_numberplate = tk.Checkbutton(root, variable=self.__detect_number_plates_var)
83.         ft = tkFont.Font(family='Times', size=10)
```

```
84.         self.__chk_detect_numberplate["font"] = ft
85.         self.__chk_detect_numberplate["fg"] = "#333333"
86.         self.__chk_detect_numberplate["justify"] = "center"
87.         self.__chk_detect_numberplate["text"] = "Detect Number Plate"
88.          self.__chk_detect_numberplate.place(x=10, y=250, width=175,
    height=30)
89.         self.__chk_detect_numberplate["offvalue"] = False
90.         self.__chk_detect_numberplate["onvalue"] = True
91.         self.__chk_detect_numberplate["state"] = tk.DISABLED
92.
93.         self.__btn_start_video.pack(pady=5)
94.         self.__btn_stop_video.pack(pady=5)
95.         self.__chk_process_video.pack(pady=5)
96.         self.__chk_detect_objects.pack()
97.         self.__chk_detect_numberplate.pack()
98.
99.     def __initialize_processing_variables(self, improcessor):
100.         self.__improcessor = improcessor
101.         self.__number_of_skipframes = None
102.         self.__video_capture = None
103.         self.__video_thread = None
```

```
104.
105.     def __start_video(self):
106.         self.__video_capture = cv2.VideoCapture(0)
107.         self.__btn_stop_video["state"] = tk.NORMAL
108.         self.__btn_start_video["state"] = tk.DISABLED
109.         self.__video_thread = threading.Thread(target=self.__process_vi
deo)
110.         self.__video_thread.daemon = True
111.         self.__video_thread.start()
112.
113.     def __stop_video(self):
114.         if self.__video_capture:
115.             self.__video_capture.release()
116.             self.__video_capture = None  # This will break the infinite loo
p in __process_video
117.             self.__video_thread.join()
118.             self.__btn_stop_video["state"] = tk.DISABLED
119.             self.__btn_start_video["state"] = tk.NORMAL
120.
121.     def __chk_process_video_command(self):
122.         process_video = self.__process_video_var.get()
123.
```

```
124.         if process_video:
125.             self.__chk_detect_objects["state"] = tk.NORMAL
126.             self.__chk_detect_numberplate["state"] = tk.NORMAL
127.         else:
128.             self.__chk_detect_objects["state"] = tk.DISABLED
129.             self.__chk_detect_numberplate["state"] = tk.DISABLED
130.
131.     def __process_video(self):
132.         if not self.__video_capture:
133.             messagebox.showerror("Error", "Video is not running. Start th
    e video first.")
134.             return
135.
136.         cv2.namedWindow("Video", cv2.WINDOW_FULLSCREEN)
137.         while True:
138.             process_video = self.__process_video_var.get()
139.             detect_objects = self.__detect_objects_var.get()
140.              detect_number_plates = self.__detect_number_plates_var.get(
    )
141.
142.             if self.__video_capture:
143.                 ret, frame = self.__video_capture.read()
```

```
144.            else:
145.                ret = False
146.
147.            if not ret:
148.                break
149.
150.            if process_video:
151.                if detect_objects:
152.                    processed_frame = self.__detect_objects(frame)
153.                else:
154.                    processed_frame = frame
155.
156.                if detect_number_plates:
157.                    processed_frame = self.__detect_number_plate(processe
d_frame)
158.            else:
159.                processed_frame = frame
160.
161.            # Display the processed frame
162.            cv2.imshow("Video", processed_frame)
163.            cv2.waitKey(1)
```

```
164.
165.         cv2.destroyAllWindows()
166.
167.     def __detect_objects(self, frame):
168.         # Implement YOLOv5 object detection here and mark
    objects in the frame
169.         # You'll need to use YOLOv5 and its weights for this part
170.         # Example code for YOLOv5 detection:
171.         # result_frame = yolov5_detection(frame)
172.
173.         result_frame = self.__improcessor.detect_objects(frame)
174.         return result_frame
175.
176.     def __detect_number_plate(self, frame):
177.         result_frame = self.__improcessor.detect_numberplate(frame)
178.         return result_frame
179.
180.     def run(self):
181.         self.__root.mainloop()
182.
```

image_processor.py

The below code processes the image processing logic for the images. This code relies on the individual model processing files and does not handle the models by itself:

```
import cv2
import object_detector as detector
import numberplate_recognizor as numplaterecog

class ImageProcessor:
    def __init__(self, object_detection_model_file, labels_file,
                 textdetection_model_file, textrecognition_model_file,
                 confidence_threshold):
        """
        Initialize the ImageProcessor object.

        Args:
            object_detection_model_file (str): File path to the object detection model.
            labels_file (str): File path to the labels file.
            textdetection_model_file (str): File path to the text detection model.
            textrecognition_model_file (str): File path to the text recognition model.
            confidence_threshold (float): Confidence threshold for object detection.
```

```
18.
19.         Raises:
20.             FileNotFoundError: If any of the provided file paths do not ex
    ist.
21.         """
22.         # Initialize member objects, e.g., load models or configure settin
    gs
23.
24.         # Create an ObjectDetector instance for object detection
25.         self.__object_detection_model = detector.ObjectDetector(
26.             object_detection_model_file=object_detection_model_file,
27.             class_labels_file=labels_file,
28.             confidence_threshold=confidence_threshold
29.         )
30.
31.         # Create a NumberPlateRecognizor instance for
    number plate recognition
32.         self.__numberplate_detection_model = numplaterecog.NumberP
    lateRecognizor(
33.             textdetection_model_file, textrecognition_model_file
34.         )
35.
36.     def detect_objects(self, image):
```

```
37.         """
38.         Detect objects in an input image.
39.
40.         Args:
41.             image (numpy.ndarray): An OpenCV image object.
42.
43.         Returns:
44.             numpy.ndarray: An image with objects marked.
45.
46.         Raises:
47.             Exception: If an error occurs during object detection.
48.         """
49.         try:
50.             # Perform object detection using self.__object_detection_mod
    el
51.             retimage = self.__object_detection_model.detect_objects(ima
    ge)
52.
53.         except Exception as e:
54.             print(f"Error in detect_objects: {str(e)}")
55.             retimage = image  # Return the original
    image in case of an error
```

```
56.
57.         return retimage
58.
59.     def detect_numberplate(self, image):
60.         """
61.         Detect number plates in an input image.
62.
63.         Args:
64.             image (numpy.ndarray): An OpenCV image object.
65.
66.         Returns:
67.             str: Recognized number plate text.
68.
69.         Raises:
70.             Exception: If an error occurs during number plate detection.
71.         """
72.         try:
73.             # Perform number plate detection using self.__numberplate_detection_model
74.             return self.__numberplate_detection_model.detect_numberplate(image)
75.
```

```
76.         except Exception as e:
77.             print(f"Error in detect_numberplate: {str(e)}")
78.             return None  # Return None in case of an error
```

numberplate_recognizor.py

This code deals with the text recognition model. The responsibilities of working with different technical requirements of the model are abstracted by this file:

```
1. import cv2
2. import numpy as np
3. 
4. class NumberPlateRecognizor:
5. 
6.     # Constants for the model and image processing
7.     __MODEL_SHAPE = (736, 736)  # Model input shape (width, hei
   ght)
8.     __BACKEND_ID = cv2.dnn.DNN_BACKEND_OPENCV
9.     __TARGET_ID = cv2.dnn.DNN_TARGET_CPU
10. 
11.     def __init__(self, textdetection_model_file, textrecognition_model
   _file):
12.         """
13.         Initialize the NumberPlateRecognizor object.
14. 
```

```
15.         Args:
16.             textdetection_model_file (str): File path to the text detection model.
17.             textrecognition_model_file (str): File path to the text recogniti on model.
18.
19.         Raises:
20.             FileNotFoundError: If any of the provided file paths do not exist.
21.         """
22.
23.         # Initialize member objects, e.g., load models or configure settings
24.
25.         # Initialize the text detection model
26.         self.__detector_model = self.__initialize_textdetector_model(tex tdetection_model_file)
27.
28.         # Initialize CRNN for text recognition
29.         self.__recognizer, self.__character_set, self.__character_size, sel f.__vertex_coordinates = self.__initialize_english_textrecognition_m odel(textrecognition_model_file)
30.
31.     def __initialize_textdetector_model(self, model_path):
```

```
        # Constants for text detection parameters
        binary_threshold = 0.3
        polygon_threshold = 0.5
        max_candidates = 200
        unclip_ratio = 2.0

        # Create a text detection model
        model = cv2.dnn_TextDetectionModel_DB(cv2.dnn.readNet(model_path))

        model.setPreferableBackend(self.__BACKEND_ID)
        model.setPreferableTarget(self.__TARGET_ID)

        model.setBinaryThreshold(binary_threshold)
        model.setPolygonThreshold(polygon_threshold)
        model.setUnclipRatio(unclip_ratio)
        model.setMaxCandidates(max_candidates)

        model.setInputParams(1.0/255.0, self.__MODEL_SHAPE, (122.67891434, 116.66876762, 104.00698793))
        return model

```

```
52.     def __initialize_english_textrecognition_model(self, model_path):
53.         # Create a text recognition model
54.         model = cv2.dnn.readNet(model_path)
55.         model.setPreferableBackend(self.__BACKEND_ID)
56.         model.setPreferableTarget(self.__TARGET_ID)
57.
58.         # Define character set and size
59.         character_set = '0123456789abcdefghijklmnopqrstuvwxyz'
60.         character_size = (100, 32)  # This must not be changed and must be in sync with next line
61.         vertex_coordinates = np.array([
62.             [0, 31],
63.             [0, 0],
64.             [99, 0],
65.             [99, 31]
66.         ],
67.         dtype=np.float32)
68.
69.         return model, character_set, character_size, vertex_coordinates
70.
71.     def __recognize_text(self, image, boxshape):
```

```
72.         # Preprocess the image
73.         vertices = boxshape.reshape((4, 2)).astype(np.float32)
74.          rotationMatrix = cv2.getPerspectiveTransform(vertices, self.__v
    ertex_coordinates)
75.         cropped_image = cv2.warpPerspective(image, rotationMatrix, se
    lf.__character_size)
76.          cropped_image = cv2.cvtColor(cropped_image, cv2.COLOR_B
    GR2GRAY)
77.         text_blob = cv2.dnn.blobFromImage(cropped_image, size=self._
    _character_size, mean=127.5, scalefactor=1 / 127.5)
78.
79.         # Forward pass
80.         self.__recognizer.setInput(text_blob)
81.         output_blob = self.__recognizer.forward()
82.
83.         # Postprocess the recognized text
84.         text = ''
85.         for i in range(output_blob.shape[0]):
86.             c = np.argmax(output_blob[i][0])
87.             if c != 0:
88.                 text += self.__character_set[c - 1]
89.             else:
90.                 text += '-'
```

```
91.
92.         # Return processed text
93.         char_list = []
94.         for i in range(len(text)):
95.             if text[i] != '-' and (not (i > 0 and text[i] == text[i - 1])):
96.                 char_list.append(text[i])
97.
98.         return ''.join(char_list)
99.
100.    def __visualize(self, image, boxes, texts):
101.        # Visualize the recognized text on the image
102.        color = (255, 255, 255)
103.        isClosed = True
104.        thickness = 2
105.        pts = np.array(boxes[0])
106.        output = cv2.polylines(image, pts, isClosed, color, thickness)
107.        for box, text in zip(boxes[0], texts):
108.            cv2.putText(output, text, (box[1].astype(np.int32)), cv2.FONT
    _HERSHEY_SIMPLEX, 0.5, (255, 255, 255))
109.        return output
110.
```

```
111.     def detect_numberplate(self, original_image):
112.         """
113.         Detect number plates in an input image.
114.
115.         Args:
116.             original_image (numpy.ndarray): An OpenCV image object.
117.
118.         Returns:
119.             numpy.ndarray: An image with number plates marked.
120.
121.         Raises:
122.             ValueError: If the provided image is not a valid numpy.ndarra
    y.
123.         """
124.         try:
125.             # Ensure the image is a valid numpy.ndarray
126.             if not isinstance(original_image, np.ndarray):
127.                 raise ValueError("Input image is not a
    valid numpy.ndarray.")
128.
129.             # Get the original image dimensions
130.             original_h, original_w, _ = original_image.shape
```

```
131.         scaleHeight = original_h / self.__MODEL_SHAPE[1]
132.         scaleWidth = original_w / self.__MODEL_SHAPE[0]
133.
134.         # Resize the image to the model's input shape
135.         image = cv2.resize(original_image, self.__MODEL_SHAPE)
136.
137.         # Detect the locations of text in the resized image
138.         results = self.__detector_model.detect(image)
139.
140.         # Recognize text in the detected locations
141.         texts = []
142.         for box, score in zip(results[0], results[1]):
143.             text = self.__recognize_text(image, box.reshape(8))
144.             texts.append(text)
145.
146.         # Scale the results bounding box back to the
    original image dimensions
147.         for i in range(len(results[0])):
148.             for j in range(4):
149.                 box = results[0][i][j]
150.                 results[0][i][j][0] = box[0] * scaleWidth
```

```
                    results[0][i][j][1] = box[1] * scaleHeight

            # Draw results on the original input image
             original_image = self.__visualize(original_image, results, texts)
            return original_image

        except Exception as e:
            print(f"Error in detect_numberplate: {str(e)}")
```

object_detector.py

This code deals with the object detection model. The responsibilities of working with different technical requirements of the model are abstracted by this class:

```
import cv2
import time
import numpy as np

class ObjectDetector:
    def __init__(self, object_detection_model_file, class_labels_file, confidence_threshold):
        """
        Initialize the ObjectDetector object.

```

```
        Args:
            object_detection_model_file (str): File path to the object detection model.
            class_labels_file (str): File path to the class labels file.
            confidence_threshold (float): Confidence threshold for object detection.

        Raises:
            FileNotFoundError: If the provided file paths do not exist.
        """
        # Check if the provided files exist
        if not all(map(lambda f: cv2.os.path.exists(f), [object_detection_model_file, class_labels_file])):
            raise FileNotFoundError("One or more provided file paths do not exist for object detection model.")

        # Initialize member objects. This should happen once per program execution to avoid repeated disk reads
        self.__model = cv2.dnn.readNet(object_detection_model_file)
        self.__classes = self.__load_labels(class_labels_file)
        self.__confidence_threshold = confidence_threshold

        # Get all the layer names of the model
```

```
28.         self.__layer_names = self.__model.getLayerNames()
29.         # Filter and choose only the output layers
30.          self.__output_layers = [self.__layer_names[i - 1] for i in self.__
    model.getUnconnectedOutLayers()]
31.
32.     def __load_labels(self, labels_file):
33.         try:
34.             # Check if the labels file exists
35.             if not cv2.os.path.exists(labels_file):
36.                 raise FileNotFoundError(f"Labels file '{labels_file}' does n
    ot exist.")
37.
38.             # Read the COCO class names
39.             with open(labels_file, 'r') as file:
40.                 lines = file.readlines()
41.                 classes = [line.strip() for line in lines]
42.
43.             return classes
44.
45.         except Exception as e:
46.             print(f"Error in load_labels: {str(e)}")
47.
```

```
48.     def __process_detection(self, image, yolo_shape, detection):
49.         height, width, channels = image.shape
50.         x_scaling = width / yolo_shape[0]
51.         y_scaling = height / yolo_shape[1]
52.
53.         center_x = detection[0]
54.         center_y = detection[1]
55.         object_width = detection[2]
56.         object_height = detection[3]
57.
58.         # Rectangle coordinates
59.         topleft_x = int(x_scaling * (center_x - object_width / 2))
60.         topleft_y = int(y_scaling * (center_y - object_height / 2))
61.
62.         object_width = int(x_scaling * object_width)
63.         object_height = int(y_scaling * object_height)
64.
65.         return (topleft_x, topleft_y, object_width, object_height)
66.
67.     def detect_objects(self, original_image):
68.         """
```

```
69.         Detect objects in an input image.
70.
71.         Args:
72.             original_image (numpy.ndarray): An OpenCV image object.
73.
74.         Returns:
75.             numpy.ndarray: An image with objects marked.
76.
77.         Raises:
78.             ValueError: If the provided image is not a valid numpy.ndarra
    y.
79.         """
80.         scaling_factor = 1 / 255
81.         nms_threshold = 0.1
82.         yolo_shape = (640, 640)
83.
84.         try:
85.             # Ensure the image is a valid numpy.ndarray
86.             if not isinstance(original_image, np.ndarray):
87.                 raise ValueError("Input image is not a valid numpy.ndarray.
    ")
88.
```

```
89.             image = np.copy(original_image)
90.
91.             # Perform object detection
92.              blob = cv2.dnn.blobFromImage(image, scaling_factor, yolo_s
    hape, (0, 0, 0), 1, crop=False)
93.
94.             # Detecting objects
95.             self.__model.setInput(blob)
96.             results = self.__model.forward(self.__output_layers)
97.
98.             # Initialize lists to store object information
99.             object_classes = []
100.            object_confidences = []
101.            object_coordinates = []
102.
103.            number_of_detections = results[0].shape[1]
104.            for inx in range(number_of_detections):
105.                one_detection = results[0][0][inx]
106.
107.                # Each detection is a 1D array with the
    following format:
```

```
108.                 # [x, y, width, height, confidence_score_for_class_0, confid
     ence_score_for_class_1, ...]
109.             # We extract the class with the highest
     confidence score.
110.             confidence_scores_for_classes = one_detection[5:]
111.              classid_with_highest_confidence = np.argmax(confidence_
     scores_for_classes)
112.              class_confidence = confidence_scores_for_classes[classid_
     with_highest_confidence]
113.
114.             if class_confidence > self.__confidence_threshold:
115.                 object_location = self.__process_detection(image, yolo_s
     hape, one_detection)
116.                 object_coordinates.append(object_location)
117.                 object_confidences.append(float(class_confidence))
118.                 object_classes.append(self.__classes[classid_with_highes
     t_confidence])
119.
120.         # Apply Non-Maximum Suppression (NMS) to
     remove overlapping bounding boxes
121.          indexes = cv2.dnn.NMSBoxes(object_coordinates, object_con
     fidences, self.__confidence_threshold, nms_threshold)
122.
123.         # Prepare the final list of objects and their
     coordinates
```

```
124.        objects_and_locations = []
125.        for inx in indexes:
126.            class_label = object_classes[inx]
127.            (x, y, width, height) = object_coordinates[inx]
128.            top_left_coordinate = (x, y)
129.            bottom_right_coordinate = (x + width, y + height)
130.
131.            one_object = {}
132.            one_object["class"] = class_label
133.            one_object["top_left"] = top_left_coordinate
134.            one_object["bottom_right"] = bottom_right_coordinate
135.            one_object["confidence"] = object_confidences[inx]
136.            objects_and_locations.append(one_object)
137.
138.        # Draw bounding boxes and class
    labels on the original image
139.        for one_object in objects_and_locations:
140.            cv2.rectangle(original_image, one_object["top_left"], one_
    object["bottom_right"], (255, 255, 255), 3)
141.            cv2.putText(original_image, one_object["class"], one_objec
    t["top_left"], cv2.FONT_HERSHEY_SIMPLEX, 1,
142.                    (255, 255, 255), 3)
```

```
143.
144.             result_image = original_image
145.
146.         except Exception as e:
147.             print(f"Error in detect_objects: {str(e)}")
148.             result_image = None
149.
150.         return result_image
```

Running the code

This code can be run by using the below command:

```
python main_app.py
```

This will show the window as shown in *Figure 11.1*:

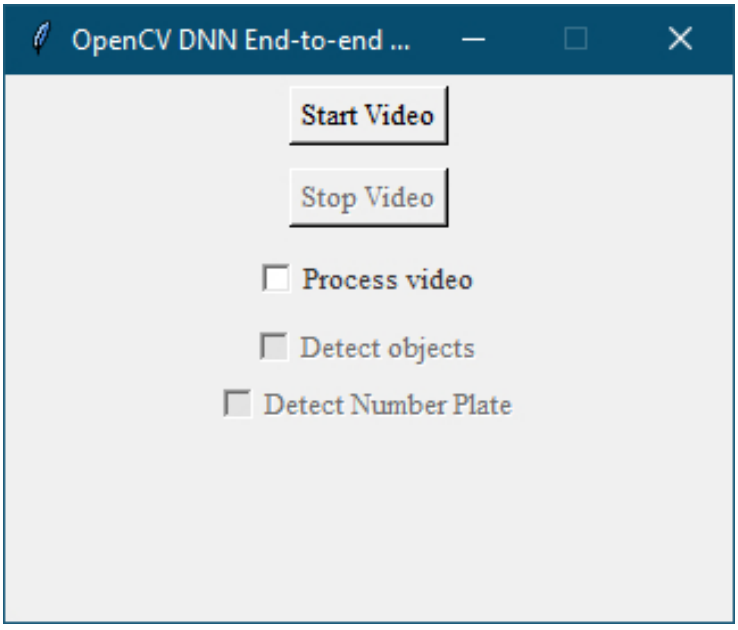


***Figure 11.1**: UI of the demo application*

Application design

This application is created using object-oriented approach in Python. The classes in this application and their purpose are explained in the below table.

File	Class	Purpose
numberplate_recognizor.py	**NumberPlate Recognizor**	This class loads the models for detecting and recognizing text. This class is the same code as shown in *Chapter 9*. The code in *Chapter 9* is refactored to an object-oriented design. This class is implemented specifically for the models discussed in *Chapter 11*. The paths of the model files are accepted as input arguments.
object_detector.py	**ObjectDetector**	This class loads the models for objects using YOLOv5. This class is the same code as shown in *Chapter 8*. The code in *Chapter 8* is refactored to an object-oriented design. This class is implemented specifically for the YOLOv5 ONNX model discussed i

		n *Chapter 8*. The paths of the model file and class labels file are accepted as input arguments.
image_processor.py	**ImageProcessor**	This class encapsulates the **NumberPlateRecognizor** and **ObjectDetector** objects and initializes them. Any additional business logic processing can be included here. This class is unconcerned with the complexities of processing the individual models. This helps in isolating business logic from the model specific complexities. This simplifies future enhancements for the application.
video_app_ui.py	**VideoAppUI**	This is the main the user interface class. It creates and manages the different UI widgets and user interactions.
main_app.py	-	Main entry point to the application.

Notes about codes

Please note the below points about the code provided in this chapter.

- Clicking the **Start Video** button shall start camera feed processing and shows the feed in a separate window.
- **Stop Video** button can be clicked to stop the feed.
- Checking the **Process Video** checkbox shall enable **Detect Objects** and **Detect Numberplate** check boxes.
- **Detect Objects** shall make use of the YOLOv5 object detection model. **Detect Numberplate** shall use the text detection and recognition models. These two check boxes can be individually selected or can be combined.
- Application occasionally freezes upon clicking **Stop Video** button. This is because of the sub-optimal thread synchronization code in the UI. This was a deliberate choice to keep the code simple and understandable for early learners.

Conclusion

In this chapter, we embarked on a journey into the world of computer vision by building an end-to-end application that showcases the power and versatility of this field. We explored the intricacies of image and video processing using the OpenCV library, and we integrated deep learning models to detect objects and recognize text within these visual data streams.

Our exploration began with the initialization of the fundamental building blocks of our application. Through the Tkinter library, we created an intuitive interface that allows users to interact with the application seamlessly. We used the YOLOv5 model for detecting objects within the camera feed. We extended our application's capabilities using CRNN for text detection and recognition. We showcased how computer vision can be used for tasks such as license plate recognition or text extraction from images.

Throughout our journey, we emphasized the importance of clean code and best practices in software development. Our application adheres to these principles, ensuring readability, and maintainability. By combining the power of GUIs, deep learning, and real-time video processing, we have opened the door to countless possibilities in areas such as surveillance, image analysis, and automation. As you continue your exploration of

computer vision, remember that this chapter is just the beginning of your exciting journey into the world of visual perception and intelligence.

Exercises

1. Try using different models in the code instead of the one provided for you. For example, try using YOLOv7 instead of v5.
2. Try using other use cases and models. For example, use ResNet model for image classification and classify each frame.
3. If you have an NVIDIA GPU on your machine, try modifying the compute flags in the code and make the inference run on the GPU. Observe the performance difference between running on CPU versus GPU.

Index

Symbols

A

B

C

- CNN, for object detection
  - activation functions
  - anchor boxes
  - architecture
  - backpropagation and optimization
  - convolutional blocks
  - convolutional layers
  - fully connected (FC) layers
  - localization and classification heads
  - loss functions
  - pooling layers
  - post-processing
- CNNs, for classification
  - activation function
  - convolutional layers
  - fully connected layers
  - input layer
  - loss function
  - output layers
  - pooling layers
- code
  - downloading
  - fetching
  - image_processor.py
  - main_app.py
  - notes

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y