

Applied Deep Learning and Computer Vision for Self-Driving Cars

Build autonomous vehicles using deep neural networks and behavior-cloning techniques



Packt

www.packt.com

Sumit Ranjan and Dr. S. Senthamilarasu

Applied Deep Learning and Computer Vision for Self-Driving Cars

Build autonomous vehicles using deep neural networks and behavior-cloning techniques

**Sumit Ranjan
Dr. S. Senthamilarasu**

Packt

BIRMINGHAM - MUMBAI

Applied Deep Learning and Computer Vision for Self-Driving Cars

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Sunith Shetty

Acquisition Editor: Girish Shet

Content Development Editor: Nathanya Dias

Senior Editor: Ayaan Hoda

Technical Editor: Sonam Pandey

Copy Editor: Safis Editing

Project Coordinator: Aishwarya Mohan

Proofreader: Safis Editing

Indexer: Manju Arasan

Production Designer: Shankar Kalbhor

First published: August 2020

Production reference: 1130820

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83864-630-1

www.packtpub.com



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.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 Packt books and eBooks.

Contributors

About the authors

Sumit Ranjan was a silver medalist for his Bachelor of Technology (Electronics and Telecommunication) degree. He is a passionate data scientist who has worked on solving business problems to build an unparalleled customer experience across domains such as automobiles, healthcare, semi-conductors, cloud virtualization, and insurance.

He is experienced in building applied machine learning, computer vision, and deep learning solutions to meet real-world needs. He was awarded Autonomous Self-Driving Car Scholar by KPIT Technologies. He has also worked on multiple research projects at Mercedes Benz Research and Development. Apart from work, his hobbies are traveling and exploring new places, wildlife photography, and blogging.

Writing a book is harder than I thought, and more rewarding than I could ever imagine. First, I would like to thank everyone at Packt Publishing who extended their support to me. Special thanks to Nathanya and Ayaan, the ever-patient editors, Girish a great Acquisition Editor who onboarded me, Utkarsha and Sonam who helped me improve the technical aspects of the book, and Gebin, the greatest Project manager. A big thanks to my parents, my siblings Amit and Anamika for always being there for me, and even to all my Gurus, Sikandar, Ashis, and Arjun due to whom I have reached here. This book would not have been possible without my amazing friends Divya, Pramod, and Ranjeet who made my life awesome, thanks for all the support you gave me while writing this book.

Dr. S. Senthamilarasu was born and raised in Coimbatore, Tamil Nadu. He is a technologist, designer, speaker, storyteller, journal reviewer, educator, and researcher. He loves to learn new technologies and solves real-world problems in the IT industry. He has published various journals and research papers and has presented at various international conferences. His research areas include data mining, image processing, and neural networks.

He loves reading Tamil novels and involves himself in social activities. He has also received silver medals at international exhibitions for his research products for children with an autism disorder. He currently lives in Bangalore and is working closely with lead clients.

First, I want to thank God without who I would be unable to do all of this. Writing a book is harder than I thought, and more satisfying than I ever expected. Without my mate Sumit, none of this would be possible. During every war and all my successes, he stood by me and wrote the book. I would like to express my special gratitude to the members of my family, who gave me this golden opportunity to write this wonderful book, which has enabled me to learn so many new things. Last but not the least, I would like to thank everyone in the publishing team, without their constant support this book would have not been possible.

About the reviewers

Ashis Roy is a data scientist working at the network infrastructure company Ericsson, in machine learning and artificial intelligence. Previously, he worked with a pure-play data analytics firm, Mu Sigma, and has worked with many Fortune 500 clients in BFSI, retail, and the telecom domain. He completed a master's in mathematics and computing from IIT Guwahati, India and a professional doctorate in engineering from TU/e, the Netherlands, in industrial mathematics. Currently, he is developing a self-healing network using AI and ML. Besides that, he has a deep interest in the sunrise areas of industries.

Dr. B. Muthukumaraswamy has a PhD in data science. He is a technical evangelist who is a result-driven software engineer with over 12 years of continuous experience dealing with all facets of AI, deep learning, and machine learning. He has worked with the most popular algorithms used in the field of deep learning today, such as **Convolutional Neural Networks (CNNs)**, isomorphic JavaScript, and the **object-oriented JavaScript (OOJS)** software development life cycle in world-class JavaScript-based systems. He has been architecting and working with React, Angular, and React Native. He writes a lot of code both for work and personally. He also works with the MEAN stack, the MERN stack, and the JAMstack, and has authored a couple of books.

Akshay GovindRao Mankar has more than 3 years of experience in the domain of computer vision and deep learning. Currently, he is working in the domain of self-driving cars in Germany as a senior software engineer. Before getting into the industry, he completed an M.Sc in electronics and an M.Tech in communication engineering from Mangalore University and Vellore Institute of Technology (V.I.T. University) respectively. During these master's programs, his research subject of interest was image processing and computer vision. Akshay is very passionate about artificial intelligence and has experience with the Python programming language, OpenCV, TensorFlow, and Keras. In his free time, he likes to go through online courses.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight 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.

Table of Contents

Preface	1
<hr/>	
Section 1: Deep Learning Foundation and SDC Basics	
<hr/>	
Chapter 1: The Foundation of Self-Driving Cars	7
Introduction to SDCs	8
Benefits of SDCs	12
Advancements in SDCs	13
Challenges in current deployments	15
Building safe systems	17
The cheapest computer and hardware	18
Software programming	24
Fast internet	24
Levels of autonomy	24
Level 0 – manual cars	25
Level 1 – driver support	25
Level 2 – partial automation	25
Level 3 – conditional automation	25
Level 4 – high automation	26
Level 5 – complete automation	26
Deep learning and computer vision approaches for SDCs	26
LIDAR and computer vision for SDC vision	28
Summary	29
Chapter 2: Dive Deep into Deep Neural Networks	30
Diving deep into neural networks	31
Introduction to neurons	33
Understanding neurons and perceptrons	35
The workings of ANNs	39
Understanding activation functions	40
The threshold function	40
The sigmoid function	41
The rectifier linear function	42
The hyperbolic tangent activation function	42
The cost function of neural networks	43
Optimizers	44
Understanding hyperparameters	46
Model training-specific hyperparameters	47
Learning rate	47

Batch size	49
Number of epochs	49
Network architecture-specific hyperparameters	50
Number of hidden layers	50
Regularization	50
L1 and L2 regularization	51
Dropout	51
Activation functions as hyperparameters	52
TensorFlow versus Keras	52
Summary	53
Chapter 3: Implementing a Deep Learning Model Using Keras	54
Starting work with Keras	55
Advantages of Keras	56
The working principle behind Keras	56
Building Keras models	58
The sequential model	58
The functional model	59
Types of Keras execution	60
Keras for deep learning	61
Building your first deep learning model	64
Description of the Auto-Mpg dataset	64
Importing the data	66
Splitting the data	68
Standardizing the data	68
Building and compiling the model	69
Training the model	70
Predicting new, unseen data	74
Evaluating the model's performance	75
Saving and loading models	76
Summary	76
Section 2: Deep Learning and Computer Vision Techniques for SDC	
Chapter 4: Computer Vision for Self-Driving Cars	79
Introduction to computer vision	80
Challenges in computer vision	81
Artificial eyes versus human eyes	83
Building blocks of an image	84
Digital representation of an image	84
Converting images from RGB to grayscale	87
Road-marking detection	90
Detection with the grayscale image	90
Detection with the RGB image	92
Challenges in color selection techniques	94
Color space techniques	94

Introducing the RGB space	95
HSV space	96
Color space manipulation	97
Introduction to convolution	107
Sharpening and blurring	110
Edge detection and gradient calculation	118
Introducing Sobel	119
Introducing the Laplacian edge detector	120
Canny edge detection	121
Image transformation	126
Affine transformation	127
Projective transformation	128
Image rotation	128
Image translation	130
Image resizing	133
Perspective transformation	135
Cropping, dilating, and eroding an image	139
Masking regions of interest	143
The Hough transform	146
Summary	153
Chapter 5: Finding Road Markings Using OpenCV	154
Finding road markings in an image	154
Loading the image using OpenCV	155
Converting the image into grayscale	156
Smoothing the image	157
Canny edge detection	158
Masking the region of interest	159
Applying bitwise_and	161
Applying the Hough transform	163
Optimizing the detected road markings	167
Detecting road markings in a video	170
Summary	172
Chapter 6: Improving the Image Classifier with CNN	173
Images in computer format	173
The need for CNNs	174
The intuition behind CNNs	175
Introducing CNNs	176
Why 3D layers?	176
Understanding the convolution layer	177
Depth, stride, and padding	181
Depth	181
Stride	181
Zero-padding	183
ReLU	184

Fully connected layers	185
The softmax function	185
Introduction to handwritten digit recognition	185
Problem and aim	186
Loading the data	187
Reshaping the data	190
The transformation of data	190
One-hot encoding the output	191
Building and compiling our model	192
Compiling the model	193
Training the model	194
Validation versus train loss	195
Validation versus test accuracy	196
Saving the model	197
Visualizing the model architecture	197
Confusion matrix	199
The accuracy report	201
Summary	202
Chapter 7: Road Sign Detection Using Deep Learning	203
Dataset overview	204
Dataset structure	204
Image format	204
Loading the data	205
Image exploration	207
Data preparation	208
Model training	210
Model accuracy	211
Summary	216

Section 3: Semantic Segmentation for Self-Driving Cars

Chapter 8: The Principles and Foundations of Semantic Segmentation	218
Introduction to semantic segmentation	220
Understanding the semantic segmentation architecture	221
Overview of different semantic segmentation architectures	222
U-Net	223
SegNet	224
Encoder	225
Decoder	226
PSPNet	226
DeepLabv3+	227
E-Net	229
Summary	231

Chapter 9: Implementing Semantic Segmentation	232
Semantic segmentation in images	233
Semantic segmentation in videos	238
Summary	242
Section 4: Advanced Implementations	
Chapter 10: Behavioral Cloning Using Deep Learning	244
Neural network for regression	245
Behavior cloning using deep learning	249
Data collection	249
Data preparation	255
Model development	268
Evaluating the simulator	271
Summary	274
Chapter 11: Vehicle Detection Using OpenCV and Deep Learning	275
What makes YOLO different?	277
The YOLO loss function	277
The YOLO architecture	278
Fast YOLO	279
YOLO v2	279
YOLO v3	280
Implementation of YOLO object detection	280
Importing the libraries	281
Processing the image function	281
The get class function	282
Draw box function	282
Detect image function	283
Detect video function	284
Importing YOLO	285
Detecting objects in images	285
Detecting objects in videos	287
Summary	288
Chapter 12: Next Steps	289
SDC sensors	293
Camera	294
RADAR	294
Ultrasonic sensors	295
Odometric sensors	295
LIDAR	295
Introduction to sensor fusion	295
Kalman filter	296
Summary	297

Table of Contents

Other Books You May Enjoy	299
Index	302

Preface

With **self-driving cars (SDCs)** being an emerging subject in the field of artificial intelligence, data scientists have now focused their interest on building autonomous cars. This book is a comprehensive guide to using deep learning and computer vision techniques to develop SDCs.

The book starts by covering the basics of SDCs and deep neural network techniques that are required to get up and running with building your autonomous car. Once you are comfortable with the basics, you'll learn how to implement the convolution neural network. As you advance, you'll use deep learning methods to perform a variety of tasks such as finding lane lines, improving the image classifier, and road sign detection. Furthermore, you'll delve into the basic structure and workings of a semantic segmentation model, and even get to grips with detecting cars using semantic segmentation. The book also covers advanced applications such as behavior cloning and vehicle detection using OpenCV and advance deep learning methodologies.

By the end of this book, you'll have learned how to implement various neural networks to develop autonomous vehicle solutions using modern libraries from the Python environment.

Who this book is for

This book is for you if you're a deep learning engineer, AI engineer, or anyone looking to implement or start with deep learning and computer vision techniques to build self-driving blueprint solutions. This book will also help you solve real-world problems using the Python ecosystem. Some Python programming experience and a basic understanding of deep learning are expected.

What this book covers

Chapter 1, *The Foundation of Self-Driving Cars*, talks about the history and evolution of SDCs. It briefs you on different approaches used in SDCs. It also covers details about the advantages and disadvantages of SDCs, the challenges in creating them, as well as the levels of autonomy of an SDC.

Chapter 2, *Dive Deep into Deep Neural Networks*, covers how to go from a simple neural network to a deep neural network. We will learn about many concepts such as the activation function, normalization, regularization, and dropouts to make the training more robust, so we can train a network more efficiently.

Chapter 3, *Implementing a Deep Learning Model Using Keras*, covers the step-by-step implementation of a deep learning model using Keras. We are going to implement a deep learning model using Keras with the Auto-mpg dataset.

Chapter 4, *Computer Vision for Self-Driving Cars*, introduces advanced computer vision techniques for SDCs. This is one of the important chapters to get into computer vision. In this chapter, we will cover different OpenCV techniques that help in image preprocessing and feature extraction in SDC business problems.

Chapter 5, *Finding Road Markings Using OpenCV*, walks you through writing a software pipeline to identify the lane boundaries in a video from the front-facing camera in a SDC. This is a starter project using OpenCV to get into SDCs.

Chapter 6, *Improving the Image Classifier with CNN*, covers how to go from a simple neural network to a advance deep neural network. In this chapter, we will learn about the theory behind the convolutional neural network, and how a convolutional neural network helps to improve the performance of an image classifier. We will implement an image classifier project using the MNIST dataset.

Chapter 7, *Road Sign Detection Using Deep Learning*, looks at the training of a neural network to implement a traffic sign detector. This is the next step toward SDC implementation. In this chapter, we will create a model that reliably classified traffic signs, and learned to identify their most appropriate features independently.

Chapter 8, *The Principles and Foundations of Semantic Segmentation*, covers the basic structure and workings of semantic segmentation models, and all of the latest state-of-the-art methods.

Chapter 9, *Implementation of Semantic Segmentation*, looks at the implementation of ENET semantic segmentation architecture to detect pedestrians, vehicles, and so on. We will learn about the techniques we can apply to semantic segmentation using OpenCV, deep learning, and the ENet architecture. We will use the pre-trained ENet model to perform semantic segmentation on both images and video streams.

Chapter 10, *Behavioral Cloning Using Deep Learning*, implements behavioral cloning. Here, cloning means that our learning program will copy and clone human behavior such as our steering actions to mimic human driving. We will implement a behavior cloning project and test it in a simulator.

Chapter 11, *Vehicle Detection Using OpenCV and Deep Learning*, implements vehicle detection for SDCs using OpenCV and the pre-trained deep learning model YOLO. Using this model, we will create a software pipeline to perform object prediction on both images and videos.

Chapter 12, *Next Steps*, summarizes the previous chapters and ways to enhance the learning. This chapter also briefs you on sensor fusion, and covers techniques that can be tried out for advanced learning in SDCs.

To get the most out of this book

Let's look at what you need to get the most out the book:

- Some Python programming experience and a basic understanding of deep learning are expected.
- The execution of the code while reading the book will help you to get the most out of it.

Software/Hardware covered in the book	OS Requirements
Install Python 3.7 with latest Anaconda environment	Window, Linux or Mac minimum 8 GB RAM
Install deep learning libraries TensorFlow 2.0 and KERAS 2.3.4	Window, Linux or Mac minimum 8 GB RAM
Install image processing library OpenCV	Window, Linux or Mac minimum 8 GB RAM

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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

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

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Applied-Deep-Learning-and-Computer-Vision-for-Self-Driving-Cars>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781838646301_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "In this function, anything below 0 will be set to 0."

A block of code is set as follows:

```
In[1]: import tensorflow as tf  
In[2]: from tensorflow import keras  
In[3]: from tensorflow.keras import layers
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "The project is now called **Waymo**."

Warnings or important notes appear like this.



Tips and tricks appear like this.



Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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 copyright@packt.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 authors.packtpub.com.

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 Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Section 1: Deep Learning Foundation and SDC Basics

In this section, we will learn about the motivation behind becoming a self-driving car engineer, and the associated learning path, and we will get an overview of the different approaches and challenges found in the self-driving car field. It covers the foundations of deep learning, which are necessary, so that we can take a step toward the implementation of self-driving cars. This section provides a step-by-step explanation to enable you to understand deep neural network libraries such as Keras. It also covers the implementation of deep learning models from scratch by using Keras.

This section comprises the following chapters:

- Chapter 1, *The Foundation of Self-Driving Cars*
- Chapter 2, *Deep Dive into Deep Neural Networks*
- Chapter 3, *Implementing a Deep Learning Model Using Keras*

1

The Foundation of Self-Driving Cars

The driverless car is popularly known as an **self-driving car (SDC)**, an autonomous vehicle, or a robot car. The purpose of an autonomous car is to drive automatically without a driver. The SDC is the sleeping giant that might improve everything from road safety to universal mobility, while dramatically reducing the costs of driving. According to McKinsey & Company, the widespread use of robotic cars in the US could save up to \$180 billion annually in healthcare and automotive maintenance alone based on a realistic estimate of a 90% reduction in crash rates.

Although self-driving automotive technologies have been in development for many decades, it is only in recent years that breakthroughs have been achieved. SDCs have proved to be much safer than human drivers, and automotive firms as well as other tech firms are investing billions in bringing this technology into the real world. They struggle to find great engineers to contribute to the field. This book will teach you what you need to know to kick-start a career in the autonomous driving industry. Whether you're coming from academia, or from within the industry, this book will provide you with the foundational knowledge and practical skills you will need to help build a future with **Advanced Driver-Assistance Systems (ADAS)** engineers or SDC engineers. Throughout this book, you will study real-world data and scenarios from recent research in autonomous cars.

This book can also help you learn and implement the state-of-the-art technologies for computer vision that are currently used in the automotive industry in the real world. By the end of this book, you will be familiar with different deep learning and computer vision techniques for SDCs. We'll finish this book off with six projects that will give you a detailed insight into various real-world issues that are important to SDC engineers.

In this chapter, we will cover the following topics:

- Introduction to SDCs
- Advancement in SDCs
- Levels of autonomy
- Deep learning and computer vision approaches for SDCs

Let's get started!

Introduction to SDCs

The following is an image of an SDC by WAYMO undergoing testing in Los Altos, California:



Fig 1.1: A Google SDC

You can check out the image at https://en.wikipedia.org/wiki/File:Waymo_Chrysler_Pacifica_in_Los_Altos,_2017.jpg.

The idea of the autonomous car has existed for decades, but we saw enormous improvement from 2002 onward when the **Defense Advanced Research Projects Agency (DARPA)** announced the first of its grand challenges, called the DARPA Grand Challenge (2004). That would forever change the world's perception of what autonomous robots can do. The first event was held in 2004 and DARPA offered the winners a one-million-dollar prize if they could build an autonomous vehicle that was able to navigate 142 miles through the Mojave Desert. Although the first event saw only a few teams get off the start line (Carnegie Mellon's red team took first place, having driven only 7 miles), it was clear that the task of driving without any human aid was indeed possible. In the second DARPA Grand Challenge in 2005, five of the 23 teams smashed expectations and successfully completed the track without any human intervention at all. Stanford's vehicle, **Stanley**, won the challenge, followed by Carnegie Mellon's **Sandstorm**, an autonomous vehicle. With this, the era of driverless cars had arrived.

Later, the 2007 installment, called the DARPA Urban Challenge, invited universities to show off their autonomous vehicles on busy roads with professional stunt drivers. This time, after a harrowing 30-minute delay that occurred due to a jumbotron screen blocking their vehicle from receiving GPS signals, the Carnegie Mellon team came out on top, while the Stanford Junior vehicle came second.

Collectively, these three grand challenges were truly a watershed moment in the development of SDCs, changing the way the public (and more importantly, the technology and automotive industries) thought about the feasibility of full vehicular autonomy. It was now clear that a massive new market was opening up, and the race was on. Google immediately brought in the team leads from both Carnegie Mellon and Stanford (Chris Thompson and Mike Monte-Carlo, respectively) to push their designs onto public roads. By 2010, Google's SDC had logged over 140 thousand miles in California, and they later wrote in a blog post that they were confident about cutting the number of traffic deaths by half using SDCs. blog by Google: *What we're driving at* — Sebastian Thrun (<https://googleblog.blogspot.com/2010/10/what-were-driving-at.html>)



According to a report by the World Health Organization, more than 1.35 million lives are lost every year in road traffic accidents, while 20-50 million end up with non-fatal injuries (https://www.who.int/health-topics/road-safety#tab=tab_1).

As per a study released by the **Virginia Tech Transportation Institute (VTTI)** and the **National Highway Traffic Safety Administration (NHTSA)**, 80% of car accidents involve human distraction (<https://seriousaccidents.com/legal-advice/top-causes-of-car-accidents/driver-distractions/>). An SDC can, therefore, become a useful and safe solution for the whole of society to reduce these accidents. In order to propose a path that an intelligent car should follow, we require several software applications to process data using **artificial intelligence (AI)**.

Google succeeded in creating the world's first autonomous car 2 years ago (at the time of writing). The problem with Google's car was its expensive 3D RADAR, which is worth about \$75,000.



The 3D RADAR is used for environmental identification, as well as the development of a high-resolution 3D map.

The solution to this cost is to use multiple, cheaper cameras that are mounted to the car to capture images that recognize the lane lines on the road, as well as the real-time position of the car.

In addition, a driverless car can reduce the distance between cars, thereby reducing the degree of road loads, reducing the number of traffic jams. Furthermore, they greatly reduce the capacity for human errors to occur while driving and allow people with disabilities to drive long distances.

A machine as a driver will never make a mistake; it will be able to calculate the distance between cars very accurately. Parking will be more efficiently spaced, and the fuel consumption of cars will be optimized.

The driverless car is a vehicle equipped with sensors and cameras for detecting the environment, and it can navigate (almost) without any real-time input from a human. Many companies are investing billions of dollars in order to advance this toward an accessible reality. Now, a world where AI takes control of driving has never been closer.

Nowadays, self-driving car engineers are exploring several different approaches in order to develop an autonomous system. The most successful and popularly used among them are as follows:

- The robotics approach
- The deep learning approach

In reality, in the development of SDCs, both robotics and deep learning methods are being actively pursued by developers and engineers.

The robotic approach works by fusing output from a set of sensors to analyze a vehicle's environment directly and prompt it to navigate accordingly. For many years, self-driving automotive engineers have been working on and improving robotic approaches. However, more recently, engineering teams have started developing autonomous vehicles using a deep learning approach.

Deep neural networks enable SDCs to learn how to drive by imitating the behavior of human driving.

The five core components of SDCs are **computer vision**, **sensor fusion**, **localization**, **path planning**, and **control** of the vehicle.

In the following diagram, we can see the five core components of SDCs:

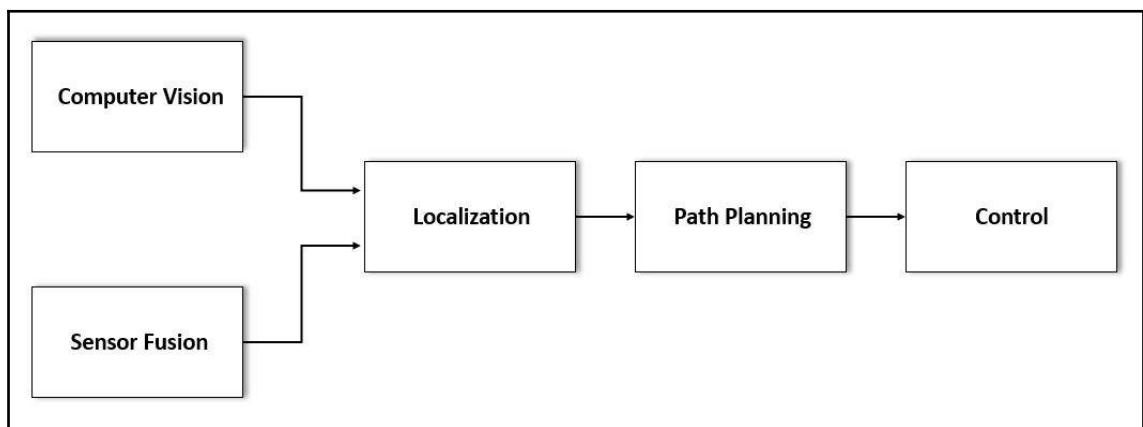


Fig 1.2: The five core components of SDCs

Let's take a brief look at these core components:

- **Computer vision** can be considered the eyes of the SDC, and it helps us figure out what the world around it looks like.
- **Sensor fusion** is a way of incorporating the data from various sensors such as RADAR, LIDAR, and LASER to gain a deeper understanding of our surroundings.
- Once we have a deeper understanding of what the world around it looks like, we want to know where we are in the world, and **localization** helps with this.

- After understanding what the world looks like and where we are in the world, we want to know where we would like to go, so we use **path planning** to chart the course of our travel. Path planning is built for trajectory execution.
- Finally, **control** helps with turning the steering wheel, changing the car's gears, and applying the brakes.

Getting the car to autonomously follow the path you want requires a lot of effort, but researchers have made the possible with the help of advanced systems engineering. Details regarding systems engineering will be provided later in this chapter.

Benefits of SDCs

Indeed, some people may be afraid of autonomous driving, but it is hard to deny its benefits. Let's explore a few of the benefits of autonomous vehicles:

- **Greater safety on roads:** Government data identifies that a driver's error is the cause of 94% of crashes (<https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115>). Higher levels of autonomy can reduce accidents by eliminating driver errors. The most significant outcome of autonomous driving could be to reduce the devastation caused by unsafe driving, and in particular, driving under the influence of drugs or alcohol. Furthermore, it could reduce the heightened risks for unbelted occupants of vehicles, vehicles traveling at higher speeds, and distractions that affect human drivers. SDCs will address these issues and increase safety, which we will see in more detail in the *Levels of autonomy* section of this chapter.
- **Greater independence for those with mobility problems:** Full automation generally offers us more personal freedom. People with special needs, particularly those with mobility limitations, will be more self-reliant. People with limited vision who may be unable to drive themselves will be able to access the freedom afforded by motorized transport. These vehicles can also play an essential role in enhancing the independence of senior citizens. Furthermore, mobility will also become more affordable for people who cannot afford it as ride-sharing will reduce personal transportation costs.
- **Reduced congestion:** Using SDCs could address several causes of traffic congestion. Fewer accidents will mean fewer backups on the highway. More efficient, safer distances between vehicles and a reduction in the number of stop-and-go waves will reduce the overall congestion on the road.

- **Reduced environmental impact:** Most autonomous vehicles are designed to be fully electric, which is why the autonomous vehicle has the potential to reduce fuel consumption and carbon emissions, which will save fuel and reduce greenhouse gas emissions from unnecessary engine idling.

There are, however, potential disadvantages to SDCs:

- The loss of vehicle-driving jobs in the transport industry as a direct impact of the widespread adoption of automated vehicles.
- Loss of privacy due to the location and position of an SDC being integrated into an interface. If it can be accessed by other people, then it can be misused for any crime.
- A risk of automotive hacking, particularly when vehicles communicate with each other.
- The risk of terrorist attacks also exists; there is a real possibility of SDCs, charged with explosives, being used as remote car bombs.

Despite these disadvantages, automobile companies, along with governments, need to come up with solutions to the aforementioned issues before we can have fully automated cars on the roads.

Advancements in SDCs

The idea of SDCs on our streets seemed like a crazy sci-fi fantasy until just a few years ago. However, the rapid progress made in recent years in both AI and autonomous technology proves that the SDC is becoming a reality. But while this technology appears to have emerged virtually overnight, it has been a long and winding path to achieving the self-driving vehicles of today. In reality, it wasn't long after the invention of the motor car when inventors started thinking about autonomous vehicles.

In 1925, former US army electrical engineer and founder of The Houdina Radio Control Co., Francis P Houdina, developed a radio-operated automobile. He equipped a Chandler motor car with a transmitting antenna and operated it from a second car that followed it using a transmitter.



In 1968, John McCarthy, one of the founding fathers of AI, referred to a concept similar to an SDC in an essay entitled *Computer-Controlled Cars*. He mentioned the idea of an automatic chauffeur that is capable of navigating a public road using a television camera's input (<http://www-formal.stanford.edu/jmc/progress/cars/cars.html>).

In the early 1990s, Dean Pomerleau, who is a PhD researcher from Carnegie Mellon University, did something interesting in the field of SDCs. Firstly, he described how neural networks could allow an SDC to take images of the road and predict steering control in real time. Then, in 1995, along with his fellow researcher Todd Jochem, he drove an SDC that they created on the road. Although their SDC required driver control of the speed and brakes, it traveled around 2,797 miles.



You can find out more information about Pomerleau at <http://www-formal.stanford.edu/jmc/progress/cars/cars.html>.

Then came the grand challenge by DARPA in 2002, which we discussed previously. This competition offered a \$1 million prize to any researcher who could build a driverless vehicle. It was stipulated that the vehicle should be able to navigate 142 miles through the Mojave Desert. The challenge kicked off in 2004, but none of the competitors were able to complete the course. The winning team traveled for less than 8 miles in a couple of hours.

In the early 2000s, when the autonomous car was still futuristic, self-parking systems began to evolve. Toyota's Japanese Prius hybrid vehicle started offering automatic parking assistance in 2003. This was later followed by BMW and Ford in 2009.

Google secretly started an SDC project in 2009. The project was initially led by Sebastian Thrun, the former director of the Stanford Artificial Intelligence Laboratory and co-inventor of Google Street View. The project is now called **Waymo**. In August 2012, Google revealed that their driverless car had driven 300,000 miles without a single accident occurring.

Since the 1980s, various companies such as General Motors, Ford, Mercedes-Benz, Volvo, Toyota, and BMW have started working on their own autonomous vehicles. As of 2019, 29 US states have passed legislation enabling autonomous vehicles.



In August 2013, Nissan pledged that they will be releasing several driverless cars by the end of 2020. Nissan Leaf has broken the record for the longest SDC journey in the UK. This autonomous model has driven itself 230 miles from Bedfordshire to Sunderland. So far, this is the longest and most complex journey that's been taken on UK roads by any autonomous vehicle.

Nvidia Xavier is an SDC chip that has incorporated AI capabilities. Nvidia also announced a collaboration with Volkswagen to transform this dream into a reality, by developing AI for SDCs.

On this journey to becoming a reality, the driverless car has launched into a frenzied race that includes tech companies and start-ups, as well as traditional automakers.



The autonomous vehicle is expected to drive a market of \$7 trillion by 2050, which explains why companies are investing heavily to achieve the first-mover advantage.



The SDC and self-driving truck market size is estimated to grow to 6.7 per thousand vehicular units globally in 2020 and is expected to increase at a **compound annual growth rate (CAGR)** of 63.1% from 2021 to 2030.

It is expected that the highest adoption of driverless vehicles will be in the US due to the increase in government support for the market gateway of the autonomous vehicle. The US transportation secretary Elaine Chao signaled strong support for SDCs in the CES tech conference, Las Vegas, which was organized by the Consumer Technology Association on January 7th, 2020.

Additionally, it is expected that Europe will also emerge as a potentially lucrative market for technological advancements in self-driving vehicles with increasing consumer preference.

In the next section, we will learn about the challenges in current deployments of autonomous driving.

Challenges in current deployments

Companies have started public testing with autonomous taxi services in the US, which are often driven at low speeds and nearly always with a security driver.

A few of these autonomous taxi services are listed in the following table:

Voyage	In the villages of Florida
Drive.ai	Arlington, Texas
Waymo One	Phoenix, Arizona
Uber	Pittsburgh, PA
Aurora	San Francisco and Pittsburgh
Optimus Ride	Union Point, MA
May Mobility	Detroit, Michigan
Nuro	Scottsdale, Arizona
Aptiv	Las Vegas, Boston, Pittsburgh, and Singapore
Cruise	San Francisco, Arizona, and Michigan

The fully autonomous vehicle announcements (including testing and beyond) are listed in the following table:

Tesla	Expected in 2019/2020
Honda	Expected in 2020
Renault-Nissan	Expected in 2020 (for urban areas)
Volvo	Expected in 2021 (for highways)
Ford	Expected in 2021
Nissan	Expected in 2020
Daimler	Expected between 2021 and 2025
Hyundai	Expected in 2021 (for highways)
Toyota	Expected in 2020 (for highways)
BMW	Expected in 2021
Fiat-Chrysler	Expected in 2021



Note: Due to the COVID-19 pandemic, global lockdown timelines might be impacted.

However, despite these advances, there is one question we must ask: SDC development has existed for decades, but why is it taking so long to become a reality? The reason is that there are lots of components to SDCs, and the dream can only become a reality with the proper integration of these components. So, what we have today is multiple prototypes of SDCs from multiple companies to showcase their promising technologies.

The key ingredients or differentiators of SDCs are the sensors, hardware, software, and algorithms that are used. Lots of system and software engineering is required to bring all these four differentiators together. Even the choice of these differentiators plays an important role in SDC development.

In this section, we will cover existing deployments and their associated challenges in SDCs. Tesla has recently revealed their advancements and the research they've conducted on SDCs. Currently, most Tesla vehicles are capable of supplementing the driver's abilities. It can take over the tedious task of maintaining lanes on highways; monitoring and matching the speeds of surrounding vehicles; and can even be summoned to you while you are not in the vehicle. These capabilities are impressive and, in some cases, even life-saving, but it is still far from a full SDC. Tesla's current output still requires regular input from the driver to ensure they are paying attention and capable of taking over when needed.

There are four primary challenges that automakers such as Tesla need to overcome in order to succeed in replacing the human driver. We'll go over these now.

Building safe systems

The **first one** is building a safe system. In order to replace human drivers, the SDC needs to be safer than a human driver. So, how do we quantify that? It is impossible to guarantee that accidents will not occur without real-world testing, which comes with that innate risk.

We can start by quantifying how good human drivers are. In the US, the current fatality rate is about one death per one million hours of driving. This includes human error and irresponsible driving, so we can probably hold the vehicles to a higher standard, but that's the benchmark nonetheless. Therefore, the SDC vehicle needs to have fewer fatalities than once every one million hours, and currently, that is not the case. We do not have enough data to calculate accurate statistics here, but we do know that Uber's SDC required a human to intervene approximately every **19 kilometers (KM)**. The first case of pedestrian fatality was reported in 2018 after a pedestrian was hit by Uber's autonomous test vehicle.

The car was in self-driving mode, sitting in the driving seat with a human backup driver. Uber halted testing of SDCs in Arizona, where such testing had been approved since August 2016. Uber opted not to extend its California self-driving trial permit when it expired at the end of March 2018. Uber's vehicle that hit the pedestrian was using LIDAR sensors that didn't work using light coming from camera sensors. However, Uber's test vehicle made no effort to slow down, even though the vehicle was occupied by the human backup driver, who wasn't careful and was not paying attention.

According to the data obtained by Uber, the vehicle first observed the pedestrian 6 seconds before the impact with its RADAR and LIDAR sensors. At the time of the hazard, the vehicle was traveling at 70 kilometers per hour. The vehicle continued at the same speed and when the paths of the pedestrian and the car converged, the classification algorithm of the machine was seen trying to classify what object was in its view. The system switched its identification from an unidentified object, to a car, to a cyclist with no identification of the driving path of the pedestrian. Just 1.3 seconds before the crash, the vehicle was able to recognize the pedestrian. The vehicle was required to perform an emergency brake but didn't as it was programmed not to brake.

As per the algorithm's prediction, the vehicle performed a speed deceleration of more than 6.5 meters per square second. Also, the human operator was expected to intervene, but the vehicle was not designed to alert the driver. The driver did intervene a few seconds before the impact by engaging the steering wheel and braking and bringing the vehicle's speed to 62 kilometers per hour, but it was too late to save the pedestrian. Nothing malfunctioned in the car and everything worked as planned, but it was clearly a case of bad programming. In this case, the internal computer was clearly not programmed to deal with this uncertainty, whereas a human would normally slow down when confronted with an unknown hazard. Even with high-resolution LIDAR, the vehicle failed to recognize the pedestrian.

The cheapest computer and hardware

Computer and hardware architecture plays a significant role in SDCs. As we know, a large part of that lies in the hardware itself and the programming that goes into it. Tesla unveiled its new, purpose-built computer; a chip specifically optimized for running a neural network. It has been designed to be retrofitted in existing vehicles. This computer is of a similar size and power to the existing self-driving computers. This has increased Tesla's SDC computer capabilities by 2,100% as it allows it to process 2,300 frames per second, 2,190 frames more than the previous iteration. This is a massive performance jump, and that processing power will be needed to analyze footage from the suit of sensors Tesla has.

The Tesla autopilot model currently consists of three forward-facing cameras, all mounted behind the windshield. One is a 120-degree wide-angle fish-eye lens, which gives situational awareness by capturing traffic lights and objects moving into the path of travel. The second camera is a narrow-angle lens that provides longer-range information needed for high-speed driving. The third is the main camera, which sits in the middle of these two cameras. There are four additional cameras on the sides of the vehicle that check for vehicles unexpectedly entering any lane, and provide the information needed to safely enter intersections and change lanes. The eighth and final camera is located at the rear, which doubles as a parking camera, but is also used to avoid crashes from rear hazards.

The vehicle does not completely rely on visual cameras. It also makes use of 12 ultrasonic sensors, which provide a 360-degree picture of the immediate area around the vehicle, and one forward-facing RADAR:

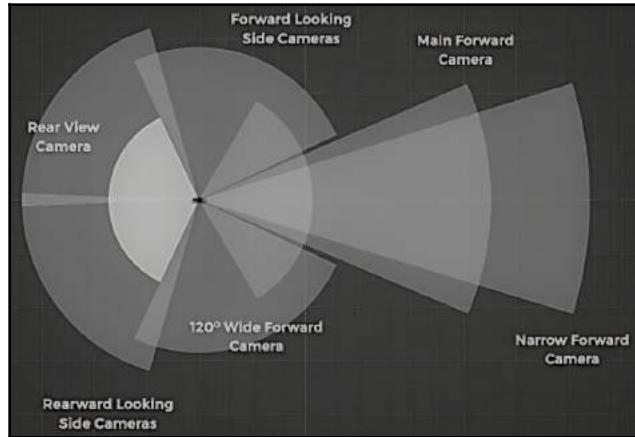


Fig 1.3: Camera views

Finding the correct sensor fusion has been a subject of debate among competing SDC companies. Elon Musk recently stated that anyone relying on LIDAR sensors (which work similarly to RADAR but utilize light instead of radio waves) is doomed. To understand why he said this, we will plot the strengths of each sensor on a RADAR chart, as follows:

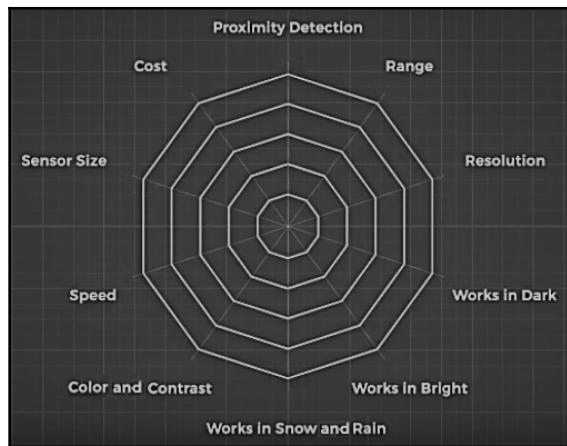


Fig 1.4: RADAR chart

RADAR has great resolution; it provides highly detailed information about what it's detecting. It works in low and high light situations and is also capable of measuring speed. It has a good range and works moderately well in poor weather conditions. Its biggest weakness is that these sensors are expensive and bulky. This is where the **second challenge** of building a SDC comes into play: building an affordable system that the average person can buy.

Let's look at the following RADAR chart:

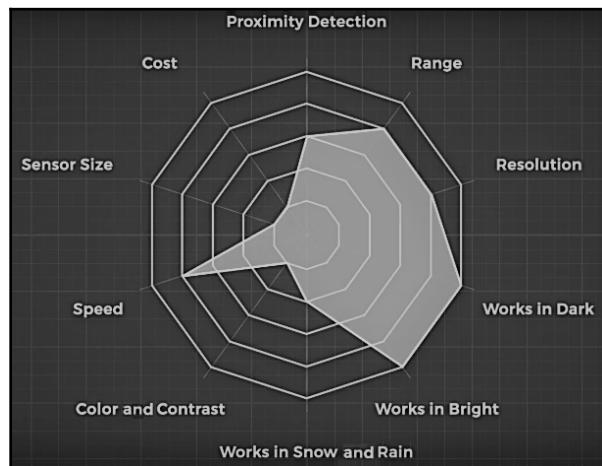


Fig 1.5: RADAR chart – strength

LIDAR sensors are the big sensors we see on Waymo, Uber, and most competing SDC companies output. Elon Musk has become more aware of LIDAR's potential after SpaceX utilized it in their dragon-eye navigation sensor. It's a drawback for Tesla for now, who focused on building not just a cost-effective vehicle, but a good-looking vehicle. Fortunately, LIDAR technology is gradually becoming smaller and cheaper.

Waymo, a subsidiary of Google's parent company Alphabet, sells its LIDAR sensors to any company that does not intend to compete with its plans for a self-driving taxi service. When they started in 2009, the per-unit cost of a LIDAR sensor was around \$75,000, but they have managed to reduce this to \$7,500 as of 2019 by manufacturing the units themselves. Waymo vehicles use four LIDAR sensors on each side of the vehicle, placing the total cost of these sensors for the third party at \$30,000. This sort of pricing does not line up with Tesla's mission as their mission is to speed up the world so that it moves toward sustainable transport. This issue has pushed Tesla toward a cheaper sensor fusion setup.

Let's look at the strength and weaknesses of the three other sensor types – RADAR, camera sensor, and ultrasonic sensor – to see how Tesla is moving forward without LIDAR.

First, let's look at RADAR. It works well in all conditions. RADAR sensors are small and cheap, capable of detecting speed, and their range is good for short- and long-distance detection. Where they fall short is in the low-resolution data they provide, but this weakness can easily be augmented by combining it with cameras. The plot for RADAR and its cameras can be seen in the following image:

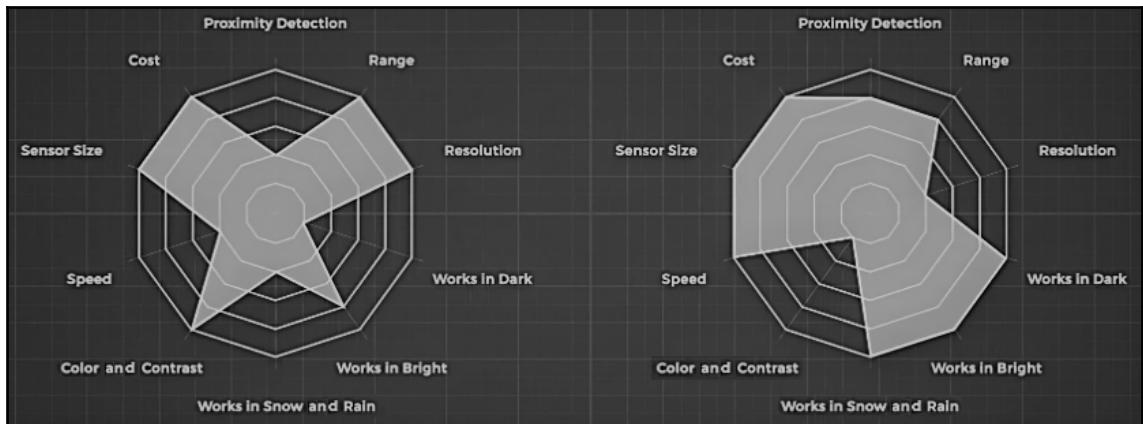


Fig 1.6: RADAR and camera plot

When we combine the two, this yields the following plot:

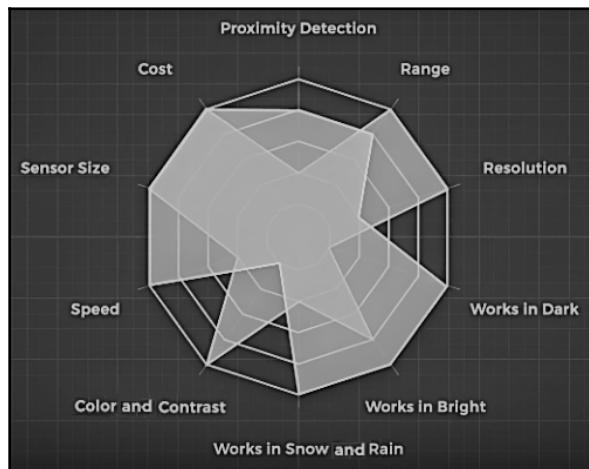


Fig 1.7: Added RADAR and camera plot

This combination has excellent range and resolution, provides color and contrast information for reading street signs, and is extremely small and cheap. Combining RADAR and cameras allows each to cover the weaknesses of the other. They are still weak in terms of proximity detection, but using two cameras in stereo can allow the camera to work like our eyes to estimate distance. When fine-tuned distance measurements are needed, we can use the ultrasonic sensor. An example of an ultrasonic sensor can be seen in the following photo:



Fig 1.8: Ultrasonic sensor

These are circular sensors dotted around the car. In Tesla cars, eight surround cameras have coverage of 360 degrees around the car at a range of up to 250 meters. This vision is complemented by 12 upgraded ultrasonic sensors, which allow the detection of both hard and soft objects that are nearly twice the distance away from the prior device. A forward-facing RADAR with improved processing offers additional data about the world at a redundant wavelength and can be seen through heavy rain, fog, dust, and even the car ahead. This is a cost-effective solution. According to Tesla, their hardware is already capable of allowing their vehicles to self-drive. Now, they just need to continue improving the software algorithms. Tesla is in a fantastic position to make it work.

In the following screenshot, we can see the object detection camera sensor used by Tesla cars:

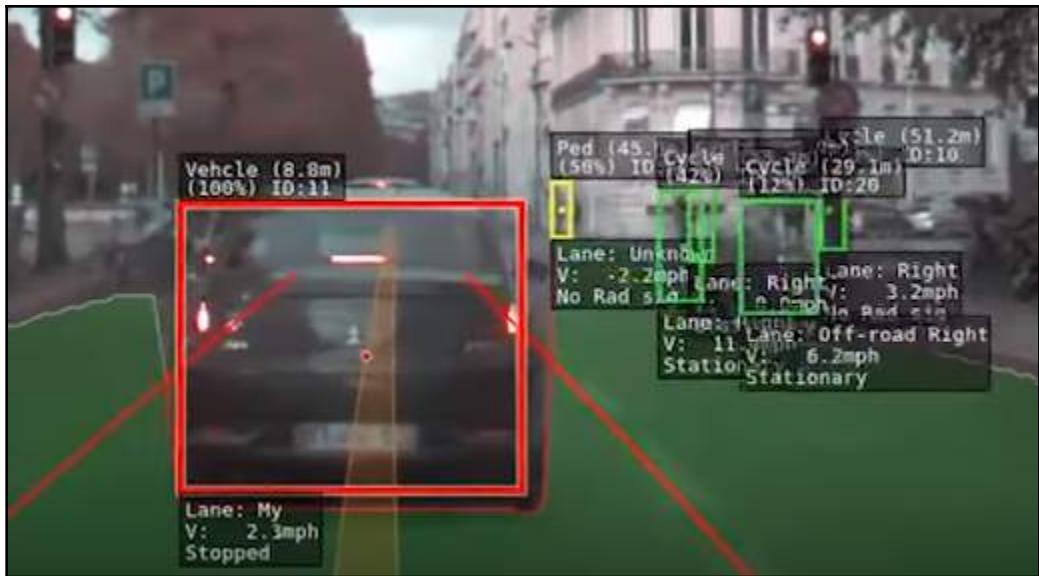


Fig 1.9: Object detection using a camera sensor

When training a neural network, data is key. Waymo has logged millions of kilometers driven in order to gain data, while Tesla has logged over a billion. 33% of all driving with Tesla cars is with the autopilot engaged. This data collection extends past autopilot engagement. Tesla cars also drive manually and collect data in areas where autopilot is not allowed, such as in the city or streets. Accounting for all the unpredictability of driving requires an immense amount of training for a machine learning algorithm, and this is where Tesla's data gives them an advantage. We will read about neural networks in later chapters. One key point to note is that the more data you have to train a neural network, the better it is going to be. Tesla's machine vision does a decent job, but there are still plenty of gaps there.

The Tesla software places bounding boxes around the objects it detects while categorizing them as cars, trucks, bicycles, and pedestrians. It labels each with a relative velocity to the vehicle, and what lane they occupy. It highlights drivable areas, marks the lane dividers, and sets a projected path between them. This frequently struggles in complicated scenarios, and Tesla is working on improving the accuracy of the models by adding new functionalities. Their latest SDC computer is going to radically increase its processing power, which will allow Tesla to continue adding functionality without needing to refresh information constantly. However, even if they manage to develop the perfect computer vision application, programming the vehicle on how to handle every scenario is another hurdle. This is a vital part of building not only a safe vehicle but a practical self-driving vehicle.

Software programming

Another challenge is programming for safety and practicality, which are often at odds with each other. If we program a vehicle purely for safety, its safest option is not to drive.

Driving is an inherently dangerous operation, and programming for the multiple scenarios that arise while driving is an insanely difficult task. It is easy to say **follow the rules of the road**, but the problem is, humans don't follow the rules of the road perfectly. Therefore, programmers need to enable SDCs to react to this. Sometimes, the computer will need to make difficult decisions and may need to make a decision that involves endangering the life of its occupants or people outside the vehicle. This is a dangerous task, but if we continue improving on the technology, we could start to see reduced road deaths, all while making taxi services drastically cheaper and freeing many people from the financial burden of purchasing a vehicle.

Tesla is in a fantastic position to gradually update their software as they master each scenario. They don't need to create the perfect SDC out of the gate, and with this latest computer, they are going to be able to continue their technological growth.

Fast internet

Finally, for many processes in SDCs, fast internet is required. The 4G network is good for online streaming and playing smartphone games, but when it comes to SDCs, next-generation technology such as 5G is required. Companies such as Nokia, Ericsson, and Huawei are researching how to bring out efficient internet technology to specifically meet the needs of SDCs.

In the next section, we will read about the **levels of autonomy** of autonomous vehicles, which are defined by the **Society of Automotive Engineering**.

Levels of autonomy

Although the terms **self-driving** and **autonomous** are often used, not all vehicles have the same capabilities. The automotive industry uses the **Society of Automotive Engineering's (SAE's)** autonomy scale to determine various levels of autonomous capacity. The levels of autonomy help us understand where we stand in terms of this fast-moving technology.

In the following sections, we will provide simple explanations of each autonomy level.

Level 0 – manual cars

In Level 0 autonomy, both the steering and the speed of the car are controlled by the driver. Level 0 autonomy may include issuing a warning to the driver, but the vehicle itself may not take any action.

Level 1 – driver support

In level 1 autonomy, the driver of the car takes care of controlling most of the features of the car. They look at all the surrounding environments, acceleration, braking, and steering. An example of level 1 is that if the vehicle gets too close to another vehicle, it will apply the brake automatically.

Level 2 – partial automation

In level 2 autonomy, the vehicle will be partially automated. The vehicle can take over the steering or speed acceleration and will try to eliminate drivers from performing a few basic tasks. However, the driver is still required in the car for monitoring critical safety functions and environmental factors.

Level 3 – conditional automation

With level 3 autonomy and higher, the vehicle itself performs all environmental monitoring (using sensors such as LIDARs). At this level, the vehicle can drive in autonomous mode in certain situations, but the driver should be ready to take over when the vehicle is in a situation that may exceed the vehicle's control limit.



Audi claimed that the next-generation A8 Luxury Sedan is going to have SAE level 3 autonomy.

Level 4 – high automation

Level 4 autonomy is just one level below full automation. In this level of autonomy, the vehicle can control the steering, brakes, and acceleration of the car. It can even monitor the vehicle itself, as well as pedestrians, and the highway as a whole. Here, the vehicle can drive in autonomous mode for the majority of the time; however, a human is still required to take over in uncontrollable situations, such as in crowded places such as cities and streets.

Level 5 – complete automation

In level 5 autonomy, the vehicle will be completely autonomous. Here, no human driver is required; the vehicle controls all the critical tasks such as steering, the brakes, and the pedals. It will monitor the environment and identify and react to all unique driving conditions such as traffic jams.



NVIDIA's announcement of an AI computer that can control a car will be a great contribution to level 5 autonomy (<https://www.nvidia.com/en-us/self-driving-cars/>).

Now, let's move on and look at the approaches of deep learning and computer vision for SDCs.

Deep learning and computer vision approaches for SDCs

Perhaps the most exciting new technology in the world today is deep neural networks, especially convolutional neural networks. This is known collectively as deep learning. These networks are conquering some of AI's and pattern recognition's most common problems. Due to the rise in computational power, the milestones in AI have been achieved increasingly commonly over recent years, and have often exceeded human capabilities. Deep learning offers some exciting features such as its ability to learn complex mapping functions automatically and being able to scale up automatically. In many real-world applications, such as large-scale image classification and recognition tasks, such properties are essential. After a certain point, most machine learning algorithms reach plateaus, while deep neural network algorithms continually perform better with more and more data. The deep neural network is probably the only machine learning algorithm that can leverage the enormous amounts of training data from autonomous car sensors.

With the use of various sensor fusion algorithms, many autonomous car manufacturers are developing their own solutions, such as LIDAR by Google and Tesla's purpose-built computer; a chip specifically optimized for running a neural network.

Neural network systems have improved in terms of gauging image recognition problems over the past several years, and have exceeded human capabilities.

SDCs can be used to process this sensory data and make informed decisions, such as the following:

- **Lane detection:** This is useful for driving correctly, as the car needs to know which side of the road it is on. Lane detection also makes it easy to follow a curved road.
- **Road sign recognition:** The system must recognize road signs and be able to act accordingly.
- **Pedestrian detection:** The system must detect pedestrians as it drives through a scene. Whether an object is a pedestrian or not, the system needs to know so that it can put more emphasis on not hitting pedestrians. It needs to drive more carefully around pedestrians than other objects that are less important, such as litter.
- **Traffic light detection:** The vehicle needs to detect and recognize traffic lights so that, just like human drivers, it can comply with road rules.
- **Car detection:** The presence of other cars in the environment must also be detected.
- **Face recognition:** There is a need for an SDC to identify and recognize the driver's face, other people inside the car, and perhaps even those who are outside it. If the vehicle is connected to a specific network, it can match those faces against a database to recognize car thieves.
- **Obstacle detection:** Obstacles can be detected using other means, such as ultrasound, but the car also needs to use its camera systems to identify any obstacles.
- **Vehicle action recognition:** The vehicle should know how to interact with other drivers since autonomous cars will drive alongside non-autonomous cars for many years to come.

The list of requirements goes on. Indeed, deep learning systems are compelling tools, but there are certain properties that can affect their practicality, particularly when it comes to autonomous cars. We will implement solutions for all of these problems in later chapters.

LIDAR and computer vision for SDC vision

Some people may be surprised to know that early generation cars from Google barely used their cameras. The LIDAR sensor is useful, but it could not see lights and color, and the camera was mostly used to recognize things such as red and green lights.

Google has since become one of the world's leading players in neural network technology. It has made a substantial effort to execute the sensor fusion of LIDARs, cameras, and other sensors. Sensor fusion is likely to be very good at using neural networks to assist Google's vehicles. Other firms, such as Daimler, have also demonstrated an excellent ability to fuse camera and LIDAR information together. LIDARs are working today, and are expected to become cheaper. However, we have still not crossed that threshold to make the leap toward new neural network technology.

One of the shortcomings of LIDAR is that it usually has a low resolution; so, while not sensing an object in front of the car is very unlikely, it may not figure out what exactly the barrier is. We have already seen an example in the section, *The cheapest computer and hardware*, on how fusing the camera with convolutional neural networks and LIDAR will make these systems much better in this area, and knowing and recognizing what things are means making better predictions regarding where they are going to be in the future.

Many people claim that computer vision systems would be good enough to allow a car to drive on any road without a map, in the same manner as a human being. This methodology applies mostly to very basic roads, such as highways. They are identical in terms of directions and that they are easy to understand. Autonomous systems are not inherently intended to function as human beings do. The vision system plays an important role because it can classify all the objects well enough, but maps are important and we cannot neglect them. This is because, without such data, we might end up driving down unknown roads.

Summary

This chapter addressed the question of how SDCs are becoming a reality. We discovered that SDC technology has existed for decades. We learned how it has evolved, and about advanced research through the arrival of computational power such as GPUs. We also learned about the advantages, disadvantages, challenges, and levels of autonomy of an SDC.

Don't worry if you feel a bit confused or overwhelmed after reading this chapter. The purpose of this chapter was to provide you with an extremely high-level overview of SDCs.

In the next chapter, we are going to study the concept of deep learning more closely. This is the most interesting and important topic of this book, and after reading *Chapter 2, Dive Deep into Deep Neural Networks*, you will be well versed in deep learning.

2

Dive Deep into Deep Neural Networks

In this chapter, you will learn about a topic that has changed the way we think about autonomous driving: **Artificial Neural Networks (ANNs)**. Throughout this chapter, you will learn how these algorithms can be used to build a self-driving car perception stack, and you'll learn about the different components needed to design and train a deep neural network. This chapter will teach you everything you need to know about ANNs. You will also learn about the building blocks of feedforward neural networks, a very useful basic type of ANN. Specifically, we'll look at the hidden layers of a feedforward neural network. These hidden layers are important as they differentiate the mode of action of neural networks from the rest of the **Machine Learning (ML)** algorithms. We'll begin by looking at the mathematical definition of feedforward neural networks so that you can start to understand how to build these algorithms for the perception stacks of self-driving cars.

Before ML became popular for object detection, we used to use the **Histogram of Oriented Gradients (HOG)** and classifiers. The main goal was to train a model that recognizes the shapes of an object by recognizing its different gradients or its orientation. HOG retains the shape and direction of the images, it counts occurrences of the gradient in a localized portion of the image.



We are not going to learn about HOG. You can refer to https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients to learn more about it.

In recent years, **Deep Learning (DL)** has become very popular for its performance due to advances in computational power, the arrival of powerful **Graphical Processor Units (GPUs)**, and its ability to accumulate more **data**. Before GPUs, it was difficult to process deep learning algorithms on our machines, so a big reason for recent developments is cheaper computation.

We will start this chapter with an introduction to neural networks and deep learning. We will also learn about neurons, the activation function, the cost function, and hyperparameters. We will implement our first deep learning model in Chapter 3, *Implementing a Deep Learning Model Using Keras*.

In this chapter, we will cover the following topics:

- Diving deep into neural networks
- Understanding neurons and perceptrons
- The workings of ANNs
- Understanding activation functions
- The cost function of neural networks
- Optimizers
- Understanding hyperparameters
- TensorFlow versus Keras

Diving deep into neural networks

Deep learning is a sub-field of ML that is based on ANNs (see Fig 2.1). Deep learning mimics the human brain and is inspired by the structure and function of the brain. The concept of deep learning is not new and has existed for a number of years. The reason for the popularity and success of deep learning in recent years is due to high powered processing units, such as GPUs, and the presence of enormous amounts of data. One of the reasons for **deep neural networks (DNNs)** performing better is the complex relationships among features and high-dimensional data:

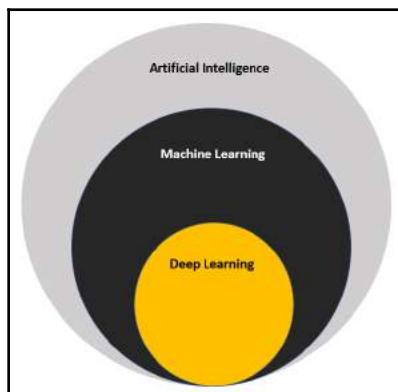


Fig 2.1: Deep learning is a sub-field of ML

One of the great things about deep learning is that it eliminates human input. It replaces the costly and inefficient effort of human beings and automates most of the extraction process from features and raw data so that it doesn't require human involvement. Before, we used to extract features ourselves to make ML algorithms work. The automated extraction of features has enabled us to work with larger datasets, eliminating the need for human intervention completely, except at the supervised labeling step at the very end. Thus, when there is huge data, the performance of deep learning algorithms is far better than most ML algorithms, as shown in the following diagram:

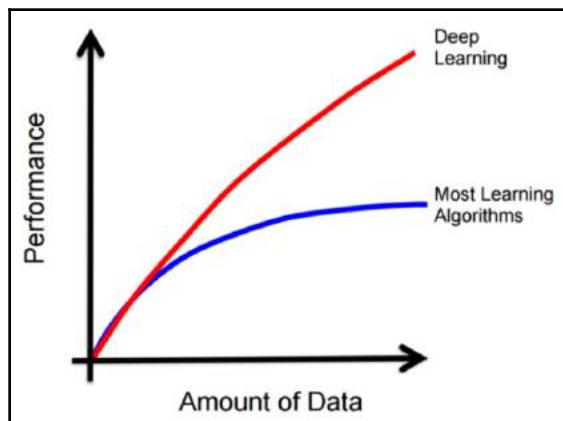


Fig 2.2: Deep learning and ML performance



It may surprise you to know that deep learning has existed since the 1940s. It has undergone various name changes, including **cybernetics**, **connectionism**, and the most well-known iteration, **ANNs**.

In the next section, we will look at the biological neuron—the foundation of neural networks.

Introduction to neurons

In this section, we will discuss neurons, which are the basic building blocks of ANNs. In the following photograph, we can see actual real-life neurons as observed through a microscope:

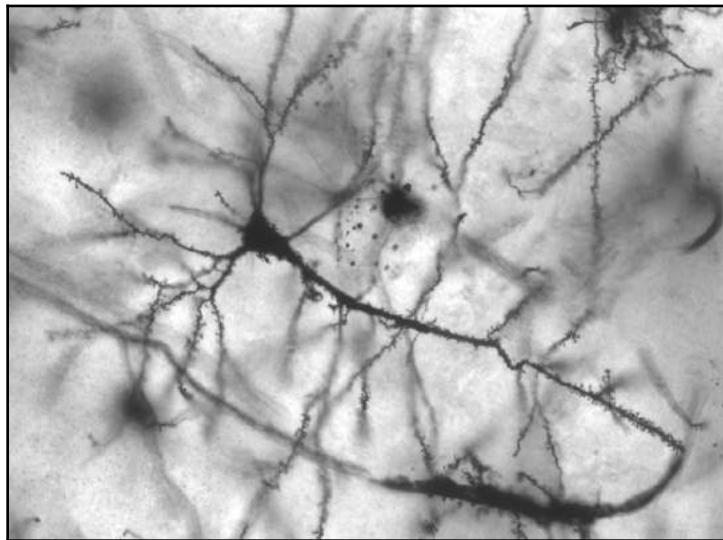


Fig 2.3: A photograph of a neuron

You can find this photograph at https://commons.wikimedia.org/wiki/Neuron#/media/File:Pyramidal_hippocampal_neuron_40x.jpg.

The question now is how can we recreate neurons in ML? We need to create them since the whole purpose of deep learning is to mimic the human brain, one of the most powerful tools on the planet. So, the first step toward creating an ANN is to recreate a neuron.

Before creating a neuron in ML, we will examine the depiction of neurons created by Spanish neuroscientist **Santiago Ramon y Cajal** in 1899.



Santiago Ramon y Cajal observed two neurons that had branches at the top and many threads below (<https://commons.wikimedia.org/wiki/File:PurkinjeCell.jpg>).

Nowadays, we have advanced technology that allows us to get a closer look at neurons. We can see a neuron in the following diagram that looks very similar to what Santiago Ramon y Cajal drew.

We can see that it has a body called a **neuron**, some branches called **dendrites**, and a long tail called an **axon**:

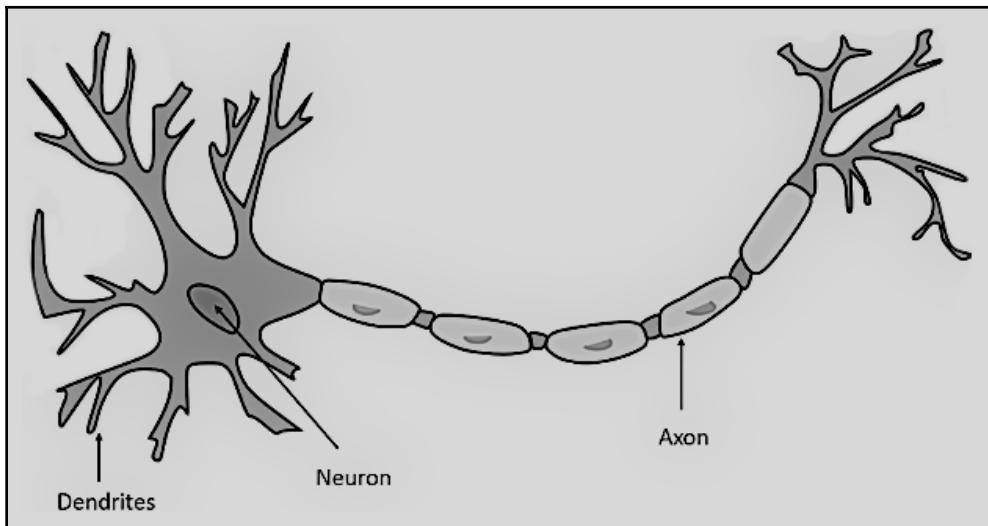


Fig 2.4: Dendrites, the neuron, and the axon

One thing to note is that Fig 2.4 shows only one neuron. One neuron alone is not that powerful, but when you have lots of neurons together, they can do wonders. You may be wondering, *how do they work together?* The answer is with the help of dendrites and axons. Dendrites act as a receiver for other neurons' signals and the axon is the transmitter of the neuron's signal.

The diagram, found at the following link, shows the **generic neurotransmitter system** (https://commons.wikimedia.org/wiki/File:Generic_Neurotransmitter_System.jpg). From the preceding diagram, we can see that the dendrites of one neuron are connected to the axon of another neuron. Then, the signal is transferred to its axon and connects or passes on to the dendrites of the next neuron; that is how they are connected. We can see, in the right-hand part of the preceding diagram, that the axon does not actually touch the dendrite. A lot of ML scientists are very adamant about the fact that it does not touch the dendrite, and it has been proven that there is no physical connection. However, we are interested in knowing about the connection between the axon and dendrites. We already know that the dendrites of neurons receive input in the form of electric signals from the axons of other neurons.



Let's move on to see how we are going to create neurons in machines; we are moving away from neuroscience now. In the next section, we will read about neurons and perceptrons in detail and create artificial neurons.

Understanding neurons and perceptrons

As discussed in the previous section, *Introduction to neurons*, before, ANNs had a basis in biology, and we mimic biological neurons with artificial neurons that are known as **perceptrons**. The perceptron is a mathematical model of a biological neuron. Later in this section, we will see how we can mimic biological neurons with artificial neurons.

As we know, the biological neuron is a brain cell. The body of the neuron has dendrites. When an electrical signal is passed from the dendrites to the body cell of the neuron, a single output or a single electrical signal comes out through an axon, and then it connects to some other neuron, as shown in the diagram of the **generic neurotransmitter system** that you can find in the link provided in the *Introduction to neurons* section. That is the basic idea we have: lots of inputs of electrical signals go through the dendrites, into the body, and then through the axon as a single output signal.

We can see this in the following example, which shows the conversion to an artificial neuron from a biological neuron:

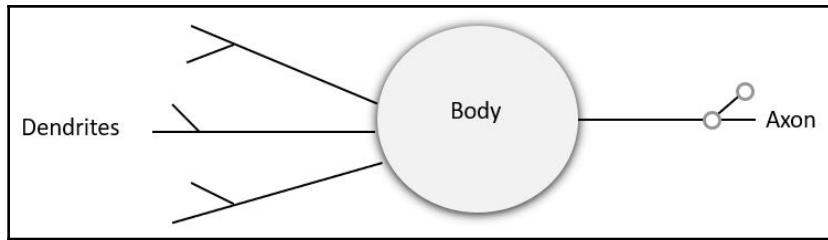


Fig 2.5: Artificial neuron

We can see that the artificial neuron also has input and outputs, so the attempt to mimic a biological neuron was done successfully. This simple model is known as a **perceptron**.

Let's see a simple example of how we are going to convert a biological neuron into an artificial neuron. The step-by-step procedure is as follows. We can see, in the following diagram, that **dendrites** are converted into **input signals**, the **axon** is converted into an **output signal**, and the body is converted into a neuron:

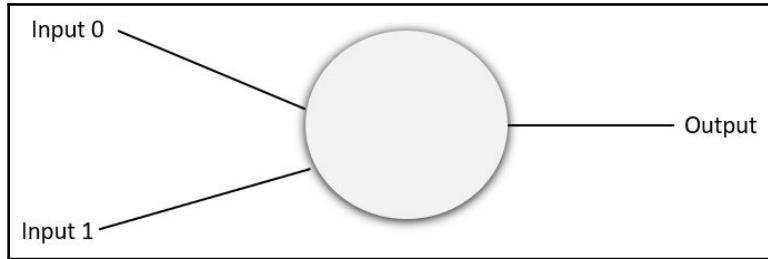


Fig 2.6: Dendrites as the input signals and the axon as the output signal

In this case, we have two inputs and a single output. We will start indexing at **0** and will have the input of **0** and **1**. The input will have the value of features. So, when you have your dataset, you are going to have various features, and these features can be anything from how many rooms a house has or how dark an image is, which is represented by some sort of pixel amount or some kind of darkness. In the following example of conversion into an artificial neuron, we will assign input values of **12** and **4**:

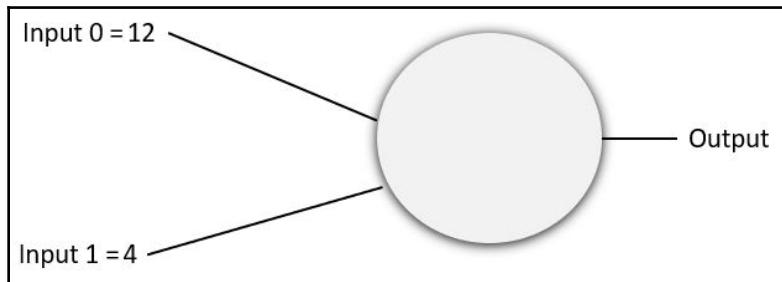


Fig 2.7: Assigning 12 to input 0 and 4 to input 1

Our next step is to multiply these input signals (input 0 and input 1) with weight. The actual neuron only fires an output signal when the total input signal intensity reaches a defined threshold. Here, we have weight **0** for input **0** and weight **1** for input **1**, as shown in the following diagram. Typically, weights are initialized through some sort of random generation as it chooses a random weight:

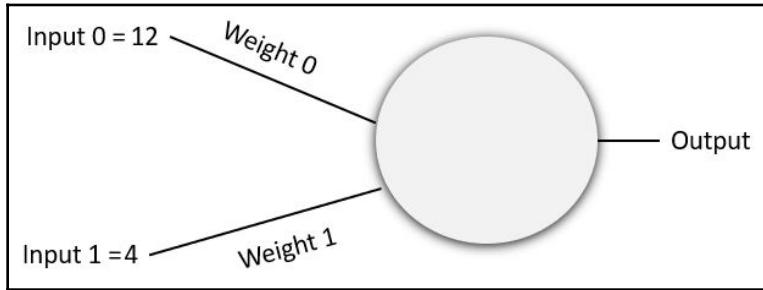


Fig 2.8: Choosing a random weight

In this case, random numbers have been chosen for the value of weights—for example, **0.5** for weight **0** and **-1** for weight **1**—and the chosen numbers are arbitrary:

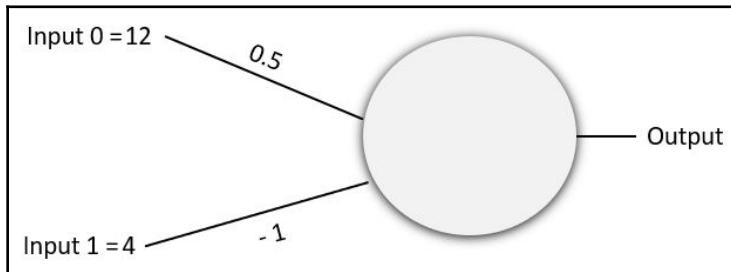


Fig 2.9: Assigning weight as 0.5 and -1

So now, these inputs are going to be multiplied by the weights, which means that input **0** is **6** and input **1** is **-4**. The next step is to multiply the inputs by their weight and pass them to the activation function, as in the following diagram:

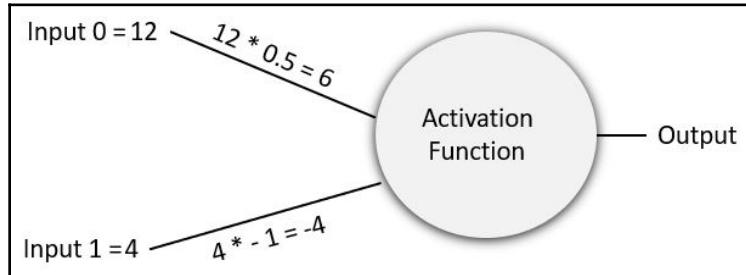


Fig 2.10: Multiplying the input with the weight and passing it to the activation function

There are many activation functions we can choose from, which we will cover in a later section of this chapter, *Understanding activation functions*. For now, we will choose a simple activation function. So, in the case of this activation function, if the sum of the inputs is positive, then it returns an output of **1**, and if the sum of the input is negative, then it returns an output of **0**. In our case, the sum of the input is positive, so it will return **1**.

There can be a case where the original input is **0**, so for every weight, it will be **0**. This can be fixed by adding a bias term, which is shown in the following diagram. The bias is the intercept, which is a linear equation of a straight line. The **bias** is the parameter that helps the neural network adjust the output and the weighted sum of the neurons. In simple terms, we can say that the bias helps the model best fit the given data. In this case, we will choose a bias of **1**, as shown in the following diagram:

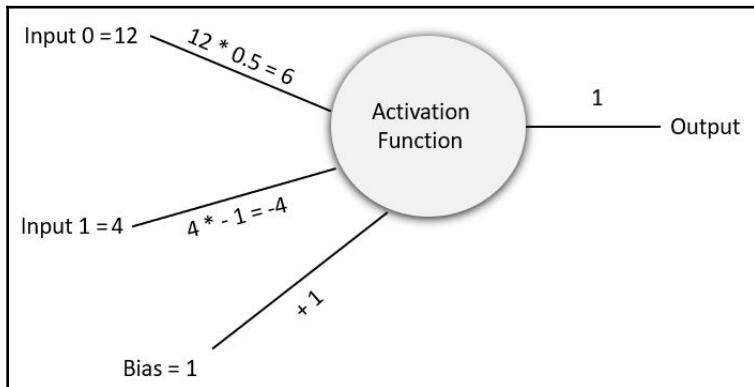


Fig 2.11: The value of the output after applying an activation function

So, we have designed an artificial neuron from a biological neuron. Now, we are ready to learn about ANNs. In the next section, we will see how a deep learning network is designed using multiple neurons.

The workings of ANNs

We have seen the concept of how a single neuron or perceptron works; so now, let's expand the concept to the idea of deep learning. The following diagram shows us what multiple perceptrons look like:

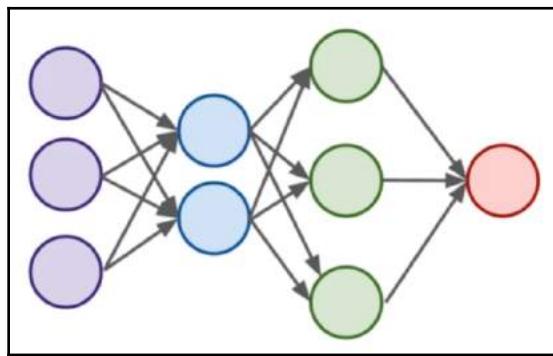


Fig 2.12: Multiple perceptrons

In the preceding diagram, we can see various layers of single perceptrons connected to each other through their inputs and outputs. The input layer is violet, the hidden layers are blue and green, and the output layer of the network is represented in red.

Input layers are real values from the data, so they take in actual data as their input. The next layers are the hidden layers, which are between the input and output layers. If three or more hidden layers are present, then it's considered a deep neural network. The final layer is the output layer, where we have some sort of final estimation of whatever the output that we are trying to estimate is. As we progress through more layers, the level of abstraction increases.

In the next section, we will understand an important topic in deep learning—activation functions and their types.

Understanding activation functions

Activation functions are so important to neural networks as they introduce non-linearity to a network. Deep learning consists of multiple non-linear transformations, and activation functions are the tools for non-linear transformation. Hence, activation functions are applied before sending an input signal to the next layer of neural networks. Due to activation functions, a neural network has the power to learn complex features.

Deep learning has many activation functions:

- The threshold function
- The sigmoid function
- The rectifier function
- The hyperbolic tangent function
- The cost function

In the next section, we will start with one of the most important activation functions, called the threshold activation function.

The threshold function

The **threshold** function can be seen in the following diagram:

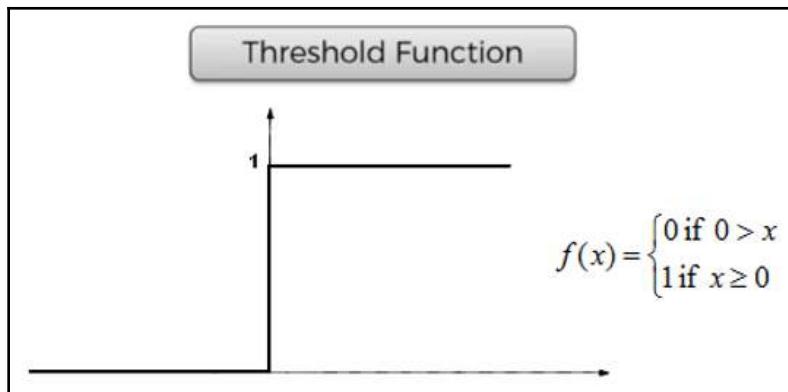


Fig 2.13: The threshold function

On the x axis, we have the weighted sum of the input, and on the y axis, we have the threshold values from 0 to 1. The threshold function is very simple: if the value is less than 0, then the threshold will be 0 and if the value is more than 0, then the threshold will be 1. This works as a yes-or-no function.

The sigmoid function

The **sigmoid** function is a very interesting type of function; we can see it in the following diagram:

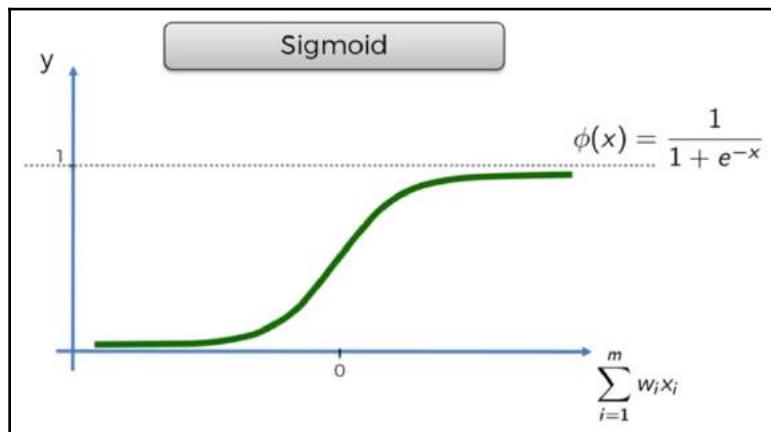


Fig 2.14: The sigmoid function

The sigmoid function is nothing but a logistic function. In this function, anything below 0 will be set to 0. This function is often used in the output layer, especially when you're trying to find the predictive probability.

The rectifier linear function

The **Rectifier Linear (ReLU)** function is one of the most popular functions in the field of ANNs. If the value is less than or equal to 0, then the value of x is set to 0, and then from there, it gradually progresses as the input value increases. We can observe this in the following diagram:

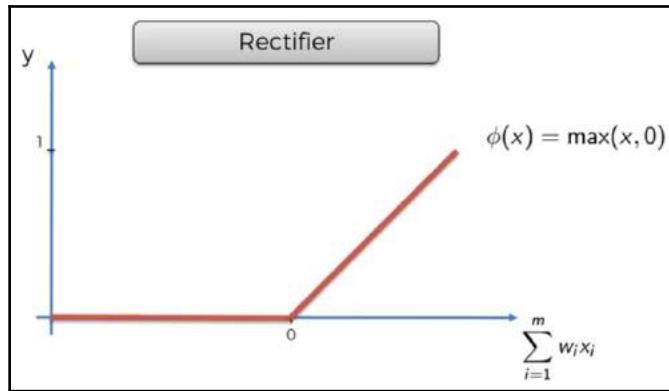


Fig 2.15: The rectifier function

In the next section, we will learn about the hyperbolic tangent activation function.

The hyperbolic tangent activation function

Finally, we have another function, called the **Hyperbolic Tangent (tanh)** function, which looks as follows:

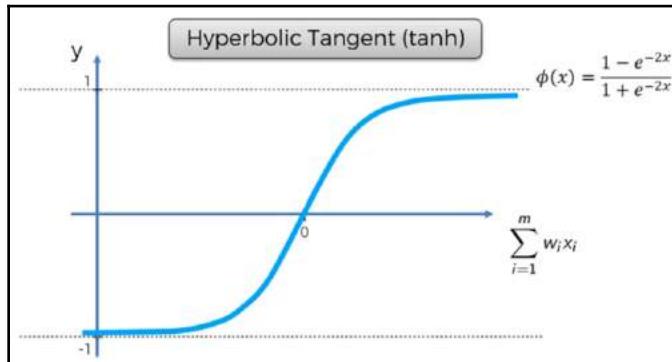


Fig 2.16: Hyperbolic tangent

The **tanh** function is very similar to the sigmoid function; the range of a tanh function is (-1,1). Tanh functions are also S-shaped, like sigmoid functions. The advantage of the tanh function is that a positive will be mapped as strongly positive, a negative will be mapped as strongly negative, and 0 will be mapped to 0, as shown in *Fig 2.16*.



For more information about the performance of the hyperbolic function (\tanh), refer to <http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf>.

In the next section of this chapter, we will learn about the cost function.

The cost function of neural networks

We will now explore how can we evaluate the performance of a neural network by using the cost function. We will use it to measure how far we are from the expected value. We are going to use the following notation and variables:

- Variable Y to represent the true value
- Variable a to represent the neuron prediction

In terms of weight and biases, the formula is as follows:

$$W * X + b = z$$

We pass z , which is the input (X) times the weight (X) added to the bias (b), into the activation function of $\sigma(z) = a$.

There are many types of cost functions, but we are just going to discuss two of them:

- The quadratic cost function
- The cross-entropy function

The first cost function we are going to discuss is the **quadratic cost** function, which is represented with the following formula:

$$C = \Sigma(Y - a)^2 / n$$

In the preceding formula, we can see that when the error is high, which means the actual value (y) is less than the predictive value (a), then the value of the cost function will be negative and we cannot use a negative value as cost. So, we are going to square the result, then the value of the cost function will be a positive value. But unfortunately, when we use the quadratic cost function, the way the formula works actually reduces the learning rate of the network.

Instead, we are going to use the **cross-entropy** function, which can be defined as follows:

$$\text{CrossEntropy}(C) = (-1/n)\Sigma(y * \ln(a) + (1 - y) * \ln(1 - a))$$

This cost function allows faster learning because the larger the difference between y and a , the faster our neurons' learning rate. This means that if the network has a large difference between the predicted value and the actual value at the beginning of the model training process, then we can essentially move toward using a cost function because the larger the difference, the faster the neurons are going to learn.

There are two key components of how neural networks learn from features. First, there are neurons and their activation functions and cost functions, but we are still missing a key step: the actual learning process. So, we need to figure out how we can use the neurons and their measurement of error (the cost function) to correct our prediction or make the network learn. Up until now, we have tried to understand neurons and perceptrons, and then linked them to get a neural net. We also understand that cost functions are essentially measurements of errors. Now, we are going to fix the errors between the actual and predicted values using gradient descent and backpropagation.

Optimizers

Optimizers define how a neural network learns. They define the value of parameters during the training such that the loss function is at its lowest.

Gradient descent is an optimization algorithm for finding the minima of a function or the minimum value of a cost function. This is useful to us as we want to minimize the cost function. So, to find the local minimum, we take steps proportional to the negative of the gradient.

Let's go through a very simple example in one dimension, shown in the following plot:

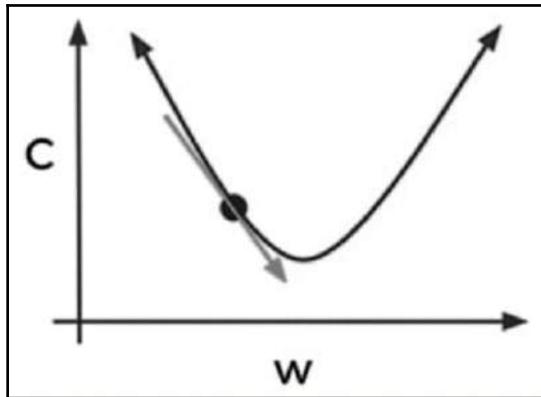


Fig 2.17: Gradient descent

On the y axis, we have the cost (the result of the cost function), and on the x axis, we have the particular weight we are trying to choose (we chose the random weight). The weight minimizes the cost function and we can see that, basically, the parameter value is at the bottom of the parabola. We have to minimize the value of the cost function to the minimum value. Finding the minimum is really simple for one dimension, but in our case, we have a lot more parameters, and we can't do this visually. We are going to use linear algebra and a deep learning library, where we can get the best parameters for minimizing the cost function.

Now, let's see how we can quickly adjust the optimal parameters or weights across our entire network. This is where we need backpropagation.

Backpropagation is used to calculate the error contribution from each neuron after a batch of data is processed. It relies heavily on the chain rule to go back through the network and calculate these errors. Backpropagation works by calculating the error at the output and then updates the weight back through the network layers. It requires a known desired output for each input value.



One of the problems with gradient descent is that the weight is only updated after seeing the entire dataset, so the gradient below is typically large and reaching the loss at the minima is really difficult. One of the solutions to this is updating the parameter more frequently, as in the case of another optimizer called **stochastic gradient descent**. Stochastic gradient descent updates the weight after seeing each data point instead of the whole dataset. It may have noise, however, as it is influenced by every single sample. Due to this, we use **mini-batch gradient descent**, which updates the parameters after only a few samples. You can read more about optimizers in the *An Overview of Gradient Descent Optimization Algorithms* paper (<https://arxiv.org/pdf/1609.04747.pdf>). Another way of decreasing the noise of stochastic gradient descent is to use Adam optimizers. Adam is one of the more popular optimizers; it is an adaptive learning rate method and computes individual learning rates for different parameters. You can check out this paper on Adam optimizers: *Adam: A Method for Stochastic Optimization* (<https://arxiv.org/abs/1412.6980>).

In the next section, we will learn about hyperparameters, which help tweak neural networks so that they can learn features more effectively.

Understanding hyperparameters

Hyperparameters serve a similar purpose to the various tone knobs on a guitar that are used to get the best sound. They are settings that you can tune to control the behavior of an ML algorithm.

A vital aspect of any deep learning solution is the selection of hyperparameters. Most deep learning models have specific hyperparameters that control various aspects of the model, including memory or the execution cost. However, it is possible to define additional hyperparameters to help an algorithm adapt to a scenario or problem statement. To get the maximum performance of a particular model, data science practitioners typically spend lots of time tuning hyperparameters as they play such an important role in deep learning model development.

Hyperparameters can be broadly classified into two categories:

- Model training-specific hyperparameters
- Network architecture-specific hyperparameters

In the following sections, we will cover model training-specific hyperparameters and network architecture-specific hyperparameters in detail.

Model training-specific hyperparameters

Model training-specific hyperparameters play an important role in model training. These are hyperparameters that live outside the model but have a direct influence on it. We will discuss the following hyperparameters:

- Learning rate
- Batch size
- Number of epochs

Let's start with the learning rate.

Learning rate

The learning rate is the mother of all hyperparameters and quantifies the model's learning progress in a way that can be used to optimize its capacity.

A too-low learning rate would increase the training time of the model as it would take longer to incrementally change the weights of the network to reach an optimal state. On the other hand, although a large learning rate helps the model adjust to the data quickly, it causes the model to overshoot the minima. A good starting value for the learning rate for most models would be 0.001 ; in the following diagram, you can see that a low learning rate requires many updates before reaching the minimum point:

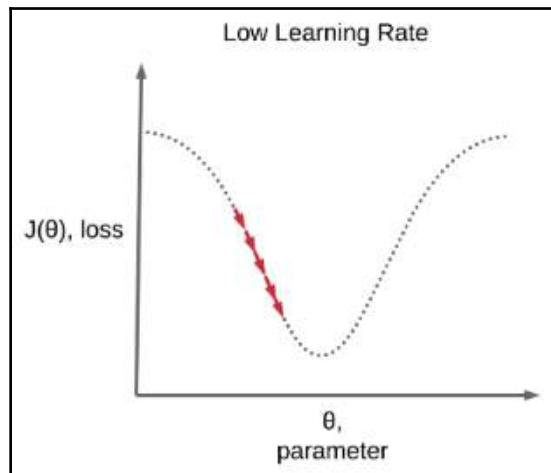


Fig 2.18: A low learning rate

However, an optimal learning rate swiftly reaches the minimum point. It requires less of an update before reaching near minima. Here, we can see a diagram with a decent learning rate:

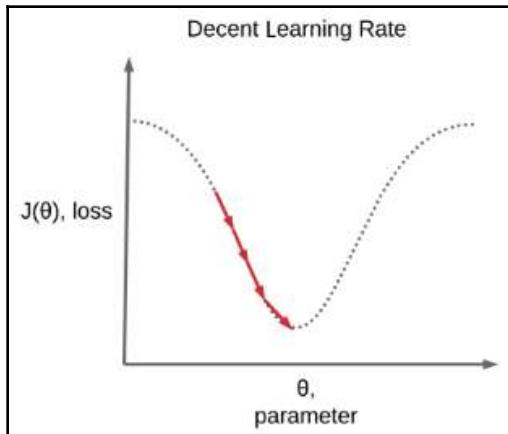


Fig 2.19: Decent learning rate

A high learning rate causes drastic updates that lead to divergent behaviors, as shown in the following diagram:

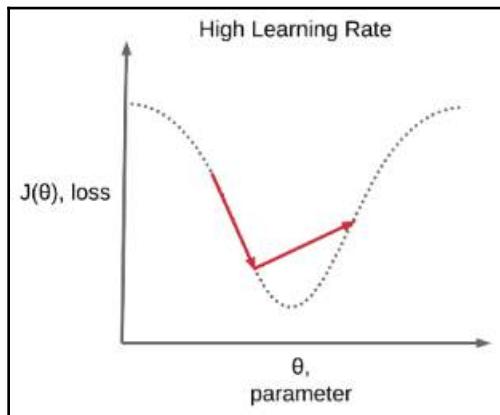


Fig 2.20: A high learning rate



Here is a paper on choosing the learning rate (*Cyclical Learning Rates for Training Neural Networks*) by Leslie Smith: <https://arxiv.org/abs/1506.01186>.

In the next section, we will learn about an important model training-specific parameter called batch size.

Batch size

Another non-trivial hyperparameter that has a huge influence on the training accuracy, time, and resource requirements is batch size. Basically, batch size determines the number of data points that are sent to the ML algorithm in a single iteration during training.

Although having a very large batch size is beneficial for huge computational boosts, in practice, it has been observed that there is a significant degradation in the quality of the model, as measured by its ability to generalize. Batch size also comes at the expense of needing more memory for the training process.

Although a smaller batch size increases the training time, it almost always yields a better model than when using a larger batch size. This can be attributed to the fact that smaller batch sizes introduce more noise in gradient estimations, which helps them converge to flat minimizers. However, the downside of using a small batch size is that training times are increased.



In general, if the training sample sizes are larger, it is usually recommended to choose a larger batch size. A good batch size value would be around 2 to 32. You can refer to a paper written by Dominic Masters (<https://arxiv.org/search/cs?searchtype=author&query=Masters%2C+D>) and Carlo Luschi (<https://arxiv.org/search/cs?searchtype=author&query=Luschi%2C+C>), *Revisiting Small Batch Training for Deep Neural Networks* (<https://arxiv.org/abs/1804.07612>), for more information. As this paper says, mini-batch sizes between $m = 2$ and $m = 32$ perform well consistently.

Number of epochs

An epoch is the number of cycles for which a model is trained. One epoch is when a whole dataset is passed forward and backward only once through the neural network. We can also say that an epoch is an easy way to track the number of cycles, while the training or validation error continues to go on. Since one epoch is too large to feed at once to the machine, we divide it into many smaller batches.

One of the techniques to do this is to use the **early stopping** Keras callback, which stops the training process if the training/validation error has not improved in the past 10 to 20 epochs.

Network architecture-specific hyperparameters

The hyperparameters that directly deal with the architecture of the deep learning model are called **network architecture-specific hyperparameters**. The different types of network-specific hyperparameters are as follows:

- Number of hidden layers
- Regularization
- Activation function as hyperparameters

In the following section, we will see how network architecture-specific hyperparameters work.

Number of hidden layers

It is easy for a model to learn simple features with a smaller number of hidden layers. However, as the features get complex or non-linearity increases, it requires more and more layers and units.

Having a small network for a complex task would result in a model that performs poorly as it wouldn't have the required learning capacity. Having a slightly larger number of units than the optimal number is not a problem; however, a much larger number will lead to the model overfitting. This means that the model will try to memorize the dataset and perform well on the training dataset, but will fail to perform well on the test data. So, we can play with the number of hidden layers and validate the accuracy of the network.

Regularization

Regularization is a hyperparameter that allows slight changes to the learning algorithm so that the model becomes more generalized. This also improves the performance of the model on the unseen data.

In ML, regularization penalizes the coefficients. In deep learning, regularization penalizes the weight matrices of the nodes.

We are going to discuss two types of regularization, as follows:

- L1 and L2 regularization
- Dropout

We will start with L1 and L2 regularization.

L1 and L2 regularization

The most common types of regularization are L1 and L2. We change the overall cost function by adding another term called regularization. The values of weight matrices decrease due to the addition of this regularization because it assumes that a neural network with smaller weight matrices leads to simpler models.

Regularization is different in L1 and L2. The formula for L1 regularization is as follows:

$$\text{CostFunction} = \text{Loss} + \frac{\lambda}{2m} * \sum ||W||$$

In the preceding formula, regularization is represented by lambda (λ). Here, we penalize the absolute weight.

The formula for L2 regularization is as follows:

$$\text{CostFunction} = \text{Loss} + \frac{\lambda}{2m} * \sum ||W||^2$$

In the preceding formula, L2 regularization is represented by lambda (λ). It is also called *weight decay* as it forces the weights to decay close to 0.

Dropout

Dropout is a regularization technique that is used to improve the generalizing power of a network and prevent it from overfitting. Generally, a dropout value of 0.2 to 0.5 is used, with 0.2 being a good starting point. In general, we have to select multiple values and check the performance of the model.

The likelihood of a dropout that has a value that is too low has a negligible impact. However, if the value is too high for the network, then the network under-learns the features during model training. If dropout is used on a larger and wider network, then you are likely to get better performance, giving the model a greater opportunity to learn independent representations.

An example of dropout can be seen as follows, showing how we are going to drop a few of the neurons from the network:

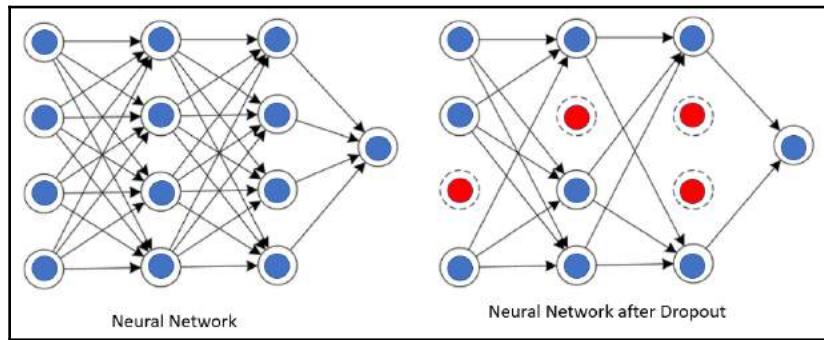


Fig 2.21: Dropout

In the next section, we will learn about activation functions as hyperparameters.

Activation functions as hyperparameters

Activation functions, which are less commonly known as transfer functions, are used to enable the model to learn nonlinear prediction boundaries. Different activation functions behave differently and are carefully chosen based on the deep learning task at hand. We have already discussed different types of activation in an earlier section of this chapter, *Understanding activation functions*.

In the next section, we will learn about the popular deep learning APIs—TensorFlow and Keras.

TensorFlow versus Keras

Primarily, there are two levels of abstraction for deep learning frameworks:

- Firstly, there is the lower level, where frameworks such as TensorFlow, Theano, and PyTorch sit. It is at this level where neural network elements such as convolutions and other generalized matrix operations are carried out.
- Then, there is a higher level, where frameworks such as Keras are present. Here, primitives from the lower levels are utilized to create neural network layers and models. User-friendly APIs for training and saving models are also implemented here.

Since they are present on different levels of abstraction, you cannot compare Keras and TensorFlow. TensorFlow, while being used for deep learning, is not a dedicated deep learning library and is used for a wide array of other applications besides deep learning. Keras, however, is a library developed from the ground up specifically for deep learning. It has very well-designed APIs that use TensorFlow or Theano as the backend, taking advantage of their powerful computational engines.

I always hear people say that if you are starting with deep learning or you are not carrying out any serious research, developing some crazy neural network, then you should only use Keras. I want to take this opportunity to say that I don't agree with the preceding statement. Keras is a beautifully written API. The functional nature of the API doesn't block access to lower-level frameworks, and it also doesn't get in your way when building complex applications. Some of the other benefits that come with using Keras are as follows:

- Using Keras helps make the code more concise and readable.
- Keras APIs for building models, callbacks, and data streaming using generators that are robust and mature.
- Keras is now officially declared as high-level abstraction for TensorFlow.

In this book, we will use Python, which is an extremely powerful and intuitive programming language and is one of the best when it comes to working with deep learning algorithms. The primary deep learning library we'll use during the course of this book is Keras. Furthermore, since Keras is added to the core TensorFlow library from Google, we can always integrate the TensorFlow code directly into our Keras models if we wish. In many ways, we are getting the best of both worlds by using Keras.

Summary

In this chapter, we learned how to convert biological neurons into artificial neurons, how ANNs work, and about various hyperparameters. We also covered an overview of deep learning APIs—TensorFlow and Keras. This chapter has provided a foundation for deep learning. Now, you are ready to start implementing a deep learning model, which is the next step toward designing your implementation of a deep learning model for autonomous cars.

In the next chapter, we are going to implement a deep learning model using Keras.

3

Implementing a Deep Learning Model Using Keras

In chapter, [Chapter 2, Deep Dive into Deep Neural Networks](#), we learned about deep learning in detail, which means we have a solid foundation in this area. We are also closer to implementing computer vision solutions for self-driving cars. In this chapter, we will learn about the deep learning API Keras. This will help us with the implementation of deep learning models. We will also examine a deep learning implementation using the **Auto-Mpg dataset**. We'll start by understanding what Keras is, and then implement our first deep learning model.

In this chapter, we will cover the following topics:

- Starting work with Keras
- Keras for deep learning
- Building your first deep learning model

Let's get started!

Starting work with Keras

What is Keras? Keras is a Python-based deep learning framework that is actually the high-level API of TensorFlow. Keras can run on top of Theano, TensorFlow, or **Microsoft Cognitive Toolkit (CNTK)**. Since it can run on any of these frameworks, Keras is amazingly simple and popular to work with; building models is as simple as stacking layers. We can make models, define layers, or set up multiple input-output models that are handled using the Keras high-level API.



Initially, Keras was developed as part of the research effort associated with the **Open-Ended Neuro-Electronic Intelligent Robot Operating System (ONEIROS)** project. Click on the following link to find out more: <http://keras.io/>.

Keras has attracted lots of attention in recent years as it is open source and, most importantly, is being actively developed by contributors from all over the world. The documentation related to Keras is endless. Yes, we may have understood the documentation, but what we need to know is how Keras performs. Since it is an API that is used to specify and train differentiable programs, high performance naturally follows.

To understand more about Keras, we need to check out some of the contributors and backers of the deep learning framework. Keras had over 4,800 contributors during its launch, and this number has since risen to 250,000 active developers. Growth has doubled every year since its launch. It gets a lot of attention from start-ups, and companies such as Microsoft, Google, NVIDIA, and Amazon actively contribute to its development.

Now, let's understand who uses Keras. It has been used in the **machine learning (ML)** development of popular firms such as Netflix, Uber, Google, Huawei, NVIDIA, and Expedia. So, the next time you watch any movie on Netflix or book an Uber ride, you will know what sort of thing Keras is being used for.

In the next section, we will look into the advantages of Keras. We are going to implement most of our projects using Keras, so this is important to know.

Advantages of Keras

Keras follows the best practices associated with reducing cognitive load. It offers simple and consistent APIs and affords us the freedom to design our own architecture.

Keras provides clear feedback on user error, which minimizes the number of user actions required. It provides high flexibility as it integrates with lower-level deep learning languages such as TensorFlow. You can implement anything that was built in the base language.

Keras also supports various programming languages. We can develop Keras in Python, as well as R. We can also run the code with TensorFlow, CNTK, Theano, and MXNet, which can be run on the CPU, TPU, and GPU as well. The best part is that it supports both NVIDIA and AMD GPUs. These advantages offered by Keras ensure that producing models with Keras is really simple. It can run with TensorFlow Serving, GPU acceleration (web Keras, Keras.js), Android (TF, TF Lite), iOS (Native CoreML), and Raspberry Pi.

In the next section, we are going to learn about the working principle of Keras.

The working principle behind Keras

The main idea behind the Keras development is to facilitate experimentation's by fast prototyping. The is great to go from an idea to result with the least possible delay is key to good research. The structure in Keras is the Model that defines the complete graph of a network. To create a custom model for a project, we simply add more layers to the existing model.

Let's look at the model architecture in Keras in the following screenshot:

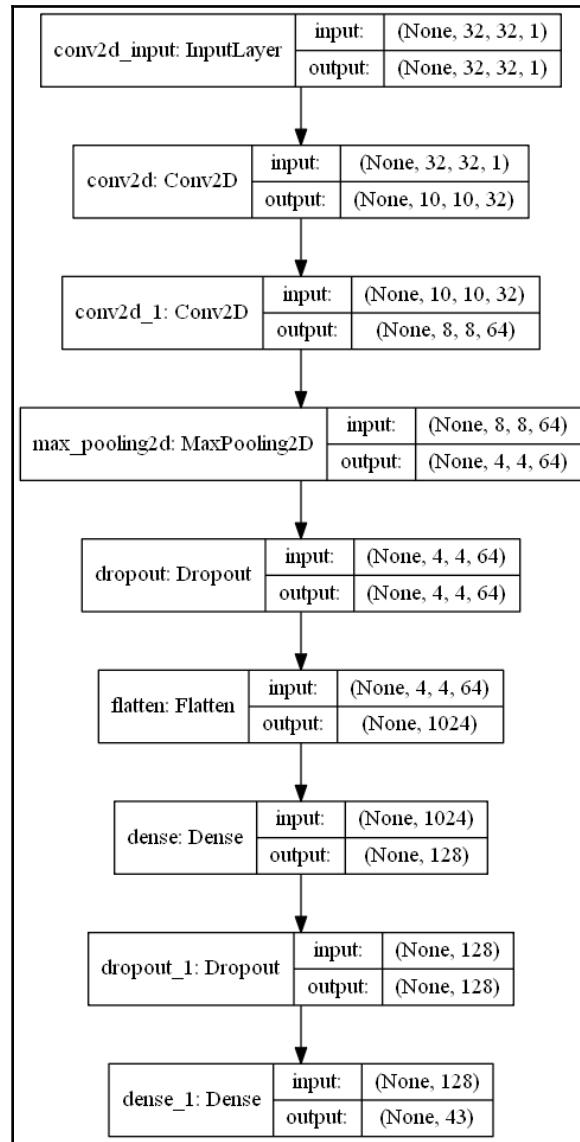


Fig 3.1: Model architecture in Keras

Keras relies on its backend for low-level operations like convolutions and tensor products. While Keras supports several backend engines, its default backend is TensorFlow, with Google as its primary supporter.

In the next section, we will learn about the sequential model and the functional model of Keras.

Building Keras models

There are two major models that Keras offers, as follows:

- The sequential model
- The functional model

We'll look at both in more detail in the following subsections.

The sequential model

The sequential model is like a linear stack of layers. It is useful for building simple models, such as the simple classification network and encoder-decoder models. It basically treats the layer as an object that is fed into the next layer.



For most problems, the sequential API lets you create layer-by-layer models. It restricts us from creating models that share layers or have multiple inputs or outputs.

Let's look at the Python code for this:

1. Let's begin by importing the key Python libraries:

```
In[1]: import tensorflow as tf  
In[2]: from tensorflow import keras  
In[3]: from tensorflow.keras import layers
```

2. We will define the model as a sequential model (In[4]) and then add a flatten layer. With the hidden layer, we have 120 neurons (In[6], In[7]), and the activation function is **Rectified Linear Units (ReLU)**. In[8] is the last layer as it has 10 neurons and a softmax function; it turns logits into probabilities that sum to one:

```
In[4]: model = tf.keras.Sequential()  
  
In[5]: model.add(tf.keras.layers.Flatten())  
  
In[6]: model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))  
  
In[7]: model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))  
  
In[8]: model.add(tf.keras.layers.Dense(10,  
activation=tf.nn.softmax))
```

3. Finally, `model.compile` (In[9]) and `model.fit` (In[10]) are used to train the network. Here, there are 10 epochs and the batch size is 32:

```
In[9]: model.compile(optimizer='adam',  
                    loss='sparse_categorical_crossentropy',  
                    metrics=['accuracy'])  
In[10]: model.fit(x, y, epochs=10, batch_size=32)
```



An **epoch** is when a training dataset is passed forward and backward through the neural network once, while the **batch size** is the number of training examples in one forward and backward pass. So, the higher the batch size, the more memory we need.

The functional model

The functional model is the more widely used of the two models. The key aspects of such a model are as follows:

- Multi-input, multi-output, and arbitrary static graph topologies
- Multi-input and multi-output models
- The complex model, which forks into two or more branches
- Models with shared layers



The functional API allows you to create models that are much more versatile as you can easily identify models that link layers to more than just the previous and next layers. You can actually connect layers to any other layer and create your own complex layer.

The following steps are similar to the sequential model's implementation, but with a number of changes. Here, we'll import the model, work on its architecture, and then train the network:

```
In[1]: import tensorflow as tf  
In[2]: from tensorflow import keras  
In[3]: from tensorflow.keras import layers  
  
In[4]: inputs = keras.Input(shape=(10,))  
In[5]: x= layers.Dense(20, activation='relu')(x)  
In[6]: x=layers.Dense(20, activations='relu')(x)  
In[7]: outputs = layers.Dense((10, activations='softmax')(x)  
  
In[8]: model = keras.Model(inputs, outputs)  
In[9]: model.fit(x, y, epochs=10, batch_size=32)
```

In the functional model, we have a concept called **domain adaption**. What we used to do is train on domain A and test on domain B. The results used to be poor for the test dataset. The solution was to adapt the model to both domains. This can be done using Keras.

Now, let's discuss the types of executions in Keras.

Types of Keras execution

There are two types of execution in Keras:

- Deferred (symbolic) execution
- Eager (imperative) execution

In deferred execution, we use Python to build a computation graph first, and then it's compiled so that it can be executed later. In eager execution, the Python runtime itself becomes the execution runtime for all the models.

In the next section, we will learn about deep learning using Keras. We will also look into visual recognition challenges, due to which deep learning became popular.

Keras for deep learning

Deep learning started to gain popularity a couple of years ago when AlexNet, a **convolutional neural network (CNN)** designed by Alex Krizhevsky and published with Ilya Sutskever and doctoral adviser Geoffrey Hinton, also referred to as the godfather of deep learning, was created. AlexNet blew away the ImageNet Large Scale Visual Recognition Challenge on 30 September 2012. Their deep neural network was significantly better than all the other submissions. Architectures such as AlexNet have revolutionized the field of computer vision. In the following diagram, you can see the top five predictions for the visual challenge where AlexNet emerged victorious:

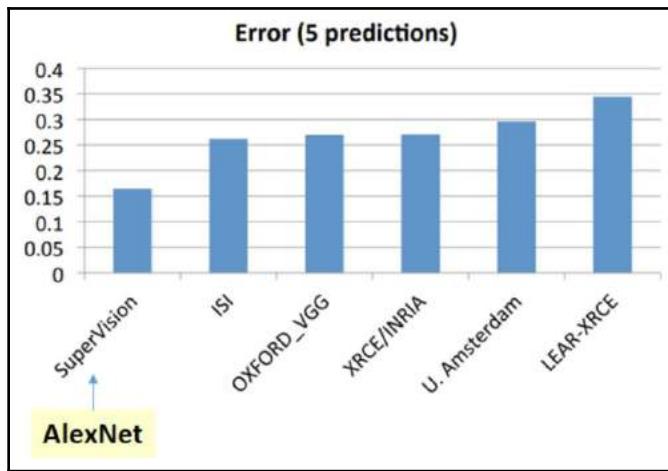


Fig 3.2: Visual Recognition Challenge 2012

Because deep learning requires lots of GPU computation and data, people began to take notice and implemented their own deep neural networks for different tasks, resulting in a deep learning library.

Theano was one of the first widely adopted deep learning libraries. It was maintained by the University of Montreal, but its development was stopped in 2018. Various open source Python deep learning frameworks have been introduced in the past couple of years, and some have acquired a lot of popularity. At the time of writing, the deep learning library that is most widely used is TensorFlow. However, other libraries are gaining popularity as well. PyTorch is an excellent example. It was introduced in January 2018 by Facebook. They ported the popular Torch framework, which was written in Lua, to Python.



The main driver behind the popularity of PyTorch was the fact that it uses a dynamic computation graph, which leads to efficient memory usage. You can read more about PyTorch at <https://pytorch.org/>.

In addition to the main TensorFlow framework, several other companion libraries were released, including the TensorFlow Fold library for dynamic computation graphs and TensorFlow Transform for data input pipelines. The team at TensorFlow also announced **eager execution mode**, which works similarly to PyTorch's dynamic computation graph.

Other tech giants have also been working on creating their own libraries. Microsoft launched CNTK, Facebook launched Caffe2, Amazon launched MXNet, and DeepMind released Sonnet.

Facebook and Microsoft revealed the **Open Neural Network Exchange (ONNX)**, an open format for sharing deep learning models across multiple frameworks. For example, you can train the model in one framework and then serve it in production in another framework, as shown in the following diagram:

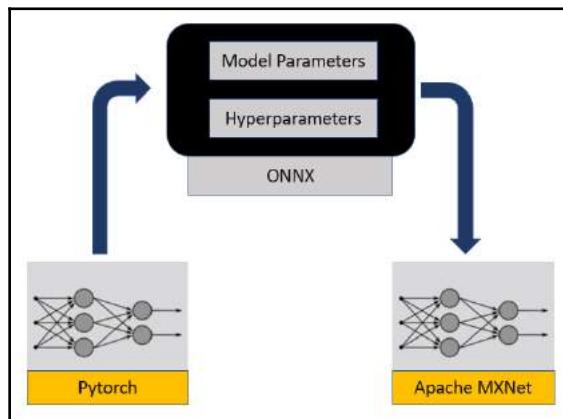


Fig 3.3: ONNX

The best way to understand how the AI concept works is by building your own neural network and then figuring it out as you go along. The best way to do that is by using a high-level library such as Keras. This is effectively an interface that wraps multiple frameworks. We can use it as an interface to TensorFlow, Theano, or CNTK (Microsoft). It works the same, no matter what backend you use.



François Chollet, a deep learning researcher at Google, created and maintains Keras. Google has announced that Keras has been chosen as the official high-level API of TensorFlow. PyTorch is a faster option when it comes to writing and debugging custom modules and layers, while Keras is the faster option when you need to quickly train and test a model built from standard layers.

Using Keras, let's look at what the pipeline for building a deep network looks like:



Fig 3.4: Deep learning pipeline

In the Keras pipeline, we define a network, compile it, fit it, and evaluate it, and then use it to make a prediction. Consider a simple three-layer deep learning network with an input layer, a hidden layer, and an output layer. Each of these layers is just a matrix operation. Input times add a bias and activate the results, repeat this procedure twice, and obtain a result where a_1, a_2, a_3, a_4, a_5 , are the matrix of the input times and addition of bias. This can be seen in the following diagram:

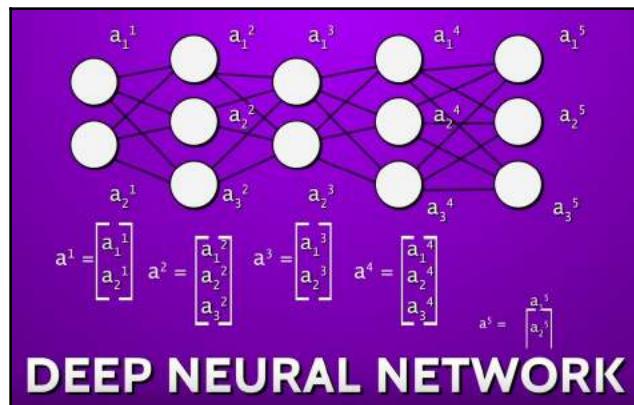


Fig 3.5: Deep learning network

The deep network has multiple hidden layers, which is why it's called **deep**. Deep networks don't use only one type of operation, and there are all sorts of layers out there for different kinds of networks – convolutional layers, dropout layers, current layers... the list goes on. The basic idea of a deep neural network is to apply a series of math operations to the input data. Each layer represents a different operation that then passes the result onto the next layer. So, in a way, we can think of these layers as building blocks. If we can list the different types of layers, we can wrap them in their own class and reuse them as modular building blocks.

Building your first deep learning model

In this section, we will be using Keras with a TensorFlow 2.0 backend to perform our deep learning operations.

We will start with a dataset that contains details regarding the technical specifications of cars. This dataset can be downloaded from the UCI Machine Learning Repository. The data we will be working with in this chapter isn't images. Right now, our focus is on how to use Keras for general machine learning. Once we've learned about CNNs, we can expand Keras so that we can feed image data into the network. This section focuses on learning the basics of building a neural network with Keras.

In the next section, we will concentrate on how to use Keras and its general syntax.

Description of the Auto-Mpg dataset

To begin with, we will use the Auto-Mpg dataset. This dataset can be downloaded from the UCI Machine Learning Repository. The following is some information about this dataset:

- **Title:** Auto-Mpg Data
- **Sources:**
 - **Origin:** This dataset was taken from the StatLib library, which is maintained at Carnegie Mellon University. The dataset was used in the 1983 American Statistical Association Exposition.
 - **Date:** July 7, 1993.

- **Past usage:**
 - See the date in the preceding bullet point.
 - Quinlan, R. (1993). Combining Instance-Based and Model-Based Learning. In *Proceedings on the Tenth International Conference of Machine Learning*, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
- **Relevant information:** This dataset is a slightly modified version of the dataset provided in the StatLib library. In line with its use by Ross Quinlan (1993) in predicting the mpg attribute, eight of the original instances were removed because they had unknown values for the mpg attribute. The original dataset is available in the auto-mpg.data-original file. The data concerns city-cycle fuel consumption in miles per gallon, to be predicted in terms of three multivalued discrete attributes and five continuous attributes. (Quinlan, 1993).
- **Number of instances:** 398.
- **Number of attributes:** Nine, including the class attribute.
- **Attribute information:**
 - **MPG:** Continuous
 - **Cylinders:** Multi-valued discrete
 - **Displacement:** Continuous
 - **Horsepower:** Continuous
 - **Weight:** Continuous
 - **Acceleration:** Continuous
 - **Model year:** Multi-valued discrete
 - **Origin:** Multi-valued discrete
 - **Car name:** String (unique for each instance)
- **Missing attribute values:** horsepower has six missing values.



In the upcoming chapters, we will cover grabbing and working with Auto-Mpg Data with Keras. More information about this dataset can be found at <https://archive.ics.uci.edu/ml/datasets/auto+mpg>.

In the next section, we will import the data and perform a few preprocessing steps.

Importing the data

We are going to start by importing one of the required libraries for this task: NumPy. Let's get started!

1. We are also going to import `pathlib`, `matplotlib`, `Seaborn`, `TensorFlow`, and `keras`. We've already learned about `TensorFlow` and `Keras`. `matplotlib` and `Seaborn` are used for visualization. `pathlib` provides a readable and easier way to build paths. Finally, `pandas` is one of the best data preprocessing libraries available:

```
In[1]: import pathlib  
In[2]: import matplotlib.pyplot as plt  
In[3]: import pandas as pd  
In[4]: import seaborn as sns  
In[5]: import tensorflow as tf  
In[6]: from tensorflow import keras  
In[7]: from tensorflow.keras import layers  
In[8]: from __future__ import absolute_import, division,  
      print_function, unicode_literals
```

2. Now, we will import the data using <https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data>:

```
In[9]: dataset_path = keras.utils.get_file("auto-mpg.data",  
                                         "https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/  
                                         /auto-mpg.data")  
  
In[10]: column_names  
      =['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight',  
          'Acceleration', 'Model Year', 'Origin']  
  
In[11]: raw_dataset = pd.read_csv(dataset_path, names=column_names,  
                               na_values = "?", comment='\t',  
                               sep=" ", skipinitialspace=True)  
In[12]: dataset = raw_dataset.copy()
```

3. When we take a look at the data, we will see that we have various columns of features (**MPG**, **Cylinders**, **Displacement**, **Horsepower**, **Weight**, **Acceleration**, **Model Year**, and **Origin**):

MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	Origin
27.0	4	140.0	86.0	2790.0	15.6	82	1
44.0	4	97.0	52.0	2130.0	24.6	82	2
32.0	4	135.0	84.0	2295.0	11.6	82	1
28.0	4	120.0	79.0	2625.0	18.6	82	1
31.0	4	119.0	82.0	2720.0	19.4	82	1

Fig 3.6: Auto-Mpg dataset

4. Creating a one-hot encoding for **Origin** as the **Origin** column is a form of categorical data. Here, we have three locations: USA, Europe, and Japan. The code block for this is as follows:

```
# We had around 6 NA values in our data, and we are dropping NA
values
In[13]: dataset = dataset.dropna()

# 'Origin' column is categorical so creating one hot encoding

In[14]: origin = dataset.pop('Origin')
In[15]: dataset['USA'] = (origin == 1)*1.0
In[16]: dataset['Europe'] = (origin == 2)*1.0
In[17]: dataset['Japan'] = (origin == 3)*1.0
In[18]: dataset.tail()
```

The following is what the dataset looks like after performing one-hot encoding on the **Origin** column:

MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	USA	Europe	Japan
27.0	4	140.0	86.0	2790.0	15.6	82	1.0	0.0	0.0
44.0	4	97.0	52.0	2130.0	24.6	82	0.0	1.0	0.0
32.0	4	135.0	84.0	2295.0	11.6	82	1.0	0.0	0.0
28.0	4	120.0	79.0	2625.0	18.6	82	1.0	0.0	0.0
31.0	4	119.0	82.0	2720.0	19.4	82	1.0	0.0	0.0

Fig 3.7: Dataset with the origin

In the next section, we will split the dataset into train and test datasets.

Splitting the data

It's time to split the data into train/test sets. Bear in mind that sometimes, people like to split their data three ways; train, test, and validation. For now, though, we'll keep things simple and just use train and test.

First, we will split the data into `train_data` and `test_data`. We are going to use `train_data` for training and `test_data` for prediction. We are going to have an 80-20 split:

```
In[19]: train_data = dataset.sample(frac=0.8, random_state=0)

In[20]: test_data = dataset.drop(train_dataset.index)
```

Now, we will separate the MPG label from the train and test data:

```
In[21]: train_labels = train_data.pop('MPG')

In[22]: test_labels = test_data.pop('MPG')
```

In the next section, we will normalize the dataset as this helps us improve the performance of the model.

Standardizing the data

Usually, when we use neural networks, we get improved performance when we standardize the data. Standardization just means normalizing the values so that they all fit between a certain range, such as 0 to 1 or -1 to +1.



The `scikit-learn` library also provides a nice function for this. Click on the following link for more information: <http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html>.

There is one more way to normalize the data: by using mean and standard deviations. The `normalization` function in the following code can standardize the data for better performance:

```
In[23]: def normalization(x):
    return (x - train_stats['mean']) / train_stats['std']
In[24]: normed_train_data = normalization(train_dataset)
In[25]: normed_test_data = normalization(test_dataset)
```



Previously, we performed standardization using mean and standard deviation. In general, it is recommended to normalize the data in one go and split it into train and test sets.

In the next section, we will start building the model. After that, we will compile and run it.

Building and compiling the model

Now, let's build a simple neural network.

In this section, we will add the layers we'll use in our deep learning model. Let's get started:

1. First, we will import tensorflow, keras, and layers:

```
In[26]: import tensorflow as tf  
In[27]: from tensorflow import keras  
In[28]: from tensorflow.keras import layers
```

2. Now, we can build our model. First, we are going to use `Sequential()` with two hidden layers and output a single continuous value. We have a wrapper function called `model_building` for this. When we compile the model, we need to choose a loss function, an optimizer, and accuracy metrics. We used RMSprop as the optimizer, `mean_square_error` as the loss function, and `mean_absolute_error` and `mean_square_error` as the required metrics. **Mean Squared Error (MSE)** is a common loss function used for regression problems. Evaluation metrics for regression is a **Mean Absolute Error (MAE)**:

```
In[29]: def model_building():  
    model = keras.Sequential([  
        layers.Dense(64, activation=tf.nn.relu,  
        input_shape=[len(train_dataset.keys())]), layers.Dense(64,  
        activation=tf.nn.relu), layers.Dense(1)]  
  
    optimizer = tf.keras.optimizers.RMSprop(0.001)  
  
    model.compile(loss='mean_squared_error',  
                  optimizer=optimizer,  
                  metrics=['mean_absolute_error',  
                           'mean_squared_error'])  
    return model  
In[30]: model = model_building()
```



`compile` defines the loss function, the optimizer, and the metrics. It will not change the weights, and we can compile a model as many times as we want without causing any problems to the pre-trained weights.

Now, we will check the model summary:

```
In[31]: model.summary()
```

The model summary looks as follows:

Layer (type)	Output Shape	Param #
<hr/>		
dense_9 (Dense)	(None, 64)	640
dense_10 (Dense)	(None, 64)	4160
dense_11 (Dense)	(None, 1)	65
<hr/>		
Total params: 4,865		
Trainable params: 4,865		
Non-trainable params: 0		

Fig 3.8: Model summary

In the next section, we will train and save the model.

Training the model

In this section, we will train the model.

Here, we can see that `model.fit` helps us start the training process:

```
# Display training progress by printing a single dot for each completed
epoch
In[32]: class PrintDot(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        if epoch % 100 == 0: print('')
        print('.', end='')

In[33]: EPOCHS = 1000

In[34]: history = model.fit(normed_train_data, train_labels,
    epochs=EPOCHS, validation_split = 0.2, verbose=0,
    callbacks=[PrintDot()])
```

Now, we will visualize the model's training progress:

```
In[35]: hist = pd.DataFrame(history.history)
In[36]: hist['epoch'] = history.epoch
In[37]: hist.tail()
In[38]: def plot_training_history(history):
    hist = pd.DataFrame(history.history)
    hist['epoch'] = history.epoch

    plt.figure()
    plt.xlabel('Epoch')
    plt.ylabel('Mean Abs Error [MPG]')
    plt.plot(hist['epoch'], hist['mean_absolute_error'],
              label='Train Error')
    plt.plot(hist['epoch'], hist['val_mean_absolute_error'],
              label = 'Val Error')
    plt.ylim([0,5])
    plt.legend()

    plt.figure()
    plt.xlabel('Epoch')
    plt.ylabel('Mean Square Error [${MPG}^2$]')
    plt.plot(hist['epoch'], hist['mean_squared_error'],
              label='Train Error')
    plt.plot(hist['epoch'], hist['val_mean_squared_error'],
              label = 'Val Error')
    plt.ylim([0,20])
    plt.legend()
    plt.show()

In[39]: plot_training_history(history)
```

In the following image, we have visualized the training progress:

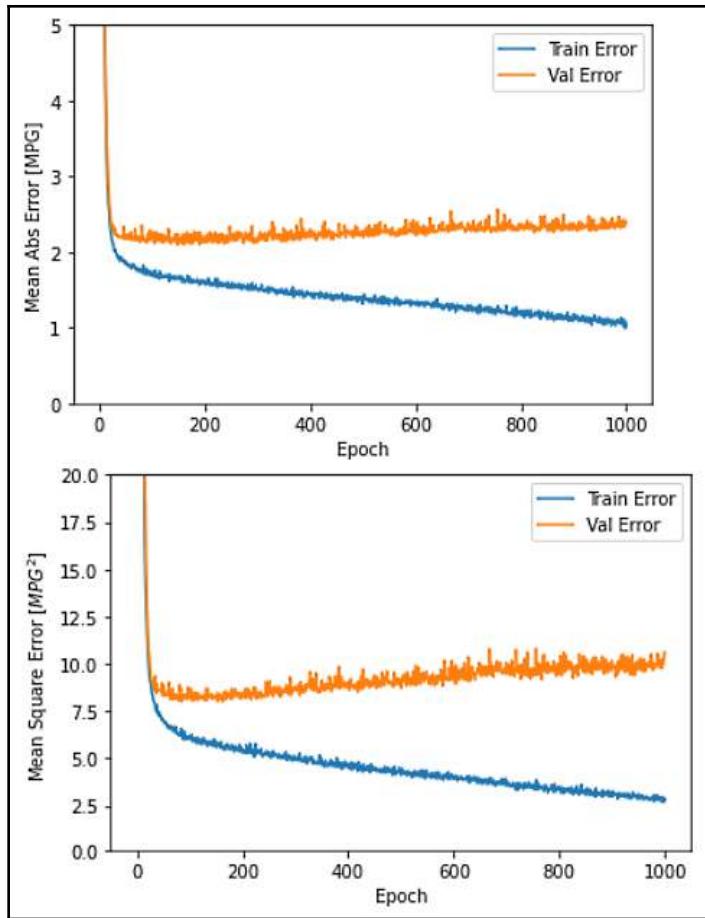


Fig 3.9: Training the model

As we can see, there's been an improvement as well as degradation in terms of validation loss. There is a way by which we can automatically stop the training process when the validation score is not improving: **callbacks**. This is a set of functions that can be applied at given stages of the training procedure. Here, we will use callbacks as an early stopping mechanism; this will help us avoid overfitting by terminating training early. The code for this is as follows:

```
model = model_building()  
  
# The patience parameter is the amount of epochs to check for improvement  
early_stop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)
```

```
history = model.fit(normed_train_data, train_labels, epochs=EPOCHS,
                     validation_split = 0.2, verbose=0,
                     callbacks=[early_stop, PrintDot()])

plot_history(history)
```

The following graph shows that, on the validation set, the average error is usually around +/- 2 MPG:

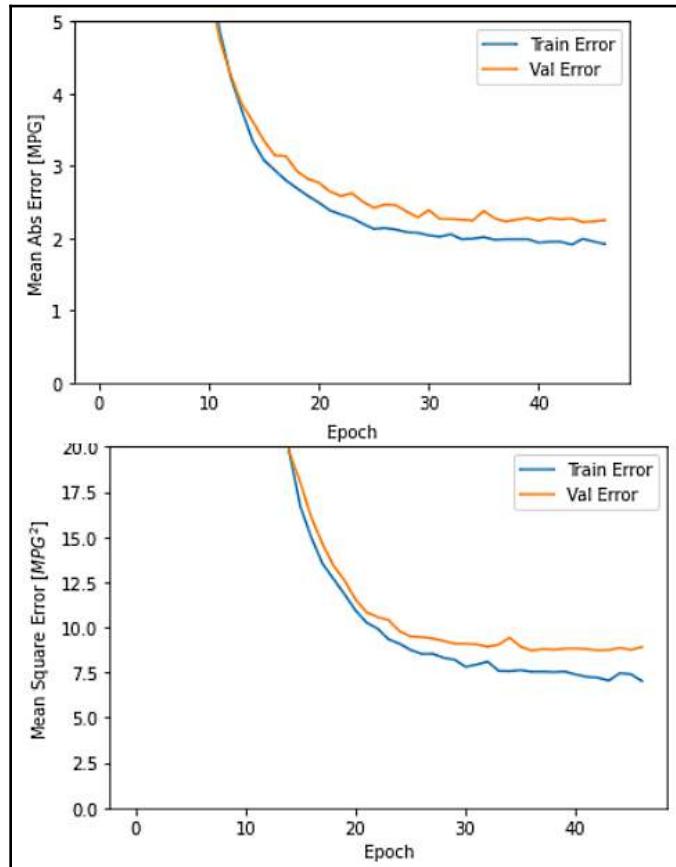


Fig 3.10: Training and test errors after using callbacks for early stopping

Let's check out the resulting code:

```
loss, mae, mse = model.evaluate(normed_test_data, test_labels, verbose=0)
print("Testing dataset Mean Abs Error(MAE): {:.5.2f} MPG".format(mae))
Testing dataset set Mean Abs Error(MAE): 1.96 MPG
```

Here, we can see the value of MAE is 1.96, which isn't bad.

With that, we have trained the model. In the next section, we will make some predictions using unseen data.

Predicting new, unseen data

Now, let's see how we did by making predictions on the test data of the Duto-Mpg dataset. Remember, our model has **never** seen the test data that we scaled previously! This process is the same process that you would use on brand-new data.

Let's look at the test data that we've just analyzed:

```
test_predictions = model.predict(normed_test_data).flatten()

plt.scatter(test_labels, test_predictions)
plt.xlabel('True Values [MPG]')
plt.ylabel('Predictions [MPG]')
plt.axis('equal')
plt.axis('square')
plt.xlim([0,plt.xlim()[1]])
plt.ylim([0,plt.ylim()[1]])
_ = plt.plot([-100, 100], [-100, 100])
plt.show()
```

The scatter plot between the predicted and true values shows the error in the model:

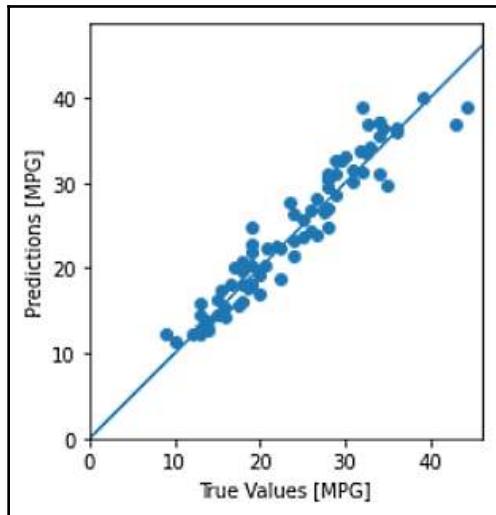


Fig 3.11: True values versus predicted values

In the next section, we will evaluate the performance of the model.

Evaluating the model's performance

So, how well did we do? How do we actually measure how well we did? It all depends on the situation.

Let's evaluate our model by plotting the error counts:

```
error = test_predictions - test_labels
plt.hist(error, bins = 25)
plt.xlabel("Prediction Error [MPG]")
_ = plt.ylabel("Count")
plt.show()
```

Now, let's view the output:

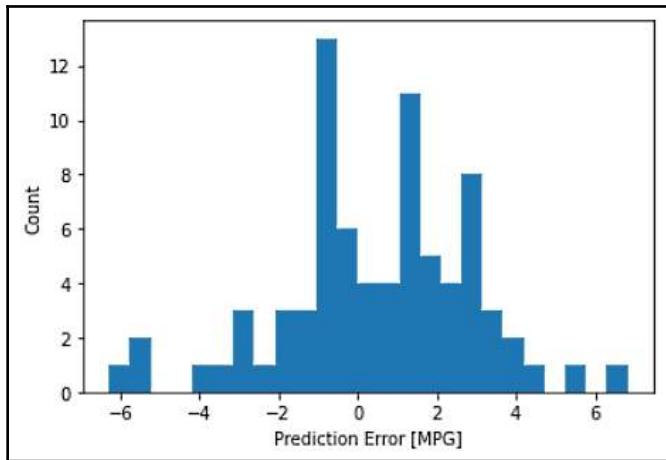


Fig 3.12: Count of predicted errors in the model

It looks like the model predicted reasonably well. The distribution error of the model shows it is not quite Gaussian or normally distributed, but we can expect non-Gaussian as the number of samples is very small.

Saving and loading models

Now that we have trained the model, we need to save and load it. Let's get started:

1. We will start by saving the model:

```
In[31]: model.save('myfirstmodel.h5')\
```

2. Next, we will import the model:

```
In[32]: from keras.models import load_model  
In[33]: newmodel = tf.keras.models.load_model('myfirstmodel.h5')
```

3. Finally, we will predict the imported model:

```
In[34]: test_predictions =  
model.predict(normed_test_data).flatten()
```

With that, you have implemented your first deep learning model!

Summary

In this chapter, we began by understanding the basics of Keras and saw why Keras is so useful. We learned about the types of Keras execution, and we also built our first deep learning model step by step. We went through the different steps of building a model: importing data, splitting data, normalizing data, building the model, compiling the model, training the model, predicting unseen data, evaluating model performance, and finally saving and loading the model.

In the next chapter, we are going to learn about computer vision techniques.

2

Section 2: Deep Learning and Computer Vision Techniques for SDC

This section of the book focuses on learning about advanced computer vision techniques used in the self-driving car field. We will learn about image preprocessing and feature extraction techniques using OpenCV. We will deep dive into **convolutional neural networks (CNNs)** and implement multiple image classification models using Keras.

This section comprises the following chapters:

- Chapter 4, *Computer Vision for Self-Driving Cars*
- Chapter 5, *Finding Road Markings Using OpenCV*
- Chapter 6, *Improving the Image Classifier with CNNs*
- Chapter 7, *Road Sign Detection Using Deep Learning*

4

Computer Vision for Self-Driving Cars

The major players in the autonomous driving industry use a camera as the primary sensor in their vehicle sensor suite. Cameras are rich sensors that capture incredible detail about the environment around the vehicle, but they require extensive processing to make use of the information that's captured. Over the course of this book, you will get hands-on experience of how to algorithmically manipulate camera images to extract information that is useful for autonomous driving.

Of all the common self-driving car sensors, the camera is the sensor that provides the most detailed visual information about objects in the environment. Information about the appearance of the surrounding environment is particularly useful for tasks requiring an understanding of the scene, such as object detection, semantic segmentation, and object identification. This appearance information is what allows us to distinguish between road signs, traffic light states, and track turn signals, and to resolve overlapping vehicles in an image into separate instances. Because of its high-resolution output, the camera is able to collect and provide orders of magnitude more information than other sensors used in self-driving while still being relatively inexpensive. The combination of high-value appearance information and a low cost makes a camera an essential component of our sensor suite.

In this chapter, we are going to cover the fundamentals of computer vision. We will learn how to represent an image, how to obtain features within images, and how to obtain the gradient of an image. Learning about images is very important when dealing with camera sensors.

In this chapter, we will cover the following topics:

- Introduction to computer vision
- Building blocks of an image
- Color space techniques
- Introduction to convolutions

- Edge detection and gradient calculation
- Image transformation

Introduction to computer vision

Computer vision is a science that is used to make computers understand what is happening within an image. Some examples of the use of computer vision in self-driving cars are the detection of other vehicles, lanes, traffic signs, and pedestrians. In simple terms, computer vision helps computers understand images and videos, and determines what the computer is seeing in the surrounding environment.

The following screenshot shows how a human sees the world:

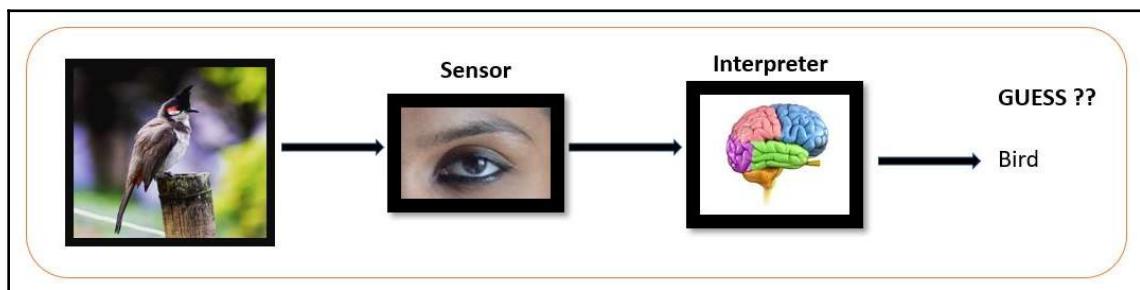


Fig 4.1: Human eye interpretation

In the preceding screenshot, we can see that humans see using their eyes. The visual information captured by their eyes is then interpreted in the brain, enabling the individual to conclude that the object is a bird. Similarly, in computer vision, the camera takes the role of the human eye and the computer takes the role of the brain, as shown in the following screenshot:

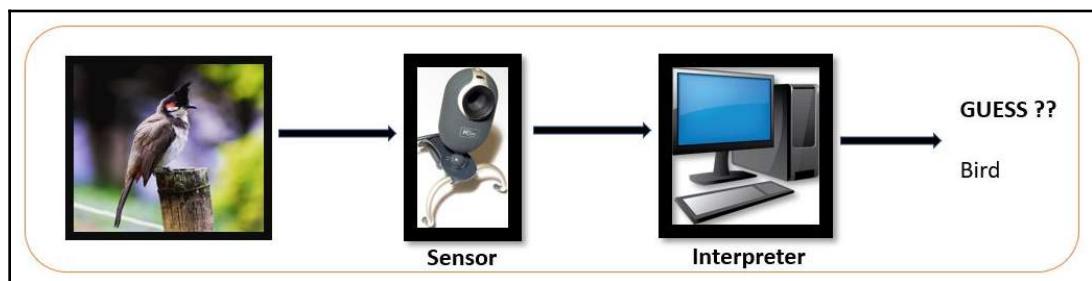


Fig 4.2: Computer interpretation

Now the question is, what process actually happens in computer vision, and how does a computer mimic the human brain? It is actually based on the same principles human vision follows: we extract the image using a camera and develop a machine learning algorithm within the computer that helps us to classify images.

Computer vision is not only applicable in self-driving cars, but also in facial recognition, object detection, handwriting recognition, license-plate number detection, medical imaging, image reconstruction, image synthesis, image-style transfer, and object segmentation, among other things.

In the next section, we will learn about the challenges in the field of computer vision.

Challenges in computer vision

We will look at the following challenges in computer vision:

- Viewpoints
- Camera limitations
- Lighting
- Scaling
- Object variation

Viewpoints: The first challenge is that of viewpoints. In the following screenshot, both pictures are of roads, but the viewpoints are different. Therefore, it is very difficult to take these images and get our computer to generalize and become smart enough to be able to detect all roads from every viewpoint. This is very simple for humans, but for computers, it is a little more challenging. We will see more challenges like this in *Chapter 5, Finding Road Markings Using OpenCV*:



Fig 4.3: Viewpoints

Camera limitations: The next challenge is **camera limitations**. A high-quality camera results in better quality pictures. Since image quality is measured in pixels, we can say that the higher the number of pixels, the better the camera.

Lighting: Lighting is also a challenge. The following photos show roads in light and dark conditions. In daylight, machine learning algorithms might help us to classify a road, but during the night, or when it's foggy, the features will not be the same, and it will be more difficult for the algorithm to identify them as roads. We want our cars to be able to drive safely in the day as well as the night, or when it's foggy. Let's look at a comparison of images of roads in foggy and sunny conditions in the following screenshot:



Fig 4.4: Foggy and sunny lighting

Scaling: In cases where we are dealing with zoomed-in images, the biggest challenge is scaling. The scaling of the image refers to the resizing of the image. Most machine learning algorithms expect inputs to be in the same dimensions, so we have to rescale the images accordingly to fit this expectation. But if the image is already zoomed-in, we lose the clarity of the image.

Object variation: Object variation is all about the differences in the structure of the object. For example, we can classify all the objects in the following photos as chairs, but for a machine learning algorithm, it will be difficult to classify all the different possible labels for the various types of chair. In the following screenshot, we can see a variety of chairs:

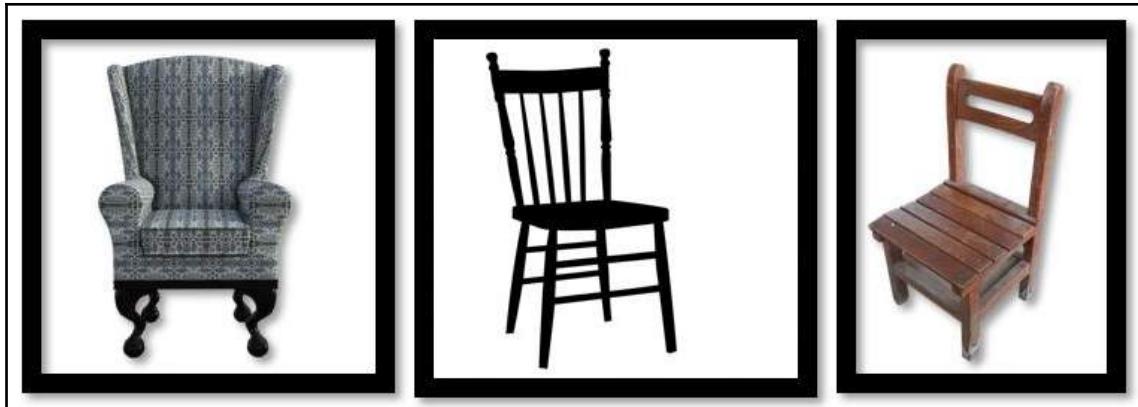


Fig 4.5: Object variation

In the next section, we will learn about the requirements of the camera sensors for self-driving cars.

Artificial eyes versus human eyes

In this section, we will compare the requirements of artificial eyes with human eyes. In the following table, we can see the differences between the requirements of artificial eyes for self-driving cars and the capabilities of human eyes:

Self-Driving Car Requirement	Human Eye
It requires 360-degree coverage around the vehicle.	It has 3D vision for 130 degrees of the field of view, resulting in a blind spot . Humans can turn their heads and bodies to mitigate this.
It must identify 3D objects that are close to and far from the vehicle.	The human eye's high resolution extends only to the central 50 degrees in the field of view. Outside the central zone, perception drops.
It must process real-time data.	In the human eye, the frame is good in the central zone and poor in the peripheries.
It should be able to work well in all lighting and weather conditions.	Human eyes perform well in various lighting conditions, but to see in the dark, human eyes depend on headlights in cars.

The preceding table ignores age, disease, and the cognitive abilities of the individual that influence their processing ability/time. The conclusion of this is that the human eye is particularly good in a few conditions, but artificial eyes, or cameras, require a lot of enhancement to be comparable with the human eye.

We have already learned about the sensors that allow self-driving cars to perceive the world. Some examples of these sensors are cameras, LIDAR, RADAR, and GPS, all of which collect data. A machine learning algorithm can then process the sensor data to take action. Let's briefly take a look at these sensors:

- **Camera sensors:** These sensors provide detailed images, but require deep learning to interpret 2D images.
- **Light Detection and Ranging (Lidar):** This sensor works like radar, but instead of sending radio waves, it emits infrared laser beams and measures how long they take to come back. It creates a 3D map of the world in real time.
- **Radar:** This sends radio waves to detect objects. It is cheaper than lidar.
- **Global Positioning System (GPS):** This provides information about the current location of the car on the road using HD maps.

In the next section of the chapter, we will learn about the building blocks of an image.

Building blocks of an image

In this section, we will learn the fundamentals of how to represent an image in a digital format and how to use images in a better way in the machine learning world for tasks such as image manipulation.

We will start by looking at how humans see color. Let's assume we have a yellow box. The brain can see the color yellow. The light waves that are observed by the human eye are translated into color by the visual cortex of the brain. When we look at a yellow box, the wavelengths of the reflected light determine what color we see. The light waves reflect off the yellow box and hit our eyes with a wavelength of 570 to 580 nanometers (the wavelength of yellow light).

In the next section, we will read about the digital representation of images. We are going to use the OpenCV library to process an image.

Digital representation of an image

Now we will see how to represent an image digitally. We will start with grayscale images. A grayscale picture is one where shades of gray are the only colors in the image. The grayscale image is a simple form of the image, and so is easy to process with multiple applications. These are also known as black and white images. Let's look at an image of a car. This image is stored digitally in the form of pixels:



Fig 4.6: Grayscale image

Each pixel has a number in it that ranges from 0 to 255. If a pixel's value is zero, that means the color is black. If its value is 255, it will be white. As this number increases, so does the pixel's brightness. In the following screenshot, we can see that the black pixels contain the number 0 and the white pixels contain the number 255. We can see pixels that are gray too, which occurs between the numbers 0 and 255. This is essentially how we represent the image in a decimal number format. Now let's assume we want to save an image in binary form, which is easily understood by computers. In binary form, 0 is equal to 00000000, and 255 is equal to 11111111. We can see a comparison between the pixel and binary values in the following screenshot:

255	255	255	11111111	11111111	11111111
155	155	155	10011011	10011011	10011011
0	0	0	00000000	00000000	00000000

Fig 4.7: Pixel range

Now we will learn how to represent an image in a colored format. In grayscale, we have one layer. This layer has a number of pixels, and each pixel is numbered from 0 to 255. For colored images, we don't have one layer: we have three layers consisting of three colors, namely red, green, and blue. In the colored format, each pixel coordinate contains three values ranging from 0 to 255, and each pixel has 8 bits. All images can be represented as a mixture of red, blue, and green.

We will take one image and divide it into several pixels, with each pixel being a different layer. We have red, green, and blue (RGB) channels, as shown in the following diagram. We will start mixing them together to come up with new colors. For example, when we mix red and green, we get yellow, and by mixing blue and red, we get pink. This is how we come up with the different colors that our eyes can see.

In the following screenshot, there are three color channels. All the colors in the photo are formed using red, green, and blue channels:

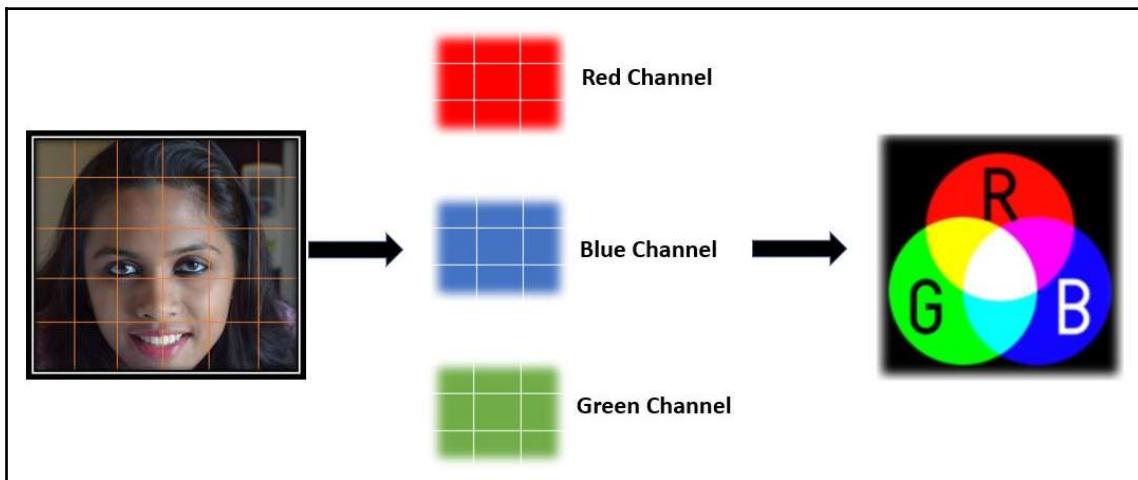


Fig 4.8: Color channel

If we have the same pixel images in both a grayscale and RGB format, then the RGB image is three times bigger than the grayscale image, as the RGB image has three channels.

The following screenshot contains an example of red, blue, and green pixels:



Fig 4.9: RGB values

In the preceding screenshot, we can see that the color red has a **Channel 1** value of 255, while **Channel 2** is 0 and **Channel 3** is 0. Similarly, if **Channel 1** is 0, **Channel 2** is 255, and **Channel 3** is 0, then the color will be green, and if **Channel 1** is 0, **Channel 2** is 0, and **Channel 3** is 255, the color will be blue.

We can also create a new color by mixing these channels. For example, if **Channel 1** is 255, **Channel 2** is 255, and **Channel 3** is 0, it will result in the color yellow.

Converting images from RGB to grayscale

In this section, we will use a powerful image-processing library called OpenCV and use it to convert an image to grayscale. We will take a color image of a road, as shown in the following screenshot:



Fig 4.10: Sample image

In the following steps, we will convert the color image into grayscale using the OpenCV library:

1. First, import the matplotlib (mpimg and pyplot), numpy, and openCV libraries:

```
In[1]: import matplotlib.image as mpimg  
In[2]: import matplotlib.pyplot as plt  
In[3]: import numpy as np  
In[4]: import cv2
```

2. Next, import the image for the operation:

```
In[5]: image_color = mpimg.imread('image.jpg')  
In[6]: plt.imshow(image_color)
```

Let's see what our image looks like:



Fig 4.11: Reading an image using matplotlib

3. In the preceding screenshot, the image has three channels because it is in an RGB format. Let's check the size of the image. We can see that the value is (515, 763, 3):

```
In [7]: image_color.shape  
Out [7]: (515, 763, 3)
```

4. Now we will convert it to grayscale. The following OpenCV code will convert the image into grayscale:

```
In [8]: image_gray = cv2.cvtColor(image_color, cv2.COLOR_BGR2GRAY)
In [9]: plt.imshow(image_gray, cmap = 'gray')
```



OpenCV represents RGB images as a multidimensional NumPy array, but in reverse order. This means that images are actually represented as BGR rather than RGB. We can see in the preceding code that the argument is `cv2.COLOR_BGR2GRAY`. Here, BGR is referred to as the Blue, Green, and Red channels.

The grayscale image is as follows:

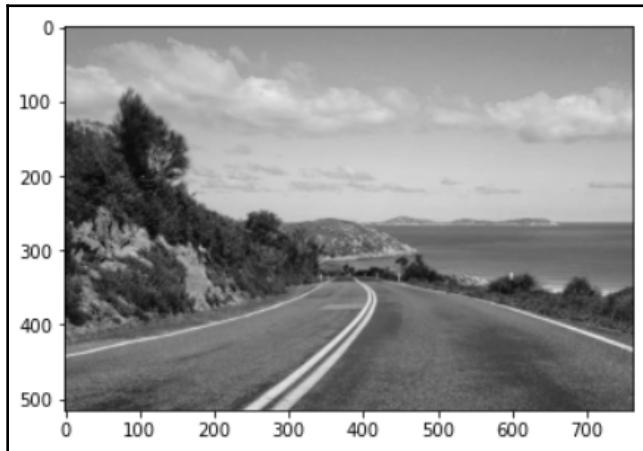


Fig 4.12: Grayscale image

5. As we know, grayscale images use single channels consisting of the colors black and white, which exclusively consist of shades of gray, as shown in the following code block. The values of the channels are (515, 763):

```
In[10]: image_gray.shape
```

```
Out [10]: (280, 660)
```

In the next section, we will learn how to detect lane lines using OpenCV.

Road-marking detection

In this section, we are going to perform image manipulation by highlighting the white sections of the road markings within a grayscale image and a color image. We will start by detecting these sections in the grayscale image.

Detection with the grayscale image

We will start by using OpenCV techniques with the grayscale image:

1. Start by importing the `matplotlib` (`mpimg` and `pyplot`), `numpy`, and `opencv` libraries as follows:

```
In[1]: import matplotlib.image as mpimg  
In[2]: import matplotlib.pyplot as plt  
In[3]: import numpy as np  
In[4]: import cv2
```

2. Next, read the image and convert it into a grayscale image:

```
In[5]: image_color = mpimg.imread('Image_4.12.jpg')  
In[6]: image_gray = cv2.cvtColor(image_color, cv2.COLOR_BGR2GRAY)  
In[7]: plt.imshow(image_gray, cmap = 'gray')
```

We have already seen what the image looks like; it is the grayscale conversion of the color image:

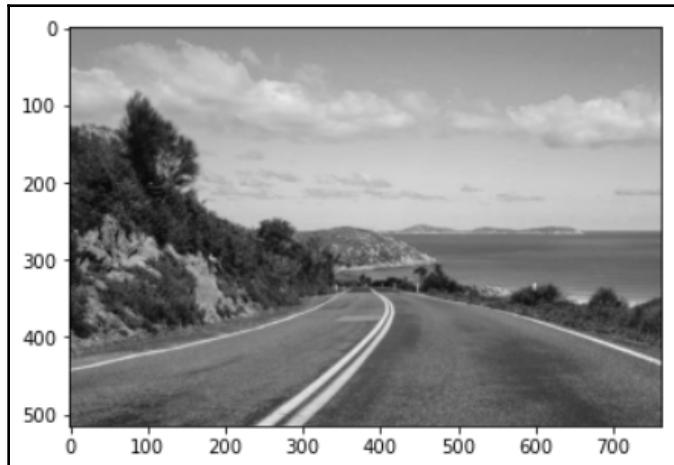


Fig 4.13: Color image to grayscale

3. Now we will check the shape of the image, which is (515, 763):

```
In[8]: image_gray.shape  
Out[8]: (515, 763)
```

4. Now we will apply a filter to identify the white pixels of the image:

```
In[9]: image_copy = np.copy(image_gray)  
  
# any value that is not white colour  
In[10]: image_copy[ (image_copy[:, :] < 250) ] = 0
```

5. Once this processing has completed, we can display the image as follows:

```
In[11]: plt.imshow(image_copy, cmap = 'gray')  
In[12]: plt.show()
```

This yields the following output:

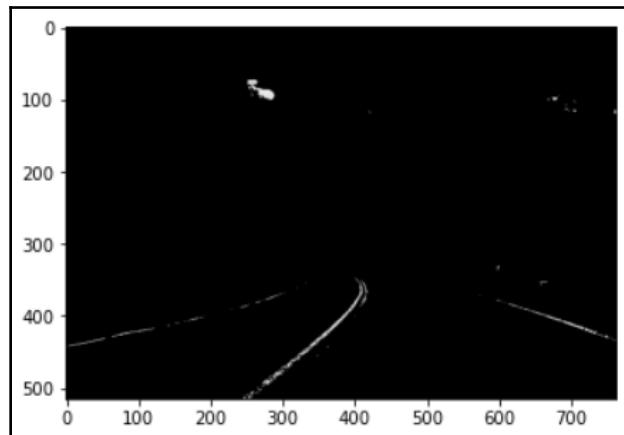


Fig 4.14: Road masking

In the next section, we will perform lane-line detection using an RGB image.

Detection with the RGB image

Now we will find the road markings in an RGB image:

1. First, import the `matplotlib (mpimg and pyplot)`, `numpy`, and `openCV` libraries:

```
In[1]: import matplotlib.image as mpimg  
In[2]: import matplotlib.pyplot as plt  
In[3]: import numpy as np  
In[4]: import cv2
```

2. Read the image as follows:

```
In[5]: image_color = mpimg.imread('image.jpg')
```

We have read the image, and this is what it looks like:



Fig 4.15: Sample image

3. Now we will check the shape of the image, which is $(280, 660, 3)$:

```
In[6]: image_color.shape  
Out[6]: (280, 660, 3)
```

4. Next, we will detect the white lines. We can play with the values in line 8 in the following code to get sharper images. We can set the values of channel-1 to less than 209, *channel 2 to less than 200, and *channel 3 to less than 200. Different images require different values to get sharper images. We will look at the following code:

```
In[7]: image_copy = np.copy(image_color)

# Any value that is not white
In[8]: image_copy[ (image_copy[:, :, 0] < 209) | (image_copy[:, :, 1] <
200) | (image_copy[:, :, 2] < 200) ] = 0
```

5. The following code displays the processed image:

```
In[9]: plt.imshow(image_copy, cmap = 'gray')
In[10]: plt.show()
```

This yields the following output:

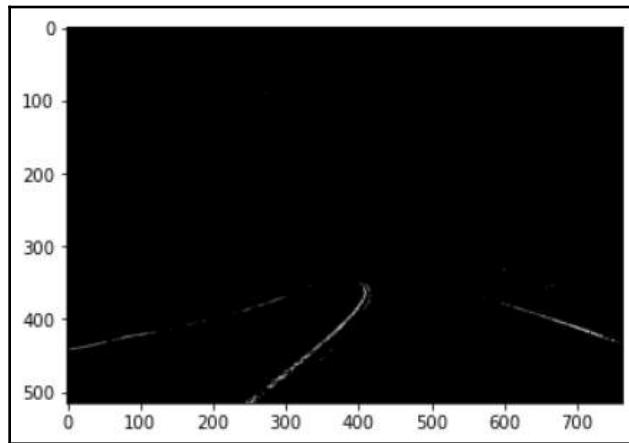


Fig 4.16: Masked image

We can see that the preceding image is more clear than the processed grayscale image. When we process them, color images seem to perform better than grayscale images, as color images have more information than grayscale images. We will see more image transformation techniques in the upcoming sections of the book.

Color space is a mathematical model that describes the range of colors in images. We'll be diving into the color space model once we've learned about the challenges in color selection techniques.

Challenges in color selection techniques

In the previous section, we learned how to extract a specific color from a grayscale and color image, and we also identified road marking pixels. But there are a few challenges that might arise when using these techniques. What if the road markings aren't white? What if it's night time, or the weather is different? These are the challenges that we face when programming self-driving cars.

One of the main challenges is the color-selection techniques. Here, we are required to develop a sophisticated algorithm that will work in all conditions, whether it is night time or snowing. There are, however, ways to overcome this challenge:

- We can use advanced computer vision techniques to extract more features from images, such as edge detection, which we will cover later in this chapter.
- We can use LIDAR to create a high-resolution 3D digital map of the SDC's surroundings. During ideal weather conditions, the LIDAR collects 2.8 million laser points per second, which is used to create a LIDAR map. This level of detail can be helpful during winter or the rainy season because the car can predict the surrounding objects using LIDAR data in circumstances where computer vision fails.

In the next section of the chapter, we will read about color space techniques in detail.

Color space techniques

In this section, we are going to firstly explore different color spaces, which are very important in image analysis for self-driving cars. We will explore the following:

- RGB color space
- HSV color space

The **red green blue (RGB)** color space describes colors in terms of red, green, and blue, whereas the **hue saturation value (HSV)** describes colors in terms of hue, saturation, and value.

The HSV color space is preferable over RGB color space when performing image analytics because it describes colors in a way that is closer to our own perception of color, accounting for vibrancy and brightness rather than simply a combination of primary colors.

In the next section, we will learn about the RGB color space.

Introducing the RGB space

We will start with the most popular color space, RGB. As we know, RGB is made up of the colors red, green, and blue. We can mix them up to produce any color:

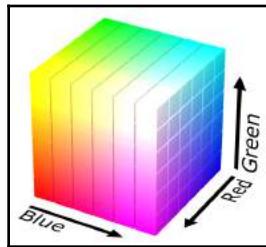


Fig 4.17: RGB color space

You can check the image and the license at https://commons.wikimedia.org/wiki/File:RGB_Cube_Show_lowgamma_cutout_b.png#/media/File:RGB_Cube_Show_lowgamma_cutout_a.



In the OpenCV library, colors are stored in BGR format, not in RGB format. So when we load the images using OpenCV, it will be reversed, starting with blue, moving to green, and ending with red.

The RGB color table for various colors is shown in the following screenshot:

Color	Name	Code
Black		(0,0,0)
White		(255,255,255)
Red		(255,0,0)
Lime		(0,255,0)
Blue		(0,0,255)
Yellow		(255,255,0)
Cyan / Aqua		(0,255,255)
Magenta / Fuchsia		(255,0,255)
Silver		(192,192,192)
Gray		(128,128,128)
Maroon		(128,0,0)
Olive		(128,128,0)
Green		(0,128,0)
Purple		(128,0,128)
Teal		(0,128,128)
Navy		(0,0,128)

Fig 4.18: RGB color table

In the next section, we are going to learn about the HSV color space in detail.

HSV space

HSV stands for hue, saturation, and value (or brightness). The HSV color space can be seen in the following screenshot:

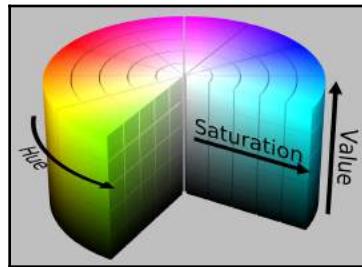


Fig 4.19: HSV color space

You can check the image and the license at https://en.wikipedia.org/wiki/HSL_and_HSV#/media/File:HSV_color_solid_cylinder_saturation_gray.png. In HSV, the color space stores the information in cylindrical format, as can be seen in the preceding screenshot.

The values of HSV are as follows:

- **Hue:** Color value (0–360)
- **Saturation:** Vibrancy of color (0–255)
- **Value:** Brightness or intensity (0–255)

Why should we use HSV color space? The HSV color model is preferred by various designers as HSV has a better representation of color than the RGB color space, which is useful when selecting color or ink. It is easy for people to relate to the colors using the HSV model as images can be seen using the three parameters of color, shade, and brightness.

We can specify the color on the basis of the angle of hue, saturation, and value as shown in *Fig 4.19*. We can see how hue, saturation, and value can be represented in the color space figure:

- **Hue:** The hue is the arrangement of the radial slice around the central axis of neutral colors. It ranges from black at the bottom to white at the top. The hue refers to the color section of the color space, as we can see in *Fig 4.19*. For example, red falls between 0 and 60 degrees, and yellow falls between 61 and 120 degrees.
- **Saturation:** The higher the saturation, the more vibrant the color is. Saturation is, therefore, the amount of gray in the color, ranging from 0 to 100 percent.
- **Value/brightness:** This combination of the hue and saturation describes the brightness of the color. It ranges from 0 to 100 percent.



In OpenCV, the hue ranges from 0–180, saturation from 0–255, and brightness from 0–255.

Color space manipulation

In this section, we will learn how to manually convert RGB to HSV and RGB to grayscale in an image using the OpenCV computer vision library.

Some examples of the conversion from RGB to HSV can be seen in the following screenshot:

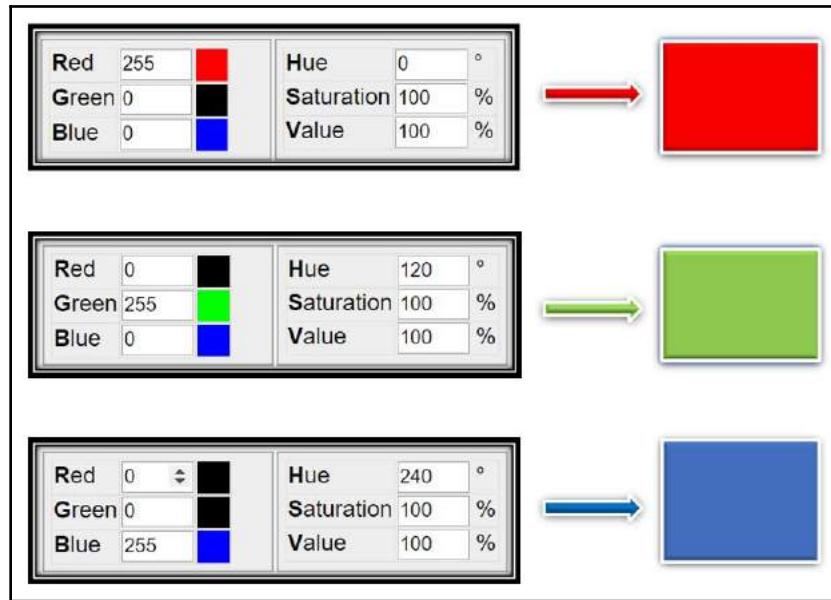


Fig 4.20: RGB to HSV conversion

In the preceding diagram, we can see how the values of the image formats are different in RGB and HSV. For example, red is represented as (255,0,0) in the RGB format and as (0,100,100) in the HSV format.

Next, we will convert RGB to HSV using Python:

1. We are going to use the `matplotlib (pyplot and mpimg)`, `numpy`, and `openCV` libraries, which can be imported as follows:

```
In[1]: import matplotlib.image as mpimg  
In[2]: import matplotlib.pyplot as plt  
In[3]: import numpy as np  
In[4]: import cv2
```

2. Then we will read and display the image using OpenCV:

```
In[5]: image = cv2.imread('Test_image.jpg')
```

3. Now print and check the dimensions of the image. Because it is a color image, it will have three channels:

```
In[6]: image.shape
```

```
(629, 943, 3)
```

4. Now we will check the height and width of the image. `image.shape[0]` is the first element of the image, and similarly, `image.shape[1]` is the second element of the image. These are the values of the height and width of the image respectively:

```
In[7]: print ('Height = ', int(image.shape[0]), 'pixels')
In[8]: print ('Width = ', int(image.shape[1]), 'pixels')
```

The values of the height and width of the image are as follows:

```
Height = 629 pixels
Width = 943 pixels
```

5. In general, *OpenCV uses BGR instead of RGB*. The `waitKey` in the following code allows us to input information when an image window is open. If we leave it blank, it just waits for any key to be pressed before continuing. Next, we will display the input image:

```
In[9]: cv2.imshow('Self Driving Car!', image)
In[10]: cv2.waitKey(0)
In[11]: cv2.destroyAllWindows()
```

The output of the input image is as follows:



Fig 4.21: Output image

6. We will plot this image using the plt command, not the OpenCV library. If we use the plt command, we must therefore swap B for R and R for B:

```
In[12]: plt.imshow(image)
In[13]: image.shape
```

The output is as follows:

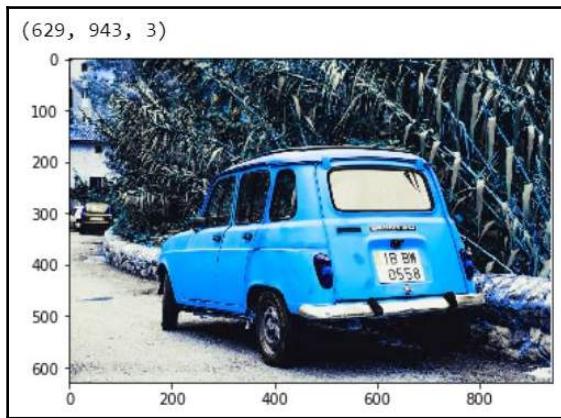


Fig 4.22: Checking the image using matplotlib

Here, we can see that the color has changed. This is because OpenCV uses BGR instead of RGB. To display the image, we imported the image using OpenCV and tried to display it using matplotlib. Because matplotlib follows RGB rather than BGR, the color of the image is different from the original, which is why we should not mix libraries when we are using OpenCV.

7. In this step, we will change the color image into a grayscale image and then check the size of the grayscale image. We can see from the following that we are using cv2.COLOR_BGR2GRAY for the conversion:

```
In[16]: gray_img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
In[17]: cv2.imshow('Self Driving Car in Grayscale!', gray_img)
In[18]: cv2.waitKey()
In[19]: cv2.destroyAllWindows()
```

The output as a grayscale image is as follows:



Fig 4.23: Grayscale image

The size of the grayscale image is as follows. It is a two-dimensional matrix as it has only one channel:

```
In[20]: gray_img.shape  
(629, 943)
```

8. Now, we will convert the RGB image into HSV. The code for converting the RGB into HSV is as shown in the following code. We can see that the image is changed to HSV using cv2.COLOR_BGR2HSV:

```
In[21]: image = cv2.imread('Test_image.jpg')  
In[22]: cv2.imshow('Self Driving Car!', image)  
  
In[23]: hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)  
In[24]: cv2.imshow('HSV Image', hsv_image)  
In[25]: cv2.waitKey()  
In[26]: cv2.destroyAllWindows()
```

The output of the conversion from RGB to HSV is as follows:



Fig 4.24: RGB to HSV conversion

9. Now, we will look at each channel separately. We will start with the hue channel. The hue channel can be seen by selecting channel one, which can be selected by putting 0 in `hsv_image[:, :, 0]`, and similarly, saturation can be selected by selecting channel two and inputting the channel value as 1—that is, `hsv_image[:, :, 1]`:

```
In[27]: plt.imshow(hsv_image[:, :, 0])
In[28]: plt.title('Hue channel')
```

The hue channel's output is as follows:

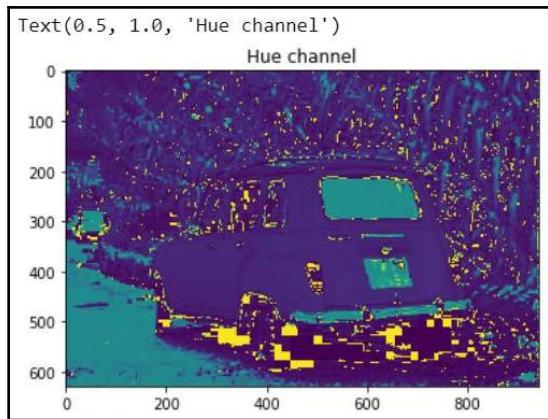


Fig 4.25: Hue channel

10. Next, we will check the saturation channel's output. We will enter the channel value as 1:

```
#Saturation Channel  
  
In[29]: plt.imshow(hsv_image[:, :, 1])  
In[30]: plt.title('Saturation channel')
```

Let's look at the output:

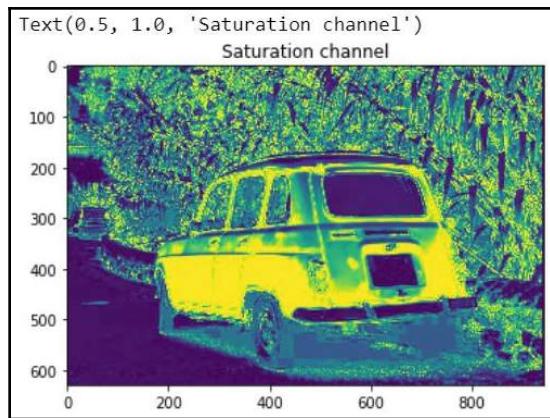


Fig 4.26: Saturation channel

11. Now we will process the image and see the form of the value channel by entering the channel value as 2:

```
#Value Channel  
  
In[31]: plt.imshow(hsv_image[:, :, 2])  
In[32]: plt.title('Value channel')
```

The output of the value channel is as follows:

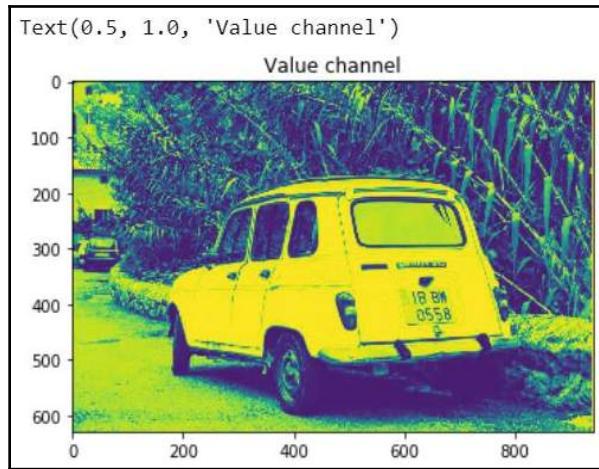


Fig 4.27: Value channel

12. Next, we will split and merge the B, G, and R channels:

```
In[33]: image = cv2.imread('Test_image.jpg')
In[34]: B, G, R = cv2.split(image)
In[35]: B.shape
```

The size of blue is (629, 943). Now we will see the shape of the green channel image:

```
In[36]: G.shape
```

The size of green is (629, 943).

13. We will now learn about the blue channel. Let's see what the blue channel looks like:

```
In[37]: cv2.imshow("Blue Channel!", B)
In[38]: cv2.waitKey(0)
In[39]: cv2.destroyAllWindows()
```

The output of the following image is a bit strange, as it is shown in grayscale; however, it is correct, as now the image has been converted into a one-dimensional channel image, and one-dimensional channel images are always grayscale. In the next section, we are going to create a blue channel image:



Fig 4.28: Blue channel version with one channel

14. Let's try to create our own three-dimensional image out of the blue channel, as we cannot see the blue channel in one dimension. Here, we are going to create the channel with all zeros. Zeros can be created using NumPy, as shown in the following code. Then, we will add a blue channel with two zero channels, (`cv2.merge([B, zeros, zeros])`), as shown in the following code:

```
In[40]: zeros = np.zeros(image.shape[:2], dtype = "uint8")  
  
In[41]: cv2.imshow("Blue Channel!", cv2.merge([B, zeros, zeros]))  
In[42]: cv2.waitKey(0)  
In[43]: cv2.destroyAllWindows()
```

The output of the blue channel is as follows:



Fig 4.29: Blue channel

15. Let's merge the RGB channels to create our original image. We can do this by using the `cv2.merge()` function, given that we already know the value of B, G, and R:

```
In[48]: image_merged = cv2.merge([B, G, R])
In[49]: cv2.imshow("Merged Image!", image_merged)

In[50]: cv2.waitKey(0)
In[51]: cv2.destroyAllWindows()
```

The output of the merged image is as follows:



Fig 4.30: Merged image

16. Let's merge our original image and add more green:

```
In[52]: image_merged = cv2.merge([B, G+100, R])
In[53]: cv2.imshow("Merged Image with some added green!", image_merged)

In[54]: cv2.waitKey(0)
In[55]: cv2.destroyAllWindows()
```

The output of the image with more green is as follows:



Fig 4.31: Image with more green

In this section, we looked at the RGB and HSV color spaces and processed images using OpenCV in Python. These techniques are really helpful in the field of computer vision as one of the steps of image preprocessing.

Now we will explore one of the important topics in image processing: **convolution**.

Introduction to convolution

Convolutions are used to scan an image and apply a filter to obtain a certain feature using a kernel matrix. An image kernel is a matrix that is used to apply effects such as blurring and sharpening. Kernels are used in machine learning for feature extraction—that is, selecting the most important pixels of an image. It also preserves the spatial relationship between pixels.

In the following screenshot, we can see that after applying kernels, the example image is transformed into feature maps:

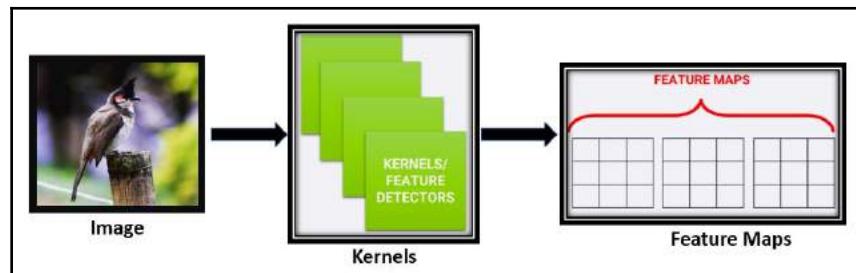


Fig 4.32: Applying kernels

In Fig 4.33, we can see how the convolution works. We have an example of a grayscale image, the blue box is the kernel, and the green box is the final image. In general, the kernel is applied to the entire image and scans the features of the image. Convolution can be used when generating a new image, scaling down the image, blurring the image, or sharpening the image, depending on the value of the kernel we use:

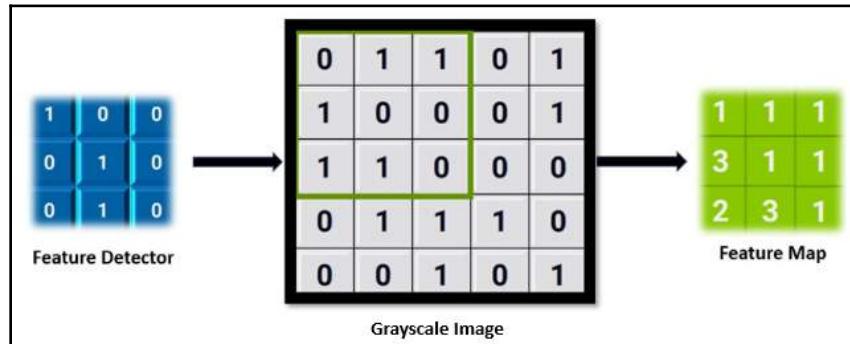


Fig 4.33: Feature map

Kernels move one by one and are applied to one pixel at a time from the top left. Each cell of the kernel is multiplied by each cell within the grayscale image, and we sum these to get the first value of the green image.

For example, the first value of the feature map is obtained by the following:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 * 0 & 0 * 1 & 0 * 1 \\ 0 * 1 & 1 * 0 & 0 * 0 \\ 0 * 1 & 1 * 1 & 0 * 0 \end{bmatrix}$$

Now, we will add the feature, which will be equal to 1:

$$[1 * 0 + 0 * 1 + 0 * 1 + 0 * 1 + 1 * 0 + 0 * 0 + 0 * 1 + 1 * 1 + 0 * 0] = 1$$

The screenshot of the value of the feature map is marked in the color black on the feature map box, which is green in the following screenshot:

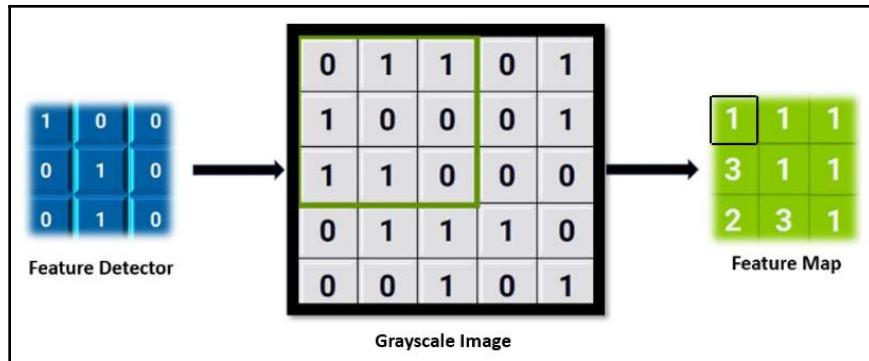


Fig 4.34: Feature map *2

Similarly, we will move the filter one cell at a time across the image from left to the right on the matrix shown in *Fig 4.34*, and then one cell at a time down until the last bottom-right cell.

The second value of the first feature map (which is 1) can be calculated by the following:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 * 1 & 0 * 1 & 0 * 0 \\ 0 * 1 & 0 * 1 & 0 * 0 \\ 0 * 1 & 1 * 0 & 0 * 0 \end{bmatrix}$$

The following screenshot shows the result of the multiplication of the feature detector and grayscale image matrix, which is shown in the orange-colored box. These are added together to get the value of the future map, 1. This is shown in the orange-colored box of the feature map in the following screenshot:

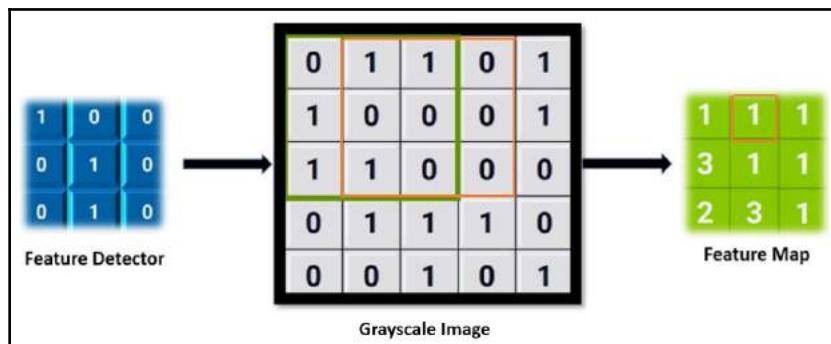


Fig 4.35: Feature map *3

The sum will again be 1, as shown in the following:

$$[1 * 1 + 0 * 1 + 0 * 0 + 0 * 1 + 0 * 1 + 0 * 0 + 0 * 1 + 1 * 0 + 0 * 0] = 1$$

Therefore, if we calculated the pixels of the new image by applying 3×3 kernels to all the pixels of the grayscale image, then we would generate the final feature map as follows:

$$\begin{bmatrix} 1 & 1 & 1 \\ 3 & 1 & 1 \\ 2 & 3 & 1 \end{bmatrix}$$

In the next section, we will learn how to sharpen and blur an image.

Sharpening and blurring

We use different types of kernels for sharpening and blurring images. The kernel for sharpening (the sharpen kernel) highlights the differences in adjacent pixel values, which emphasizes detail by enhancing contrast.

We will look at different examples of sharpening by multiplying the image pixels by 9 or 5 kernels and the other pixels around them by -1 or 0, as shown in the following matrix. The sharpening kernel is simply a way of enhancing the pixel of the image at any point.

Sharpening kernel type 1:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Sharpening kernel type 2:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Next, we will look at blurring kernels.

A blurring kernel is used to blur an image by averaging each pixel value and its neighbors. The blurring kernel is an $N \times N$ matrix filled with ones. Normalization has to be performed to achieve blurring. The values in the matrix have to collectively total to 1. If the sum doesn't add up to 1, then the image will be brighter or darker, as shown in Fig 4.36.

In the following matrix, we will multiply the entire picture by 1, add them together, and divide by 9. This matrix is also called the **blurring kernel**:

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The different operations, kernels, and resulting images are as follows:

Operation	Kernel	Resulting Image
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur 3×3 (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Fig 4.36: Kernels and operators



The choice of the kernel should not always have to be the square matrix, and the elements of the kernel matrix need not necessarily be predetermined. Usually, we use a matrix of any dimension. The selection of the filter matrix also depends on the type of images that we are applying convolution operations to in order to extract features.

Next, we will implement convolution using the OpenCV Python library, as follows:

1. First, we import the `matplotlib` (`mpimg` and `pyplot`), `numpy`, and `openCV` libraries using the following code block:

```
In[1]: import matplotlib.image as mpimg
In[2]: import matplotlib.pyplot as plt
In[3]: import numpy as np
In[4]: import cv2
```

2. Next, we import the image and open it:

```
In[5]: import cv2  
  
In[6]: image = cv2.imread('Test_image.jpg')  
  
In[7]: cv2.imshow('My Image', image)  
  
In[8]: cv2.waitKey()  
  
In[9]: cv2.destroyAllWindows()
```

The imported image is as follows:



Fig 4.37: Input image

3. The following code will change the image to grayscale:

```
In[10]: gray_img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)  
In[11]: cv2.imshow('Self Driving Car in Gray!', gray_img)  
In[12]: cv2.waitKey()  
In[13]: cv2.destroyAllWindows()
```

The grayscale image is as follows:



Fig 4.38: Grayscale images

4. Now, we will apply a sharpening kernel. Here, we will apply two types of sharpening kernel. Let's start with sharpening kernel type 1:

```
In[14]: Sharp_Kernel_1 = np.array([[0,-1,0],  
                                [-1,5,-1],  
                                [0,-1,0]])  
  
In[15]: Sharpened_Image_1 = cv2.filter2D(gray_img, -1,  
                                         Sharp_Kernel_1)  
In[16]: cv2.imshow('Sharpened Image', Sharpened_Image_1)  
  
In[17]: cv2.waitKey(0)  
In[18]: cv2.destroyAllWindows()
```

We can see the sharpened image in the following screenshot:



Fig 4.39: Sharpening kernel type 1 result

5. Next, we will process the image with sharpening kernel type 2:

```
In[19]: Sharp_Kernel_2 = np.array([[[-1,-1,-1],  
                                 [-1, 9,-1],  
                                 [-1,-1,-1]]])  
  
In[20]: Sharpened_Image_2 = cv2.filter2D(gray_img, -1,  
                                         Sharp_Kernel_2)  
In[21]: cv2.imshow('Sharpened Image_2', Sharpened_Image_2)  
  
In[22]: cv2.waitKey(0)  
In[23]: cv2.destroyAllWindows()
```

The output image is sharper than the previous output:



Fig 4.40: Sharpening kernel type 2 result

6. Next, we will look at the blurring kernel:

```
# Blurring Kernel  
In[24]: Blurr_Kernel = np.ones((3,3))  
In[25]: Blurr_Kernel
```

The output of the blurring kernel is as follows:

```
array([[1.,  1.,  1.],  
       [1.,  1.,  1.],  
       [1.,  1.,  1.]])
```

7. In the following code, we will check the blurred image that is made using the blurring kernel:

```
In[26]: Blurred_Image = cv2.filter2D(gray_img, -1, Blurr_Kernel)
In[27]: cv2.imshow('Blurred Image', Blurred_Image)

In[28]: cv2.waitKey(0)
In[29]: cv2.destroyAllWindows()
```

The output image is as follows:

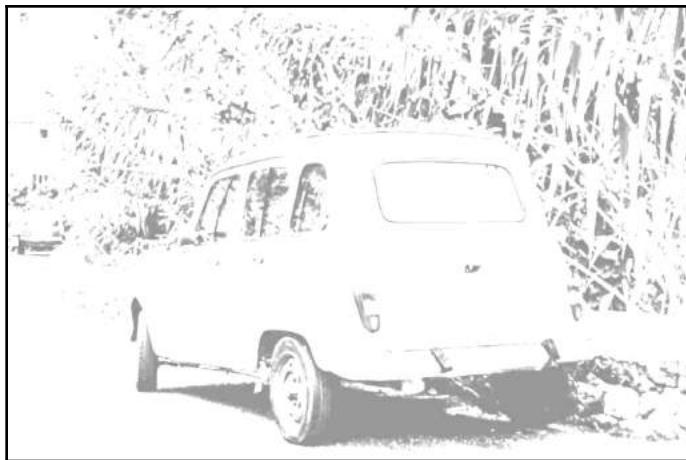


Fig 4.41: Blurring kernel type result

8. Next, we will normalize the kernel and check the image:

```
# Blurring Kernel normalized
In[30]: Blurr_Kernel = np.ones((3,3)) * 1/9
In[31]: Blurr_Kernel

In[32]: Blurred_Image = cv2.filter2D(gray_img, -1, Blurr_Kernel)
In[33]: cv2.imshow('Blurred Image', Blurred_Image)

In[34]: cv2.waitKey(0)
In[35]: cv2.destroyAllWindows()
```

The output looks like this:



Fig 4.42: Blurring kernel normalization

9. Now, we will try blurring with a more powerful kernel:

```
In[36]: Blurr_Kernel = np.ones((8,8))
In[37]: Blurr_Kernel
```

Let's look at the output in the following code block:

```
array([[1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.]])
```

10. Now, we will check the output of the blurring kernel:

```
In[38]: Blurr_Kernel = np.ones((8,8)) * 1/64
In[39]: Blurr_Kernel
```

Let's look at the output in the following code block:

```
array([[0.015625, 0.015625, 0.015625, 0.015625, 0.015625, 0.015625,
       0.015625, 0.015625],
       [0.015625, 0.015625, 0.015625, 0.015625, 0.015625, 0.015625,
       0.015625, 0.015625],
```

```
[0.015625, 0.015625, 0.015625, 0.015625, 0.015625, 0.015625,  
0.015625, 0.015625],  
[0.015625, 0.015625, 0.015625, 0.015625, 0.015625, 0.015625,  
0.015625, 0.015625],  
[0.015625, 0.015625, 0.015625, 0.015625, 0.015625, 0.015625,  
0.015625, 0.015625],  
[0.015625, 0.015625, 0.015625, 0.015625, 0.015625, 0.015625,  
0.015625, 0.015625],  
[0.015625, 0.015625, 0.015625, 0.015625, 0.015625, 0.015625,  
0.015625, 0.015625],  
[0.015625, 0.015625, 0.015625, 0.015625, 0.015625, 0.015625,  
0.015625, 0.015625],  
[0.015625, 0.015625, 0.015625, 0.015625, 0.015625, 0.015625,  
0.015625, 0.015625],  
[0.015625, 0.015625, 0.015625, 0.015625, 0.015625, 0.015625,  
0.015625, 0.015625])
```

11. Now we will look at the image that is processed with the blurring kernel:

```
In[40]: Blurred_Image = cv2.filter2D(gray_img, -1, Blurr_Kernel)  
  
In[41]: cv2.imshow('Blurred Image', Blurred_Image)  
  
In[42]: cv2.waitKey(0)  
In[43]: cv2.destroyAllWindows()
```

The new image looks like this:



Fig 4.43: Blurred image

In this section, we have looked at some examples of convolution. In the next section, we will learn about a concept that is used in self-driving cars called edge detection.

Edge detection and gradient calculation

Edge detection is a very important feature-extraction technique in computer vision that is used in self-driving cars to go beyond convolution, which we discussed in the previous section. In the previous section, we learned how to extract edges within an image. We converted a color image to grayscale or HSV, and later applied convolution to an image to extract features from it. In this section, we will learn about edge detection and gradient calculation.

Edge detection is a computer-vision feature-extraction tool that is used to detect the sharp changes in an image.

Let's say that we have three pixels. The first pixel is white, which is represented by 255 (as we have already learned in a previous section of this chapter); the next pixel is 0, which represents black; and the third pixel is also 255. So this means that we are going from white to black and then back to white. Edge detection happens when pixels change from 255 to 0 and from 0 to 255, as shown in the following screenshot:

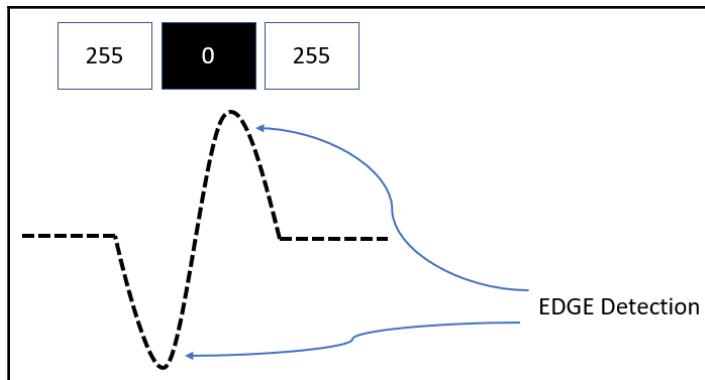


Fig 4.44: Edge detection

This is also called the first order of differentiation. Now we will look at the mathematical steps of edge detection.

We will take an image and apply the kernel application or the gradient application in a specific direction. We will learn about Sobel in the next section, and we will learn how it can be applied along the x and y axes.

Introducing Sobel

The gradient-based method based on the first-order derivatives is called the **Sobel edge detector**. The Sobel edge detector calculates the first-order derivatives of the image separately for the x axis and y axis. Sobel uses two 3×3 kernels that convolve over the original image to calculate the derivatives. For image A , G_x and G_y are two images that represent the horizontal and vertical derivative approximations:

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

The $*$ character indicates the 2D signal processing convolution operation.



The Sobel kernels compute the gradient with smoothing, as it can be decomposed a product of the averaging and differentiation kernels.

Sobel computes the gradient using smoothing. For example, $*$ can be written as follows:

$$\mathbf{G}_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * ([-1 \ 0 \ +1] * \mathbf{A}) \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 \\ 0 \\ +1 \end{bmatrix} * ([1 \ 2 \ 1] * \mathbf{A})$$

Here, the x -coordinate shows an increase in a **right** direction, and the y -coordinate shows an increase in a **downward** direction.

The resulting gradient approximations at each point in the image can be merged to give the gradient magnitude using the following formula:

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$$

Using the preceding information, we can also calculate the direction of the gradient:

$$\Theta = \text{atan}\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right)$$

For example, the direction of the gradient (Θ) is 0 for a vertical edge, where the gradient is lighter in the right direction.

We will see a hands-on example of the Sobel edge detector in a later section. Firstly, however, we will learn about the Laplacian edge detector.

Introducing the Laplacian edge detector

The Laplacian edge detector uses only one kernel. It calculates second-order derivatives in a single pass and detects zero crossings. In general, the second-order derivative is extremely sensitive to noise.

The kernel for the Laplacian edge detector is shown in the following screenshot:

0	-1	0
-1	4	-1
0	-1	0

The laplacian operator

Fig 4.45: The Laplacian operator

The following is an example of gradient-based edge detection and Laplacian-based edge detection. We can see that the first-order derivative is calculated using gradient-based edge detection, and second-order derivatives are calculated using Laplacian edge detection:

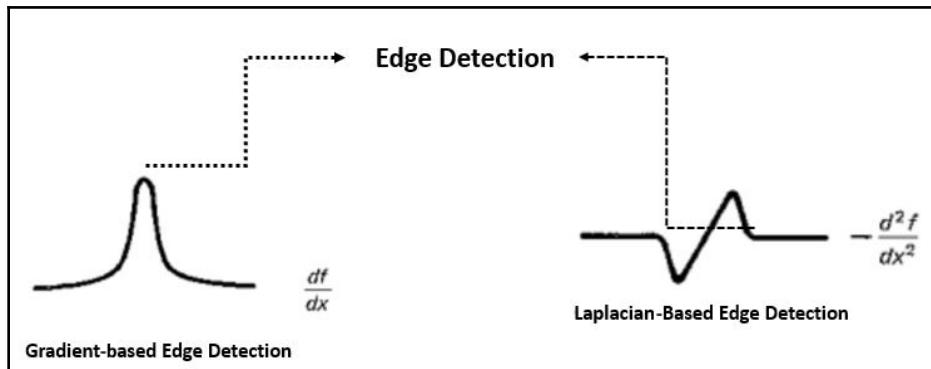


Fig 4.46: Gradient versus Laplacian edge detection



The objective of this book is to introduce you to the different edge detection concepts. If you want to read about these in more detail, you can go to https://en.wikipedia.org/wiki/Edge_detection.

In the next section, we will learn about an important concept called **Canny edge detection**.

Canny edge detection

The Canny edge is a popular edge-detection algorithm. It can detect a wide range of edges. The Canny edge detection algorithm was developed by John F. Canny in 1986. The Canny edge is widely used in the field of computer vision, as it has a wide range of applications.

The process of Canny edge detection has the following criteria:

- The edges of images should be detected with high accuracy.
- Only one marks should be created for one image; there should not be any duplicate marks.
- The detected edges should be correctly localized on the image.
- Granular edges should also be detected.

The Canny edge detection algorithm is applied using the following steps:

1. In the first step, a Gaussian filter is applied to smooth the image. Smoothing the image removes the noise.
2. Next, we find the intensity gradient of the image.
3. Then, we apply nonmaximum suppression to remove any fake edge detection response.
4. Next, we apply a double-threshold on the image to determine the accuracy of the edge detection.
5. Finally, we track edges using hysteresis. The edge is detected by suppressing all of the weak edges and considering only the strong ones.

In the following steps, we will implement Sobel, Laplacian, and Canny edge detection using the OpenCV library:

1. We will import the `matplotlib (mpimg and pyplot)`, `numpy`, and `openCV` libraries:

```
In[1]: import matplotlib.image as mpimg  
In[2]: import matplotlib.pyplot as plt  
In[3]: import numpy as np  
In[4]: import cv2
```

2. Next, we will read and display the image using OpenCV:

```
In[5]: import cv2  
  
In[6]: image = cv2.imread('Test_image.jpg')  
  
In[7]: cv2.imshow('My Test Image', image)  
  
In[8]: cv2.waitKey()  
In[9]: cv2.destroyAllWindows()
```

The imported image is as follows:



Fig 4.47: Input image

3. Then we change the image into grayscale using OpenCV:

```
In[10]: gray_img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)  
In[11]: cv2.imshow('Self Driving Car in Gray!', gray_img)  
In[12]: cv2.waitKey()  
In[13]: cv2.destroyAllWindows()
```

The output of the grayscale is as follows:



Fig 4.48: Grayscale image

4. Next, we will perform a Sobel X(G_x) calculation of the preceding image:

```
In[14]: x_sobel = cv2.Sobel(gray_img, cv2.CV_64F, 0, 1, ksize = 7)
In[15]: cv2.imshow('Sobel - X direction', x_sobel)
In[16]: cv2.waitKey()
In[17]: cv2.destroyAllWindows()
```

The output looks like this:

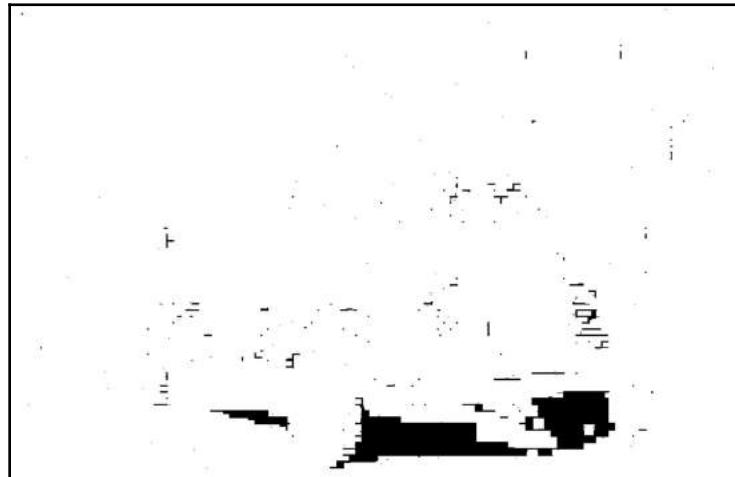


Fig 4.49: Sobel X output

5. Now we will calculate Sobel $\text{Y}(G_y)$ using the OpenCV library:

```
In[18]: y_sobel = cv2.Sobel(gray_img, cv2.CV_64F, 1, 0, ksize = 7)
In[19]: cv2.imshow('Sobel - Y direction', y_sobel)
In[20]: cv2.waitKey()
In[21]: cv2.destroyAllWindows()
```

The Sobel $\text{Y}(G_y)$ looks like this:

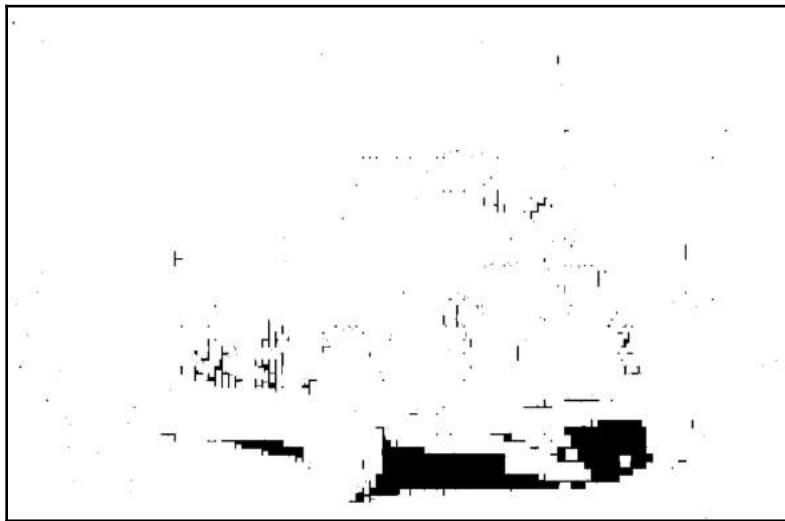


Fig 4.50: Sobel Y

6. The Laplacian edge is calculated using OpenCV as follows:

```
In[22]: laplacian = cv2.Laplacian(gray_img, cv2.CV_64F)
In[23]: cv2.imshow('Laplacian', laplacian)
In[24]: cv2.waitKey()
In[25]: cv2.destroyAllWindows()
```

The Laplacian looks like this:



Fig 4.51: Laplacian detector

7. Now it's time to calculate the Canny edge detection. `threshold_1` is the first threshold for the hysteresis procedure and `threshold_2` is the second threshold for the hysteresis procedure:

```
In[26]: threshold_1 = 120
```

```
In[27]: threshold_2 = 200
```

```
In[28]: canny = cv2.Canny(gray_img, threshold_1, threshold_2)
```

```
In[29]: cv2.imshow('Canny', canny)
```

```
In[30]: cv2.waitKey()
```

```
In[31]: cv2.destroyAllWindows()
```

The Canny edge detection looks like this:



Fig 4.52: Edge detection

Canny edge detection is the best method for edge detection, as it has numerous computer-vision applications. It also plays an important role in the field of autonomous cars. In Chapter 5, *Finding Road Markings Using OpenCV*, we will see an application of Canny edge detection.



If you want to learn about Canny edge detection in more detail, you can refer to the information at https://en.wikipedia.org/wiki/Canny_edge_detector.

In the next section, we will learn about image transformation.

Image transformation

In this section, we will learn about different image-transformation techniques, such as rotation, translation, resizing, and masking a region of interest. Image transformations are used to correct distorted images or to change the perspective of an image. With regard to self-driving cars, there are lots of applications of image transformation. We have different cameras mounted in the car, and most of the time they are required to transform the image. Sometimes, by transforming the image, we allow the car to concentrate on an area of interest. We will also look at a project on behavioral cloning in Chapter 9, *Implementation of Semantic Segmentation*.

There are two types of image transformation:

- Affine transformation
- Projective transformation

Affine transformation

A linear mapping method that preserves points, straight lines, and planes is called **affine transformation**. After affine transformation, sets of parallel lines will remain parallel. In general, the affine transformation technique is used in the correction of geometric distortions that occur with nonideal camera angles.

An example of affine transformation is as follows. Here, we are applying affine transformation to a rectangle, causing it to shear offsets—that is, a set of points that are moved a distance proportional to their axis:

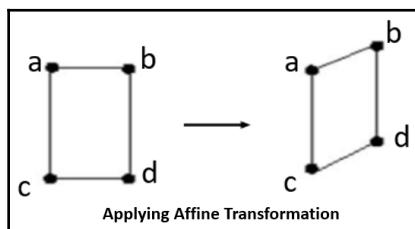


Fig 4.53: Applying affine transformation to a rectangle



You can learn about affine transformation in more detail at https://en.wikipedia.org/wiki/Affine_transformation.

In the preceding diagram, we can see how a rectangle with four points changes after we apply affine transformation.

Projective transformation

A transformation that maps lines to lines is a projective transformation. Here, parallel lines can be formed with an angle that may intersect at some point. For example, the lines between **a** - **d** and **d** - **c** on the left side of the following diagram are not parallel and could meet at some point in time. At the same time, the lines between **a** - **d** and **b** - **c** are parallel and will never meet. After applying the projective transform, none of the sides will have parallel lines:

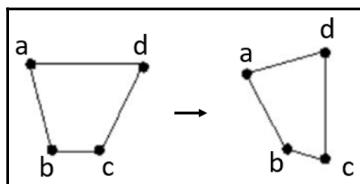


Fig 4.54: Applying projective transformation



You can learn more about projective transformation at <https://en.wikipedia.org/wiki/Homography>.

In the next section of the chapter, we will learn about image rotation using the OpenCV library.

Image rotation

In this section, we will learn how we can perform a rotation by using OpenCV and the rotation matrix, M . A rotation matrix is a matrix that is used to perform a rotation in Euclidean space. It rotates points in the xy plane counterclockwise through an angle, θ , around the origin.

Now we will implement image rotation using OpenCV:

1. We will first import the `matplotlib` (`mpimg` and `pyplot`), `numpy`, and `openCV` libraries:

```
In[1]: import cv2  
In[2]: import numpy as np  
In[3]: import matplotlib.image as mpimg  
In[4]: from matplotlib import pyplot as plt  
In[5]: %matplotlib inline
```

2. Next, we will read the input image:

```
In[5]: image = cv2.imread('test_image.jpg')
In[6]: cv2.imshow('Original Image', image)
In[7]: cv2.waitKey()
In[8]: cv2.destroyAllWindows()
```

The input image looks like this:



Fig 4.55: Input image

3. The height and width of the image are as follows:

```
In[9]: height, width = image.shape[:2]
In[10]: height
579
In[11]: width
530
```

4. Then, we will perform rotation using OpenCV:

```
In[12]: M_rotation = cv2.getRotationMatrix2D((width/2, height/2),
90, 0.5) # Rotate around the center

In[13]: rotated_image = cv2.warpAffine(image, M_rotation, (width,
height)) # insert size of the output image
In[14]: cv2.imshow('Rotated', rotated_image)
In[15]: cv2.waitKey()
In[16]: cv2.destroyAllWindows()
```

The output image looks like this:

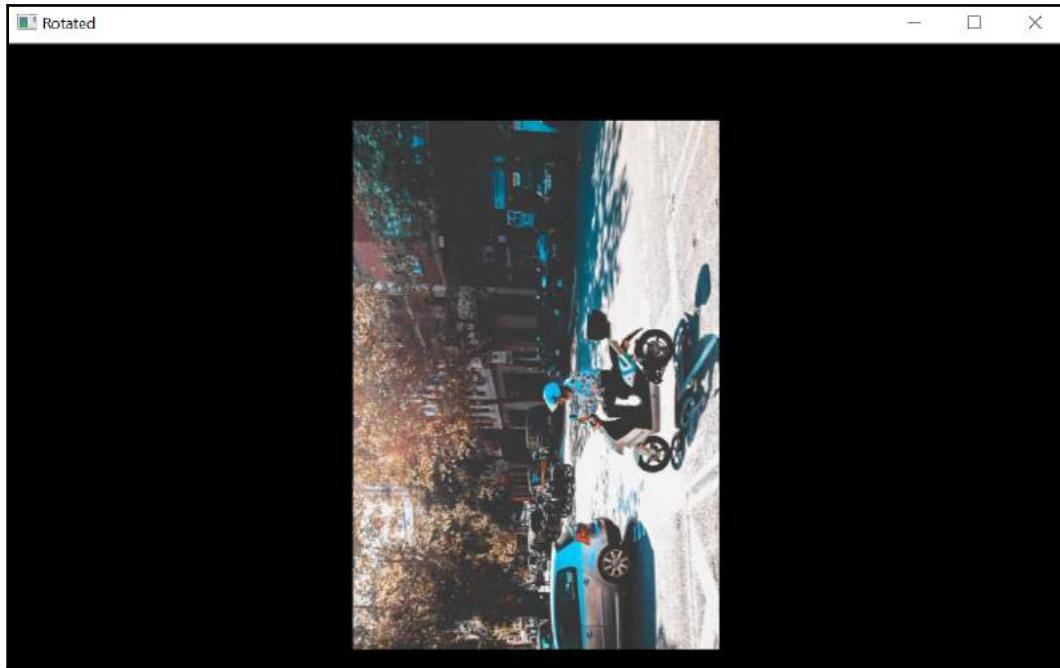


Fig 4.56: Output image

In the next section, we will look at image translation.

Image translation

In this section, we will learn about image translation. Image translation involves shifting an object's position in the x and/or y direction. OpenCV uses a translational matrix, T , as follows:

$$T = \begin{bmatrix} 0 & 1 & Tx \\ 1 & 0 & Ty \end{bmatrix}$$

Now, we will perform image translation:

1. We will first import the `matplotlib (mpimg and pyplot)`, `numpy`, and `openCV` libraries:

```
In[1]: import cv2  
In[2]: import numpy as np  
In[3]: import matplotlib.image as mpimg  
In[4]: from matplotlib import pyplot as plt  
In[5]: %matplotlib inline
```

2. Then we read in the input image:

```
In[6]: image = cv2.imread('test_image.jpg')  
In[7]: cv2.imshow('Original Image', image)  
In[8]: cv2.waitKey()  
In[9]: cv2.destroyAllWindows()
```

The input image looks like this:

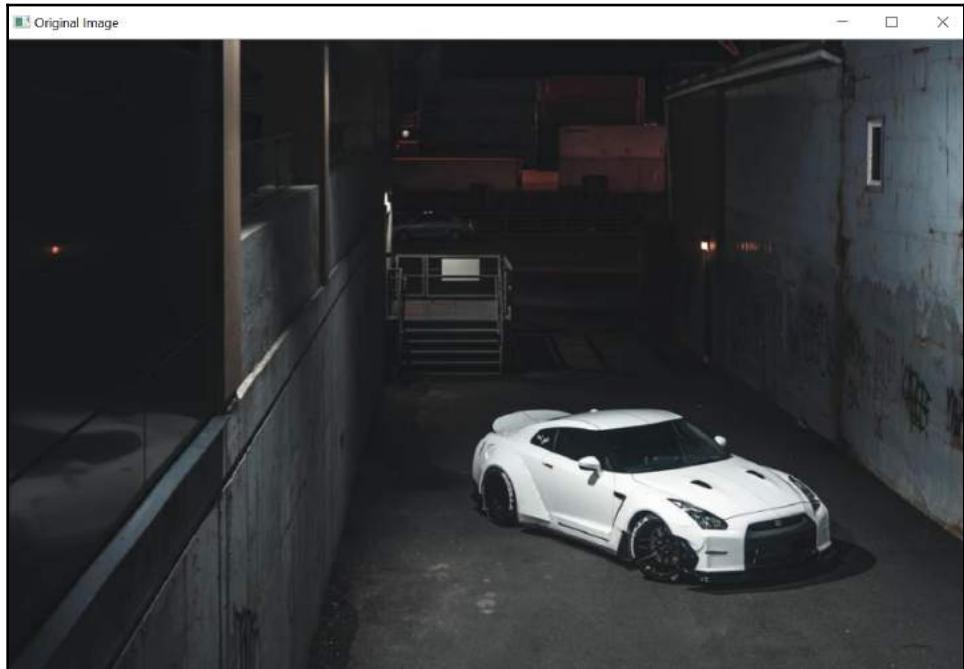


Fig 4.57: Input image

3. The height and width of the image are as follows:

```
In[10]: height, width = image.shape[:2]
In[11]: height
579
In[12]: width
530
```

The translation matrix is defined as follows:

```
In[13]: Translational_Matrix = np.float32([[1, 0, 120],
                                             [0, 1, -150]])
```

4. Then we perform translation using OpenCV:

```
In[14]: translated_image = cv2.warpAffine(image,
                                            Translational_Matrix, (width, height))
In[15]: cv2.imshow('Translated Image', translated_image)
In[16]: cv2.waitKey()
In[17]: cv2.destroyAllWindows()
```

The output snapshot of the translated image looks like this:

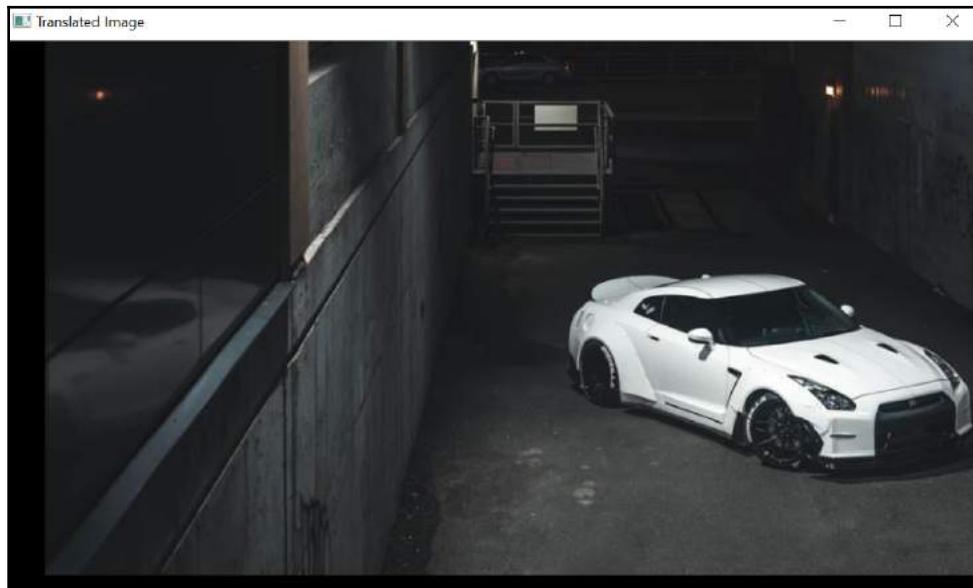


Fig 4.58: Output image

In the next section, we will learn about image resizing.

Image resizing

This section is all about image resizing. Resizing using OpenCV can be performed by using `cv2.resize()`. The preferred interpolation methods are `cv.INTER_AREA` for shrinking and `cv.INTER_CUBIC` for zooming. By default, the interpolation method used is `cv.INTER_LINEAR` for all resizing purposes:

1. First, we will import the `numpy`, `openCV`, and `matplotlib` (`mpimg` and `pyplot`) libraries:

```
In[1]: import cv2  
In[2]: import numpy as np  
In[3]: import matplotlib.image as mpimg  
In[4]: from matplotlib import pyplot as plt  
In[5]: %matplotlib inline
```

2. Then we read in the input image:

```
In[6]: image = cv2.imread('test_image.jpg')  
In[7]: cv2.imshow('Original Image', image)  
In[8]: cv2.waitKey()  
In[9]: cv2.destroyAllWindows()
```

The input image looks like this:

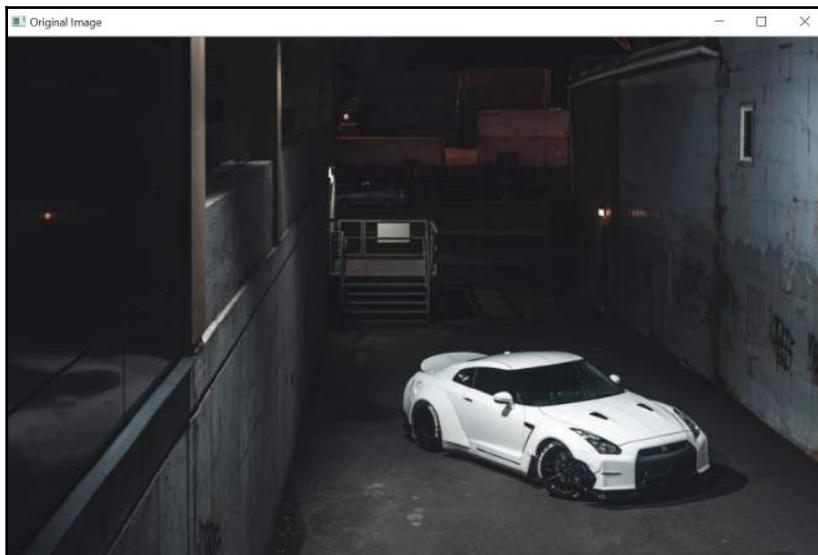


Fig 4.59: Input image

3. The height and width of the image are as follows:

```
In[10]: height, width = image.shape[:2]
In[11]: height
579
In[12]: width
530
```

4. Next, we perform the resize using OpenCV:

```
In[13]: resized_image = cv2.resize(image, None, fx=0.5, fy=0.5,
interpolation = cv2.INTER_CUBIC) # Try 0.5

In[14]: cv2.imshow('Resized Image', resized_image)
In[15]: cv2.waitKey()
In[16]: cv2.destroyAllWindows()
```

The resized image looks like this:

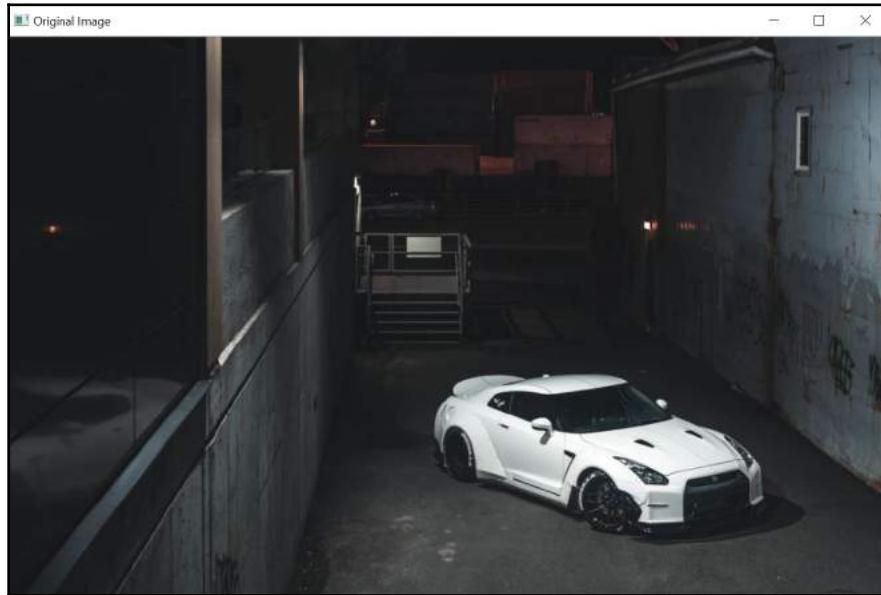


Fig 4.60: Resized image

In the next section, we will learn about one of the most important topics in the field of autonomous driving: perspective transformation.

Perspective transformation

Perspective transformation is an important aspect of programming self-driving cars. Perspective transformation is more complicated than affine transformation. In perspective transformation, we use a 3×3 transformation matrix to transform images from the 3D world into 2D images.

An example of perspective transformation is shown in *Fig 4.61*. In the following screenshot, we can see the tilted chessboard, and once the perspective transform is applied, the board is transformed into a normal chessboard with a top-down view. This has numerous applications in the field of self-driving cars, as roads have many objects that require perspective transformation to be processed:

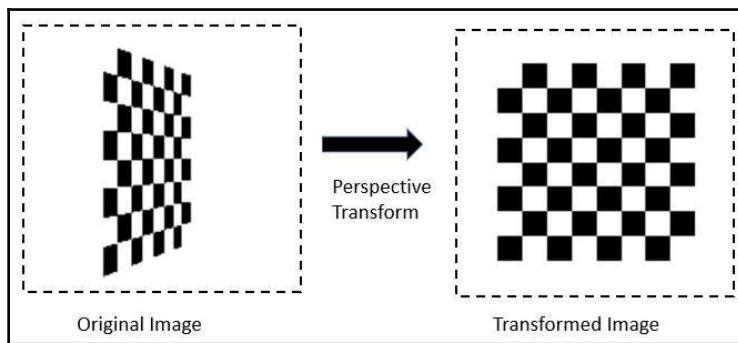


Fig 4.61: Perspective transform using a chessboard

Now we will implement perspective transformation to a traffic signboard using the OpenCV library:

1. We will first import the `matplotlib (mpimg and pyplot)`, `numpy`, and `openCV` libraries:

```
In[1]: import cv2  
In[2]: import numpy as np  
In[3]: import matplotlib.image as mpimg  
In[4]: from matplotlib import pyplot as plt  
In[5]: %matplotlib inline
```

2. Next, we will read the input image using OpenCV:

```
In[6]: image = cv2.imread('Speed_Sign_View_2.jpg')  
In[7]: cv2.imshow('Original Image', image)  
In[8]: cv2.waitKey()  
In[9]: cv2.destroyAllWindows()
```

The image, as read by OpenCV, is as follows:



Fig 4.62: Output image

The image, as read by matplotlib, is as follows:



Fig 4.63: Output image

3. The height and width of the image are as follows:

```
In[10]: height, width = image.shape[:2]
In[11]: height
426
In[12]: width
640
```

4. If we want to select any particular section from an image, then the following coordinates of the four corners of the original image must be used for the selection points:

- **First entry:** The top-left of the image is represented by the yellow point in the following image.
- **Second entry:** The top-right of the image is represented by the green point in the following image.
- **Third entry:** The bottom-right of the image is represented by the blue point in the following image.
- **Fourth entry:** The bottom-left of the image is represented by the red point in the following image.

You can see these points in the following image:



Fig 4.64: Image with points

The value of each point in the preceding image is as follows:

- Top-left of the image (shown in yellow): [420,70]
- Top-right of the image (shown in green): [580, 50]
- Bottom-right of the image (shown in blue): [590,210]
- Bottom-left of the image (shown in red): [430, 220]]

Now we will enter the values in the selected coordinates as follows:

```
In[13]: Source_points = np.float32([[420,70], [580, 50], [590,210], [430, 220]])
```

The coordinates of the four corners of the desired output are as follows:

```
In[14]: Destination_points = np.float32([[0,0], [width,0],  
[width,height], [0,height]])
```

5. Use the two sets of four points to compute the perspective transformation matrix, M:

```
In[15]: M = cv2.getPerspectiveTransform(Source_points,  
Destination_points)
```

```
In[16]: warped = cv2.warpPerspective(image, M, (width, height))
```

```
In[17]: cv2.imshow('warped Image', warped)
```

```
In[18]: cv2.waitKey()
```

```
In[19]: cv2.destroyAllWindows()
```

The output image looks like the following. As you can tell, perspective transformation has an important role in the field of image preprocessing, especially when used in self-driving cars:



Fig 4.65: Output image

In the next section, we will learn about using cropping, dilation, and erosion on an image.

Cropping, dilating, and eroding an image

Image cropping is a type of image transformation. In this section, we will crop an image using OpenCV:

1. Firstly, we will import the matplotlib (mpimg and pyplot), numpy, and openCV libraries:

```
In[1]: import cv2  
In[2]: import numpy as np  
In[3]: import matplotlib.image as mpimg  
In[4]: from matplotlib import pyplot as plt  
In[5]: %matplotlib inline
```

2. Next, we will read the input image:

```
In[6]: image = cv2.imread('Test_auto_image.jpg')  
In[7]: cv2.imshow('Original Image', image)  
In[8]: cv2.waitKey()  
In[9]: cv2.destroyAllWindows()
```

The input image is as follows:



Fig 4.66: Input image

The height and width of the image are as follows:

```
In[10]: height, width = image.shape[:2]
In[11]: height
800
In[12]: width
1200
```

3. Next, we perform cropping with the following code. The top-left coordinates of the desired cropped area are w_0 and h_0 :

```
In[13]: w0 = int(width * 0.5)
In[14]: h0 = int(height * 0.5)
```

4. The bottom-right coordinates of the desired cropped area are w_1 and h_1 :

```
In[15]: h1 = int(height * 1)
In[16]: w1 = int(width * 1)
```

5. Next, we crop the image using OpenCV:

```
In[17]: Image_cropped = image[h0:h1 , w0:w1]

In[18]: cv2.imshow("Cropped Image", Image_cropped)
In[19]: cv2.waitKey()
In[20]: cv2.destroyAllWindows()
```

The cropped image looks like this:

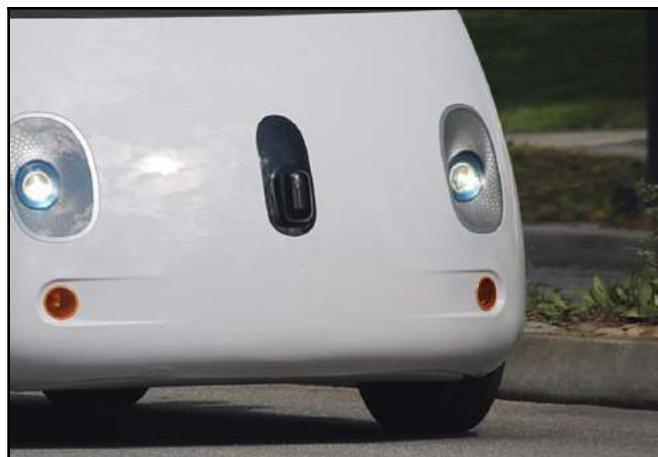


Fig 4.67: Cropped image

Now, we will look at dilation and erosion.

Erosion means the removal of pixels at the boundaries of the objects in an image and **dilation** means the addition of extra pixels to the boundaries of the objects in an image. An example of using OpenCV for this is as follows:

1. We will first import the `matplotlib (mpimg and pyplot)`, `numpy`, and `openCV` libraries:

```
In[1]: import cv2  
In[2]: import numpy as np  
In[3]: import matplotlib.image as mpimg  
In[4]: from matplotlib import pyplot as plt  
In[5]: %matplotlib inline
```

2. Then we read the input image:

```
In[6]: image = cv2.imread('Speed_Sign_View_2.jpg')  
In[7]: cv2.imshow('Original Image', image)  
In[8]: cv2.waitKey()  
In[9]: cv2.destroyAllWindows()
```

The image looks like this:



Fig 4.68: Input image

3. In the following step, we process the image with erosion and dilation using OpenCV:

```
In[10]: kernel = np.ones((3,3), np.uint8)
In[11]: image_erosion = cv2.erode(image, kernel, iterations=3)
In[12]: image_dilation = cv2.dilate(image, kernel, iterations=3)
```

Let's see what the image looks like after applying erosion and dilation:

```
In[13]: cv2.imshow('Input', image)
In[14]: cv2.imshow('Erosion', image_erosion)
In[15]: cv2.imshow('Dilation', image_dilation)
In[16]: cv2.waitKey()
In[17]: cv2.destroyAllWindows()
```

The eroded image looks like this:



Fig 4.69: Image after erosion

The dilated image looks like this:



Fig 4.70: Dilated image

Now, we will learn about masking regions of interest.

Masking regions of interest

The main goal of masking regions of interest is to color filter an image to perform different operations. For instance, when an autonomous car is driving on the road, the region of interest for the car is the lane lines because it must be driven on the road. In the following steps, we will see an example of how to determine a region of interest when applied to the lane lines on a road:

1. Firstly, import the `matplotlib (mpimg and pyplot)`, `numpy`, and `openCV` libraries:

```
In[1]: import cv2  
In[2]: import numpy as np  
In[3]: import matplotlib.image as mpimg  
In[4]: from matplotlib import pyplot as plt  
In[5]: %matplotlib inline
```

2. Next, we will read the input image:

```
In[6]: image_color = cv2.imread('lanes.jpg')
In[7]: cv2.imshow('Original Image', image_color)
In[8]: cv2.waitKey()
In[9]: cv2.destroyAllWindows()
```

The input image looks like this:



Fig 4.71: Input image

The height and width of the image are as follows:

```
In[10]: height, width = image_color.shape[:2]
In[11]: height
426
In[12]: width
640
```

3. Now, we will convert the image into grayscale using OpenCV:

```
In[13]: image_gray = cv2.cvtColor(image_color, cv2.COLOR_BGR2GRAY)
In[14]: plt.imshow(image_gray, cmap = 'gray')
```

This is the output:

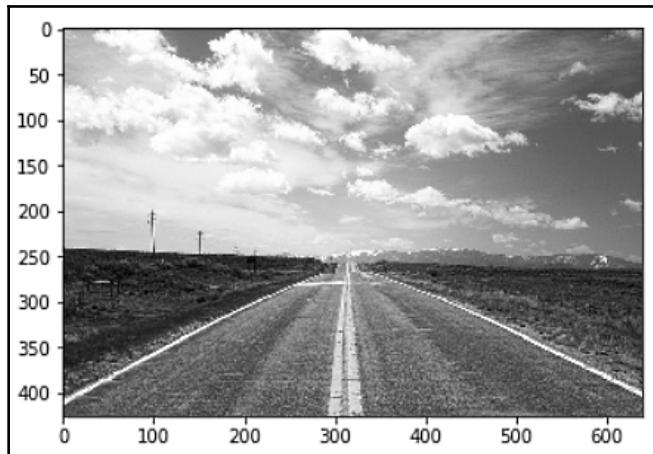


Fig 4.72: Grayscale image

Now let's mask the regions of interest:

1. First, we will select the points of the region of interest (ROI):

```
In[15]: ROI = np.array([(0, 400), (300, 250), (450, 300), (700, height)]], dtype=np.int32)
```

2. Next, we will define a blank image with all zeros (black):

```
In[16]: blank = np.zeros_like(image_gray)
In[17]: blank.shape
(519, 939)
```

3. Then we fill the region of interest in white (255):

```
In[18]: mask = cv2.fillPoly(blank, ROI, 255)
```

4. Next, we will perform a bit-wise AND operation to select only the region of interest:

```
In[19]: masked_image = cv2.bitwise_and(image_gray, mask)
In[20]: plt.imshow(masked_image, cmap = 'gray')
```

We can see the region of interest masking; the road under the lane lines is our area of interest. We will implement a real-time project in Chapter 5, *Finding Road Markings Using OpenCV*. The masked image appears as follows:

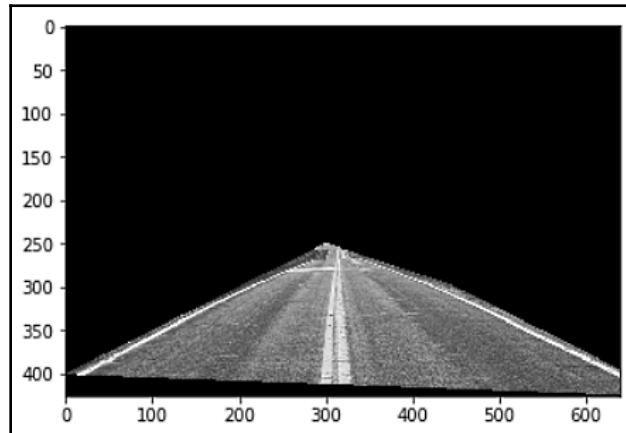


Fig 4.73: Masked image

In the next section, we will read about the Hough transform, one of the most important topics of computer vision.

The Hough transform

The Hough transform is one of the most important topics of computer vision. It is used in feature extraction and image analysis. The Hough transform was invented in 1972 by Richard Duda and Peter Hart, and it was originally called the **generalized Hough transform**. In general, the technique is used to find instances of objects that are not perfectly within a certain class by means of a voting procedure.

We can use the Hough transform along with region of interest masking. We will see an example of the detection of road markings in Chapter 5, *Finding Road Markings Using OpenCV*, using the Hough transform and region of interest masking together.

We will learn about the Hough transform in more detail with the drawing of a 2D coordinate space of x and y inside a straight line, as shown in Fig 4.74.

We know that the equation of a straight line is $y = mx + c$. The straight line has two parameters, m and c , and we are currently plotting it as a function of x and y . We can also represent these lines in a parametric space, which we will call the Hough space.

The plot will be c versus m , as shown in the following screenshot:

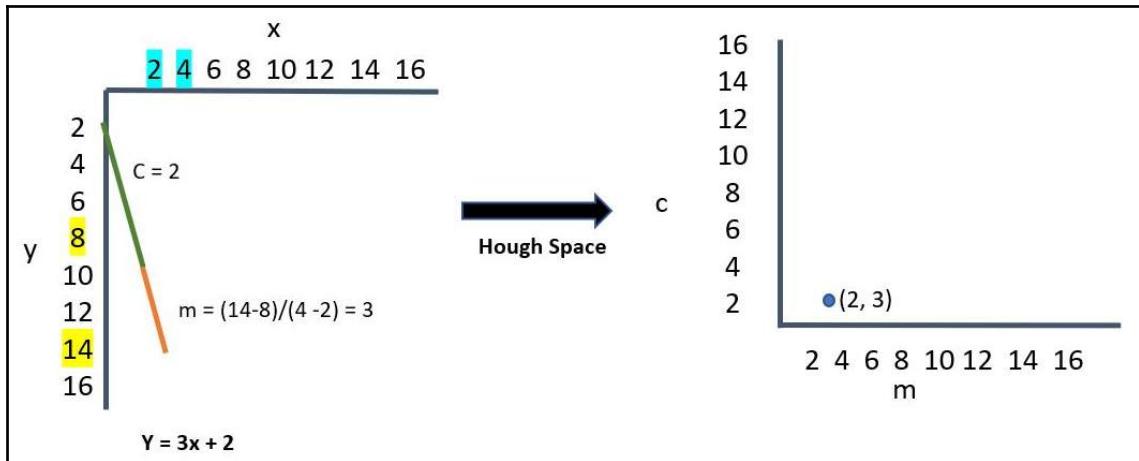


Fig 4.74: The Hough space

Now, imagine that instead of a line we had a single dot located at coordinates (12,2). There are many possible lines that could pass through this dot, each with different values of m and c . In the following screenshot, you can see how a single point in the x and y space is represented by a line in the Hough space:

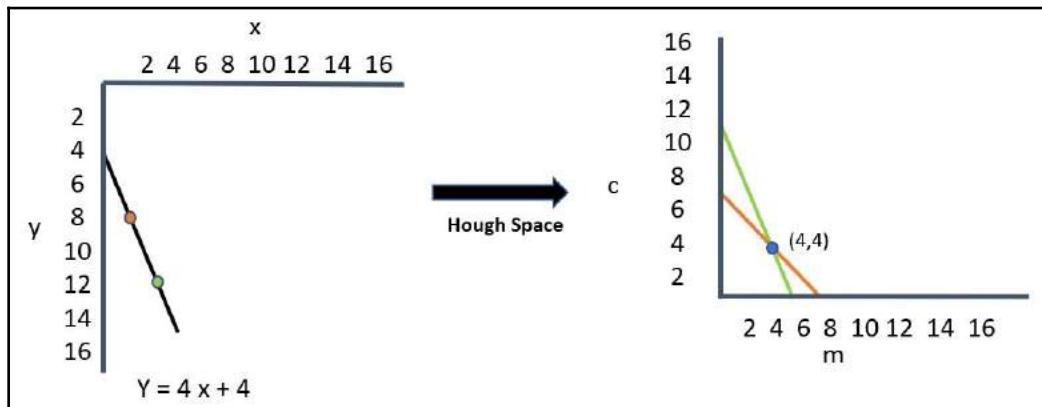


Fig 4.75: The Hough space 2

This ability to identify possible lines from a series of points is how we're going to find lines in the gradient of the image.

We have one more tiny problem. If you try to compute the slope of a vertical line, the change in x is zero, which ultimately will always evaluate to a gradient of infinity, which is not something that we can represent in the Hough space. Infinity is not really something we can work with anyway, as shown in the following diagram:

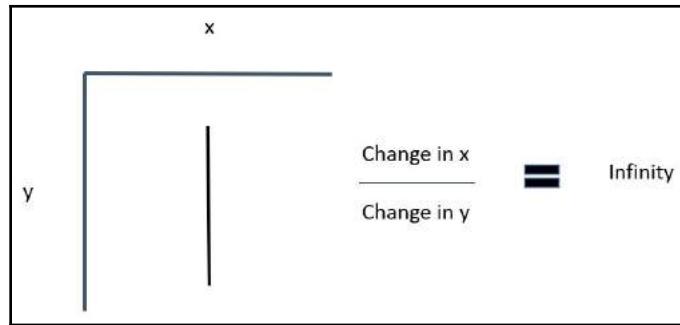


Fig 4.76: The Hough space 3

We need a more robust representation of lines so that we don't encounter any numeric problems because clearly in the form $y = mx + c$, c cannot represent vertical lines. That being said, instead of expressing our line with the Cartesian coordinate parameters m and c , we can express it in the polar coordinate system—rho and theta—so that our line equation can be read as $\rho = x\cos(\theta) + y\sin(\theta)$. This is shown in the following screenshot:

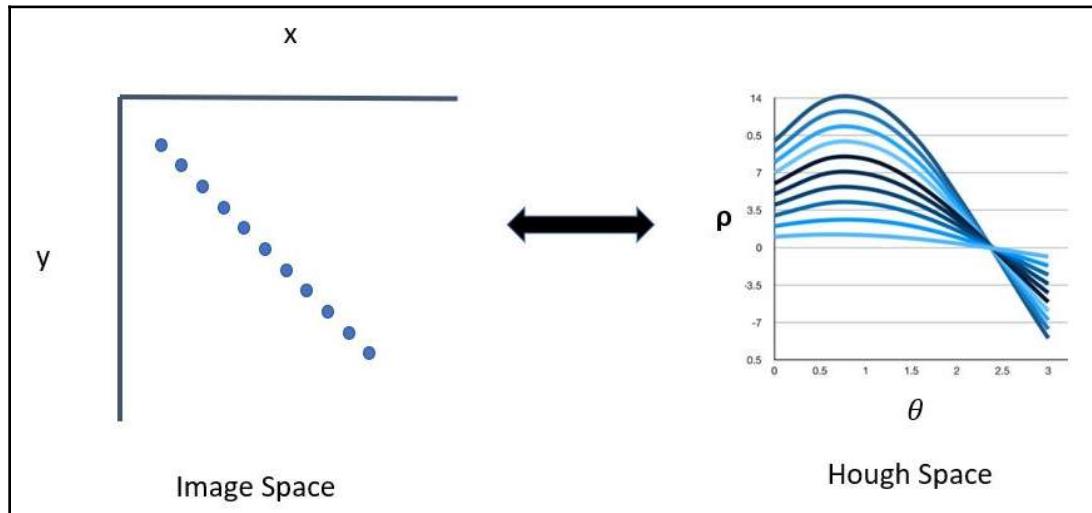


Fig 4.77: Image space versus Hough space

By using polar coordinates, it is easy to represent the lines in forms, which cannot be done with the Cartesian form. The formula for the polar form is as follows:

$$x \cos(\theta) + y \sin(\theta) = \rho$$

This equation contains the following two elements:

- ρ is the perpendicular distance from the origin to the line.
- θ is the angle formed in the horizontal axis by the perpendicular line.

A graphical representation of the equations is shown in the following screenshot:

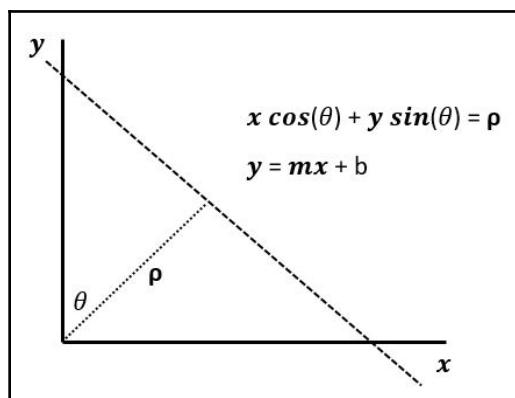


Fig 4.78: Polar form

Next, we will see an example of the space of an image represented in the Hough space:

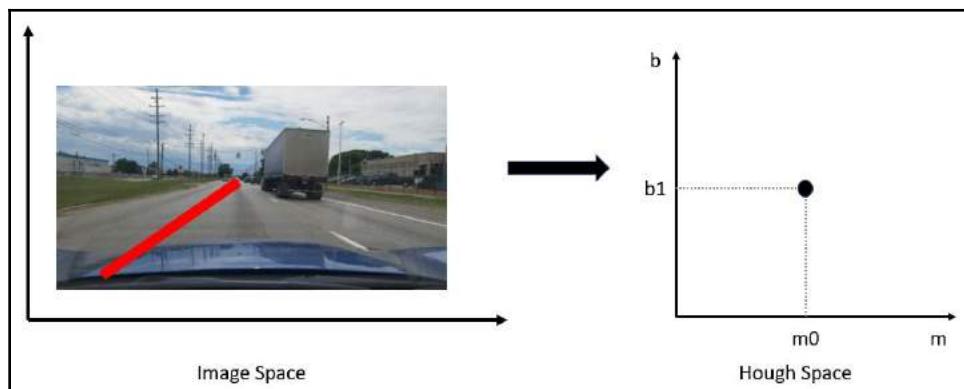


Fig 4.79: Image in the Hough space

In the image space, a line is plotted as x versus y and modeled as $y = mx + b$ or $xcos(\theta) + ysin(\theta) = \rho$. In the preceding screenshot, we can see in the parameter space (the Hough space) that a line is represented by a point in m versus b . Each line is represented as a single point with (m, b) coordinates or (ρ, θ) parameters.



You can refer to https://en.wikipedia.org/wiki/Hough_transform to learn more about the Hough transform.

In the following section, we will see the Python implementation of the Hough transform:

1. Import the matplotlib, numpy, and openCV libraries:

```
In[1]: import cv2  
In[2]: import numpy as np  
In[3]: import matplotlib.pyplot as plt
```

2. Next, read the input image:

```
In[4]: image_c = cv2.imread('calendar.jpg')  
In[5]: cv2.imshow('Given Image', image_c)  
In[6]: cv2.waitKey(0)  
In[7]: cv2.destroyAllWindows()
```

3. Then, read the image using matplotlib:

```
In[8]: plt.imshow(image_c)
```

The output is as follows:

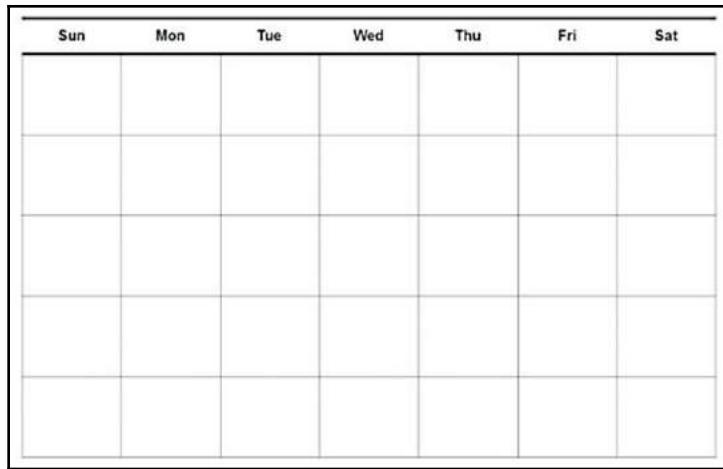


Fig 4.80: Input image

4. Convert the image by applying Canny edge detection:

```
In[9]: image_g = cv2.cvtColor(image_c, cv2.COLOR_BGR2GRAY)  
  
In[10]: image_canny = cv2.Canny(image_g, 50, 200, apertureSize= 3)  
  
In[11]: image_canny
```

We can see the output in the following screenshot:

```
array([[ 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]], dtype=uint8)
```

Fig 4.81: Matrix of the input image

We will use the following code to view the Canny image:

```
In[12]: cv2.imshow('canny image', image_canny)  
In[13]: cv2.waitKey(0)  
In[14]: cv2.destroyAllWindows()
```

The Canny image appears as follows:

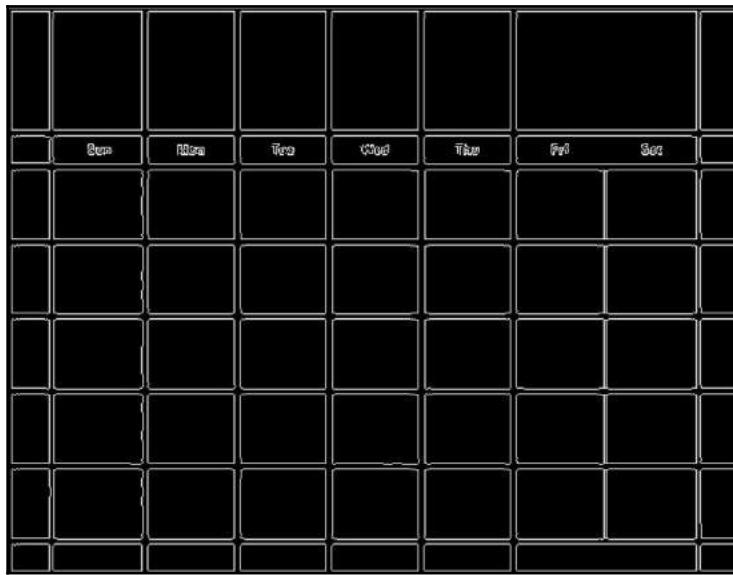


Fig 4.82: Canny image of a calendar

5. Next, we will apply the Hough transform:

```
In[15]: lines = cv2.HoughLines(image_canny, 1, np.pi/180, 300)
In[16]: if lines is not None:
            for i in range(0, len(lines)):
                rho = lines[i][0][0]
                theta = lines[i][0][1]

In[17]: x0 = rho * np.cos(theta)
In[18]: y0 = rho * np.sin(theta)

In[19]: a = np.cos(theta)
In[20]: b = np.sin(theta)

In[21]: x1 = int(x0 + 1000 * (-b))
In[22]: y1 = int(y0 + 1000 * (a))
In[23]: x2 = int(x0 - 1000 * (-b))
In[24]: y2 = int(y0 - 1000 * (a))
In[25]: cv2.line(image_c, (x1, y1), (x2, y2), (255, 0, 0), 2)

In[26]: cv2.imshow('Hough Lines', image_c)
In[27]: cv2.waitKey(0)
In[28]: cv2.destroyAllWindows()
```

The output image is as follows:

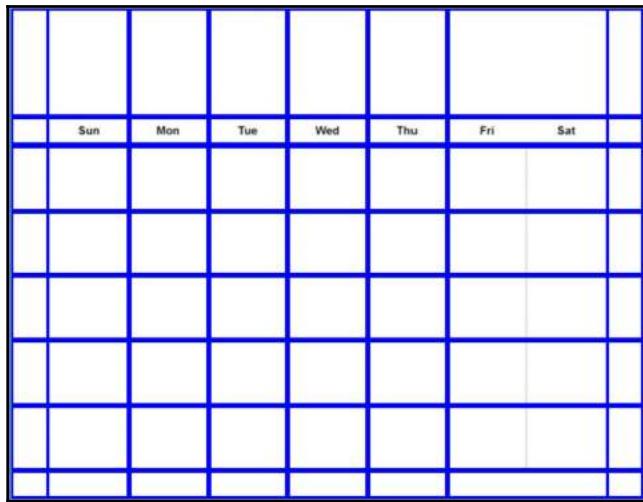


Fig 4.83: Hough transform of a calendar

We can see in the preceding image that all the lines in blue have been detected by the Hough transform. Here, we have used OpenCV for the Hough transform. In Chapter 5, *Finding Road Markings Using OpenCV*, we are going to apply the Hough transform to a real-time project.

Summary

In this chapter, we learned about the importance of computer vision and the challenges we face in the field of computer vision. We also learned about color spaces, edge detection, and the different types of image transformation, as well as the many examples of using OpenCV. We are going to use a few of these techniques in later chapters.

We learned about the building blocks of an image and how a computer sees an image. We also learned about the importance of color space techniques, such as convolution. We are going to apply all the techniques that we covered here in future chapters.

In the next chapter, we are going to apply computer-vision techniques and implement a software pipeline for detecting road markings. We will first apply this process to an image and then apply it to a video. In the next chapter, we are going to apply several of the techniques that we covered in this chapter, such as edge detection and Hough transformation.

5

Finding Road Markings Using OpenCV

In this chapter, we will use the concepts from [Chapter 4, Computer Vision for Self-Driving Cars](#), to design a pipeline to identify road markings for self-driving cars using the OpenCV library. In general, we will preprocess our data with the OpenCV library and then feed in a deep learning network.

The main purpose of this chapter is to build a program that can identify road markings in a picture or a video. When we drive a car, we can see where the road markings are. A car obviously doesn't have any eyes, which is where computer vision comes in. In this chapter, we will use a complex algorithm that helps the computer see the world as we do. We will be using a series of camera images to identify road markings.

In this chapter, we will cover the following topics:

- Finding road markings in an image
- Detecting road markings in a video

Finding road markings in an image

Finding road markings is the first step toward building a self-driving car. If a camera sensor can detect road markings correctly, it can follow the lanes and drive safely.

The main steps for detecting road markings are as follows:

1. Loading the image using OpenCV
2. Converting the image into grayscale
3. Smoothing the image
4. Canny edge detection

5. Masking the region of interest in an image
6. Applying `bitwise_and`
7. Implementing the Hough transform
8. Optimizing the detected road markings
9. Detecting road markings in an image
10. Detecting road masking in a video

We will look at these steps in-depth in the following sections.

Loading the image using OpenCV

The first step involves importing the image using OpenCV:

1. We will load the image using the `imread` function and display it using the `imshow` function. Let's import the libraries and load the image:

```
In[1]: import cv2  
In[2]: image = cv2.imread('test_image.jpg')  
In[3]: cv2.imshow('input_image', image)  
In[4]: cv2.waitKey(0)  
In[5]: cv2.destroyAllWindows()
```

2. The imported image looks as follows:



Fig 5.1: The input image

In the next section, we will convert the RGB image into a grayscale image.

Converting the image into grayscale

We learned in Chapter 4, *Computer Vision for Self-Driving Cars*, that a three-channel color image has red, green, and blue channels (each pixel being a combination of these three channel values). A grayscale image has only one channel for each pixel (with 0 being black and 255 being white). Naturally, processing a single-channel image is faster than processing a three-channel color image, and it is less computationally expensive, too.

Also, in this chapter, we will develop an edge-detection algorithm. The edge-detection algorithm's main goal is to identify the boundaries of the objects within an image. Later in this chapter, we will be detecting edges to find a region in an image with a sharp change in the pixels.

Now, as a first step, we will convert the image into grayscale:

1. Import the following libraries, which we need to convert the image into grayscale:

```
In[1]: import cv2  
In[2]: import numpy as np
```

2. Read in and display the image, and then convert it into grayscale:

```
In[3]: image = cv2.imread('test_image.jpg')  
In[4]: lanelines_image = np.copy(image)  
In[5]: gray_conversion= cv2.cvtColor(lanelines_image,  
cv2.COLOR_RGB2GRAY)  
In[6]: cv2.imshow('input_image', gray_conversion)  
In[7]: cv2.waitKey(0)  
In[8]: cv2.destroyAllWindows()
```

The output image is as follows:



Fig 5.2: Grayscale image

In the next section, we will reduce noise and smooth our image. This is because it is important to detect all the true edges and sharp changes in the image, while also filtering out any false edges that may arise as a result of noise.

Smoothing the image

In this section, we will smooth the image with a Gaussian filter. Applying the `GaussianBlur` function from the OpenCV library reduces the noise in our image.

We will apply it using OpenCV, as follows:

1. Start by importing OpenCV and numpy:

```
In[1]: import cv2  
In[2]: import numpy as np
```

2. In the next step, read the input image:

```
In[3]: image = cv2.imread('test_image.jpg')  
In[4]: lanelines_image = np.copy(image)
```

3. Now, we will convert the image into grayscale:

```
In[5]: gray_conversion= cv2.cvtColor(lanelines_image,  
cv2.COLOR_RGB2GRAY)
```

4. Apply `GaussianBlur` using the OpenCV library:

```
In[6]: blur_conversion = cv2.GaussianBlur(gray_conversion, (5,5),0)  
In[7]: cv2.imshow('input_image', blur_conversion)  
In[8]: cv2.waitKey(0)  
In[9]: cv2.destroyAllWindows()
```

The output is as follows:



Fig 5.3: Image after applying Gaussian blur

We smoothed the image and removed noise from it. In the next section, we will perform canny edge detection on the input image.

Canny edge detection

We covered canny edge detection in [Chapter 4, Computer Vision for Self-Driving Cars](#). In this section, we will apply canny edge detection to identify the edges in the image. As we learned in [Chapter 4, Computer Vision for Self-Driving Cars](#), an edge is a region in an image where there is a sharp change in intensity or a sharp change in color between adjacent pixels in an image. This change over a series of pixels is known as the gradient. As we already know, the canny function computes the gradient in all directions of a blurred image and will trace the strongest gradient as a series of pixels.

Now, we will implement canny edge detection using the OpenCV library:

1. Import the libraries we need:

```
In[1]: import cv2  
In[2]: import numpy as np
```

2. Now, apply canny edge detection using OpenCV—specifically, using `cv2.Canny`:

```
In[3]: image = cv2.imread('test_image.jpg')  
In[4]: lanelines_image = np.copy(image)
```

```
In[5]: gray_conversion= cv2.cvtColor(lanelines_image,  
cv2.COLOR_RGB2GRAY)  
In[6]: blur_conversion = cv2.GaussianBlur(gray_conversion, (5,5),0)  
In[7]: canny_conversion = cv2.Canny(blur_conversion, 50,155)  
In[8]: cv2.imshow('input_image', canny_conversion)  
In[9]: cv2.waitKey(0)  
In[10]: cv2.destroyAllWindows()
```

The output is as follows:

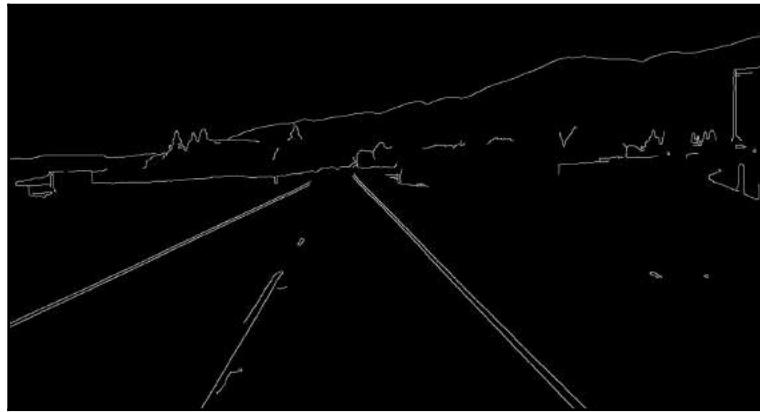


Fig 5.4: The image following canny edge detection

We can see that the strongest gradients in our screenshot are represented by the color white.

In the next section, we will mask the region of interest.

Masking the region of interest

We learned about region-of-interest masking in [Chapter 4, Computer Vision for Self-Driving Cars](#). The next step toward identifying road markings is to mask the region of interest in our image:

1. Import the required libraries:

```
In[1]: import cv2  
In[2]: import numpy as np  
In[3]: import matplotlib.pyplot as plt
```

2. Next, write a function for canny edge detection:

```
In[4]: def canny_edge(image):
    gray_conversion= cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    blur_conversion = cv2.GaussianBlur(gray_conversion,
(5,5),0)
    canny_conversion = cv2.Canny(blur_conversion, 50,150)
    return canny_conversion
```

3. Now, we will write a function for the region-of-interest masking. We will check the image manually to identify the inputs as polygons. We will plot the image and then verify the points mentioned in the polygons:

```
In[5]: def reg_of_interest(image):
    Image_height = image.shape[0]
    polygons = np.array([[(200, Image_height), (1100,
Image_height), (550, 250)]] )
    image_mask = np.zeros_like(image)
    cv2.fillPoly(image_mask, polygons, 255)
    return image_mask
```

The output is as follows; you can see the points marked in the image:

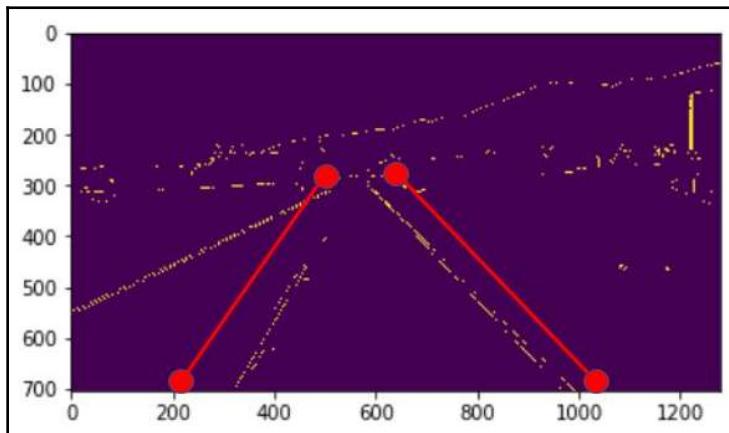


Fig 5.5: The image with marked points

4. Let's now read the input image:

```
In[6]: image = cv2.imread('test_image.jpg')
In[7]: lanelines_image = np.copy(image)
In[8]: canny_conversion = canny_edge(lanelines_image)
```

5. Next, we will apply region-of-interest masking using the OpenCV library's `reg_of_interest` function:

```
In[9]: cv2.imshow('result', reg_of_interest(canny_conversion))
In[10]: cv2.waitKey(0)
In[11]: cv2.destroyAllWindows()
```

The output is as follows:

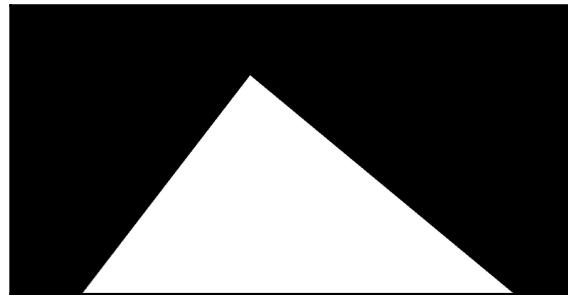


Fig 5.6: The image with region-of-interest masking

In the next section, we will convert the image using `bitwise_and`.

Applying `bitwise_and`

In this section, we will apply `bitwise_and` and multiply all the bits in the black region of the image by 0000 and the white region by 1111, as shown in the following screenshot:

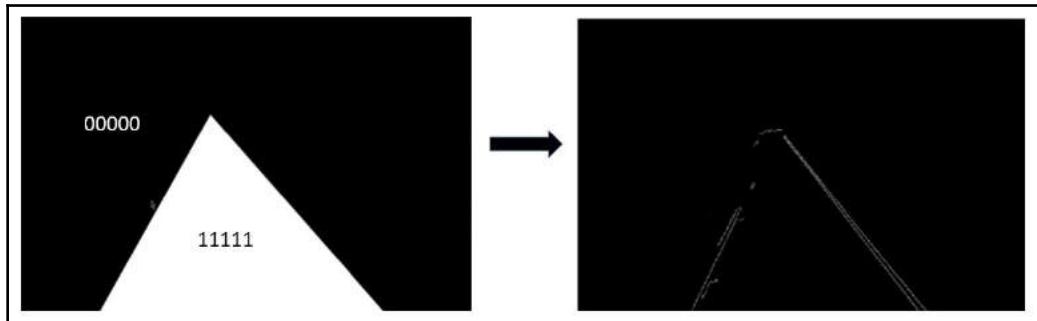


Fig 5.7: `bitwise_and` used on the black and white images

The `bitwise_and` conversion is as follows:

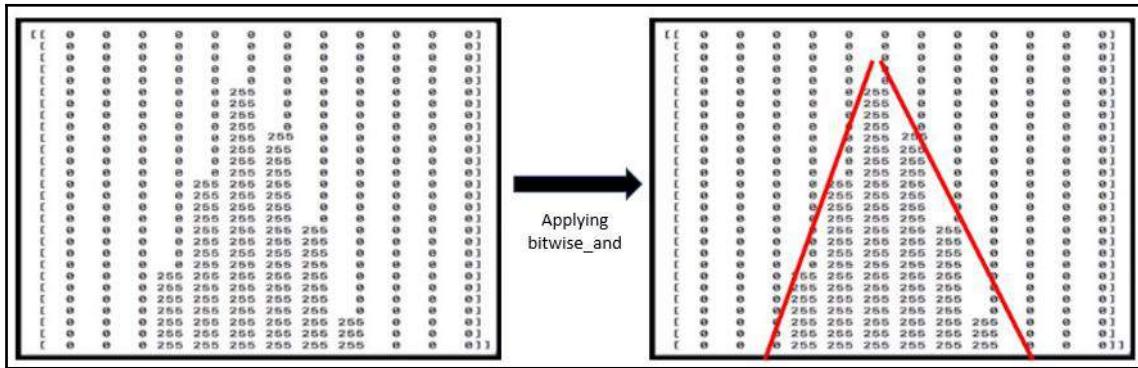


Fig 5.8: The `bitwise_and` conversion

Now, we will implement `bitwise_and` using OpenCV:

1. First, import the required libraries:

```
In[1]: import cv2
In[2]: import numpy as np
In[3]: import matplotlib.pyplot as plt
```

2. Then, write a canny edge detection function:

```
In[4]: def canny_edge(image):
    gray_conversion= cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    blur_conversion = cv2.GaussianBlur(gray_conversion,
(5,5),0)
    canny_conversion = cv2.Canny(blur_conversion, 50,150)
    return canny_conversion
```

3. Modify the region-of-interest masking function by adding `bitwise_and`:

```
In[5]: def reg_of_interest(image):
    image_height = image.shape[0]
    polygons = np.array([[(200, image_height), (1100,
image_height), (551, 250)]])
    image_mask = np.zeros_like(image)
    cv2.fillPoly(image_mask, polygons, 255)
    masking_image = cv2.bitwise_and(image,image_mask)
    return masking_image
```

4. Let's import the input image:

```
In[6]: image = cv2.imread('test_image.jpg')
In[7]: lanelines_image = np.copy(image)
In[8]: canny_conversion = canny_edge(lanelines_image)
```

5. Next, check the masked image:

```
In[9]: cropped_image = reg_of_interest(canny_conversion)
In[10]: cv2.imshow('result', cropped_image)
In[11]: cv2.waitKey(0)
In[12]: cv2.destroyAllWindows()
```

The masked image is as follows:

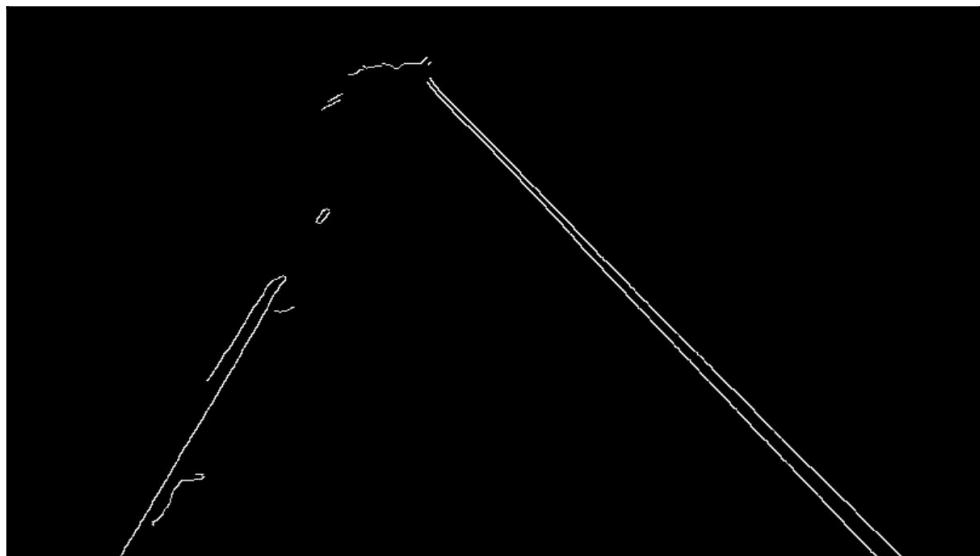


Fig 5.9: The masked image after applying bitwise_and

In the next step, we will apply the Hough transform.

Applying the Hough transform

In Chapter 4, *Computer Vision for Self-Driving Cars*, we looked at the theory behind the Hough transform. We also saw the differences between points in image space and Hough space, as shown in the following screenshot:

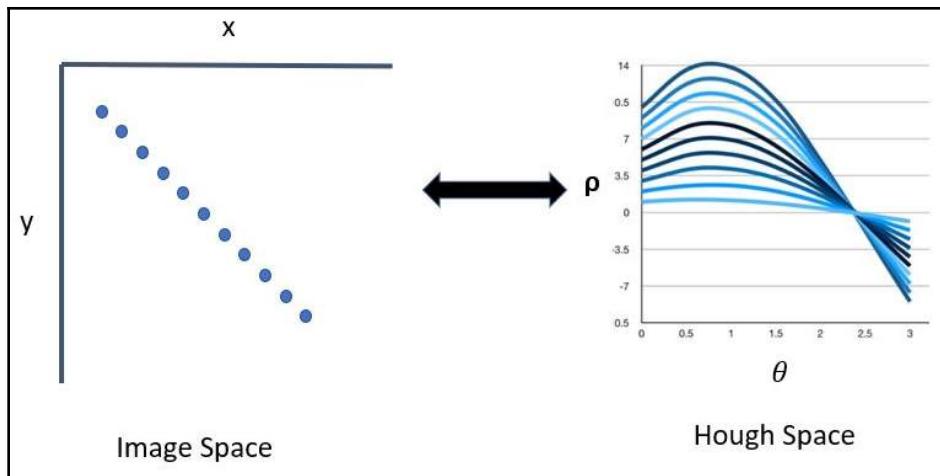


Fig 5.10: Image space to Hough space

Now, we will implement the Hough transform using OpenCV:

1. Import the required libraries:

```
In[1]: import cv2
In[2]: import numpy as np
In[3]: import matplotlib.pyplot as plt
```

2. Use the `canny_edge` function to detect of edges in the image:

```
In[4]: def canny_egde(image):
    gray_conversion= cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    blur_conversion = cv2.GaussianBlur(gray_conversion,
(5,5),0)
    canny_conversion = cv2.Canny(blur_conversion, 50,150)
    return canny_conversion
```

3. We will reuse the `region-of-interest` function from the previous section:

```
In[5]: def reg_of_interest(image):
    image_height = image.shape[0]
    polygons = np.array([[(200, image_height), (1100,
image_height), (551, 250)]])
    image_mask = np.zeros_like(image)
    cv2.fillPoly(image_mask, polygons, 255)
    masking_image = cv2.bitwise_and(image,image_mask)
    return masking_image
```

4. Write a function to display the lines:

```
In[6]: def show_lines(image, lines):
    lines_image = np.zeros_like(image)
    if lines is not None:
        for line in lines:
            X1, Y1, X2, Y2 = line.reshape(4)
            cv2.line(lines_image, (X1, Y1), (X2, Y2),
(255,0,0), 10)
    return lines_image
```

5. Add the Hough transform codes, as follows:

```
In[7]: image = cv2.imread('test_image.jpg')
In[8]: lanelines_image = np.copy(image)
In[9]: canny_conv = canny_edge(lanelines_image)
In[10]: cropped_image = reg_of_interest(canny_conv)
In[11]: lane_lines = cv2.HoughLinesP(cropped_image, 2, np.pi/180,
100, np.array([]), minLineLength= 40, maxLineGap=5)
In[12]: linelines_image = show_lines(lanelines_image, lane_lines)
```

6. Next, we will look at the processed image and check the output:

```
In[14]: cv2.imshow('result', linelines_image)
In[15]: cv2.waitKey(0)
In[16]: cv2.destroyAllWindows()
```

The output image is as follows:

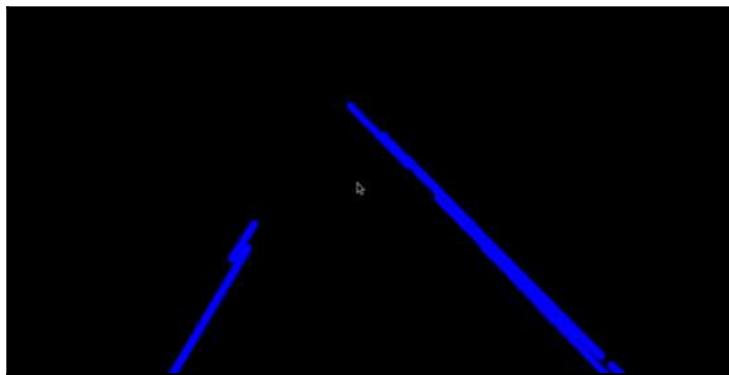


Fig 5.11: The image after applying the Hough transform

The image we detected is displayed on a black background. Therefore, the final step is to combine this with our original image.

7. Add `combine_image` to merge the result of *Fig 5.11* with the input image, using `cv2.addWeighted`:

```
In[17]: image = cv2.imread('test_image.jpg')
In[18]: lane_image = np.copy(image)
In[19]: canny = canny_edge(lane_image)
In[20]: cropped_image = reg_of_interest(canny)
In[21]: lines = cv2.HoughLinesP(cropped_image, 2, np.pi/180, 100,
np.array([]), minLineLength= 40, maxLineGap=5)
In[22]: line_image = show_lines(lane_image, lines)
In[23]: combine_image = cv2.addWeighted(lane_image, 0.8,
line_image, 1, 1)
In[24]: cv2.imshow('result', combine_image)
In[25]: cv2.waitKey(0)
In[26]: cv2.destroyAllWindows()
```

The output image is as follows:



Fig 5.12: The output image

This is how to identify road markings. We have learned how to identify lane lines in an image. We took these lines and placed them on a random black image that has the same dimensions as our original image. By blending the two, we were able to ultimately place our detected lines back onto our original image. In the next section, we will optimize the detected road markings.

Optimizing the detected road markings

We have already identified road markings in our image from a series of points in the gradient image using the Hough transform detection algorithm. We then took these lines and placed them in a blank image, which we then merged with our color image. We then displayed the lines on the input image. Now, we will further optimize it.

It is important to first recognize that the lines currently displayed correspond to the section that exceeded the voting threshold. They were voted as the lines that best described the data. Instead of having multiple lines, as seen on the left line in our image, we will now average out their slopes and y -intercepts into a single line that traces both of the lanes.

We will do this by adding two new functions to the code `make_coordinates` and `average_slope_intercept`:

1. Import the required libraries:

```
In[1]: import cv2
In[2]: import numpy as np
In[3]: import matplotlib.pyplot as plt
```

2. Now, we will define the `make_coordinates` function. This function will collect the `line_params` value (the average of all the slopes) from the `average_slope_intercept` function and unpack it. We will set y_2 to $3/5 * y_1$ as we want to consider the line up to $3/5$ of the y -axis. We know that the equation of a straight line is $y = mx + c$, so we can rewrite it. We are going to find x_1 and x_2 by using the following formula:

```
In[4]: def make_coordinates(image, line_params):
    slope, intercept = line_params
    y1 = image.shape[0]
    y2 = int(y1*(3/5))
    x1 = int((y1 - intercept)/slope)
    x2 = int((y2 - intercept)/slope)
    return np.array([x1, y1, x2, y2])
```

3. Next, we will define the `average_slope_intercept` function to average out the slopes and y -intercepts into a single line:

```
In[5]: def average_slope_intercept(image, lines):
    left_fit = []
    right_fit = []
    for line in lines:
        x1, y1, x2, y2 = line.reshape(4)
        parameter = np.polyfit((x1, x2), (y1, y2), 1)
```

```

slope = parameter[0]
intercept = parameter[1]
if slope < 0:
    left_fit.append((slope, intercept))
else:
    right_fit.append((slope, intercept))
left_fit_average = np.average(left_fit, axis=0)
right_fit_average = np.average(right_fit, axis=0)
left_line = make_coordinates(image, left_fit_average)
right_line = make_coordinates(image, right_fit_average)
return np.array([left_line, right_line])

```

In the previous function, `left_fit` and `right_fit` are the lists that collected the coordinates of the average value of the lines on the left and right sides. Here, we looped all the lines and reshaped them into a four-dimensional array using `line.reshape(4)`. Next, we used `np.polyfit`, which fits a first-degree polynomial (a linear function). It fits the polynomial of x and y and returns a vector of coefficients that describes the slope and intercept of a line. The values of the slope and intercept are collected from the matrix of the parameters. We know that the value of the slope is always negative for the left side of the line, so we wrote a condition to append all the slope values on the left side and right side of the image. Then, we averaged out the intercepts of the left side and right side using `np.average`. The averaged values are stored in `left_fit_average` and `right_fit_average`. Finally, we found the x and y coordinates of the line using the `make_coordinates` function.

4. Next, define the `canny_edge` function:

```

In[6]: def canny_edge(image):
    gray_conversion = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    blur_conversion = cv2.GaussianBlur(gray_conversion,
(5, 5), 0)
    canny_conversion = cv2.Canny(blur_conversion, 50, 150)
    return canny_conversion

```

5. Then, define the `display_line` function:

```

In[7]: def show_lines(image, lines):
    lanelines_image = np.zeros_like(image)
    if lines is not None:
        for line in lines:
            X1, Y1, X2, Y2 = line.reshape(4)
            cv2.line(lanelines_image, (X1, Y1), (X2, Y2),
(255, 0, 0), 10)
    return lanelines_image

```

6. Finally, define the region-of-interest masking function:

```
In[8]: def reg_of_interest(image):
    image_height = image.shape[0]
    polygons = np.array([[(200, image_height), (1100,
    image_height), (551, 250)]])
    image_mask = np.zeros_like(image)
    cv2.fillPoly(image_mask, polygons, 255)
    masking_image = cv2.bitwise_and(image, image_mask)
    return masking_image
```

7. To display the final output image, use the following code:

```
In[9]: image = cv2.imread('test_image.jpg')
In[10]: lanelines_image = np.copy(image)
In[11]: canny_image = canny_edge(lanelines_image)
In[12]: cropped_image = reg_of_interest(canny_image)
In[13]: lines = cv2.HoughLinesP(cropped_image, 2, np.pi/180, 100,
np.array([]), minLineLength= 40, maxLineGap=5)
In[14]: averaged_lines = average_slope_intercept(lanelines_image,
lines)
In[15]: line_image = show_lines(lanelines_image, averaged_lines)
In[16]: combine_image = cv2.addWeighted(lanelines_image, 0.8,
line_image, 1, 1)
In[17]: cv2.imshow('result', combine_image)
In[18]: cv2.waitKey(0)
In[19]: cv2.destroyAllWindows()
```

We have created an end-to-end pipeline to detect road markings. This is the final image:



Fig 5.13: The output image with optimization

In the next section, we will detect road markings in a video.

Detecting road markings in a video

In this section, we will detect road markings in a video. To do so, we will use OpenCV:

1. First, import the required libraries:

```
In[1]: import cv2  
In[2]: import numpy as np  
In[3]: import matplotlib.pyplot as plt
```

2. Define the `make_coordinates` function:

```
In[3]: def make_coordinates(image, line_parameters):  
    slope, intercept = line_parameters  
    y1 = image.shape[0]  
    y2 = int(y1*(3/5))  
    x1 = int((y1- intercept)/slope)  
    x2 = int((y2 - intercept)/slope)  
    return np.array([x1, y1, x2, y2])
```

3. Define the `average_slope_intercept` function:

```
In[4]: def average_slope_intercept(image, lines):  
    left_fit = []  
    right_fit = []  
    for line in lines:  
        x1, y1, x2, y2 = line.reshape(4)  
        parameter = np.polyfit((x1, x2), (y1, y2), 1)  
        slope = parameter[0]  
        intercept = parameter[1]  
        if slope < 0:  
            left_fit.append((slope, intercept))  
        else:  
            right_fit.append((slope, intercept))  
    left_fit_average = np.average(left_fit, axis=0)  
    right_fit_average = np.average(right_fit, axis =0)  
    left_line = make_coordinates(image, left_fit_average)  
    right_line = make_coordinates(image, right_fit_average)  
    return np.array([left_line, right_line])
```

4. Define the `canny_edge` function:

```
In[5]: def canny_edge(image):  
    gray_conversion= cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)  
    blur_conversion = cv2.GaussianBlur(gray_conversion,  
(5,5),0)  
    canny_conversion = cv2.Canny(blur_conversion, 50,150)  
    return canny_conversion
```

5. Define the `display_line` function:

```
In[6]: def show_lines(image, lines):
    line_image = np.zeros_like(image)
    if lines is not None:
        for line in lines:
            x1, y1, x2, y2 = line.reshape(4)
            cv2.line(line_image, (x1, y1), (x2, y2), (255,0,0),
10)
    return line_image
```

6. Define the region-of-interest masking function:

```
In[7]: def reg_of_interest(image):
    image_height = image.shape[0]
    polygons = np.array([(200, image_height), (1100,
image_height), (550, 250)])
    image_mask = np.zeros_like(image)
    cv2.fillPoly(image_mask, polygons, 255)
    masking_image = cv2.bitwise_and(image,image_mask)
    return masking_image
```

7. Finally, display the video:

```
In[8]: cap = cv2.VideoCapture("test2.mp4")
In[9]: while(cap.isOpened()):
    _, frame = cap.read()
    canny_image = canny_edge(frame)
    cropped_canny = reg_of_interest(canny_image)
    lines = cv2.HoughLinesP(cropped_canny, 2, np.pi/180,
100,
                           np.array([]), minLineLength=40,maxLineGap=5)
    averaged_lines = average_slope_intercept(frame, lines)
    line_image = show_lines(frame, averaged_lines)
    combo_image = cv2.addWeighted(frame, 0.8, line_image,
1, 1)
    cv2.imshow("result", combo_image)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
In[10]: cap.release()
In[11]: cv2.destroyAllWindows()
```

The preceding code results in the following video:



Fig 5.14: A screenshot of the video

We have successfully developed a lane line-marking algorithm, which is an important task in the development of self-driving cars.

Summary

This chapter explained how to find road markings in images and videos using OpenCV techniques. This is a starter project where we have implemented a road-marking detection software pipeline using OpenCV. We have applied various techniques, such as canny edge detection, the Hough transform, and grayscale conversion, which we covered in Chapter 4, *Computer Vision for Self-Driving Cars*.

In the next chapter, we will look at how to use deep learning to handle image data, starting with convolutional neural networks. We will learn about the underlying concepts of convolution neural networks and implement a traffic sign classifier.

6

Improving the Image Classifier with CNN

If you've been following the latest news on **self-driving cars (SDCs)**, you will have heard about **convolutional neural networks (CNNs, or ConvNets)**. We use ConvNets to perform a multitude of perception tasks for **SDCs**. In this chapter, we will take a deeper look at this fascinating architecture and understand its importance. Specifically, you will learn how convolutional layers use cross-correlation, instead of general matrix multiplication, to tailor neural networks to the image input data. We'll also cover the advantages of these models over standard feed-forward neural networks.

ConvNets have neurons with learnable weights and biases. Similar to neural networks, each neuron in a ConvNet receives input, and then performs a dot product and follows non-linearity as well.

The pixels of raw images of the network only express a single, differentiable score function. There is still a loss function (for example, softmax) on the last layer.

In this chapter, we will cover the following topics:

- Images in computer format
- Introducing CNNs
- Introduction to handwritten digit recognition

Images in computer format

We've already read about image formatting in computers in Chapter 4, *Computer Vision for Self-Driving Cars*. Basically, there are three channels red, green, and blue, which are popularly known as **RGB**. They have their respective pixel values. So, if we will say the size of an image is $B \times A \times 3$, this means there are **B** rows, **A** columns, and 3 channels. If the image size is $28 \times 28 \times 3$, this means there are 28 rows, 28 columns, and 3 channels.

This is how our computer sees images. For black and white images, there are only two channels.

In the following screenshot, you can see a visual example of a computer viewing an image:

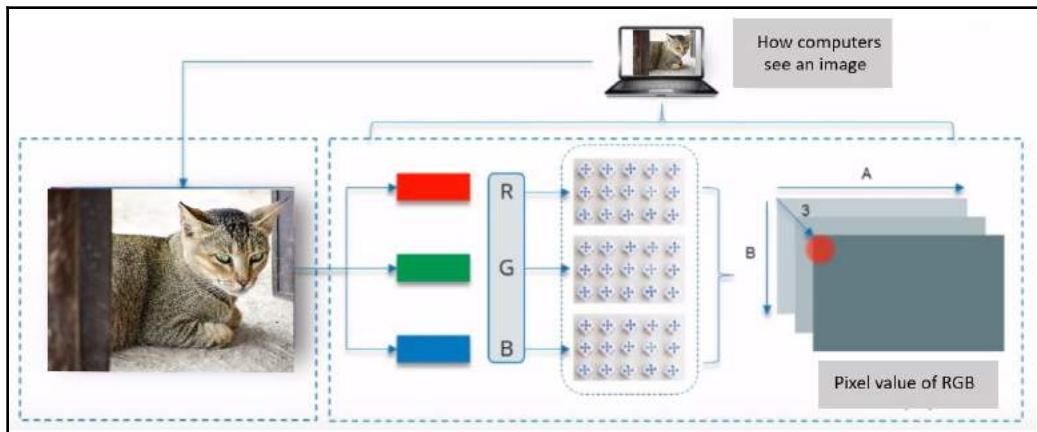


Fig 6.1: Computer viewing an image

In the next section, we will read about why we need CNNs.

The need for CNNs

We need CNNs because neural networks do not scale well to image data. In Chapter 4, *Computer Vision for Self-Driving Cars*, we discussed how images are stored. When we build a simple image classifier, it takes color images with a size of 64×64 (height x width).

So, the input size for the neural network will be $64 \times 64 \times 3 = 12,288$.

Therefore, our input layer will have 12,288 weights. If we use an image with a size of $128 \times 128 \times 3$, we will have 49,152 weights. If we add hidden layers, we will see that it will exponentially increase in training time. The CNN doesn't actually reduce the weights in the input layer. It finds a representation internally in hidden layers to basically take advantage of how images are formed. This way, we can actually make our neural network much more effective at dealing with image data.

In the next section, we will read about the intuition behind these neural networks.

The intuition behind CNNs

A CNN is a type of feed-forward artificial neural network where the connection between its neurons is inspired by an animal's visual cortex.

The visual cortex is the part of the brain's cerebral cortex that processes visual information:

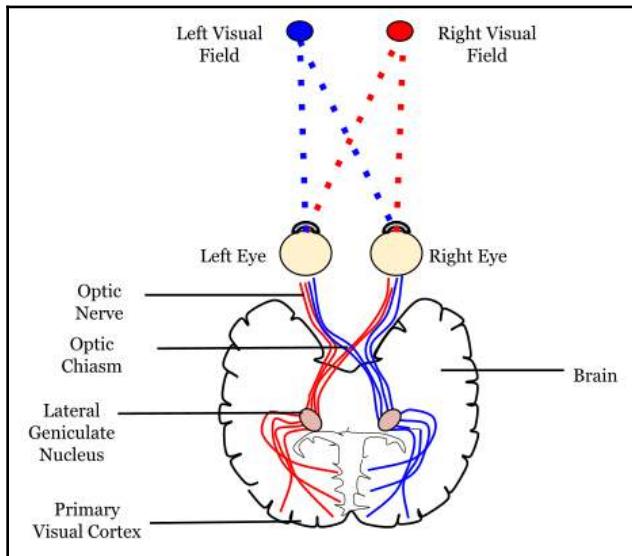


Fig 6.2: Visual cortex

The visual cortex is a small region of cells that is sensitive to a specific region of the visual field. For example, some neurons in the visual cortex fire when exposed to vertical edges, some fire when exposed to horizontal edges, and some will fire for diagonal edges. That is the process behind CNNs.



You can read more about the visual cortex at https://en.wikipedia.org/wiki/Visual_cortex.

We have already studied, in Chapter 2, *Dive Deep into Deep Neural Networks*, how biological neural networks can be converted into artificial networks.

In the next section, we will study CNNs in-depth.

Introducing CNNs

In the following screenshot, we can see all the layers of a CNN. We will go through each of them in detail:

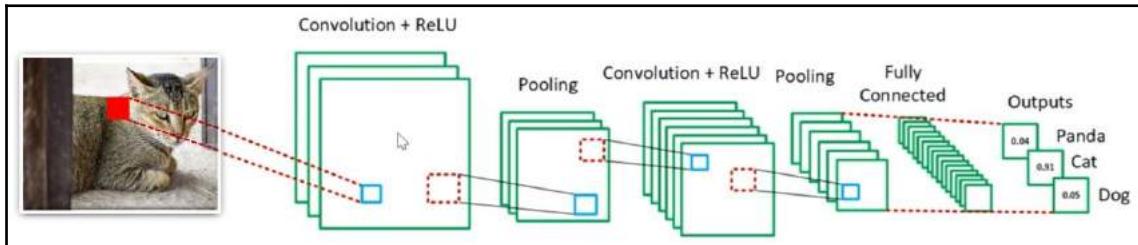


Fig 6.3: CNN layers

We know that in this neural network, we have an input layer and hidden layers.

The layers of a CNN are as follows:

- The input layer
- The convolution layer
- The ReLU layer
- The pooling layer
- The fully connected layer

In the following section, we will learn about 3D layers.

Why 3D layers?

3D layers allow us to use convolutions to learn the image features. This helps the network decrease its training time as there will be less weight in the deep network.

The three dimensions of an image are as follows:

- Height
- Width
- Depth (RGB)

The 3D layers of an image can be seen in the following screenshot:

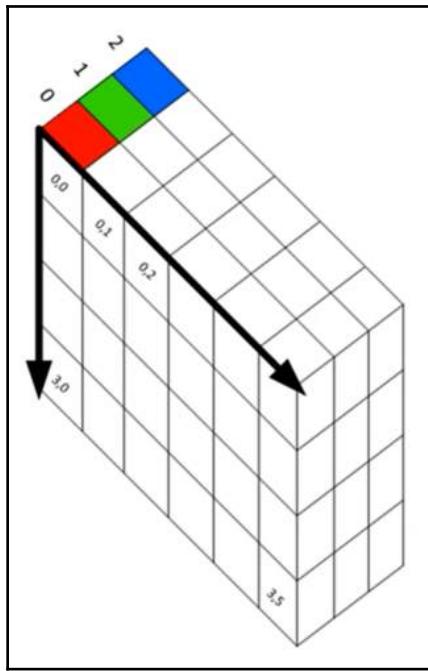


Fig 6.4: 3D layers

In the next section of this chapter, we will understand the convolution layer.

Understanding the convolution layer

The convolution layer is the most important part of a CNN as it is the layer that learns the image features. Before we dive deep into convolutions, we will learn about image features. Image features are the part of the image that we are most interested in.

Some examples of image features are as follows:

- Edges
- Colors
- Patterns/shapes

Before CNNs, the extraction of features from an image was a tedious process—the feature engineering done for one set of images would not be appropriate for another set of images.

Now, we will see what exactly a convolution is. In simple terms, convolution is a mathematical term to describe the process of combining two functions to produce a third function. The third function, or the output, is called the **feature map**. Convolution is the action of using a kernel or filter applied to an input image, and the output is the feature map.

The convolution feature is executed by sliding a kernel over the input image. Generally, the convolution-sliding process is done with a simple matrix multiplication or dot product.

Now, we will see an example of how convolution works:

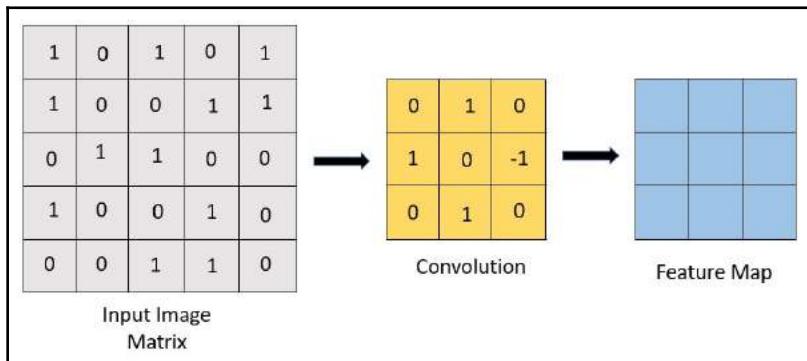


Fig 6.5: Example of convolution

In the following steps, we will learn to create a feature map using convolution:

1. The first step is to slide over the image matrix, as shown in the following screenshot:

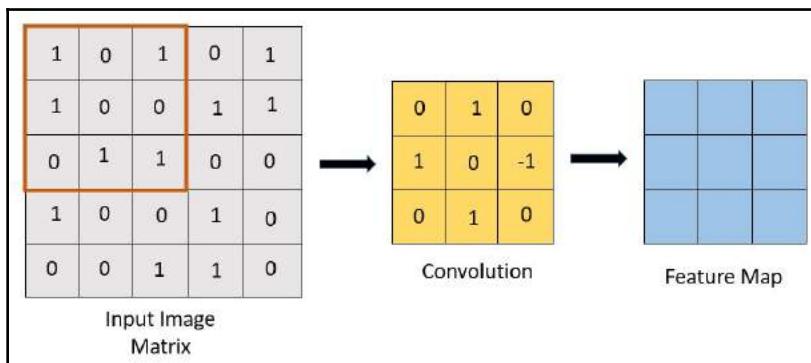


Fig 6.6: Sliding over an image

2. The next step is to multiply the convolution by the input image matrix:

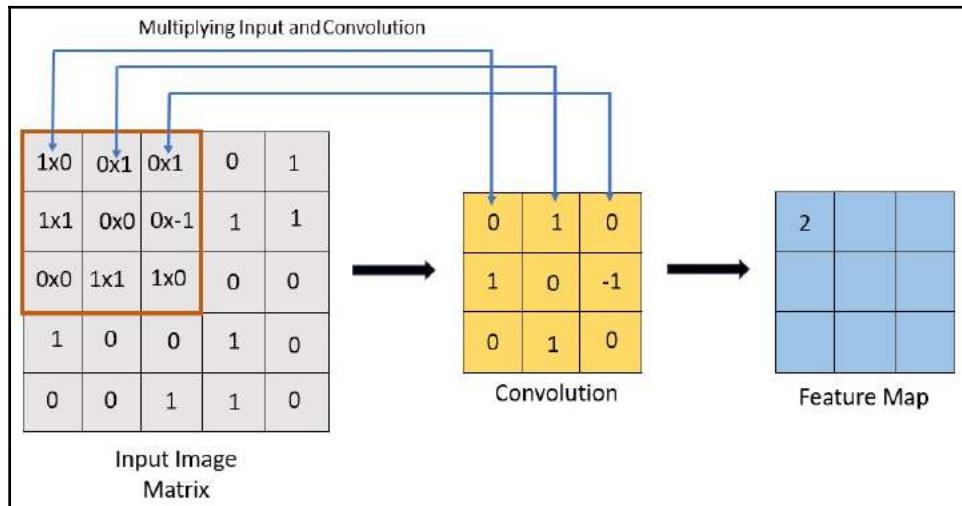


Fig 6.7: Multiplying the convolution by the image matrix

In the following screenshot, we can see how the feature map is created after the convolution kernel is applied:

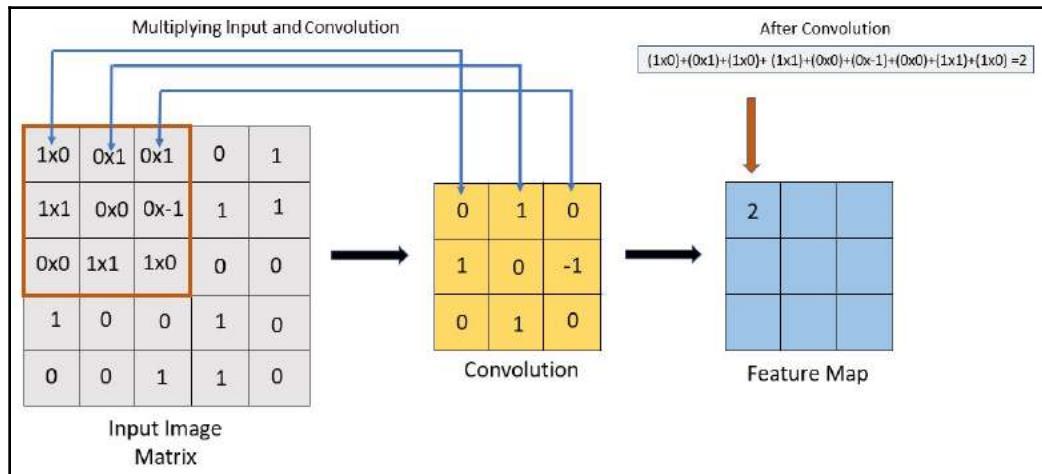


Fig 6.8: Creation of the feature map

3. Now, we will slide the kernel to get the next feature map output:

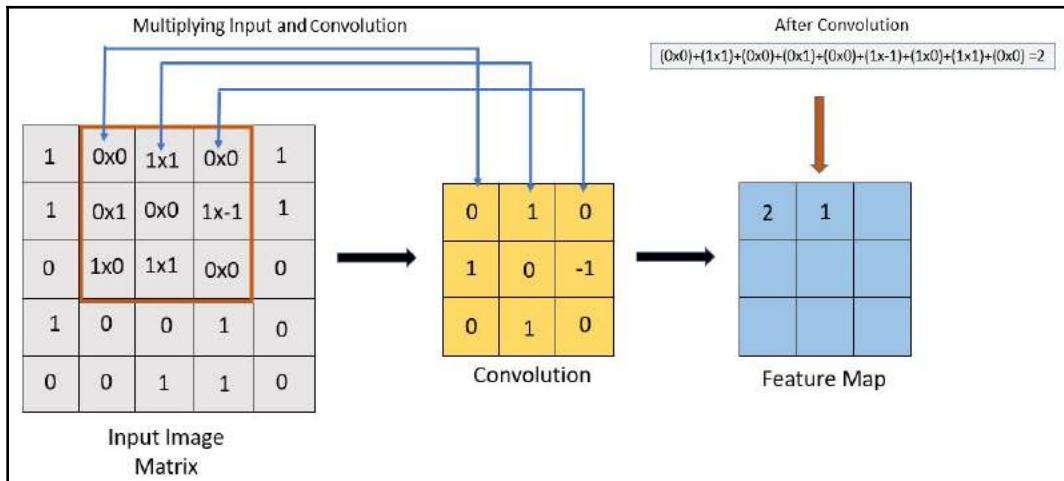


Fig 6.9: Creation of the feature map's next value

In the preceding example, the 3×3 kernel slides 9 times. We will get the 9 values as the feature map result. Here are a few important points on the effect of kernels:

- The kernel produces scalar outputs.
- The result of the feature map depends on the value of the kernels.
- Convolving with different kernels produces interesting feature maps that can be used to detect different features.
- After compressing the feature in the form of a feature map, the convolution keeps the spatial relationship by retaining the compressed information of the image.

In CNNs, we can use many filters, as we saw in [Chapter 4, Computer Vision for Self-Driving Cars](#).

For example, if we have 12 filters in a CNN, and we apply 12 filters with a size of $3 \times 3 \times 3$ to our input image with a size of $28 \times 28 \times 3$, we will produce 12 feature maps, also known as activation maps. These outputs are stacked together and treated as another 3D matrix, but in our example, the size of the output will be $28 \times 28 \times 12$. This 3D output will then be inputted to the next layer of the CNN.



Each cell of the activation map matrix can be considered a feature extractor or a neuron that looks at a specific region of the image. In the first few layers, the neurons are activated when the edges or other lower-level features are detected. In deeper layers, the neurons will detect high-level or bigger pictures.

In the next section, we will read about the depth, stride, and padding hyperparameters.

Depth, stride, and padding

Depth, stride, and padding are the hyperparameters used to tweak the size of the convolutional filters. In the previous section, *Understanding the convolution layer*, we applied 3×3 filters or kernels for the convolution of a CNN. But the question is, *does the filter have to be 3×3 ?* How many filters do we need? Are we going to shift over pixel by pixel?

We can have filters of greater size than 3×3 . It is possible to do this by tweaking the following parameters. You can also tweak these parameters to control the size of the feature maps:

- Kernel size ($K \times K$)
- Depth
- Stride
- Padding

Depth

Depth tells us the number of filters used. It does not relate to the image depth, nor to the number of hidden layers in the CNN. Each filter or kernel learns different feature maps that are activated in the presence of different image features, such as edges, patterns, and colors.

Stride

Stride basically refers to the step size we take when we slide the kernel across the input image.

An example of a stride of 1 is shown in the following screenshot:

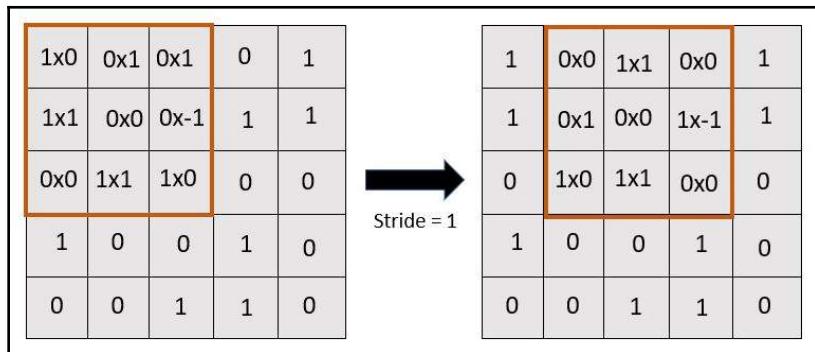


Fig 6.10: Stride of 1

In a stride of 1, the value of the feature map is 9. Similarly, a stride of 2 looks as follows:

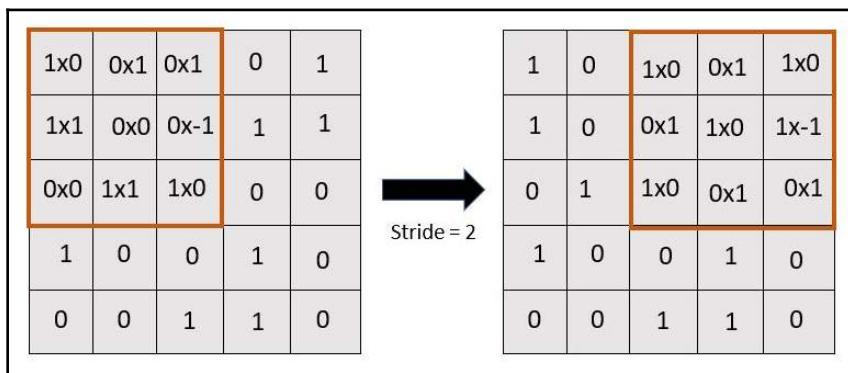


Fig 6.11: Stride of 2

When we have a stride of 2, the value of the feature map will be 4, which is equivalent to 2×2 .

Stride is important because of the following points:

- The stride controls the size of the convolution layer output.
- Using larger strides produces less overlap in kernels.
- Stride is one of the methods to control the spatial input size—that is, passing information to other input layers without losing it.

Zero-padding

Zero-padding is a very simple concept that we apply to the border of our input. With a stride of 1, the output of the feature map will be a 3×3 matrix. We can see that after applying a stride of 1, we end up with a tiny output. This output will be the input for the next layer. In this way, there are high chances of losing information. So, we add a border of zeros around the input, as shown in the following screenshot:

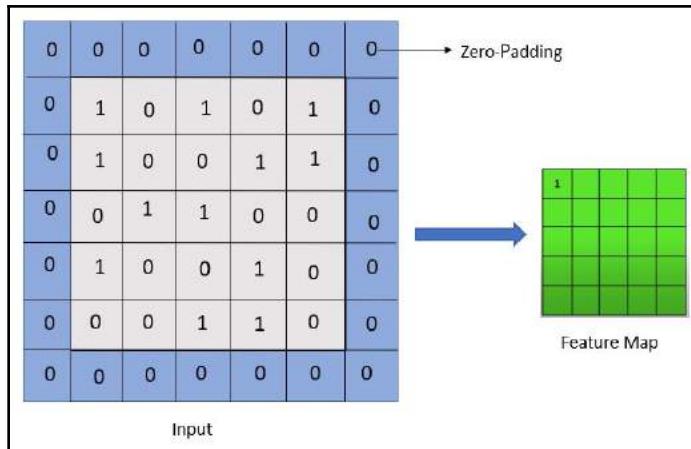


Fig 6.12: Zero-padding

Adding zeros around the border is equivalent to adding a black border around an image. We can also set the padding to 2 if required.

Now, we will calculate the output of the convolution mathematically. We have the following parameters:

- Kernal/filter size, K
- Depth, D
- Stride, S
- Zero-padding, P
- Input image size, I

To ensure that the filters cover the full input image symmetrically, we'll use the following equation to do the sanity check; it is valid if the result of the equation is an integer:

$$(1 - K + 2P) \div S + 1$$

In the next section, we will learn about the activation function known as **Rectified Linear Unit (ReLU)**.

ReLU

ReLU is the activation layer of choice for CNNs. We studied activation layers in [Chapter 2](#), *Dive Deep into Deep Neural Networks*. As we know, we need to introduce non-linearity to our model as the convolution process is linear. So, we apply an activation function to the output of the CNN.

The ReLU function simply changes all the negative values to 0, while positive values are unchanged, as shown in the following screenshot:

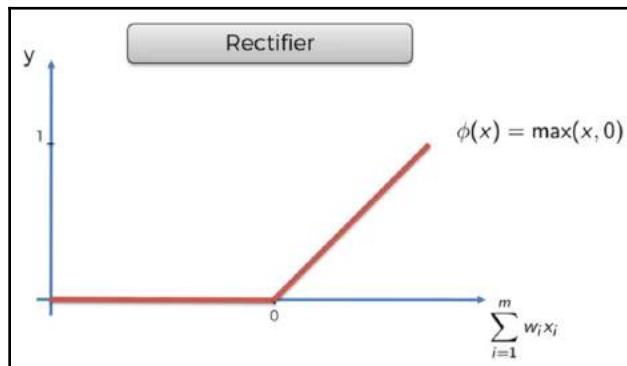


Fig 6.13: ReLU

An example of ReLU introducing non-linearity in the feature map's output can be seen in the following screenshot:

2	1	-3		2	1	0
1	2	1	→	1	2	1
2	1	-1	Applying ReLU	2	1	0

Fig 6.14: Applying ReLU

In the next section, we will learn about fully connected layers.

Fully connected layers

We have already learned about fully connected layers in [Chapter 2, Dive Deep into Deep Neural Networks](#). Having fully connected layers simply means that all the nodes in one layer are connected to the outputs of the next layers. The output of the fully connected layer is a class of probabilities, where each class is assigned a probability. All probabilities must sum up to 1. The activation function used at the output of the layer is called the **softmax function**.

The softmax function

The activation function used to produce the probabilities per class is called the **softmax function**. It turns the output of the fully connected layer, or the last layer, into probabilities. The sum of the probabilities of the classes is $1 - \text{Panda} = 0.04$, $\text{Cat} = 0.91$, and $\text{Dog} = 0.05$, which totals 1.

We can see the values of the softmax function in the following screenshot:

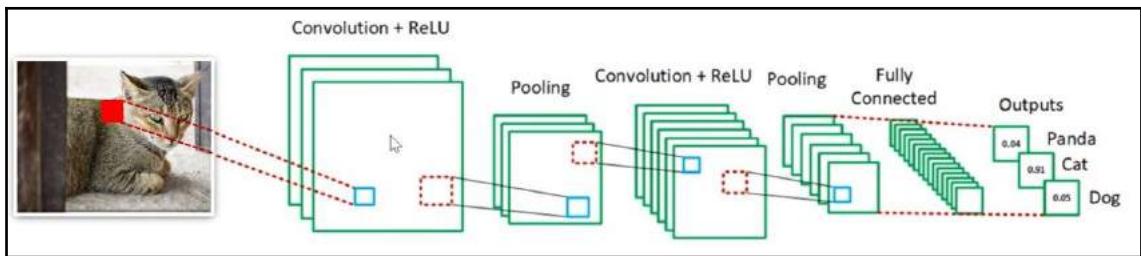


Fig 6.15: Output of softmax

In the next section of this chapter, we will implement a handwritten digit recognition CNN model using the Keras API.

Introduction to handwritten digit recognition

The MNIST dataset is one of the most popular datasets in the field of computer vision. It is a fairly large dataset, consisting of 60,000 training images and 10,000 test images.

We can see a sample of it in the following screenshot:

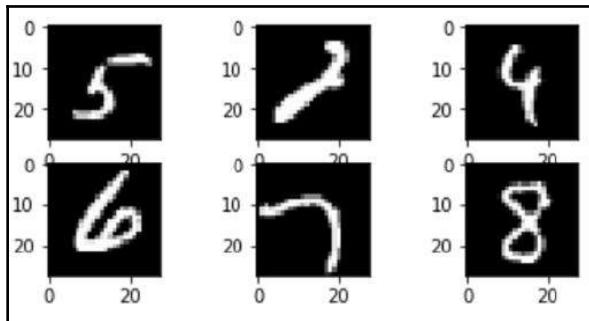


Fig 6.16: The MNIST dataset



You can find out more about the MNIST dataset at <http://yann.lecun.com/exdb/mnist/> and https://en.wikipedia.org/wiki/MNIST_database.

Now, we will learn about the problem statement and implement a CNN using Keras.

Problem and aim

The MNIST dataset was developed for the US postal service to automatically read written postcodes on mail. The aim of our classifier is simple: to take the digits in the format provided and correctly identify the given digits. Digit identification has multiple applications in self-driving cars; one of the important applications is traffic sign detection—for example, detecting the speed limit.

You can see an example in the following screenshot:



Fig 6.17: Speed limit traffic signs

We are going to build a traffic sign detector in the next chapter using a German traffic sign dataset.

In the next section, we will start by loading the data.

Loading the data

Loading the data is a simple but obviously integral first step to creating a deep learning model. Fortunately, Keras has some built-in data loaders that are simple to execute. Data is stored in an array:

1. First, we will import the keras dataset from TensorFlow:

```
from keras.datasets import mnist
```

2. Then, we will create the test and train datasets:

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

3. Now, we will print and check the shape of the x_train data:

```
print(x_train.shape)
```

4. The shape of x_train is as follows:

```
(60000, 28, 28)
```

One of the confusing things that newcomers face when using Keras is getting their dataset in the correct shape (dimensionality) required for Keras.

5. When we first load our dataset to Keras, it comes in the form of 60,000 images, 28 x 28 pixels. Let's inspect this in Python by printing the initial shape, the dimension, and the number of samples and labels in our training data:

```
print ("Initial shape & Dimension of x_train:", str(x_train.shape))
print ("Number of samples in our training data:",
str(len(x_train)))
print ("Number of labels in our test data:", str(len(x_test)))
```

Here, we can check the result:

```
Initial shape & Dimension of x_train: (60000, 28, 28)
Number of samples in our training data: 60000
Number of labels in our test data: 10000
```

6. We will print the number of samples and labels of the test data:

```
print("Number of samples in test data:"+ str(len(x_test)))
print("Number of labels in the test data:"+str(len(y_test)))
```

The result of the test data is as follows:

```
Number of samples in test data:10000
Number of labels in the test data:10000
```

Now, we will look into the data using OpenCV. First, we will import the OpenCV and numpy libraries:

```
import cv2
import numpy as np
for i in range(0,6):
    random_num = np.random.randint(0, len(x_train))
    img = x_train[random_num]
    window_name = 'Random Sample #' +str(i)
    cv2.imshow(window_name, img)
    cv2.waitKey()
cv2.destroyAllWindows()
```

The resulting output images looks like this:

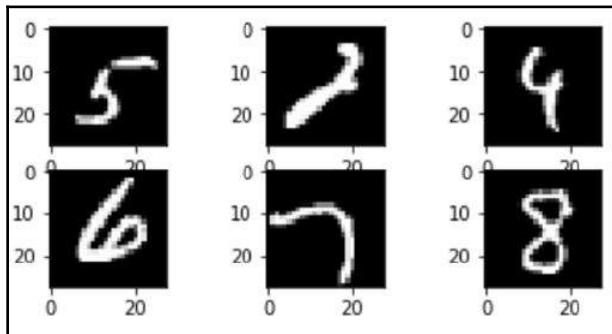


Fig 6.18: Displaying images using OpenCV

Now, we will look into the data using matplotlib:

```
import matplotlib.pyplot as plt
plt.subplot(334)
random_num = np.random.randint(0,len(x_train))
plt.imshow(x_train[random_num], cmap=plt.get_cmap('gray'))
plt.subplot(335)
random_num = np.random.randint(0,len(x_train))
plt.imshow(x_train[random_num], cmap=plt.get_cmap('gray'))
plt.subplot(336)
random_num = np.random.randint(0,len(x_train))
plt.imshow(x_train[random_num], cmap=plt.get_cmap('gray'))
plt.subplot(337)
```

```
random_num = np.random.randint(0, len(x_train))
plt.imshow(x_train[random_num], cmap=plt.get_cmap('gray'))
plt.subplot(338)
random_num = np.random.randint(0, len(x_train))
plt.imshow(x_train[random_num], cmap=plt.get_cmap('gray'))
plt.subplot(339)
random_num = np.random.randint(0, len(x_train))
plt.imshow(x_train[random_num], cmap=plt.get_cmap('gray'))
plt.show()
```

The result is as follows:

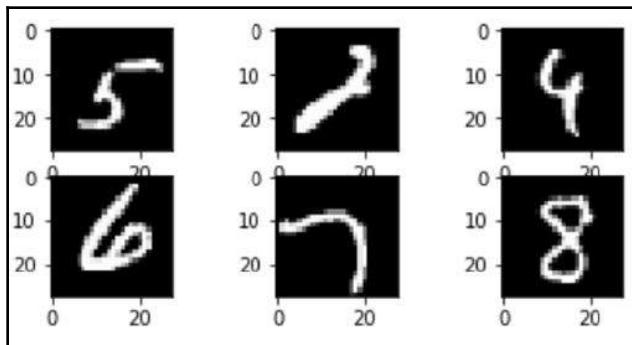


Fig 6.18: Displaying images using matplotlib

However, the input required by Keras means we have to define it in the following format:

- Number of samples, rows, columns, depth

Therefore, since our MNIST dataset is in grayscale, we need it in the following form:

- The grayscale form is $60000 \times 28 \times 28 \times 1$, where the count of the image is 60000, the height of the image is 28, the width is 28, and the channel is 1 as it is grayscale (if it were in color, then the channel would be 3 and it would be represented as $60000 \times 28 \times 28 \times 3$).
- Using NumPy's `reshape` function, we can easily add a fourth dimension to our data.

In the next section, we will look at reshaping the data for Keras.

Reshaping the data

The following code helps us to reshape the Keras input:

```
img_rows = x_train[0].shape[0]
img_cols = x_train[1].shape[0]
x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

In the next section, we will normalize the data between 0 and 1.

The transformation of data

We're going to look at transformations on the training and test image data. For `x_train` and `x_test`, we need to do the following:

1. Add a fourth dimension going from (60000, 28, 28) to (60000, 28, 28, 1),
2. Change it to the `Float32` data type.
3. Normalize it between 0 and 1 (by dividing by 255).

In the following code block, we will perform normalization on the data:

```
x_train /=255
x_test /=255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

The shape of the data doesn't change after normalization:

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

In the next section, we will perform one-hot encoding on our target variables.

One-hot encoding the output

In this section, we're going to one-hot encode the output data. By using one-hot encoding we can convert a categorical variable, and the variable with a new format helps to do a better machine learning prediction. It is easier for the computer as well to interpret the inputs in the form of one-hot encoding.

An example of one-hot encoding can be seen in the following screenshot:

The diagram illustrates the one-hot encoding process. At the top is a table with three rows of data:

Product Name	Categorical Value	Price
Product A	1	100
Product B	2	200
Product C	3	300
Product C	3	500

A large black arrow points downwards from this table to another table below, labeled "One-Hot Encoding".

Product A	Product B	Product C	Price
1	0	0	100
0	1	0	200
0	0	1	300
0	0	1	500

Fig 6.19: One-hot encoding

In the preceding screenshot, we have three products, and their categorical values are **1**, **2**, and **3**. We can see how products are represented by one-hot encoding: for **Product A**, it is **(1, 0, 0)** and for **Product B**, it is **(0, 1, 0)**. Similarly, if we want to do the same for our data, we will get **(0, 0, 0, 0, 1, 0, 0, 0, 0, 0)** for 5.

The following code will help us to one-hot encode the output:

```
from keras.utils import np_utils

y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)

print ("Number of classes: " + str(y_test.shape[1]))

num_classes = y_test.shape[1]
num_pixels = x_train.shape[1] * x_train.shape[2]

y_train[0]
```

The output after conversion into one-hot encoding looks as follows:

```
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

In the next section, we will build and compile the model.

Building and compiling our model

We have already read about building and compiling models in [Chapter 3, Implementing a Deep Learning Model Using Keras](#). Let's build a simple neural network, and then we will start building the model. In this section, we will add the layers to be used in our deep learning model:

1. We will first import the important libraries from Keras:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

2. Let's design the CNN with the following code. We add our first layer as the convolution layer with a filter of 32, kernel_size set to (3, 3), and ReLU as our activation function. Then, we add a convolution layer with a filter value of 64 with ReLU as our activation function. Then, we added a maxpooling layer. Finally, we drop out and flatten a dense layer with a filter size of 128 and our ReLU activation function. Finally, we add one more dropout layer:

```
model = tf.keras.Sequential()

model.add(tf.keras.layers.Conv2D(10, kernel_size=(3,3), activation
= 'relu', input_shape = input_shape))
model.add(tf.keras.layers.Conv2D(64, (3,3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2,2)))
model.add(tf.keras.layers.Dropout(0.25))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dropout(0.5))

model.add(tf.keras.layers.Dense(num_classes, activation='softmax'))
```

Note: We have used conv2D and MaxPooling2D; here, we are using conv2D because we are developing a model for spatial data. You can learn more about the application of conv1D, conv2D, conv3d, MaxPooling1D, MaxPooling2D, and MaxPooling3D at <https://keras.io/layers/convolutional/> and <https://keras.io/layers/pooling/>.



In the next section, we will compile the model.

Compiling the model

For the completion of the model, we need to choose a loss in the optimizer and the metrics that we're concerned with during fitting:

```
model.compile(loss = "categorical_crossentropy", optimizer= 'SGD', metrics =  
['accuracy'])  
  
print(model.summary())
```

Let's look at the output after compiling the model:

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_5 (Conv2D)	(None, 26, 26, 32)	320
conv2d_6 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_3 (MaxPooling2D)	(None, 12, 12, 64)	0
dropout_5 (Dropout)	(None, 12, 12, 64)	0
flatten_3 (Flatten)	(None, 9216)	0
dense_5 (Dense)	(None, 128)	1179776
dropout_6 (Dropout)	(None, 128)	0
dense_6 (Dense)	(None, 10)	1290
<hr/>		
Total params: 1,199,882		
Trainable params: 1,199,882		
Non-trainable params: 0		
<hr/>		
None		

Fig 6.20: Compiling the model

In the next section, we will train the model.

Training the model

We are taking a batch size of 32 and 6 epochs. We can play with these parameters to increase the accuracy.

Write the following code to train the model:

```
batch_size = 32
epochs = 6

history = model.fit(x_train,
                     y_train,
                     batch_size= batch_size,
                     epochs = epochs,
                     verbose=1,
                     validation_data= (x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)
print('Test_loss:', score[0])
print('Test_accuracy:', score[1])
```

Here are the training results:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/6
60000/60000 [=====] - 160s 3ms/step - loss: 0.5886 - acc: 0.8166 - val_loss: 0.2143 - val_acc: 0.93
51
Epoch 2/6
60000/60000 [=====] - 162s 3ms/step - loss: 0.3181 - acc: 0.9033 - val_loss: 0.1571 - val_acc: 0.95
43
Epoch 3/6
60000/60000 [=====] - 160s 3ms/step - loss: 0.2590 - acc: 0.9224 - val_loss: 0.1287 - val_acc: 0.96
09
Epoch 4/6
60000/60000 [=====] - 163s 3ms/step - loss: 0.2133 - acc: 0.9369 - val_loss: 0.1052 - val_acc: 0.96
95
Epoch 5/6
60000/60000 [=====] - 161s 3ms/step - loss: 0.1745 - acc: 0.9483 - val_loss: 0.0854 - val_acc: 0.97
46
Epoch 6/6
60000/60000 [=====] - 160s 3ms/step - loss: 0.1487 - acc: 0.9563 - val_loss: 0.0694 - val_acc: 0.97
84
```

Fig 6.21: Model training

The training accuracy for the model is 95.84 and the test loss is 0.069.

Validation versus train loss

We will compare validation to the training loss by plotting a graph. We are going to use `matplotlib` for this:

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values)+ 1)
line1 = plt.plot(epochs, val_loss_values, label = 'Validation/Test Loss')
line2 = plt.plot(epochs, loss_values, label= 'Training Loss')
plt.setp(line1, linewidth=2.0, marker = '+', markersize=10.0)
plt.setp(line2, linewidth=2.0, marker= '4', markersize=10.0)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
plt.show()
```

Let's look at the output:

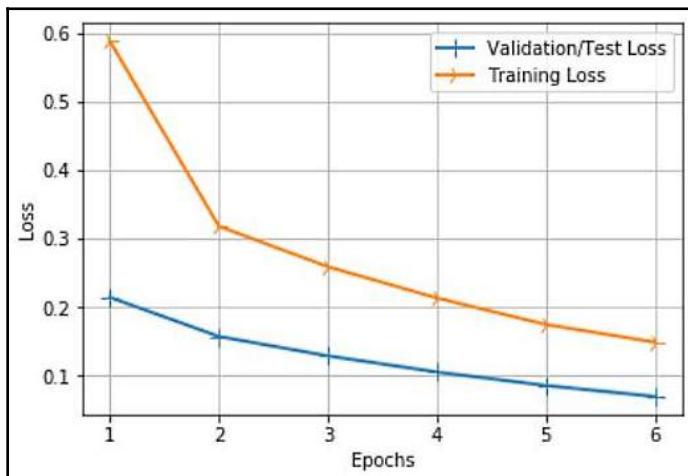


Fig 6.22: Validation versus training loss plot

The training loss started from 0.6 and ended at 0.16, and the validation loss started from 0.22 and ended at 0.06. Our model performed well, as the loss has decreased to a minimum.

Validation versus test accuracy

In this section, we will plot the graph for validation and test accuracy.

In this step, we will compare validation and test accuracy:

```
import matplotlib.pyplot as plt

history_dict = history.history
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']
epochs = range(1, len(loss_values) + 1)
line1 = plt.plot(epochs, val_acc_values, label = 'Validation/Test Accuracy')
line2 = plt.plot(epochs, acc_values, label= 'Training Accuracy')
plt.setp(line1, linewidth=2.0, marker = '+', markersize=10.0)
plt.setp(line2, linewidth=2.0, marker= '4', markersize=10.0)
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.grid(True)
plt.legend()
plt.show()
```

Here's our output:

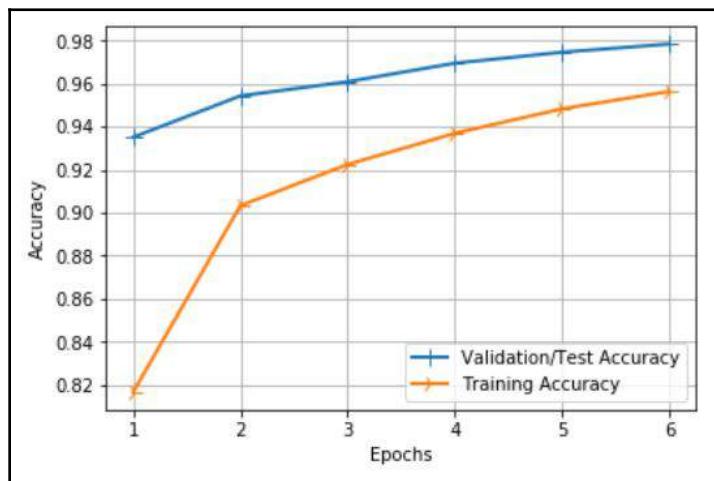


Fig 6.23: Validation versus train accuracy plot

We can see that our training accuracy is around 96% and the test accuracy is around 97.84%; this shows that our model performed well.

Saving the model

We need to save our model so that it can be reused later.

Here is the code for saving your model:

```
model.save("./mnist.h5")
```

`model.save` will save the model and `load_model` is used to reload the model:

```
from keras.models import load_model
model = load_model('./mnist.h5')
```

In the next section, we will visualize the model architecture.

Visualizing the model architecture

Keras has a great functionality, and in this section, we will use it to visualize the model architecture.

The following code will help you create a visualization of your image:

```
from keras.utils.vis_utils import plot_model
%matplotlib inline

from keras.utils import plot_model
plot_model(model, to_file='model.png',
           show_shapes= True,
           show_layer_names = True)

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

img = mpimg.imread('model.png')
plt.imshow(img)
plt.show()
```

The visualized model architecture looks as follows:

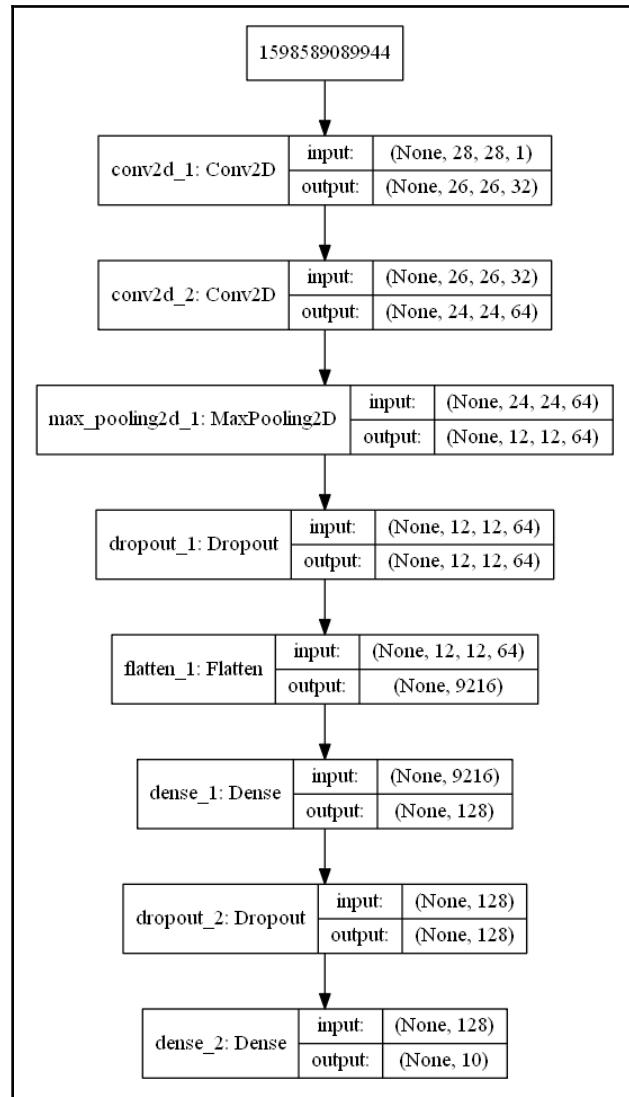


Fig 6.24: The model architecture

Now, we will validate the performance using a confusion matrix.

Confusion matrix

We use a confusion matrix to validate **the performance of a classification model** on the test data for which the true values are known.

To view the confusion matrix of the model, execute the following code:

```
y_pred=model.predict(x_test)
y_pred=np.argmax(y_pred, axis=1)
y_test=np.argmax(y_test, axis=1)
from sklearn.metrics import confusion_matrix
confusion_matrix = confusion_matrix(y_test, y_pred)
confusion_matrix
```

The confusion matrix of the model looks as follows:

```
array([[ 973,      0,      0,      0,      0,      0,      2,      1,      4,      0],
       [  0, 1120,      3,      3,      0,      0,      4,      1,      4,      0],
       [  4,      0, 1005,      4,      2,      0,      2,      7,      8,      0],
       [  0,      0,      3,  990,      0,      3,      0,      7,      7,      0],
       [  1,      0,      3,      0,  959,      0,      5,      0,      2,     12],
       [  2,      0,      0,      6,      0,  871,      5,      1,      3,      4],
       [  8,      3,      0,      0,      2,      2,  940,      0,      3,      0],
       [  2,      2,     16,      2,      1,      0,      0,  998,      1,      6],
       [  5,      1,      2,      4,      1,      1,      3,      2,  952,      3],
       [  5,      4,      0,      4,      6,      1,      0,      5,      8,  976]],  
dtype=int64)
```

Fig 6.25: The confusion matrix

You can create the confusion matrix in a more advanced way with the following code:

```
# Confusion matrix
import itertools
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(cm, classes,
                        normalize=False,
                        title='Confusion matrix',
                        cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=75)
    plt.yticks(tick_marks, classes)
```

```
if normalize:
    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, cm[i, j],
             horizontalalignment="center",
             color="white" if cm[i, j] > thresh else "black")

plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')

class_names = range(10)
cm = confusion_matrix(y_pred, y_test)

plt.figure(2)
plot_confusion_matrix(cm, classes=class_names, title='Confusion matrix')
```

Here's our output:

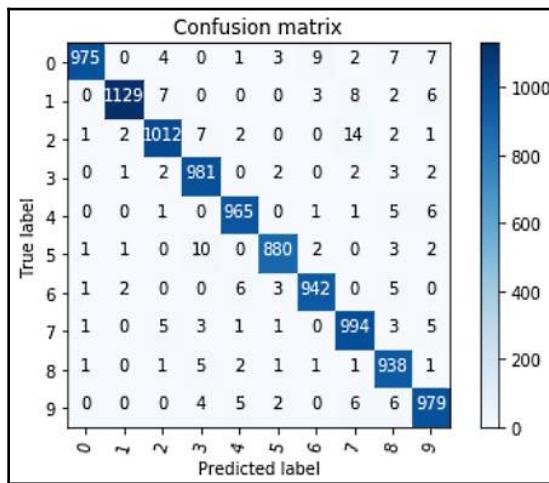


Fig 6.26: Confusion matrix II

Now, we should check the accuracy report of the model.

The accuracy report

In this step, we will check the accuracy report of the model. We will get the following values.

Accuracy: The accuracy is the most important and popular metric for model validation. The ratio of the correctly predicted observation to the total observation is called the accuracy. In general, a high accuracy model is not always preferable, as the accuracy metric only works well with symmetric datasets where values of false positives and false negatives are almost the same.

Now, we will have a look at the formula of accuracy:

$$\text{Accuracy} = (TP + TN) / (TP + FP + FN + TN)$$

Here, we have the following:

- **TP** is true positive
- **TN** is true negative
- **FP** is false positive
- **TP** is true positive

Precision: The ratio of correctly predicted positive observations (**TP**) to the total predicted positive observations (**TP + FP**) is called precision. This is the formula for precision:

$$\text{Precision} = TP / (TP + FP)$$

Recall: The ratio of correctly predicted positive observations (**TP**) to all the observations in an actual class (**TP + FN**) is called recall, or sensitivity:

$$\text{Recall} = TP / (TP + FN)$$

F1 score: The weighted average of precision and recall is called the **F1 score**. It is very difficult to understand **accuracy** intuitively. In general, the F1 score is usually more useful than accuracy, especially if you have an uneven class distribution.

The formula for the F1 score is as follows:

$$\text{F1 Score} = 2 * (\text{Recall} * \text{Precision}) / (\text{Recall} + \text{Precision})$$

Now, we will find the accuracy report for our model:

```
from sklearn.metrics import classification_report
predictions = model.predict_classes(x_test)
print(classification_report(y_test, predictions))
```

Here's the resulting report:

	precision	recall	f1-score	support
0	0.97	0.99	0.98	980
1	0.99	0.99	0.99	1135
2	0.97	0.97	0.97	1032
3	0.98	0.98	0.98	1010
4	0.99	0.98	0.98	982
5	0.99	0.98	0.98	892
6	0.98	0.98	0.98	958
7	0.98	0.97	0.97	1028
8	0.96	0.98	0.97	974
9	0.98	0.97	0.97	1009
micro avg	0.98	0.98	0.98	10000
macro avg	0.98	0.98	0.98	10000
weighted avg	0.98	0.98	0.98	10000

Fig 6.27: Accuracy report

There we have it—we have implemented a CNN for the MNIST dataset.

Summary

In this chapter, we learned about CNNs and the different ways of tweaking them, and also implemented a handwritten digit recognition model using Keras. We also learned about the hyperparameters for image-based problems and different accuracy metrics, such as accuracy, F1 score, precision, and recall.

In the next chapter, we will implement an image classifier for traffic sign detection. With this project, we will be one step closer to seeing the real-time application of autonomous vehicles.

7

Road Sign Detection Using Deep Learning

Traffic road signs are an important part of our road infrastructure. Without road signs, the percentage of road accidents would increase; drivers wouldn't know, for example, how fast they should be going or whether there are roadworks, sharp turns, or school crossings ahead. In this chapter, we are going to solve an important problem in **self-driving car** (SDC) technology – traffic sign detection. We will implement a model that will achieve 95% accuracy.



We are going to use the German traffic signs dataset, which is free to use. I have taken inspiration from the following publication: J. Stallkamp, M. Schlippsing, J. Salmen, and C. Igel. *The German Traffic Sign Recognition Benchmark: A multi-class classification competition* (<https://ieeexplore.ieee.org/document/6033395>). In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 1453–1460. 2011.

In this chapter, we will cover the following topics:

- Dataset overview
- Loading the data
- Image exploration
- Data preparation
- Model training
- Model accuracy

Let's get started!

Dataset overview

Traffic sign detection using the German traffic sign dataset is a multi-class classification problem that contains more than 40 classes and more than 50,000 images. The physical traffic sign instances are unique within the dataset, and each real-world traffic sign only occurs once.



You can follow this link to find out more about the German traffic sign detection dataset: J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. *The German Traffic Sign Recognition Benchmark: A multi-class classification competition*. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pages 1453–1460. 2011 (<http://benchmark.ini.rub.de/>).

Dataset structure

The training set archive is structured as follows:

- One directory per class.
- Each directory contains one **Comma Separated Value (CSV)** file with annotations (GT-<ClassID>.csv), as well as the training images.
- Training images are grouped by tracks.
- Each track contains 30 images of one single physical traffic sign.

Image format

The image format is structured as follows:

- The images contain one traffic sign each.
- Images contain a border of 10 % around the actual traffic sign (at least 5 pixels) to allow for edge-based approaches.
- Images are stored in PPM format – Portable, Pixmap, and P6 (http://en.wikipedia.org/wiki/Netpbm_format).
- Image sizes vary between 15 x 15 and 250 x 250 pixels.
- Images are not necessarily squared.
- The actual traffic sign is not necessarily centered within the image. This is true for images that were close to the image border in the full camera image.
- The bounding box of the traffic sign is a part of the annotations, which we will see in the following section.

The following are examples of a few classes:

- (0, b'Speed limit (20 km/h)') (1, b'Speed limit (30 km/h)')
- (2, b'Speed limit (50 km/h)') (3, b'Speed limit (60 km/h)')
- (4, b'Speed limit (70 km/h)') (5, b'Speed limit (80 km/h)')
- (6, b'End of speed limit (80 km/h)') (7, b'Speed limit (100 km/h)')
- (8, b'Speed limit (120 km/h)') (9, b'No passing')
- (10, b'No passing for vehicles over 3.5 metric tons')
- (11, b'Right-of-way at the next intersection') (12, b'Priority road')
- (13, b'Yield') (14, b'Stop') (15, b'No vehicles')
- (16, b'Vehicles over 3.5 metric tons prohibited') (17, b'No entry')
- (18, b'General caution') (19, b'Dangerous curve to the left')
- (20, b'Dangerous curve to the right') (21, b'Double curve')
- (22, b'Bumpy road') (23, b'Slippery road')
- (24, b'Road narrows on the right') (25, b'Road work')
- (26, b'Traffic signals') (27, b'Pedestrians') (28, b'Children crossing')
- (29, b'Bicycles crossing') (30, b'Beware of ice/snow')
- (31, b'Wild animals crossing')
- (32, b'End of all speed and passing limits') (33, b'Turn right ahead')
- (34, b'Turn left ahead') (35, b'Ahead only') (36, b'Go straight or right')
- (37, b'Go straight or left') (38, b'Keep right') (39, b'Keep left')
- (40, b'Roundabout mandatory') (41, b'End of no passing')
- (42, b'End of no passing by vehicles over 3.5 metric tons')

In the next section, we will load the data.

Loading the data

In this section, we will start building and training our convolutional neural network:

1. We will start by importing the necessary libraries, including pandas, numpy, and matplotlib:

```
import warnings
warnings.filterwarnings("ignore")
import libraries
import pickle
#Import Pandas for data manipulation using dataframes
```

```
import pandas as pd
#Importing Numpy for data statistical analysis
import numpy as np
#Importing matplotlib for data visualisation
import matplotlib.pyplot as plt
import random
```

2. Next, we import three pickle files – the test, training, and validation datasets:

```
with open("./traffic-signs-data/train.p", mode='rb') as
    training_data:
        train = pickle.load(training_data)
with open("./traffic-signs-data/valid.p", mode='rb') as
    validation_data:
        valid = pickle.load(validation_data)
with open("./traffic-signs-data/test.p", mode='rb') as
    testing_data:
        test = pickle.load(testing_data)
X_train, y_train = train['features'], train['labels']
X_validation, y_validation = valid['features'], valid['labels']
X_test, y_test = test['features'], test['labels']
```

3. Now, let's check the shape of the data:

```
X_train.shape
y_train.shape
```

4. The shape of X_train is as follows:

```
(34799, 32, 32, 3)
```

5. The shape of y_train is as follows:

```
(34799,)
```

In the next section, we will explore the dataset.

Image exploration

In this section, we will explore the `images` class and see what the German traffic sign dataset looks like.

Let's look at the image and check whether it has been imported correctly:

```
i = 1001
plt.imshow(X_train[i]) # Show images are not shuffled
y_train[i]
```

The output looks like a signboard, so we can see that the image has been imported correctly:

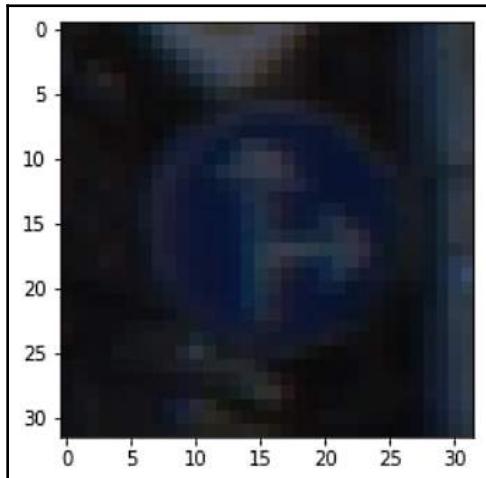


Fig 7.1: Output of the imported image

In the following image, you can see all the classes of the German traffic sign detection dataset, which we discussed in the *Image format* section:



Fig 7.2: 43 image classes

In the next section, we will start the first step of the modeling process – data preparation.

Data preparation

Data preparation is an important part of any data science project. In this section, we will prepare the data for analysis. Data preparation helps us achieve better accuracy:

1. We will start by shuffling the dataset:

```
from sklearn.utils import shuffle
X_train, y_train = shuffle(X_train, y_train)
```

2. Now, we will transform the data into grayscale and normalize it:

```
X_train_gray = np.sum(X_train/3, axis=3, keepdims=True)
X_test_gray = np.sum(X_test/3, axis=3, keepdims=True)
X_validation_gray = np.sum(X_validation/3, axis=3, keepdims=True)

X_train_gray_norm = (X_train_gray - 128)/128
X_test_gray_norm = (X_test_gray - 128)/128
X_validation_gray_norm = (X_validation_gray - 128)/128
```

3. Next, we will check the images following the grayscale conversion:

```
i = 610  
plt.imshow(X_train_gray[i].squeeze(), cmap='gray')  
plt.figure()  
plt.imshow(X_train[i])
```

The output image should look as follows:

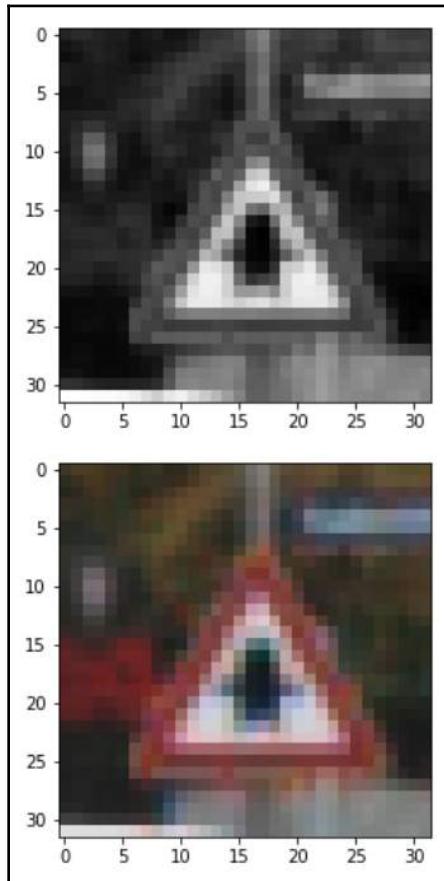


Fig 7.3: Grayscale image

In the next section, we will start the model training process.

Model training

In this section, we will train our model with the same convolution neural network architecture that we used in *Chapter 6, Improving the Image Classifier with CNN*. Let's get started:

1. We will start by importing the `keras` and `sklearn` libraries. We are importing `Sequential`, `Conv2D`, `MaxPooling2D`, `Dense`, `Flatten`, `Dropouts`, `Adam`, `TensorBoard`, and `check_output` from Keras. We have imported `train_test_split` from `sklearn`:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from subprocess import check_output
from sklearn.model_selection import train_test_split
```

2. Next, we will build the following model:

```
cnn_model = tf.keras.Sequential()
cnn_model.add(tf.keras.layers.Conv2D(32, 3, 3, input_shape =
image_shape, activation='relu'))
cnn_model.add(tf.keras.layers.Conv2D(64, (3,3), activation='relu'))
cnn_model.add(tf.keras.layers.MaxPooling2D(pool_size = (2, 2)))
cnn_model.add(tf.keras.layers.Dropout(0.25))
cnn_model.add(tf.keras.layers.Flatten())
cnn_model.add(tf.keras.layers.Dense(128, activation = 'relu'))
cnn_model.add(tf.keras.layers.Dropout(0.5))
cnn_model.add(tf.keras.layers.Dense(43, activation = 'sigmoid'))
```

3. Then, we will compile the model. We are using `sparse_categorical_crossentropy` because we have 43 different classes. We are using `Adam` as an optimizer as it is appropriate for the sparse gradient problems; its parameters require little tuning as they are computationally efficient and invariant to any diagonal rescaling of gradients:

```
cnn_model.compile(loss ='sparse_categorical_crossentropy',
optimizer=keras.optimizers.Adam(0.001, beta_1=0.9, beta_2=0.999,
epsilon=1e-07, amsgrad=False),metrics =['accuracy'])
```

4. Now, we will train the model using the `cnn_model.fit` function, here, batch size is 100 and epoch is 50:

```
history = cnn_model.fit(X_train_gray_norm,
y_train,
batch_size=500,
```

```
nb_epoch=50,
verbose=1,
validation_data =
(X_validation_gray_norm,y_validation))
```

- The model has started training. The following screenshot shows the resulting output:

```
Train on 34799 samples, validate on 4410 samples
Epoch 1/50
34799/34799 [=====] - 11s 321us/step - loss: 3.3032 - acc: 0.0469 - val_loss: 3.2339 - val_acc: 0.0671
Epoch 2/50
34799/34799 [=====] - 13s 381us/step - loss: 2.1492 - acc: 0.3623 - val_loss: 1.8857 - val_acc: 0.4859
Epoch 3/50
34799/34799 [=====] - 15s 419us/step - loss: 1.2375 - acc: 0.6828 - val_loss: 1.4569 - val_acc: 0.5805
Epoch 4/50
34799/34799 [=====] - 13s 363us/step - loss: 0.9197 - acc: 0.7704 - val_loss: 1.2320 - val_acc: 0.6494
Epoch 5/50
34799/34799 [=====] - 12s 356us/step - loss: 0.7306 - acc: 0.8225 - val_loss: 1.1188 - val_acc: 0.6957
```

Fig 7.4: Model training

Now that the model has been trained, we will look at the results.

Model accuracy

Once training is complete, the next step is to validate the model accuracy. Let's get started:

- Validate the accuracy of the model:

```
score = cnn_model.evaluate(X_test_gray_norm, y_test,verbose=0)
print('Test Accuracy : {:.4f}'.format(score[1]))
```

- The accuracy of the model is as follows:

```
Test Accuracy : 0.9523
```

- The training loss versus validation loss graph looks like this:

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values)+ 1)
```

```
line1 = plt.plot(epochs, val_loss_values, label = 'Validation/Test Loss')
line2 = plt.plot(epochs, loss_values, label= 'Training Loss')
plt.setp(line1, linewidth=2.0, marker = '+', markersize=10.0)
plt.setp(line2, linewidth=2.0, marker= '4', markersize=10.0)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
plt.show()
```

The following image shows the training versus test loss graph:

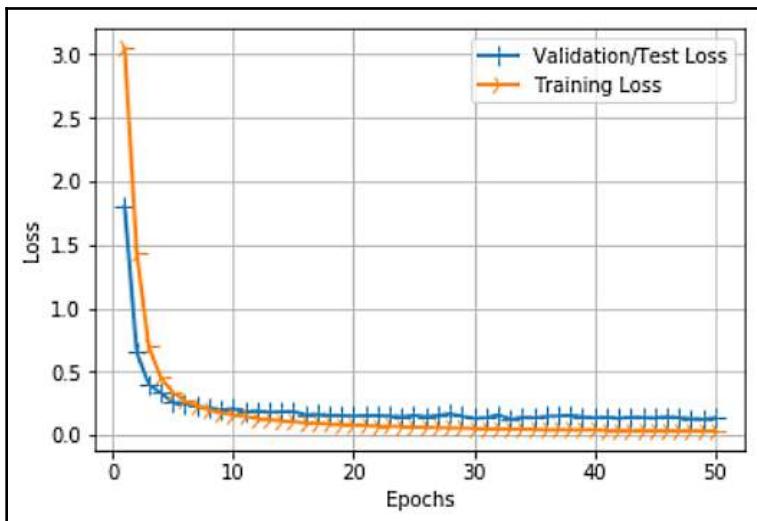


Fig 7.5: Validation versus test loss

4. Now, we will check the training and validation accuracy:

```
import matplotlib.pyplot as plt

history_dict = history.history
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']
epochs = range(1, len(loss_values)+ 1)
line1 = plt.plot(epochs, val_acc_values, label = 'Validation/Test Accuracy')
line2 = plt.plot(epochs, acc_values, label= 'Training Accuracy')
plt.setp(line1, linewidth=2.0, marker = '+', markersize=10.0)
plt.setp(line2, linewidth=2.0, marker= '4', markersize=10.0)
plt.xlabel('Epochs')
```

```
plt.ylabel('Accuracy')
plt.grid(True)
plt.legend()
plt.show()
```

The following image shows training accuracy versus the validation accuracy graph:

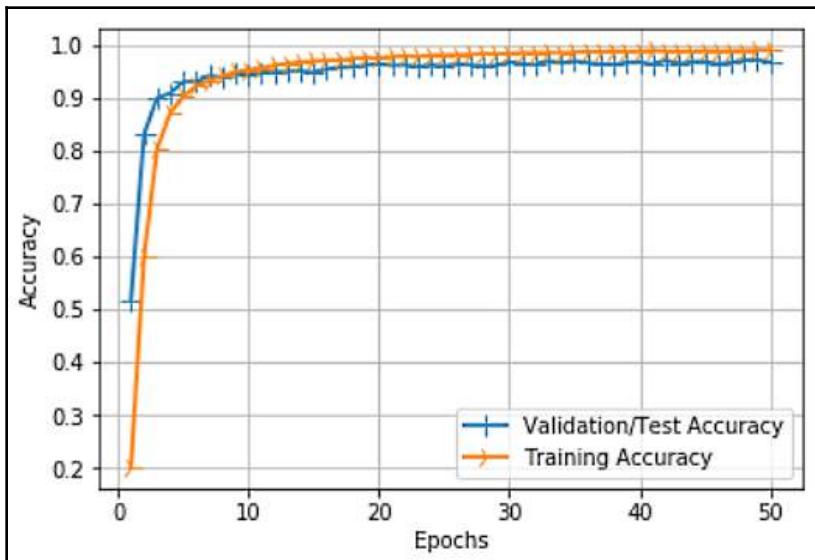


Fig 7.6: Validation versus test accuracy

By observing these two graphs, we are sure our model is performing really well. We have built a traffic sign detector classifier with 95% accuracy. The model has limited training loss, so the model is not overfitting.

5. Now, let's save the model:

```
cnn_model.save("./trafficSign.h5")
```

6. Then, we will reload the model:

```
from keras.models import load_model
model = load_model('./trafficSign.h5')
```

7. We can also check the prediction:

```
for i in range(0,12):
    plt.subplot(4,3,i+1)
    plt.imshow(X_test_gray_norm[i+10].squeeze(), cmap='gray',
```

```

    interpolation='none')
    plt.title("Predicted {}, Class
    {}".format(predicted_classes[i+10], y_true[i+10]))
    plt.tight_layout()

```

Let's check the output:

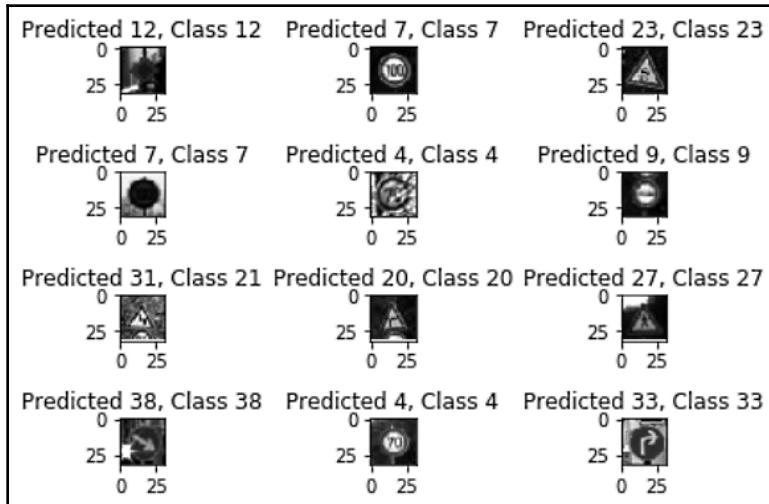


Fig 7.7: Prediction report

With this output, we can check the correctly and incorrectly classified images.

8. Now, we will check the confusion matrix:

```

# Confussion matrix
import itertools
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(cm, classes,
                        normalize=False,
                        title='Confusion matrix',
                        cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=75)
    plt.yticks(tick_marks, classes)

    if normalize:

```

```
cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]),
range(cm.shape[1])):
    plt.text(j, i, cm[i, j],
    horizontalalignment="center",
    color="white" if cm[i, j] > thresh else "black")

plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')

class_names = range(43)
cm = confusion_matrix(predictions,y_test)

plt.figure(2)
plot_confusion_matrix(cm, classes=class_names, title='Confusion matrix')
```

Here, we can see what the confusion matrix looks like. The confusion matrix is not clear as there were 43 classes, but you can follow the classification report, which can be found in the Chapter 7 Python notebook:

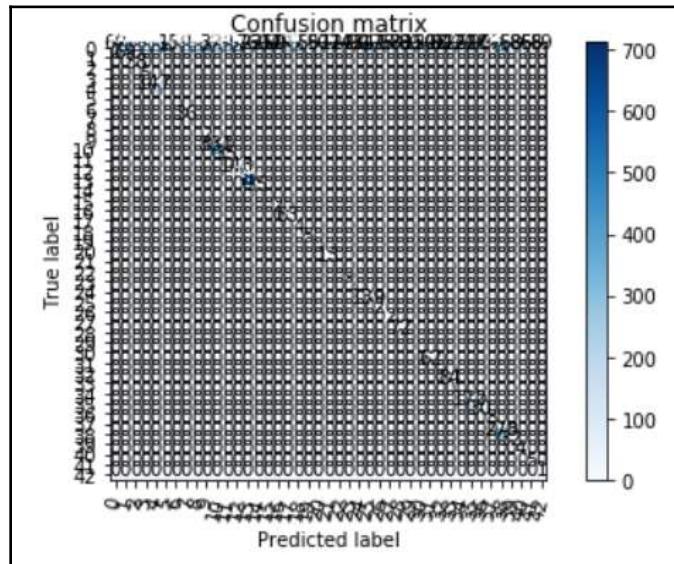


Fig 7.8: Confusion matrix

With that, we have implemented a CNN model for German traffic sign detection.

Summary

As we already know, it is really important for an autonomous driving car to follow traffic signals. In this chapter, we have implemented traffic sign detection with 95% accuracy. This project is one of the important steps toward creating autonomous driving cars. We created a model that classifies traffic signs and identifies the most appropriate features by itself.

In the next chapter, we will learn about semantic segmentation, which is the next step toward building SDCs.

3

Section 3: Semantic Segmentation for Self-Driving Cars

In this section, you will learn the basic structure and workings of semantic segmentation models and all of the latest state-of-the-art methods, including ENet, SegNet, PSPNet, and Deeplabv3. We will also implement a real-time semantic segmentation self-driving car use case using ENet.

This section comprises the following chapters:

- Chapter 8, *The Principles and Foundations of Semantic Segmentation*
- Chapter 9, *Implementation of Semantic Segmentation*

8

The Principles and Foundations of Semantic Segmentation

In this chapter, we are going to talk about how deep learning and **convolutional neural networks (CNNs)** can be adapted to solve semantic segmentation tasks in computer vision.

In a **self- driving car (SDC)**, the vehicle must know exactly where another vehicle is on the road or where a person is crossing the road. Semantic segmentation helps make these identifications. Semantic segmentation with CNNs effectively means classifying each pixel in the image. Thus, the idea is to create a map of fully detectable object areas in the image. Basically, what we want is an output image in the slide where every pixel has a label associated with it.

For example, semantic segmentation will label all the cars in an image, as shown here:

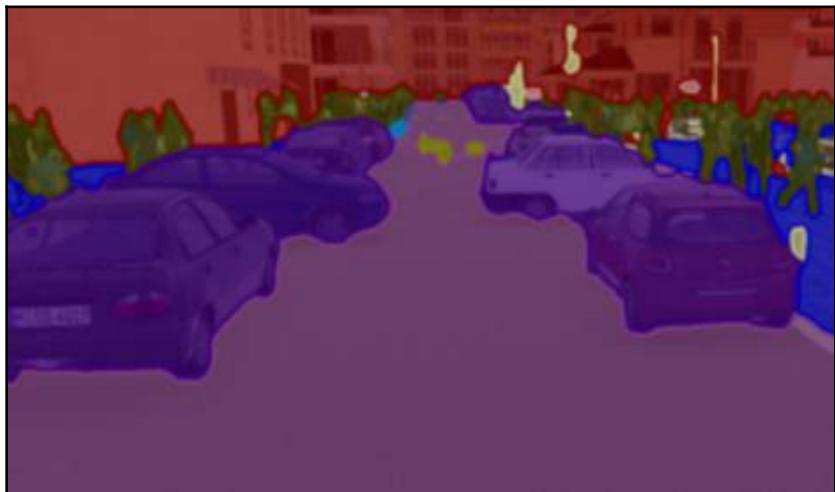


Fig 8.1: Semantic segmentation output

The demand for understanding data has increased in the field of computer vision due to the availability of data from mobile phones, surveillance systems, and automotive vehicles. The advancement of computational power in recent years has enabled deep learning to take strides toward better visual understanding. Deep neural networks have achieved performance equal to humans on tasks such as image classification and traffic sign recognition, similar to what we implemented in Chapter 6, *Improving the Image Classifier with CNNs*. However, deep learning has a high computational cost as an increase in performance is accomplished by increasing the network's size. Large neural networks are difficult to use with SDCs.

As we already know, the first step in autonomous driving systems is based on perception or visual inputs, namely object recognition, object localization, and semantic segmentation. Semantic segmentation labels each pixel in an image belonging to a given semantic class. Typically, these classes could be streets, traffic signs, street markings, cars, pedestrians, or sidewalks. When we deploy deep learning for semantic segmentation, the recognition of major objects in the images, such as people or vehicles, is done at the higher levels of a neural network. The biggest benefit of this strategy is that slight variance at the pixel level doesn't impact identification. Semantic segmentation requires a pixel-exact classification of small features that usually only occur in lower layers.

In this chapter, we will cover the following topics:

- Introduction to semantic segmentation
- Understanding the semantic segmentation architecture
- Overview of different semantic segmentation architectures
- Deep learning for semantic segmentation

Let's get started!

Introduction to semantic segmentation

Numerous technology systems have emerged in recent years that have been designed to identify a car's surroundings. Understanding the scene around our surroundings turns out to be an important area of research for analyzing the geometry of scenes and the associated objects in the surroundings. CNNs have proved to be the most effective vision computing tool in image classification, object detection, and semantic segmentation. In an automated environment, it is important to make some critical decisions in order to understand a given scene in the surroundings at the pixel level. Semantic segmentation has proven to be one of the most effective methods of assigning labels to individual pixels in an image.

Researchers have proposed numerous ways for semantic pixel-wise labeling; some approaches have tried deep architecture pixel-wise labeling, and the results have been impressive. Since segmentation at the pixel level provides better performance, researchers started using those methods for real-time automated systems. Lately, driving assistant systems have become top research areas as they provide various opportunities for boosting driving experiences. Driver performance could be improved by using semantic segmentation techniques available in the **Advanced Driving Assistance System (ADAS)**.

Semantic segmentation is the process that associates each pixel of an image with a class label, where the classes can be a person, street, road, the sky, the ocean, or a car.

A semantic segmentation algorithm consists of the following steps:

1. It **creates a partition of the image** and puts it into meaningful categories.
2. It **associates every pixel** in an input image with a class label such as a person, tree, street, road, car, bus, and so on.

In the next section, we will understand the semantic segmentation architecture.

Understanding the semantic segmentation architecture

The semantic segmentation network generally consists of an encoder-decoder network. The encoder produces high-level features using convolution, while the decoder helps in interpreting these high-level features using classes. The encoder is a common encoding mechanism that is used by pre-trained networks and the decoder weight that's learned while training a segmentation network. The following diagram shows the architecture of the encoder-decoder-based FCN architecture for semantic segmentation:

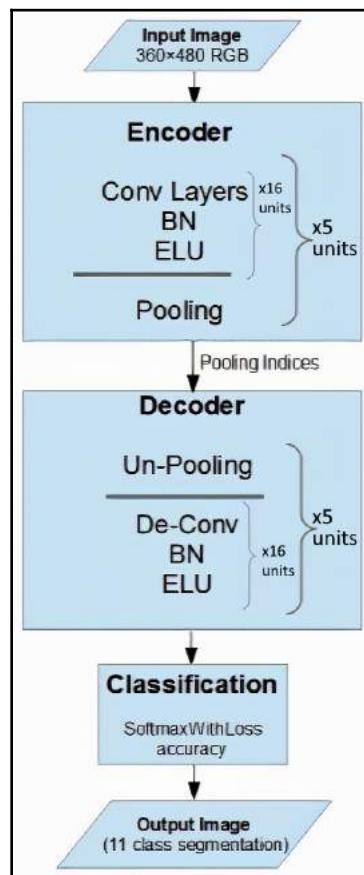


Fig 8.2: Semantic segmentation architecture

You can check out the preceding diagram at the following link: <https://www.mdpi.com/2313-433X/4/10/116/pdf>.



The encoder gradually reduces the spatial dimension with the help of pooling layers, while the decoder recovers the features of the object and spatial dimensions. You can read more about semantic segmentation in the paper on ECRU: *An Encoder-Decoder-Based Convolution Neural Network (CNN) for Road-Scene Understanding*.

One of the important concepts to understand is how semantic segmentation works in convolutional networks. The concept behind semantic segmentation is finding meaningful parts of an image. We can see how pixels belonging to one class occur in correlation with the pixels of another class. Let's consider the first layer of encoding on CNNs. The basic operation of a convolution is to encode the image into a higher-level representation, with the idea of expressing the image as a combination of parts, such as edges or gradients. The encoded features, such as edges, for example, are not exclusive but actually carry the context of a neighborhood as well. When we upsample and decode, these features are decoded with the associated class, guided by per-pixel maps during the backpropagation of the network.

Hence, the main goal of semantic segmentation is to represent an image in a way that is easy to analyze. More precisely, we can say that image segmentation is the process of labeling every pixel in an image so that pixels with the same label have similar characteristics.

In the next section, we will read about popular semantic segmentation architectures.

Overview of different semantic segmentation architectures

There are lots of deep learning architectures and pre-trained models for semantic segmentation that have been released in recent times. In this section, we will discuss the popular semantic segmentation architectures, which are as follows:

- U-Net
- SegNet
- PSPNet
- DeepLabv3+
- E-Net

We will start by introducing U-Net.

U-Net

U-Net won the award for the most challenging **Grand Challenge for the Computer-Automated Detection of Caries in Bitewing Radiography** at the **International Symposium on Biomedical Imaging (ISBI) 2015** and also won the **Cell Tracking Challenge** at **ISBI** in 2015.

U-Net is the fastest and most precise semantic segmentation architecture. It outperformed methods such as the sliding window CNN at the ISBI challenge for semantic segmentation of neuron structures in electron microscopic stacks.

At ISBI 2015, it also won the two most challenging **transmitted light microscopy** categories, **Phase contrast** and **DIC microscopy**, by a large margin.

The main idea behind U-Net is to add successive layers to a normal contracting network, where upsampling operators replace pooling operations. Due to this, the layers of U-Net increase the resolution of the output. The most important modification in U-Net occurs in the upsampling component, which contains a large number of feature channels that enable the network to propagate contextual information to higher-resolution layers.

The network is composed of a contracting path and an expansive path, which gives it the U-shaped architecture. The contracting path is a standard convolutional neural network comprised of repeated convolution, followed by a **rectified linear unit (ReLU)** and a max-pooling operation. Spatial information is reduced while feature information is increased during the contraction.

The expansive pathway is used to combine the spatial information and image features through a sequence of concatenations with high-resolution features from the contracting path and up-convolutions.

The architecture of U-Net can be seen in the following diagram:

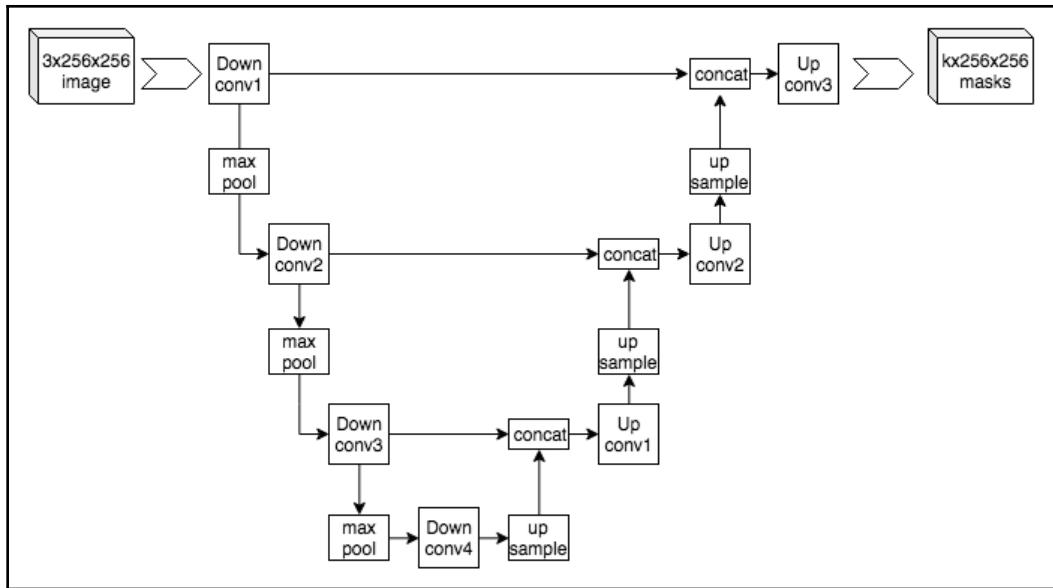


Fig 8.3: U-Net architecture producing a 256 * 256 mask using a 256 * 256 image



U-Net: Convolutional Networks for Biomedical Image Segmentation is a paper by Olaf Ronneberger, Philipp Fischer, and Thomas Brox. Click on the following link for more details: <https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/>.

In the next section, we will cover SegNet.

SegNet

SegNet is a deep encoder-decoder architecture for multi-class pixel-wise segmentation that was researched and developed by members of the Computer Vision and Robotics Group (<http://mi.eng.cam.ac.uk/Main/CVR>) at the University of Cambridge, UK.

The SegNet architecture consists of an encoder network, a corresponding decoder network, and a final classification pixel-wise layer. It also consists of a series of non-linear processing layers (encoders) and a corresponding collection of decoders, accompanied by a pixel-wise classifier.

The architecture of SegNet can be seen in the following diagram:

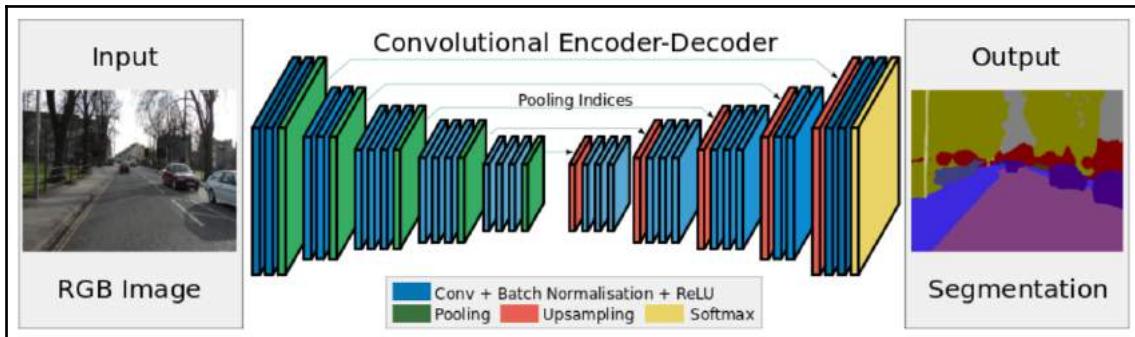


Fig 8.4: SegNet architecture

You can also check out this diagram at <https://mi.eng.cam.ac.uk/projects/segnets/>.

The encoder typically consists of one or more convolutional layers with batch normalization and a ReLU, accompanied by non-overlapping max-pooling and sub-sampling. Sparse encoding, which results from the pooling process, is upsampled in the decoder using the encoding sequence's max-pooling indices. SegNet uses max-pooling indices in the decoders to upsample the feature maps with low resolution. This has the significant advantage of keeping high-frequency details in the segmented images, as well as reducing the total number of trainable parameters in the decoders. SegNet uses stochastic gradient descent to train the network.

In the next section, we will provide an overview of the encoder and decoder parts of the SegNet architecture.

Encoder

Convolutions and max-pooling are performed in the encoder, where 13 convolutional layers are taken from VGG-16. The corresponding max-pooling indices are stored while performing 2×2 max-pooling.

Decoder

Upsampling and convolutions are conducted in the decoder's softmax classifier, at the end of each pixel. The max-pooling indices at the corresponding encoder layer are recalled and upsampled during the upsampling process. Then, a K-class softmax classifier is used for predicting each pixel.



A Deep Convolutional Encoder-Decoder Architecture for Robust Semantic Pixel-Wise Labeling was researched and developed by members of the Computer Vision and Robotics Group at the University of Cambridge, UK. Click on the following link for more details: <http://mi.eng.cam.ac.uk/projects/segnets/>.

In the next section, we'll cover the **Pyramid Scene Parsing Network (PSPNet)**.

PSPNet

PSPNet-Full-Resolution Residual Networks were really computationally intensive and using them on full-scale images was really slow. In order to deal with this problem, PSPNet came into the picture. It applies four different max-pooling operations with four different window sizes and strides. Using the max-pooling layers allows us to extract feature information from different scales with more efficiency.



PSPNet achieved state-of-the-art performance on various datasets. It became popular after the ImageNet scene parsing challenge in 2016. It hit the **PASCAL VOC 2012** benchmark and the Cityscapes benchmark with a **mIoU** record of 85.4% accuracy on **PASCAL VOC 2012**, and also achieved 80.2% on Cityscapes. The following is a link to the relevant paper: <https://arxiv.org/pdf/1612.01105.pdf>.

The following diagram shows the architecture of PSPNet:

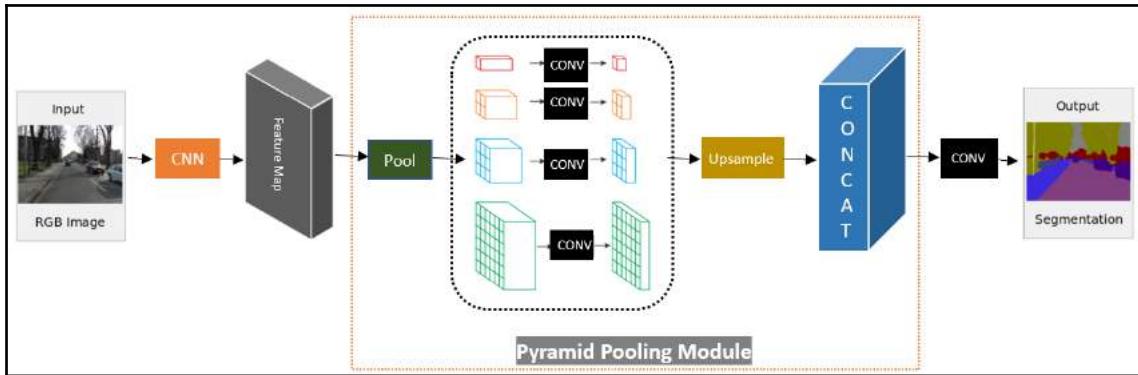


Fig 8.5: PSPNet architecture

Check out <https://hszhao.github.io/projects/pspnet/> to find out more about the PSPNet architecture and its implementation.

In the preceding diagram, we can see the proposed architecture for PSPNet. For the given input image, a feature map is extracted using the convolutional neural network. Then, the pyramid parsing module is used to harvest different representations of the sub-regions. This is followed by upsampling and concatenation layers to form the final representation of features, which contain both local and global context information. Finally, the output from the previous layer is fed into a convolution layer to get the final prediction per pixel.

In the next section, we will look at DeepLabv3+.

DeepLabv3+

DeepLab is the semantic segmentation state-of-the-art model. In 2016, it was developed and open sourced by Google. Multiple versions have been released and many improvements have been made to the model since then. These include DeepLab V2, DeepLab V3, and DeepLab V3+.

Before the release of DeepLab V3+, we were able to encode multi-scale contextual information using filters and pooling operations at different rates; the newer networks could capture the objects with sharper boundaries by recovering spatial information. DeepLabv3+ combines these two approaches. It uses both the encoder-decoder and the spatial pyramid pooling modules.

The following diagram shows the architecture of DeepLabV3+, which consists of encoder and decoder modules:

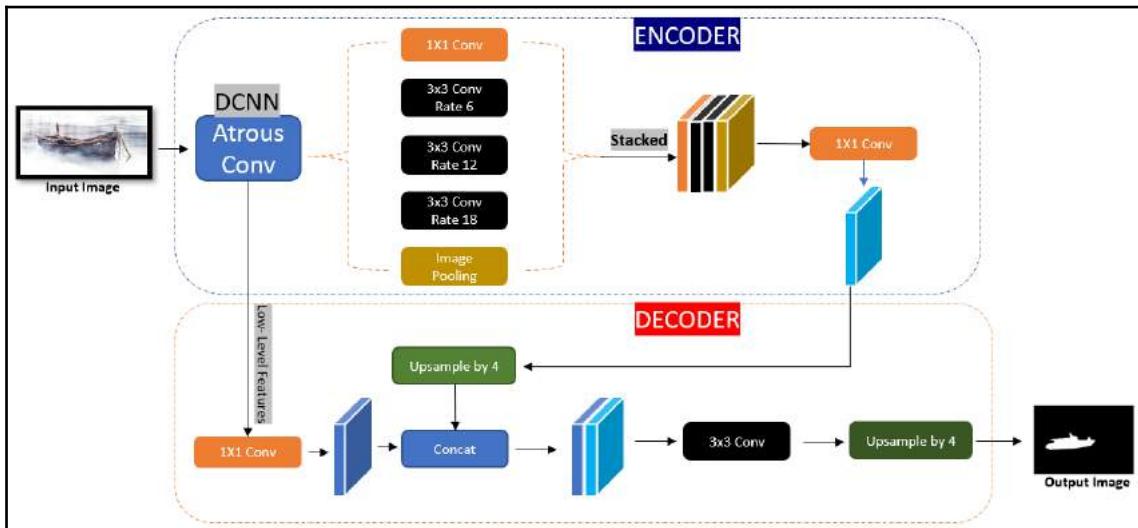


Fig 8.6: DeepLabV3+ architecture

Let's look at the encoder and decoder modules in more detail:

- **Encoder:** In the encoder step, essential information from the input image is extracted using a pre-trained convolutional neural network. The essential information for segmentation tasks is the objects present in the image and their locations.
- **Decoder:** The information that's extracted from the encoder process is used to create an output that is the same as the original input image's size.



If you want to learn more about DeepLabV3+ you can read the *Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation* paper at: <https://arxiv.org/pdf/1802.02611.pdf>.

In the next section, we will look at the E-Net architecture.

E-Net

Real-time pixel-wise semantic segmentation is one of the great applications of semantic segmentation for SDCs. Accuracy can increase in SDCs, but deploying semantic segmentation is still a challenge. In this section, we'll look at an **efficient neural network (E-Net)** that aims to run on low-power mobile devices while improving accuracy.

E-Net is a popular network due to its ability to perform real-time pixel-wise semantic segmentation. E-Net is up to 18x faster, requires 75x fewer FLOPs, and has 79x fewer parameters than existing models such as U-Net and SegNet, leading to much better accuracy. E-Net networks are tested on the popular CamVid, Cityscapes, and SUN datasets.

The architecture of E-Net is as follows:

Name	Type	Output size
initial		$16 \times 256 \times 256$
bottleneck1.0	downsampling	$64 \times 128 \times 128$
4× bottleneck1.x		$64 \times 128 \times 128$
bottleneck2.0	downsampling	$128 \times 64 \times 64$
bottleneck2.1		$128 \times 64 \times 64$
bottleneck2.2	dilated 2	$128 \times 64 \times 64$
bottleneck2.3	asymmetric 5	$128 \times 64 \times 64$
bottleneck2.4	dilated 4	$128 \times 64 \times 64$
bottleneck2.5		$128 \times 64 \times 64$
bottleneck2.6	dilated 8	$128 \times 64 \times 64$
bottleneck2.7	asymmetric 5	$128 \times 64 \times 64$
bottleneck2.8	dilated 16	$128 \times 64 \times 64$
<i>Repeat section 2, without bottleneck2.0</i>		
bottleneck4.0	upsampling	$64 \times 128 \times 128$
bottleneck4.1		$64 \times 128 \times 128$
bottleneck4.2		$64 \times 128 \times 128$
bottleneck5.0	upsampling	$16 \times 256 \times 256$
bottleneck5.1		$16 \times 256 \times 256$
fullconv		$C \times 512 \times 512$

Fig 8.7: E-Net architecture

You can check out the preceding screenshot at <https://arxiv.org/pdf/1606.02147.pdf>.

This is a framework with one master and several branches that split from the master but also merge back via element-wise addition. The model architecture consists of an initial block and five bottlenecks. The first three bottlenecks are used to encode the input image, while the other two are used to decode it. Let's learn more about the initial block and the bottleneck block.

Initial block: Let's say the resolution of the input image is 512x512. The following diagram shows that this results in an output size of 16x256x256 after concatenating the convolution of 13 filters and performing max-pooling of 2x2 without an overlap:

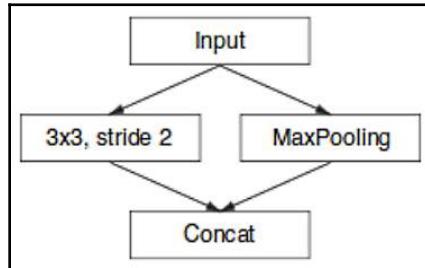


Fig 8.8: Initial E-Net architecture block

You can find the preceding diagram at <https://arxiv.org/abs/1606.02147>.

Bottleneck block: As shown in the following diagram, each branch consists of three convolutional layers. The 1x1 projection initially reduces the dimensionality, and then later expands it. In between these convolutions, a regular asymmetric dilated or full convolution with no annotation also takes place. We can also see that batch normalization and PReLU are present between all convolutions. Also, spatial dropout is used:

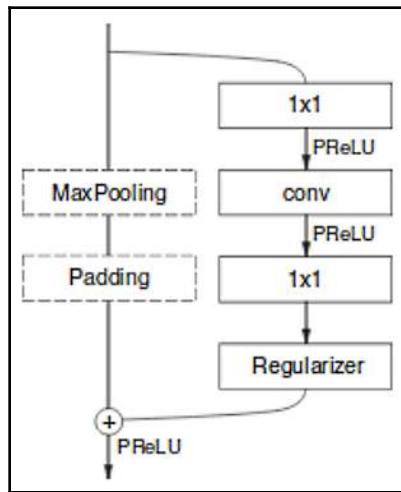


Fig 8.9: Bottleneck block of E-Net

You can check out the preceding diagram at <https://arxiv.org/abs/1606.02147>.

We should note that max-pooling on the master will only be applied when the bottleneck is downsampled. Simultaneously, a non-overlapping 2x2 convolution replaces the first projection in the branch, and the activations get zero-padded so that they equal the number of feature maps. Max-unpooling replaces max-pooling in the decoder and performs spatial convolution without bias.



You can learn more about E-Net in the paper written by Adam Paszke, Abhishek Chaurasia, Sangpil Kim, and Eugenio Culurciello: *ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation*. Click on the following link for more information: https://openreview.net/forum?id=HJy_5Mc1l.

Summary

In this chapter, we learned about the importance of semantic segmentation in the field of SDCs. We also looked at an overview of a few popular deep learning architectures related to semantic segmentation: U-Net, SegNet, PSPNet, DeepLabv3+, and E-Net.

In the next chapter, we will implement semantic segmentation using E-Net. We will use it to detect various objects in images and videos.

9

Implementing Semantic Segmentation

Deep learning has provided great accuracy in the field of computer vision, particularly for object detection. In the past, segmenting images was done by partitioning images into grab cuts, superpixels, and graph cuts. The main problem with the traditional process was that the algorithm was unable to recognize parts of the images.

On the other hand, semantic segmentation algorithms aim to divide the image into relevant categories. They associate every pixel in an input image with a class label: person, tree, street, road, car, bus, and so on. Semantic segmentation algorithms are dynamic and have many use cases, including **self-driving cars (SDCs)**.

In this chapter, you will learn how to perform semantic segmentation using OpenCV, deep learning, and the ENet architecture. As you read this chapter, you will learn how to apply semantic segmentation to images and videos using OpenCV.

In this chapter, we will cover the following topics:

- Semantic segmentation in images
- Semantic segmentation in videos

Let's get started!

Semantic segmentation in images

In this section, we are going to implement one project on semantic segmentation using a popular network called ENet.

Efficient Neural Network (ENet) is one of the more popular networks out there due to its ability to perform real-time, pixel-wise semantic segmentation. ENet is up to 18x faster, requires 75x fewer FLOPs, and has 79x fewer parameters than other networks. This means ENet provides better accuracy than the existing models, such as U-Net and SegNet. ENet networks are typically tested on CamVid, CityScapes, and SUN datasets. The model's size is 3.2 MB.

The model we are using has been trained on 20 classes:

- Road
- Sidewalk
- Building
- Wall
- Fence
- Pole
- TrafficLight
- TrafficSign
- Vegetation
- Terrain
- Sky
- Person
- Rider
- Car
- Truck
- Bus
- Train
- Motorcycle
- Bicycle
- Unlabeled

We will start with the semantic segmentation project:

1. First, we will import the necessary packages and libraries, such as numpy, openCV, and argparse:

```
import argparse
import cv2
import numpy as np
import imutils
import time
```

2. Next, we will read the sample input image, resize the image, and construct a blob from the sample image:

```
start = time.time()
SET_WIDTH = int(600)

normalize_image = 1 / 255.0
resize_image_shape = (1024, 512)

sample_img = cv2.imread('../images/example_02.jpg')
sample_img = imutils.resize(sample_img, width=SET_WIDTH)

blob_img = cv2.dnn.blobFromImage(sample_img, normalize_image,
resize_image_shape, 0, swapRB=True, crop=False)
```

3. Then, we will load our serialized ENET model from disk:

```
print("[INFO] loading model...")
cv_enet_model = cv2.dnn.readNet('../enet-cityscapes/enet-model.net')
```

4. Now, we'll perform a forward pass using the segmentation model:

```
cv_enet_model.setInput(blob_img)

cv_enet_model_output = cv_enet_model.forward()
```

5. Then, we'll load the class name labels:

```
label_values = open('../enet-cityscapes/enet-
classes.txt').read().strip().split("\n")
```

6. In the following code, we're inferring the shape of the total number of classes, along with the spatial dimensions of the mask image:

```
IMG_OUTPUT_SHAPE_START =1
IMG_OUTPUT_SHAPE_END =4
(classes_num, h, w) =
cv_enet_model_output.shape[IMG_OUTPUT_SHAPE_START:IMG_OUTPUT_SHAPE_
END]
```

The output class ID map will be $numclasses \times height \times width$ in size. Therefore, we take `argmax` to find the class label with the highest probability for each and every (x, y) coordinate:

```
class_map = np.argmax(cv_enet_model_output[0], axis=0)
```

7. If we have a colors file, we can load it from disk; otherwise, we need to randomly generate RGB colors for each class. A list of colors is initialized to represent each class:

```
if os.path.isfile('./enet-cityscapes/enet-colors.txt'):
    CV_ENET_SHAPE_IMG_COLORS = open('./enet-cityscapes/enet-
colors.txt').read().strip().split("\n")
    CV_ENET_SHAPE_IMG_COLORS =
    [np.array(c.split(",")).astype("int") for c in
    CV_ENET_SHAPE_IMG_COLORS]
    CV_ENET_SHAPE_IMG_COLORS = np.array(CV_ENET_SHAPE_IMG_COLORS,
    dtype="uint8")

else:
    np.random.seed(42)
    CV_ENET_SHAPE_IMG_COLORS = np.random.randint(0, 255,
    size=(len(label_values) - 1, 3),
    dtype="uint8")
    CV_ENET_SHAPE_IMG_COLORS = np.vstack([[0, 0, 0],
    CV_ENET_SHAPE_IMG_COLORS]).astype("uint8")
```

8. Now, we will map each class ID with the given class ID:

```
mask_class_map = CV_ENET_SHAPE_IMG_COLORS[class_map]
```

9. We will resize the mask and class map in such a way that its dimensions match the original size of the input image:

```
mask_class_map = cv2.resize(mask_class_map, (sample_img.shape[1],
sample_img.shape[0]),
interpolation=cv2.INTER_NEAREST)
```

```
class_map = cv2.resize(class_map, (sample_img.shape[1],
sample_img.shape[0]),
interpolation=cv2.INTER_NEAREST)
```

10. We will get a weighted combination of the input image and the mask to form a visualized output. Here, mask to image means filtering the image. The sum of weight in the convolution mask affects the overall intensity of the resulting image. The convolution mask can have weight sum of 1 or 0. In our case it is the sum of 0.4 and 0.6 that is, 1. Pixels with negative values may be generated using masks with negative weights:

```
cv_enet_model_output = ((0.4 * sample_img) + (0.6 *
mask_class_map)).astype("uint8")
```

11. Then, we initialize the legend's visualization:

```
my_legend = np.zeros(((len(label_values) * 25) + 25, 300, 3),
dtype="uint8")
```

12. It will loop over the class names and colors, thereby drawing the class name and color on the legend:

```
for (i, (class_name, img_color)) in enumerate(zip(label_values,
CV_ENET_SHAPE_IMG_COLORS)):
    # draw the class name + color on the legend
    color_info = [int(color) for color in img_color]
    cv2.putText(my_legend, class_name, (5, (i * 25) + 17),
                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)
    cv2.rectangle(my_legend, (100, (i * 25)), (300, (i * 25) + 25),
                  tuple(color_info), -1)
```

13. Now, we can show the input and output images:

```
cv2.imshow("My_Legend", my_legend)
cv2.imshow("Img_Input", sample_img)
cv2.imshow("CV_Model_Output", cv_enet_model_output)
cv2.waitKey(0)

end = time.time()
```

14. After that, we can show the amount of time the inference took:

```
print("[INFO] inference took {:.4f} seconds".format(end - start))
```

The legend for the semantic segmentation process can be seen in the following screenshot:



Fig 9.1: Legends of the ENet architecture

The input image for the model is as follows:

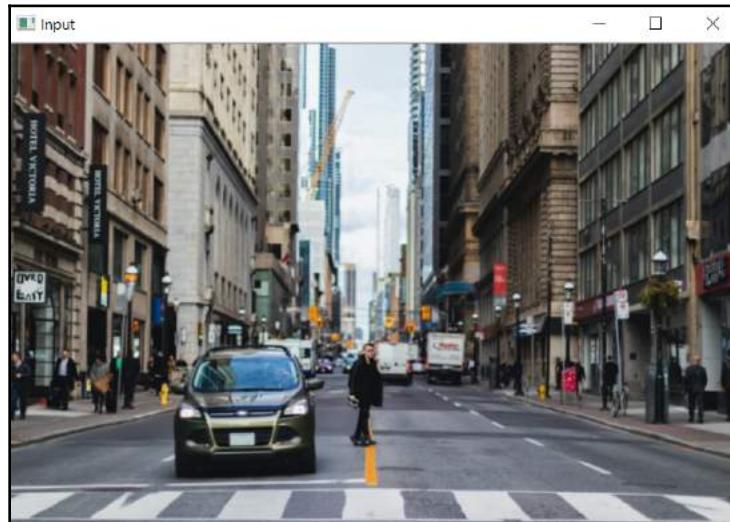


Fig 9.2: Input image

Let's take a look at the output image:



Fig 9.3: Output image

In the preceding image, we can observe the segmentation. We can see that a person is segmented in blue, a car is segmented in red, the sidewalks are segmented in pink, and that the buildings are segmented in gray. You can play with the implementation by applying it to images containing objects.

In the next section, we will develop a software pipeline for semantic segmentation for videos.

Semantic segmentation in videos

In this section, we are going to write a software pipeline using the OpenCV and ENet models to perform semantic segmentation on videos. Let's get started:

1. Import the necessary packages, such as numpy, imutils, and openCV:

```
import os
import time
import cv2
import imutils
import numpy as np
```

```
DEFAULT_FRAME = 1
WIDTH = 600
```

2. Then, load the class label names:

```
class_labels = open('./enet-cityscapes/enet-
classes.txt').read().strip().split("\n")
```

3. We can load the files from disk if we are supplied with the color file; otherwise, we will need to create the RGB colors for each class:

```
if os.path.isfile('./enet-cityscapes/enet-colors.txt'):
    CV_ENET_SHAPE_IMG_COLORS = open('./enet-cityscapes/enet-
colors.txt').read().strip().split("\n")
    CV_ENET_SHAPE_IMG_COLORS =
    [np.array(c.split(",")).astype("int") for c in
    CV_ENET_SHAPE_IMG_COLORS]
    CV_ENET_SHAPE_IMG_COLORS = np.array(CV_ENET_SHAPE_IMG_COLORS,
    dtype="uint8")

else:
    np.random.seed(42)
    CV_ENET_SHAPE_IMG_COLORS = np.random.randint(0, 255,
    size=(len(class_labels) - 1, 3),
    dtype="uint8")
    CV_ENET_SHAPE_IMG_COLORS = np.vstack([[0, 0, 0],
    CV_ENET_SHAPE_IMG_COLORS]).astype("uint8")
```

4. Now, load the model:

```
print("[INFO] loading model...")
cv_enet_model = cv2.dnn.readNet('./enet-cityscapes/enet-model.net')
```

5. Let's initialize the video stream so that we can output the video file:

```
sample_video = cv2.VideoCapture('./videos/toronto.mp4')
sample_video_writer = None
```

6. Now, we need to try and determine the total number of frames the video file contains:

```
try:
    prop = cv2.cv.CV_CAP_PROP_FRAME_COUNT if imutils.is_cv2() \
        else cv2.CAP_PROP_FRAME_COUNT
    total_time = int(sample_video.get(prop))
    print("[INFO] {} total_time video_frames in
video".format(total_time))
```

7. Now, we will write `except` if any errors occurred when determining the video frames:

```
except:  
    print("[INFO] could not determine # of video_frames in video")  
    total_time = -1
```

8. In the following code block, we loop over the frames from the video file stream and then read the next frame from the file. If a frame isn't grabbed, then this means we have reached the end of the stream:

```
while True:  
    (grabbed, frame) = sample_video.read()  
  
    if not grabbed:  
        break
```

9. In the following code, we are constructing a blob from the video frame and performing a forward pass using the segmentation model:

```
normalize_image = 1 / 255.0  
resize_image_shape = (1024, 512)  
video_frame = imutils.resize(video_frame, width=SET_WIDTH)  
blob_img = cv2.dnn.blobFromImage(video_frame, normalize_image,  
resize_image_shape, 0,  
                                    swapRB=True, crop=False)  
cv_enet_model.setInput(blob_img)  
start = time.time()  
cv_enet_model_output = cv_enet_model.forward()  
end = time.time()
```

10. Now, we can infer all the classes, along with the spatial dimensions of the mask image, through the output array shape:

```
(classes_num, h, w) = cv_enet_model_output.shape[1:4]
```

Here, the output class' ID map is `num_classes` (height (x) and width (y)) in size. We are going to take `argmax` to find the most likely class label for each and every coordinate (x, y) in the image:

```
class_map = np.argmax(enet_output[0], axis=0)
```

11. After getting the class's ID map, we can map each of the class IDs to their corresponding color codes:

```
mask_class_map = CV_ENET_SHAPE_IMG_COLORS[class_map]
```

12. Now, we will resize the mask in such a way that its dimensions should match the original size of its input frame:

```
mask_class_map = cv2.resize(mask_class_map, (video_frame.shape[1],  
                                              video_frame.shape[0]),  
                               interpolation=cv2.INTER_NEAREST)
```

13. We are going to perform a weighted combination of the input frame and the mask to form an output visualization:

```
cv_enet_model_output = ((0.3 * video_frame) + (0.7 *  
mask_class_map)).astype("uint8")
```

14. Now, we are going to check whether the video writer is `None`. If the writer is `None`, we have to initialize the video writer:

```
if sample_video_writer is None:  
    fourcc_obj = cv2.VideoWriter_fourcc(*"MJPG")  
    sample_video_writer =  
        cv2.VideoWriter('./output/output_toronto.avi', fourcc_obj, 30,  
                           (cv_enet_model_output.shape[1],  
                            cv_enet_model_output.shape[0]), True)  
  
    if total_time > 0:  
        execution_time = (end - start)  
        print("[INFO] single video_frame took {:.4f}  
seconds".format(execution_time))  
        print("[INFO] estimated total_time time:  
{:.4f}{}".format(  
            execution_time * total_time))
```

15. The following code helps us write the output frame to disk:

```
sample_video_writer.write(cv_enet_model_output)
```

16. The following code will help us verify whether we should display the output frame to our screen:

```
if DEFAULT_FRAME > 0:  
    cv2.imshow("Video Frame", cv_enet_model_output)  
    if cv2.waitKey(0) & 0xFF == ord('q'):  
        break
```

17. Release the file pointers and check the output video:

```
print("[INFO] cleaning up...")  
sample_video_writer.release()  
sample_video.release()
```

The following image shows the output of the video:

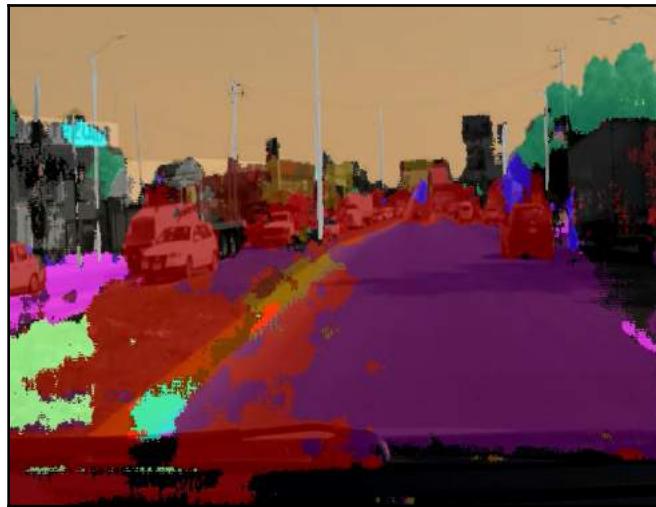


Fig 9.4: Output of the video

Here, we can see that the performance of the ENet architecture is good. You can also test the model with different videos and images. As we can see, it is able to segment the video and images with good accuracy. Using this architecture with autonomous cars will help them identify objects in real time.

Summary

In this chapter, we learned how to apply semantic segmentation using OpenCV, deep learning, and the ENet architecture. We used the pretrained ENet model on the Cityscapes dataset and performed semantic segmentation for both images and video streams. There were 20 classes in the context of SDCs and road scene segmentation, including vehicles, pedestrians, and buildings. We implemented and performed semantic segmentation on an image and a video. We saw that the performance of ENet is good for both videos and images. This will be one of the great contributions to making SDCs a reality as it helps them detect different types of objects in real time and ensures the car knows exactly where to drive.

In the next chapter, we are going to implement an interesting project called behavioral cloning. In this project, we are going to apply all the computer vision and deep learning knowledge we have gained from the previous chapters.

4

Section 4: Advanced Implementations

This section includes two important and advanced, real-time, self-driving car projects. We will implement a behavior cloning project and test in a virtual simulator. We will also implement vehicle detection using OpenCV and YOLO deep learning architecture. And, in Chapter 12, *Next Steps*, we will get an overview of sensor fusion and see what we can learn next.

This section comprises the following chapters:

- Chapter 10, *Behavior Cloning Using Deep Learning*
- Chapter 11, *Vehicle Detection Using OpenCV and Deep Learning*
- Chapter 12, *Next Steps*

10

Behavioral Cloning Using Deep Learning

Behavioral cloning is a means of identifying and reproducing human sub-cognitive abilities within a computer program. For example, a person performs tasks such as driving a vehicle, recording their actions along with the situation, and subsequently feeding the log of these records into the input of the learning algorithm. Then, the learning algorithm generates a series of rules replicating the behaviors of humans, which is called behavioral cloning. This approach can be used to construct automatic control systems for complex tasks where traditional control theory is inadequate.

In this chapter, we are going to implement behavioral cloning. Here, we need to train a neural network to predict the steering angles of a vehicle based on input images from different cameras. In this chapter we are going to drive a car autonomously in a simulator. This implementation is inspired from a NVIDIA research paper called *End to End Learning*. The simulator consists of 3 cameras, and records, center, left, and right images which are associated with the steering angle, speed, throttle, and brakes. You can read more about NVIDIA paper at: <https://arxiv.org/abs/1604.07316>.

So here, the neural network we are training is feeded with camera images and the output is a steering angle. This is a regression problem, so we will start this chapter with learning neural network for regression, and then deep learning for behavior cloning.



In 1990, as per the paper *Cognitive models from subcognitive skills*, published by Michie, Bain, and Hayes-Michie, behavioral cloning is a type of imitation learning technique whose main motivation is to create a model of a human's actions while performing complex skills. Source: <https://pascal-francis.inist.fr/vibad/index.php?action=getRecordDetail&idt=5218570>.

In this chapter, we will cover the following topics:

- Neural network for regression
- Behavior cloning using deep learning

Let's get started!

Neural network for regression

So far, we have only learned about neural networks for classification problems. But in this chapter, we will learn about one of the most important topics surrounding behavioral cloning: neural network for regression. The main aim of classification problems is to group the classes of objects into categories and predict the category of the class later. For example, cat and dog can be two different classes.

Now, we will look into a regression type example, where we will use linear regression to predict continuous values based on the relationship between the independent variable (X) and the dependent variable (y). In general, linear regression helps us predict values on a continuous spectrum, rather than on discrete classes.

The following is a linear regression plot:

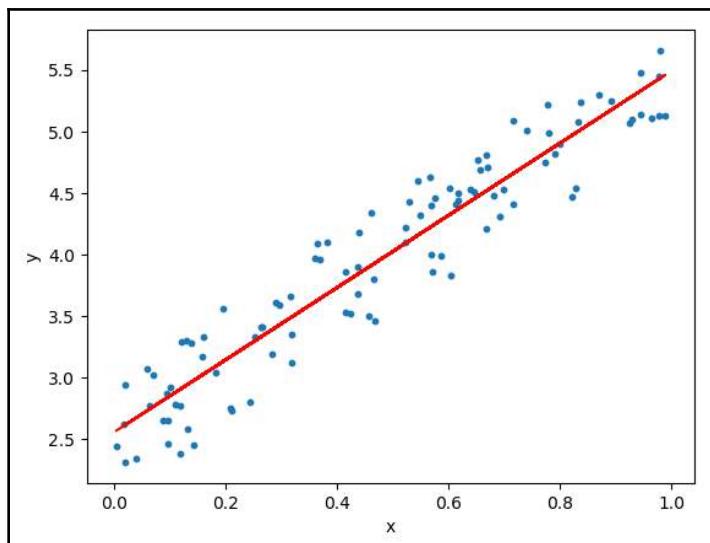


Fig 10.1: Linear regression plot

Linear regression is the simplest form of regression. The goal is to find the line parameterized by the set of slope and y-intercept parameters that provides the best fit to the training data and the train network. Later, it should be able to make predictions with the unlabeled test input.

Now, we are going to train a neural network to perform linear regression. We will do this by minimizing the loss over the dataset. The loss function for the neural network will be the **mean squared error (MSE)**, which will be covered in *Step 4* in the following steps. We will understand the process of creating a neural network for a regression type example and implement it with Keras. Let's get started:

1. First, we need to import the `numpy`, `matplotlib`, and `keras` libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

We will train the neural network so that we can fit a model into this non-linear data. First, let's set up this non-linear data by declaring the number of points, which is 500. This data forms a simple sinusoidal curve. This curve can be obtained by setting the dependent variable (y) to the sin of the corresponding independent variable (x). To add variation to the data, we will add `np.random.uniform`. This will grab a random value that will specify a range of negatives from -0.5 to +0.5.

2. Now, we'll train a neural network so that we get a model that fits this data:

```
np.random.seed(0)
points = 500
X = np.linspace(-3, 3, points)
y = np.sin(X) + np.random.uniform(-0.5, 0.5, points)
plt.scatter(X,y)
```

The output looks like this:

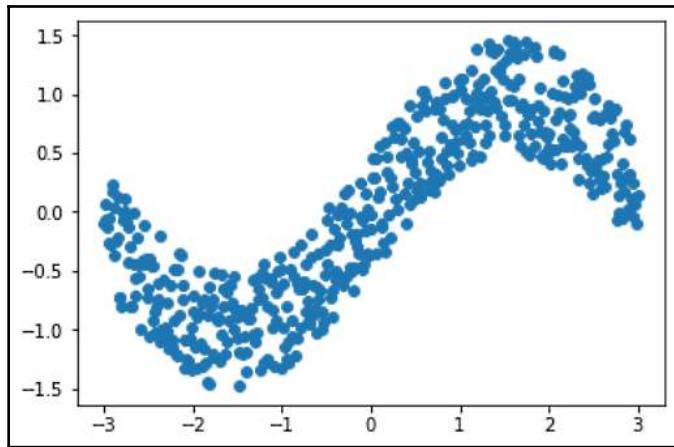


Fig 10.2: Non-linear data

3. Now, we will define our neural network architecture using Keras:

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(50, activation='sigmoid',
                               input_dim=1))
model.add(tf.keras.layers.Dense(30, activation='sigmoid'))
model.add(tf.keras.layers.Dense(1))
```

4. In the following steps, we will compile our model using the Adam optimizer and run the model. Our loss function is mse. MSE measures the average of the squares of the error, which is the average squared difference between the estimated values and what is estimated.

The formula for MSE is as follows:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

Here, we have the following values:

- Y_i is the actual value.
- \hat{Y}_i is the predicted value.
- n is the data point for all the variables.

5. In the following code block, we are specifying our learning rate as 0.01, the loss as the **mean square error**, the epoch as 50, and the optimizer as adam:

```
adam = Adam(lr=0.01)
model.compile(loss='mse', optimizer=adam)
model.fit(X, y, epochs=50)
```

6. Now that the model has been trained, we can start our prediction:

```
predictions = model.predict(X)
plt.scatter(X, y)
plt.plot(X, predictions, 'ro')
plt.show()
```

The output looks like this:

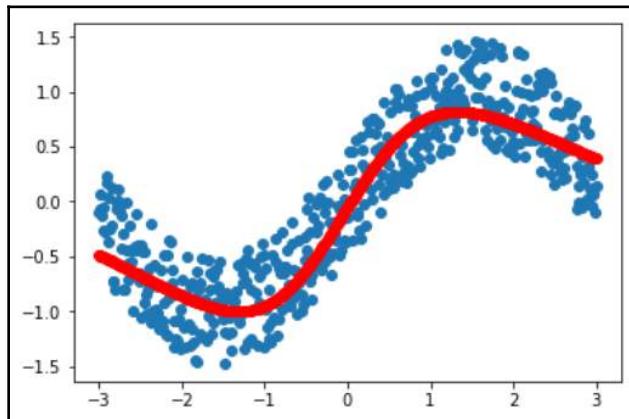


Fig 10.3: The predicted value for linear regression

In the preceding plot, the model estimated values on the y coordinate for every x coordinates, which led to it displaying the model that was trained to fit our data. Now that we have an understanding of regression type examples, we can move on to the most exciting section of this book, where we'll start building our **self-driving car (SDC)**. In this regression type example, we will train the neural network so that it fits a model into this non-linear data.

In the next section, we are going to start the behavior cloning process. We have completed all the necessary steps to move on to behavioral cloning, where we will simulate a fully functional SDC. This is going to be the most challenging but most exciting section of this chapter as we get to combine and apply all of the skills that we've learned so far in this book.

Behavior cloning using deep learning

This section will focus on a very useful technique called behavioral cloning. This chapter is relatively intense and will combine all the previous techniques we have dealt with in this book, such as deep learning, feature extraction from images, CNNs, and continuous regression.

We are going to follow these steps:

1. Download an open source SDC simulator by Udacity.
2. Collect the training data by driving the car in manual mode in the simulator. The training data consists of images from the surrounding environment of the car and the steering angles.
3. Clean the collected dataset using various OpenCV techniques.
4. Train a convolution neural network model.
5. Evaluate the model in Autonomous mode of the Udacity simulator.

This project isn't easy as it requires complex deep learning techniques and image preprocessing techniques. For this reason, I have structured this book so that you have all the necessary skills to complete this project; what you learned in the previous chapters can now be applied to this challenging task.

Let's get started by collecting the data, which we will use later to train the deep learning model.

Data collection

In this section, we will download the simulator that will enable us to begin our behavioral cloning process. We're going to start by driving the car through the simulator using our keyboard keys. That way, we're able to train a convolutional neural network to monitor the controlled operation and movement of the vehicle. Depending on how you're driving, it copies your behavior to then drive on its own in Autonomous mode. How well the neural network drives the car is determined by how well you're able to drive the car in the simulator.

We'll be using the following simulator, which is open source and available on GitHub. You can find it at <https://github.com/udacity/self-driving-car-sim>. There are other simulators that you can make use of, such as AirSim (<https://github.com/microsoft/AirSim>), which is another open source SDC simulator that's based on Unreal Engine.

We have two versions of the simulator, but we will go with **version 1**. Make sure that you download the correct file – the one that's compatible with the computer you are working with. Once you've finished downloading it, make sure to unzip the file. After you've done this, a window should appear. From this window, you can launch the **simulator** and set the screen resolution to **800x600** and the graphics quality to **fastest** – this will give you the best experience possible. When we press **Play**, we'll get a window with two options: **Training mode** and **Autonomous mode**.

Training mode is when we drive the car ourselves and gather training data, which we will use to train the model. This will help us train the neural network to emulate our behavior. Autonomous mode is when we test our neural network to drive the car on its own.

The **Training** and **Autonomous mode** options look like this:



Fig 10.4: Training mode and Autonomous mode

After we have collected the training data, we will drive the car on the flat terrain track and check the accuracy of how our car drives autonomously on the same track. Later, we'll test it on the second track with a much rougher terrain that's composed of steeper hills.

The following screenshot shows the simulator in action:



Fig 10.5: Udacity Simulator

Based on the preceding screenshot, we will do the following to complete the data collection process:

1. First, we'll select **Training mode** in the simulator.
2. Then, we will drive the car, which is very similar to playing a video game. We will make use of the arrow keys on the keyboard – the up button to drive forward, the left and right buttons to steer in the appropriate direction, and the down button to reverse.



Before collecting data using the car simulator, practice using the controls; any data with poor driving will result in poor data and thus we will get a poorly trained model. Once we are used to the controls of the simulator, we can start collecting data by pressing the **RECORD** button on the top right (as shown in the following image). The collected data will be stored in the user directory where the simulator exists.

The following image shows the simulator in recording mode:



Fig 10.6: Recording mode

We can also save the data in our current location, which is where we have our simulator. Once we've done that, we're going to drive roughly three laps around the track. These three laps will provide us with a fair amount of data; you should have enough data for your model to work with. The track is structured in such a way that it will challenge your neural network to overcome sharp turns.



Different parts of the track have different textures, edges, and borders – there are different curvatures, layouts, and landscapes throughout the track. All of these are different features that will be extracted with the convolutional neural network and are associated with a corresponding steering angle through a regression-based algorithm.

Our aim is to create a successful neural model that drives down the center of the road. So, while making turns in the simulator, try to stay as close to the middle of the road as possible. Even if the car suddenly changes direction, try to readjust the steering to come back to the middle of the track. Remember, your vehicle's driving is only as good as the behavior of the driver.

Once you've finished your three laps, you can run a couple of laps in the reverse direction in order to gather more data that will help the model generalize. Another reason to drive laps in the reverse direction is that we need to guarantee that our data is balanced. Also, the track in the simulator has more left turns, which makes its bias toward the left-hand side. This means that, while driving forward, you're mostly steering left, so driving the car in one direction around the track will skew your data to one side.

Another way to eliminate data imbalance is by flipping the images. You will find a better visual of data balances later on in this section.

As we already know, if the MSE is high on both the training and validation sets, then we've got an underfitting problem. However, if the MSE is low on the training set but high on the validation set, then the model has an overfitting problem. In these cases, a larger dataset can help improve the model.

In other cases, you might realize that, upon testing your model, whenever you encounter a turn, the car falls off the track. You can choose to add more specific and helpful data by only re-recording specific turns, thus providing the model with more turning data. Your car is equipped with three cameras: one camera mounted on the left, one mounted in the middle, and one mounted on the right of the windshield. The simulator collects the values for the steering angle, speed, throttle, and braking while you drive the car and record.

Once the data has been collected, we should have a folder that contains all the image data and all the frames that we recorded while driving. We should also have a CSV file. If you were to open up this CSV file, you will see that it contains a log of the recorded images. The first column contains the data related to the images that were taken by the camera in the center, the images that were taken by the camera adjacent to the center camera, the images that were taken by the camera mounted on the left of the windshield, and also the images that were taken by the camera placed on the right of the windshield. All these images were taken at a specific point in time, and each corresponds to a specific steering angle, throttle, brake pressure, and speed. Now, all of this extra information is nice to have, but the steering angle is the only thing that needs to be predicted in order to ensure that our car knows how to steer. These predictions are based on the images that contain features corresponding to a specific steering angle. I'm sure other advanced models would also make use of the throttle and brake, but for our purposes, we will only focus on the steering angle. This corresponds to various radian values that have been arranged from **-1** to **1**.

The training dataset of each set of images has a labeled steering angle. This is going to help the network learn features for each image and the steering angle. Whenever there is a straight path, the steering angle is most likely **zero**, whereas whenever there is a curvature to the left, the steering angle is **negative** to denote a left turn and **positive** to denote a right turn.

We are using the training images in order to train the neural network to predict the appropriate steering angles for the car, thus preparing us for when we test the car to drive on its own.

In summary, this is not a classification problem, but a regression-based problem. This is because we are trying to predict the steering angle based on a continuous spectrum. This is why we have learned about the regression example for this task; normally, we would use convolutional neural networks for classification purposes.

Let's open the set of images stating the date when they were captured or their **timestamp**. Notice how we have three of the same images, except each image is taken from a different viewpoint. This approach of making use of three cameras was proposed by NVIDIA and essentially helps us generalize the model when we collect more samples of the same scene.

The following images were taken from the different cameras in the simulator:



Fig 10.7: Camera images taken from the simulator

Here's our CSV file:

8	C:\Users\	C:\Users\	C:\Users\	0	0	0	15.64832
9	C:\Users\	C:\Users\	C:\Users\	0	0	0	15.47731
10	C:\Users\	C:\Users\	C:\Users\	0	0	0	15.85111
11	C:\Users\	C:\Users\	C:\Users\	0	0	0	15.17079
12	C:\Users\	C:\Users\	C:\Users\	0	0	0	14.98707
13	C:\Users\	C:\Users\	C:\Users\	0	0	0	15.35287
14	C:\Users\	C:\Users\	C:\Users\	0	0	0	14.68478
15	C:\Users\	C:\Users\	C:\Users\	0	0.105303	0	15.05394
16	C:\Users\	C:\Users\	C:\Users\	0	0.387209	0	14.7339
17	C:\Users\	C:\Users\	C:\Users\	0	0.758071	0	15.56607

Fig 10.8: The steering angle CSV file

The next step is to prepare the data. Data preparation is one of the most important steps for any deep learning project because it is important for model performance.

Data preparation

In this section, we are going to perform data preparation. We have to prepare the data before running a deep learning model.

We will start by exploring the data. Later, we will apply the computer vision techniques that we studied in Chapter 4, *Computer Vision for Self-Driving Cars*. Let's get started:

1. First, add the necessary libraries, including numpy, pandas, and keras, as shown in the following code block:

```
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Convolution2D, MaxPooling2D, Dropout,
Flatten, Dense
import cv2
import pandas as pd
import random
import os
import ntpath
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
import matplotlib.image as mpimg
from imgaug import augmenters as iaa
```

2. The next step is to initialize the folder name where the training images and driving logs will be stored, as well as the names of the columns for loading the drive log file. Then, we will read the drive log file with specific columns by using the pandas library:

```
datadir = 'track'

columns = ['center', 'left', 'right', 'steering', 'throttle',
'reverse', 'speed']

data = pd.read_csv(os.path.join(datadir, 'driving_log.csv'), names
= columns)
pd.set_option('display.max_colwidth', -1)

data.head()
```

The following screenshot shows the output of the preceding code:

left		right	steering	throttle	reverse	speed
top\new_track\IMG\left_2018_07_16_17_11_43_382.jpg	C:\Users\Amer\Desktop\new_track\IMG\right_2018_07_16_17_11_43_382.jpg	0.0	0.0	0.0	0.649786	
top\new_track\IMG\left_2018_07_16_17_11_43_670.jpg	C:\Users\Amer\Desktop\new_track\IMG\right_2018_07_16_17_11_43_670.jpg	0.0	0.0	0.0	0.627942	
top\new_track\IMG\left_2018_07_16_17_11_43_724.jpg	C:\Users\Amer\Desktop\new_track\IMG\right_2018_07_16_17_11_43_724.jpg	0.0	0.0	0.0	0.622910	
top\new_track\IMG\left_2018_07_16_17_11_43_792.jpg	C:\Users\Amer\Desktop\new_track\IMG\right_2018_07_16_17_11_43_792.jpg	0.0	0.0	0.0	0.619162	
top\new_track\IMG\left_2018_07_16_17_11_43_860.jpg	C:\Users\Amer\Desktop\new_track\IMG\right_2018_07_16_17_11_43_860.jpg	0.0	0.0	0.0	0.615438	

Fig 10.9: Driving log

- This module provides `os.path` functionality on Windows platforms, while `ntpath.split()` provides us with the filename and folder path after removing the location details:

```
def path_leaf(path):
    # : This module provides os.path functionality on Windows
    # platforms.
    # : ntpath.split() gives the file name and folder path
    head, tail = ntpath.split(path)
    return tail

data['center'] = data['center'].apply(path_leaf)
data['left'] = data['left'].apply(path_leaf)
data['right'] = data['right'].apply(path_leaf)
data.head()
```

The output file is shown in the following screenshot:

	center	left	right	steering	throttle	reverse	speed
0	center_2018_07_16_17_11_43_382.jpg	left_2018_07_16_17_11_43_382.jpg	right_2018_07_16_17_11_43_382.jpg	0.0	0.0	0.0	0.649786
1	center_2018_07_16_17_11_43_670.jpg	left_2018_07_16_17_11_43_670.jpg	right_2018_07_16_17_11_43_670.jpg	0.0	0.0	0.0	0.627942
2	center_2018_07_16_17_11_43_724.jpg	left_2018_07_16_17_11_43_724.jpg	right_2018_07_16_17_11_43_724.jpg	0.0	0.0	0.0	0.622910
3	center_2018_07_16_17_11_43_792.jpg	left_2018_07_16_17_11_43_792.jpg	right_2018_07_16_17_11_43_792.jpg	0.0	0.0	0.0	0.619162
4	center_2018_07_16_17_11_43_860.jpg	left_2018_07_16_17_11_43_860.jpg	right_2018_07_16_17_11_43_860.jpg	0.0	0.0	0.0	0.615438

Fig 10.10: Driving logs with the image name

4. Now, we will initialize `num_bins`. It's usually specified as the consecutive, non-overlapping intervals of a variable. We will visualize our data and check the distribution:

```
num_bins = 25

samples_per_bin = 400

hist, bins = np.histogram(data['steering'], num_bins)
center = (bins[:-1] + bins[1:]) * 0.5
plt.bar(center, hist, width=0.05)
plt.plot((np.min(data['steering']), np.max(data['steering'])),
          (samples_per_bin, samples_per_bin))
```

Here's the output:

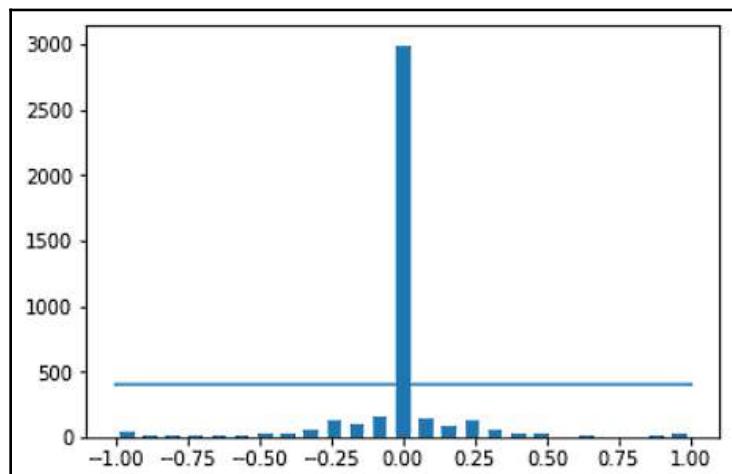


Fig 10.11: Steering angle visualization

5. Here, we can see that the data has more data points on the number of 0 angles. We will take the number of data points up to 400 and plot them:

```
print('total data:', len(data))
remove_list = []
for j in range(num_bins):
    list_ = []
    for i in range(len(data['steering'])):
        if data['steering'][i] >= bins[j] and data['steering'][i]
        <= bins[j+1]:
            list_.append(i)
    list_ = shuffle(list_)
```

```
list_ = list_[samples_per_bin:]
remove_list.extend(list_)

print('removed:', len(remove_list))
data.drop(data.index[remove_list], inplace=True)
print('remaining:', len(data))

hist, _ = np.histogram(data['steering'], (num_bins))
plt.bar(center, hist, width=0.05)
plt.plot((np.min(data['steering'])), np.max(data['steering'])),
(samples_per_bin, samples_per_bin))
```

Here is the count of data after taking 400 samples:

- Total data: 4053
- Removed: 2590
- Remaining: 1463

We can see the output of the visualization process in the following image:

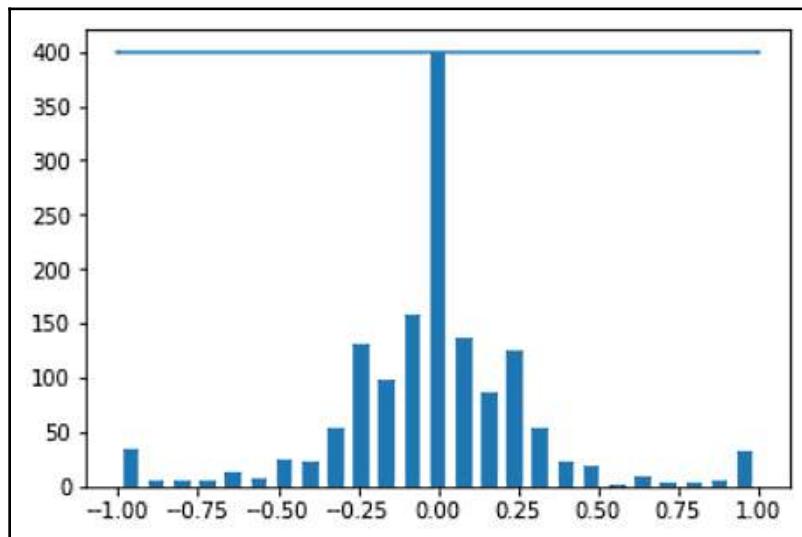


Fig 10.12: Steering angle with 400 samples

6. Now, we will check the data points, as follows:

```
print(data.iloc[1])
def load_img_steering(datadir, df):
    image_path = []
```

```

steering = []
for i in range(len(data)):
    indexed_data = data.iloc[i]
    center, left, right = indexed_data[0], indexed_data[1],
    indexed_data[2]
    image_path.append(os.path.join(datadir, center.strip()))
    steering.append(float(indexed_data[3]))
    # left image append
    image_path.append(os.path.join(datadir, left.strip()))
    steering.append(float(indexed_data[3])+0.15)
    # right image append
    image_path.append(os.path.join(datadir, right.strip()))
    steering.append(float(indexed_data[3])-0.15)
image_paths = np.asarray(image_path)
steerings = np.asarray(steering)
return image_paths, steerings

image_paths, steerings = load_img_steering(datadir + '/IMG', data)

```

The output looks as follows:

center	center_2018_07_16_17_11_44_069.jpg
left	left_2018_07_16_17_11_44_069.jpg
right	right_2018_07_16_17_11_44_069.jpg
steering	0
throttle	0
reverse	0
speed	0.601971

7. Now, we need to split the data using the sklean library:

```

X_train, X_valid, y_train, y_valid = train_test_split(image_paths,
steerings, test_size=0.2, random_state=6)
print('Training Samples: {} \nValid Samples: {}'.
format(len(X_train), len(X_valid)))

```

Here's the output:

```

Training Samples: 3511
Valid Samples: 878

```

8. Now, we will check the distributions of the training and test datasets:

```

fig, axes = plt.subplots(1, 2, figsize=(12, 4))
axes[0].hist(y_train, bins=num_bins, width=0.05, color='blue')
axes[0].set_title('Training set')
axes[1].hist(y_valid, bins=num_bins, width=0.05, color='red')
axes[1].set_title('Validation set')

```

The following image shows these distributions in detail:

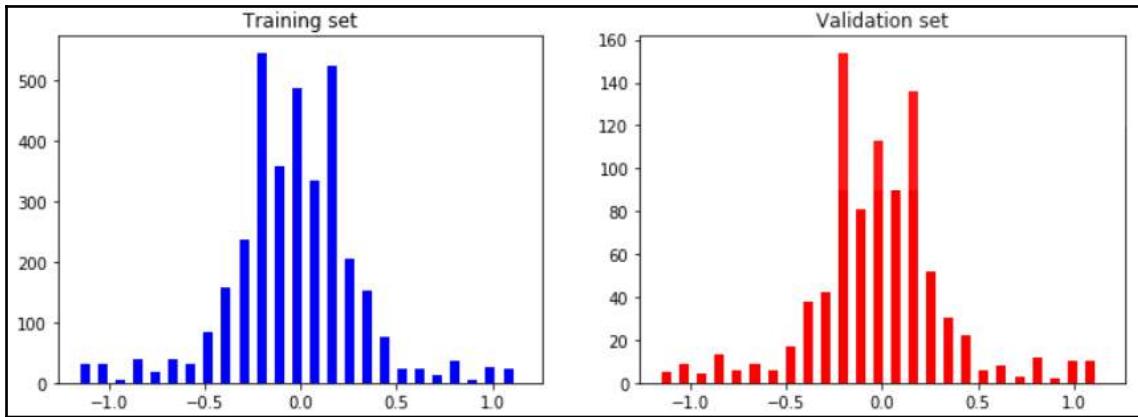


Fig 10.13: Training set and validation set distributions

9. Now, we will explore the image and clean it. We will only crop the useful part of the image. Then, we'll apply various computer vision filtering techniques to it. We'll start by zooming in on the image:

```
def zoom(image):
    zoom = iaa.Affine(scale=(1, 1.3))
    image = zoom.augment_image(image)
    return image

image = image_paths[random.randint(0, 1000)]
original_image = mpimg.imread(image)
zoomed_image = zoom(original_image)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()

axs[0].imshow(original_image)
axs[0].set_title('Original Image')

axs[1].imshow(zoomed_image)
axs[1].set_title('Zoomed Image')
```

Here's a screenshot of the output:

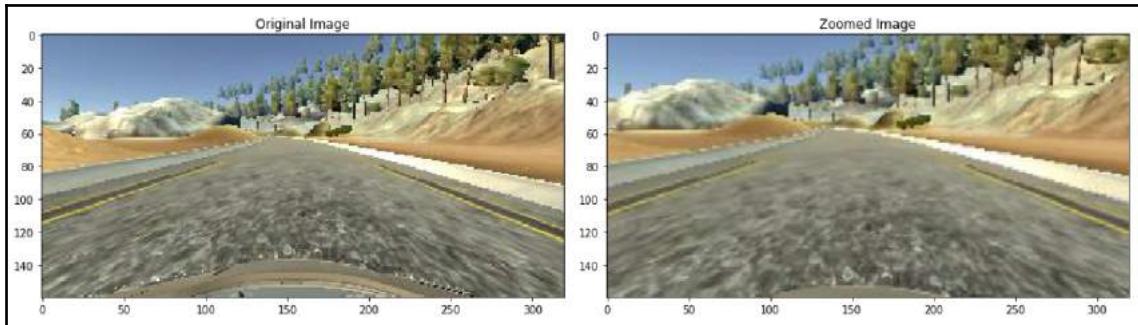


Fig 10.14: Original image and the zoomed image

10. Using the following function, we will pan the image:

```
def pan(image):
    pan = iaa.Affine(translate_percent= {"x" : (-0.1, 0.1), "y": (-0.1, 0.1)})
    image = pan.augment_image(image)
    return image

image = image_paths[random.randint(0, 1000)]
original_image = mpimg.imread(image)
panned_image = pan(original_image)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()

axs[0].imshow(original_image)
axs[0].set_title('Original Image')

axs[1].imshow(panned_image)
axs[1].set_title('Panned Image')
```

Here's a screenshot of the panned image:

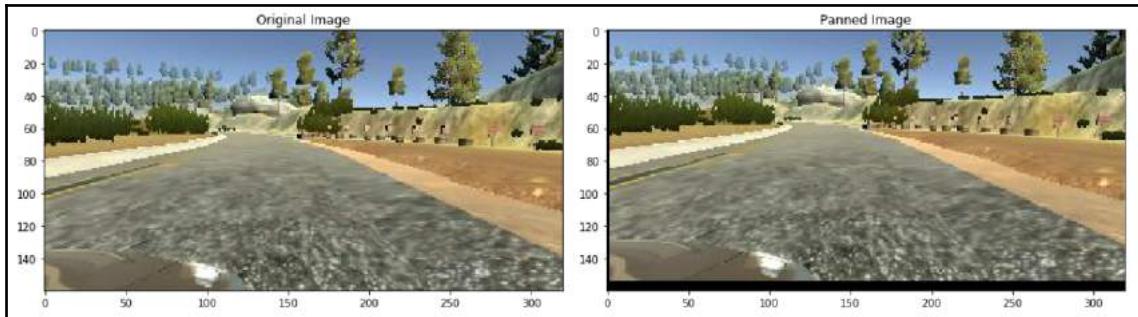


Fig 10.15: Original image and the panned image

11. Now, we will play with the brightness of the image:

```
def img_random_brightness(image):
    brightness = iaa.Multiply((0.2, 1.2))
    image = brightness.augment_image(image)
    return image

image = image_paths[random.randint(0, 1000)]
original_image = mpimg.imread(image)
brightness_altered_image = img_random_brightness(original_image)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()

axs[0].imshow(original_image)
axs[0].set_title('Original Image')

axs[1].imshow(brightness_altered_image)
axs[1].set_title('Brightness altered image ')
```

Here's our output:

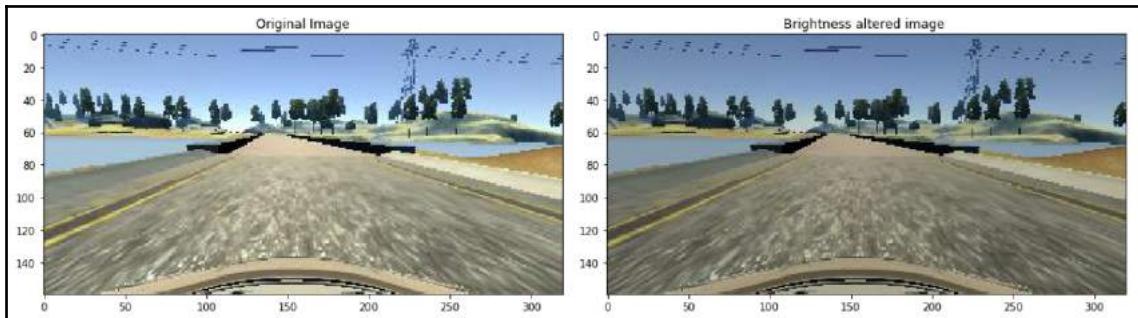


Fig 10.16: Original image and the brightness altered image

12. Now, we will flip the image horizontally and vertically:

```
def img_random_flip(image, steering_angle):
    image = cv2.flip(image,1)
    steering_angle = -steering_angle
    return image, steering_angle

random_index = random.randint(0, 1000)
image = image_paths[random_index]
steering_angle = steerings[random_index]

original_image = mpimg.imread(image)
flipped_image, flipped_steering_angle =
img_random_flip(original_image, steering_angle)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()

axs[0].imshow(original_image)
axs[0].set_title('Original Image - ' + 'Steering Angle:' +
str(steering_angle))

axs[1].imshow(flipped_image)
axs[1].set_title('Flipped Image - ' + 'Steering Angle:' +
str(flipped_steering_angle))
```

Here's a screenshot of the flipped image:

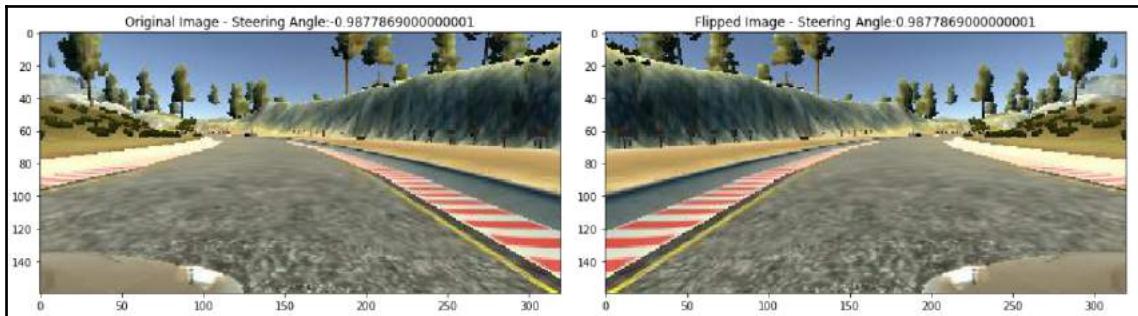


Fig 10.17: Original image and the flipped image

13. Now, we will flip the image on the basis of the steering angles:

```
def random_augment(image, steering_angle):
    image = mpimg.imread(image)
    if np.random.rand() < 0.5:
        image = pan(image)
    if np.random.rand() < 0.5:
        image = zoom(image)
    if np.random.rand() < 0.5:
        image = img_random_brightness(image)
    if np.random.rand() < 0.5:
        image, steering_angle = img_random_flip(image,
                                                steering_angle)

    return image, steering_angle

ncol = 2
nrow = 10

fig, axs = plt.subplots(nrow, ncol, figsize=(15, 50))
fig.tight_layout()

for i in range(10):
    randnum = random.randint(0, len(image_paths) - 1)
    random_image = image_paths[randnum]
    random_steering = steerings[randnum]

    original_image = mpimg.imread(random_image)
    augmented_image, steering = random_augment(random_image,
                                                random_steering)
```

```
    axs[i][0].imshow(original_image)
    axs[i][0].set_title("Original Image")

    axs[i][1].imshow(augmented_image)
    axs[i][1].set_title("Augmented Image")
```

Here's a screenshot of the output:

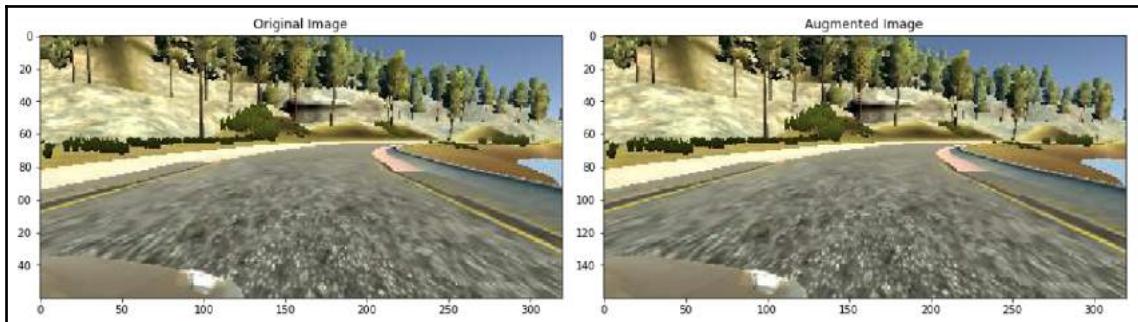


Fig 10.18: Original image and the augmented image

14. Now, we will apply the final preprocessing step – cropping the image and applying Gaussian blur:

```
def img_preprocess(img):
    img = img[60:135,:,:]
    img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
    img = cv2.GaussianBlur(img, (3, 3), 0)
    img = cv2.resize(img, (200, 66))
    img = img/255
    return img

image = image_paths[100]
original_image = mpimg.imread(image)
preprocessed_image = img_preprocess(original_image)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()
axs[0].imshow(original_image)
axs[0].set_title('Original Image')
axs[1].imshow(preprocessed_image)
axs[1].set_title('Preprocessed Image')
```

Here's a screenshot of the output:

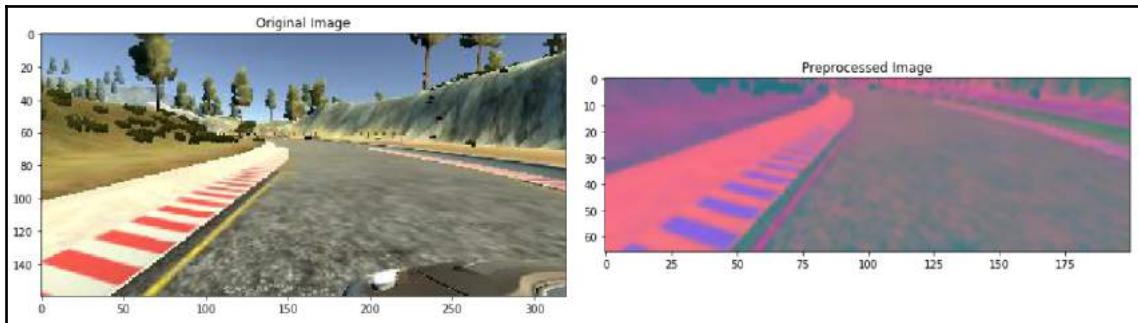


Fig 10.19: Original image and the preprocessed Image

15. We are going to write a method called the `batch_generator` method that uses a dataset instance of the `Sequence` object (`keras.utils.Sequence`) in order to avoid duplicate data when we use multiprocessing. It provides tuple objects; either `(input, target)` or `(input, target, sample_weight)`:

```
def batch_generator(image_paths, steering_ang, batch_size,
istraining):  
  
    while True:  
        = []  
        batch_steering = []  
  
        for i in range(batch_size):  
            random_index = random.randint(0, len(image_paths) - 1)  
            if istraining:  
                im, steering =  
random_augment(image_paths[random_index],  
steering_ang[random_index])  
            else:  
                im = mpimg.imread(image_paths[random_index])  
                steering = steering_ang[random_index]  
            im = img_preprocess(im)  
            batch_img.append(im)  
            batch_steering.append(steering)  
        yield (np.asarray(batch_img), np.asarray(batch_steering))
```

16. In the following code, we're exploring the training and test data:

```
x_train_gen, y_train_gen = next(batch_generator(X_train, y_train,
1, 1))
x_valid_gen, y_valid_gen = next(batch_generator(X_valid, y_valid,
1, 0))

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()

axs[0].imshow(x_train_gen[0])
axs[0].set_title('Training Image')

axs[1].imshow(x_valid_gen[0])
axs[1].set_title('Validation Image')
```

Here's the result of the test and training data after processing the image:

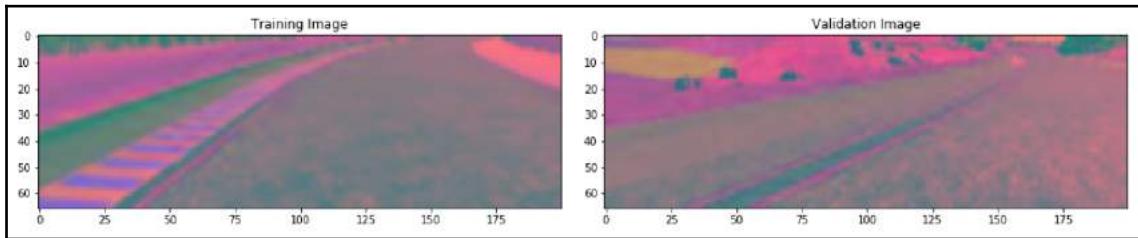


Fig 10.20: Training image and the validation image

In the next section, we will develop a deep learning model.

Model development

In this section, we will design our model architecture. We are going to use the model architecture provided by NVIDIA.

1. The code for developing the NVIDIA architecture for behavioral cloning can be seen in the following code block. Here we are going to use ADAM as optimizer, loss as MSE as output is steering angle and it is a regression problem. Also **Exponential Linear Unit (ELU)** is used as activation function. ELU is better than ReLU as it reduces cost function faster to zero. ELU is more accurate and good at solving vanishing gradient problem. You can read more about ELU at, <https://arxiv.org/abs/1511.07289>. Let's get started:

```
def nvidia_model():
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Convolution2D(24, 5, 5, subsample=(2, 2),
                                           input_shape=(66, 200, 3), activation='elu'))
    model.add(tf.keras.layers.Convolution2D(36, 5, 5, subsample=(2, 2),
                                           activation='elu'))
    model.add(tf.keras.layers.Convolution2D(48, 5, 5, subsample=(2, 2),
                                           activation='elu'))
    model.add(tf.keras.layers.Convolution2D(64, 3, 3,
                                           activation='elu'))

    model.add(tf.keras.layers.Convolution2D(64, 3, 3,
                                           activation='elu'))
    # model.add(tf.keras.layers.Dropout(0.5))

    model.add(tf.keras.layers.Flatten())

    model.add(tf.keras.layers.Dense(100, activation = 'elu'))
    model.add(tf.keras.layers.Dense(50, activation = 'elu'))

    model.add(tf.keras.layers.Dense(10, activation = 'elu'))

    model.add(tf.keras.layers.Dense(1))

    optimizer = Adam(lr=1e-3)
    model.compile(loss='mse', optimizer=optimizer)
    return model
```

2. Next, we will print the model summary:

```
model = nvidia_model()
print(model.summary())
```

Here's a screenshot of the model summary:

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_1 (Conv2D)	(None, 31, 98, 24)	1824
conv2d_2 (Conv2D)	(None, 14, 47, 36)	21636
conv2d_3 (Conv2D)	(None, 5, 22, 48)	43248
conv2d_4 (Conv2D)	(None, 3, 20, 64)	27712
conv2d_5 (Conv2D)	(None, 1, 18, 64)	36928
flatten_1 (Flatten)	(None, 1152)	0
dense_1 (Dense)	(None, 100)	115300
dense_2 (Dense)	(None, 50)	5050
dense_3 (Dense)	(None, 10)	510
dense_4 (Dense)	(None, 1)	11
<hr/>		
Total params:	252,219	
Trainable params:	252,219	
Non-trainable params:	0	
<hr/>		
None		

Fig 10.21: NVIDIA model summary

- Now, we're going to train the model on data generated batch by batch by a Python generator:

```
history = model.fit_generator(batch_generator(X_train, y_train,
100, 1),
                               steps_per_epoch=300,
                               epochs=10,
                               validation_data=batch_generator(X_valid, y_valid, 100, 0),
                               validation_steps=200,
                               verbose=1,
                               shuffle = 1)
```

The model training log can be seen in the following screenshot:

```

Epoch 1/10
300/300 [=====] - 407s 1s/step - loss: 0.1373 - val_loss: 0.0824
Epoch 2/10
300/300 [=====] - 411s 1s/step - loss: 0.0775 - val_loss: 0.0584
Epoch 3/10
300/300 [=====] - 418s 1s/step - loss: 0.1006 - val_loss: 0.0795
Epoch 4/10
300/300 [=====] - 401s 1s/step - loss: 0.0779 - val_loss: 0.0608
Epoch 5/10
300/300 [=====] - 373s 1s/step - loss: 0.0596 - val_loss: 0.0432
Epoch 6/10
300/300 [=====] - 383s 1s/step - loss: 0.0527 - val_loss: 0.0413
Epoch 7/10
300/300 [=====] - 378s 1s/step - loss: 0.0478 - val_loss: 0.0380
Epoch 8/10
300/300 [=====] - 401s 1s/step - loss: 0.0447 - val_loss: 0.0341
Epoch 9/10
300/300 [=====] - 407s 1s/step - loss: 0.0429 - val_loss: 0.0315
Epoch 10/10
300/300 [=====] - 397s 1s/step - loss: 0.0428 - val_loss: 0.0301

```

Fig 10.22: Model training log

4. Now, we will check the model's training and validation loss:

```

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['training', 'validation'])
plt.title('Loss')
plt.xlabel('Epoch')

```

Here's our output:

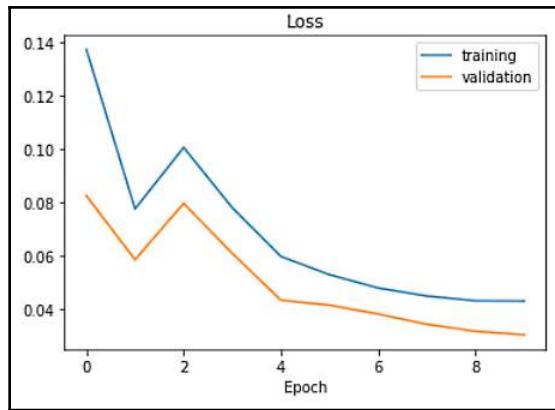


Fig 10.23: Training and validation loss

- Finally, we'll save the model as we are going to use it for autonomous driving later:

```
model.save('model.h5')
```

Now, we will evaluate the model in the simulator.

Evaluating the simulator

The final step is to check how our model is performing. To validate the simulator, we have to run the model in Autonomous mode. To run the model in Autonomous mode, we have to write a script that will set up bidirectional client-server communication and connect the model to the simulator. An example of this can be seen in the following diagram:

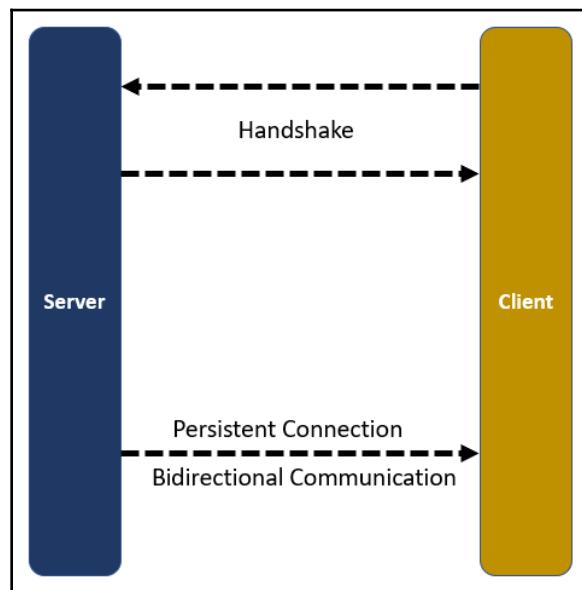


Fig 10.24: Bidirectional client-server communication

We have to run the following code to establish a connection to the simulator. This section is not related to deep learning; it is just about connecting to the simulator:

1. First, we will import the required libraries, including `socketio`, `eventlet`, `numpy`, and `OpenCV`:

```
import socketio
import eventlet
import numpy as np
from flask import Flask
from keras.models import load_model
import base64
from io import BytesIO
from PIL import Image
import cv2
```

2. Next, we will connect to the socket:

```
sio = socketio.Server()

app = Flask(__name__) #'__main__'
speed_limit = 10
def img_preprocess(img):
    img = img[60:135,:,:]
    img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
    img = cv2.GaussianBlur(img, (3, 3), 0)
    img = cv2.resize(img, (200, 66))
    img = img/255
    return img
```

3. Then, we will define the event handler so that we can load the model:

```
@sio.on('telemetry')
def telemetry(sid, data):
    speed = float(data['speed'])
    image = Image.open(BytesIO(base64.b64decode(data['image'])))
    image = np.asarray(image)
    image = img_preprocess(image)
    image = np.array([image])
    steering_angle = float(model.predict(image))
    throttle = 1.0 - speed/speed_limit
    print ('{} {} {}'.format(steering_angle, throttle, speed))
    send_control(steering_angle, throttle)
```

- After that, we need to write a function that will control the steering angle and throttle of the car:

```
@sio.on('connect')
def connect(sid, environ):
    print('Connected')
    send_control(0, 0)

def send_control(steering_angle, throttle):
    sio.emit('steer', data = {
        'steering_angle': steering_angle.__str__(),
        'throttle': throttle.__str__()
    })
```

- Next, we load the model and establish the server-client connection:

```
if __name__ == '__main__':
    model = load_model('model.h5')
    app = socketio.Middleware(sio, app)
    eventlet.wsgi.server(eventlet.listen(' ', 4567)), app)
```

By doing this, you'll see the following messages in your screen, and you will also be able to run your simulator:

```
(34704) wsgi starting up on http://0.0.0.0:4567
(34704) accepted ('127.0.0.1', 57704)
```

Open the simulator and run it in Autonomous mode. You should see your car running smoothly, as shown in the following screenshot:



Fig 10.25: Output of Autonomous mode

If the car's performance in Autonomous mode is not good, then you'll need to recollect the data and retrain the model.

Summary

In this chapter, we learned about behavioral cloning. We downloaded the Udacity open source simulator, ran the car in manual mode, and collected the data. Later, we cloned the model using the deep learning architecture provided by NVIDIA. We also applied various computer vision techniques as part of our data preparation process. In the end, our autonomous car drove smoothly. We hope you enjoyed building an autonomous car!

In the next chapter, we will learn about object detection. We are going to use a popular pre-trained model called YOLO and implement a deep learning project for vehicle detection.

11

Vehicle Detection Using OpenCV and Deep Learning

Object detection is one of the important applications of computer vision used in self-driving cars. Object detection in images means not only identifying the kind of object but also localizing it within the image by generating the coordinates of a **bounding box** that contains the object. We can summarize object detection as follows:

$$\text{ObjectDetection} = \text{classification} + \text{localization}$$

An example of object detection can be seen in the following image:



Fig 11.1: Object detection

Here, you can see that the biker is detected as a **person** and that the bike is detected as a **motorbike**.

In this chapter, we are going to use OpenCV and **You Only Look Once (YOLO)** as the deep learning architecture for vehicle detection. Due to this, we'll learn about the state-of-the-art image detection algorithm known as YOLO. YOLO can view an image and draw bounding boxes over objects it perceives as belonging to pre-identified classes.

We will also use a pre-trained network and apply transfer learning to create our final model. YOLO is a state-of-the-art, real-time object detection system. It achieved a **mean average precision (mAP)** of 78.6% on the **Visual Object Classes (VOC)** in 2007 and a **mAP** of 48.1% on the **COCO test data**.

The YOLO architecture applies just a single neural network to an image. Then, the network divides the image into parts. After that, the network predicts the bounding boxes and probabilities for each region. The prediction is completed by weighting the predicted probabilities for the bounding boxes.

There are other models for object detection applications available, such as R-CNN and Fast R-CNN. R-CNN is slow and expensive. Fast R-CNN is faster and more efficient than R-CNN. These algorithms use selective search to distinguish between regions in an image. However, YOLO trains on complete images and optimizes the detected output directly.

In this chapter, we will cover the following topics:

- What makes YOLO different?
- The YOLO loss function
- The YOLO architecture
- Implementation of YOLO object detection

Let's get started!

What makes YOLO different?

In this chapter, we will be using version 3 of the YOLO object detection algorithm, which further improves upon the old version of YOLO in terms of both speed and accuracy. Let's see how YOLO is different from other object detection networks:

- YOLO looks at the whole image during the testing process, so the prediction of YOLO is informed by the global context of the image.
- In general, networks such as R-CNN require thousands of networks to predict a single image, but in the case of YOLO, only one network is required to look into the image and make predictions.
- Due to the use of a single neural network, YOLO is 1,000x faster than other object detection networks (<https://pjreddie.com/darknet/yolo/>).
- YOLO treats detection as a regression problem.
- YOLO is extremely fast and accurate.

YOLO works as follows:

1. YOLO takes the input image and divides it into a grid of SxS. Every grid cell predicts one entity.
2. YOLO applies image classification and localization to each grid.
3. The grid cells are responsible for detecting the object if part of the object falls within a grid cell.
4. All of the grid cells predict bounding boxes with a respective confidence score.

In the next section, we will learn about the YOLO loss function.

The YOLO loss function

The YOLO loss function is calculated in the following steps:

1. First, we find the bounding boxes with the highest **intersection over union (IoU)** and with the correct bounding boxes.
2. Then, we calculate the confidence loss, which means the probability of the object being present inside a given bounding box.
3. Next, we calculate the classification loss, which indicates the present class of objects within the bounding box.
4. Finally, we calculate the coordinate loss for matching the detected boxes.

In summary, the total loss function is as follows:

$$YOLOLossFunction = CoordinateLoss + ClassificationLoss + ConfidenceLoss$$

In the next section, we will learn about the YOLO architecture.

The YOLO architecture

The YOLO architecture is inspired by the image classification model created by GoogLeNet. The YOLO network consists of 24 convolutional layers, followed by two fully connected layers. It also has alternating 1×1 convolutional layers, which reduce the feature spaces from preceding layers.

The convolution layers that are used in YOLO are from the pre-trained model of the ImageNet task, sampled at half the resolution (244x244), and then double the resolution. YOLO uses leaky ReLU for all the layers and a linear activation function for the final layers.

The following diagram shows the model architecture of YOLO:

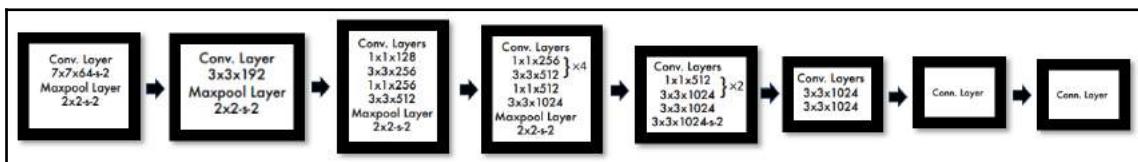


Fig 11.2: YOLO architecture



The following is a link to the official YOLO website: <https://pjreddie.com/darknet/yolo/>.

In the next section, we will learn about the different types of YOLO.

Fast YOLO

Fast YOLO is – as the name suggests – a faster version of YOLO. Fast YOLO uses nine convolutional layers and fewer filters than YOLO. The training and testing parameters are the same in both models. The output of Fast YOLO is $7 \times 7 \times 30$ tensors.

YOLO v2

YOLO v2 (also known as YOLO9000) increased YOLO's original input size from 224×224 to 448×448 . It was observed that this increase in size resulted in an improved mAP. YOLO v2 also uses batch normalization, which leads to a significant improvement in the accuracy of the model. It also resulted in an improvement in the detection of small objects, which was achieved by dividing the entire image using a 13×13 grid. In order to obtain good priors (anchors) for the model, YOLO v2 runs k-means clustering on the bounding box scale. YOLO v2 also uses five anchor boxes, as shown in the following image:

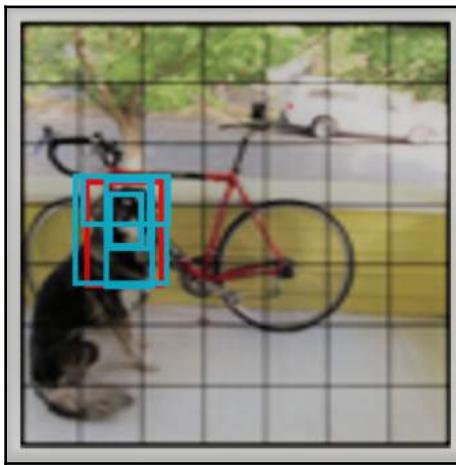


Fig 11.3: Anchor boxes

In the preceding image, the boxes in blue are anchor boxes, while the box in red is the ground truth box for the object.

YOLOv2 uses the Darknet architecture for object classification and has 19 convolution layers, five max-pooling layers, and a softmax layer.

YOLO v3

YOLO v3 is the most popular model of YOLO. It uses nine anchor boxes. YOLO V3 uses logistic regression for prediction instead of Softmax (which is used in YOLO v2). YOLO v3 also uses the Darknet-53 network for feature extraction, which consists of 53 convolutional layers.

In the next section, we will implement YOLO for object detection in both image and video.

Implementation of YOLO object detection

Now, let's explore how to implement YOLO v3 with Python. We will be using an implementation of YOLO v3 that has been trained on the COCO dataset.

The COCO dataset contains over 1.5 million object instances within 80 different object categories. We will use a pre-trained model that has been trained on the COCO dataset and explore its capabilities. Realistically, it would take many hours of training, even after using a high-end GPU, to achieve a reasonable model that can predict the required classes with good accuracy. Therefore, we will download the weights of the pre-trained network. This network is hugely complex, and the actual H5 file for the weights is over 200 MB in size.

Common objects in content (COCO) is a large-scale object detection, segmentation, and captioning dataset. The official website for COCO is <http://cocodataset.org/#home>.

COCO has several features:

- Object segmentation
- Recognition in context
- Superpixel stuff segmentation
- 330K images (>200K labeled)
- 1.5 million object instances
- 80 object categories
- 91 stuff categories
- 5 captions per image
- 250,000 people with keypoints



In the following sections, we will implement the data pipeline with YOLO.

Importing the libraries

The first step is to import the libraries.

We will import the numpy, openCV, and YOLO libraries for implementation purposes:

```
import os
import time
import cv2
import numpy as np
from model.yolo_model import YOLO
```

In the next section, we will write code for the image function so that we can resize the image, as per the architecture.

Processing the image function

Next, we will write a process_image function. This function will reduce or expand the image, depending on its original size.

Here, we want to transform the image as per the input for the YOLO model:

```
def process_image(img):
    """Resize, reduce and expand image.

    # Argument:
        img: original image.

    # Returns
        image_org: ndarray(64, 64, 3), processed image.
    """
    image_org = cv2.resize(img, (416, 416),
                          interpolation=cv2.INTER_CUBIC)
    image_org = np.array(image_org, dtype='float32')
    image_org /= 255.
    image_org = np.expand_dims(image_org, axis=0)

    return image_org
```

In the next section, we will write a class function so that we can access the classes from a text file.

The get class function

The class function will help us grab classes from the `classes` text file. We can find the `coco_classes.txt` file in the `data` folder in the project, in the section, *Importing YOLO*.

The COCO dataset contains almost 80 classes. It can detect them as follows:

```
def get_classes(file):
    """Get classes name.

    # Argument:
        file: classes name for database.

    # Returns
        class_names: List, classes name.

    """
    with open(file) as f:
        name_of_class = f.readlines()
        name_of_class_names = [c.strip() for c in name_of_class]

    return name_of_class
```

In the next section, we will write code for the `box` function, which is useful for drawing boxes around the objects in images.

Draw box function

The `draw` function will draw a box inside the identified image and put text on the image as a label, as well as place a prediction percentage for the identified class.

We will use OpenCV techniques in this step:

```
def draw_box(image, image_boxes, image_scores, image_classes,
            image_all_classes):
    """Draw the boxes on the image.

    # Argument:
        image: original image.
        image_boxes: ndarray, boxes of objects.
        image_classes: ndarray, classes of objects.
        image_scores: ndarray, scores of objects.
        image_all_classes: all classes name.

    """

```

```
for box, score, cl in zip(image_boxes, image_scores, image_classes):
    x, y, w, h = box

    image_top = max(0, np.floor(x + 0.5).astype(int))
    image_left = max(0, np.floor(y + 0.5).astype(int))
    image_right = min(image.shape[1], np.floor(x + w + 0.5).astype(int))
    image_bottom = min(image.shape[0], np.floor(y + h + 0.5).astype(int))

    cv2.rectangle(image, (image_top, image_left), (image_right,
    image_bottom), (255, 0, 0), 2)
    cv2.putText(image, '{0} {1:.2f}'.format(image_all_classes[cl], score),
    (image_top, image_left - 6),
    cv2.FONT_HERSHEY_SIMPLEX,
    0.6, (0, 0, 255), 1,
    cv2.LINE_AA)

    print('class: {0}, score: {1:.2f}'.format(image_all_classes[cl],
    score))
    print('box coordinate x,y,w,h: {0}'.format(box))

print()
```

In the next section, we will write some code for the predict function of YOLO.

Detect image function

The detect image function takes the image and uses the YOLO3 network to predict the class of objects within the image using `yolo.predict`. The code for this is as follows:

```
def detect_image(image, yolo, all_classes):
    """Use yolo v3 to detect images.

    # Argument:
        image: original image.
        yolo: YOLO, yolo model.
        all_classes: all classes name.

    # Returns:
        image: processed image.
    """
    pimage = process_image(image)

    start = time.time()
    image_boxes, image_classes, image_scores = yolo.predict(pimage,
    image.shape)
    end = time.time()
```

```
print('time: {:.2f}s'.format(end - start))

if boxes is not None:
    draw_boxes(image, image_boxes, image_scores, image_classes,
image_all_classes)

return image
```

In the next section, we will write some code that will detect objects in videos.

Detect video function

If we want to track a person or vehicle in a video, we can use the following function:

```
def detect_video(video, yolo, all_classes):
    """Use yolo v3 to detect video.

    # Argument:
        video: video file.
        yolo: YOLO, yolo model.
        all_classes: all classes name.
    """
    video_path = os.path.join("videos", "test", video)
    camera = cv2.VideoCapture(video_path)
    cv2.namedWindow("detection", cv2.WINDOW_AUTOSIZE)

    # Prepare for saving the detected video
    sz = (int(camera.get(cv2.CAP_PROP_FRAME_WIDTH)),
          int(camera.get(cv2.CAP_PROP_FRAME_HEIGHT)))
    fourcc = cv2.VideoWriter_fourcc(*'mpeg')

    vout = cv2.VideoWriter()
    vout.open(os.path.join("videos", "res", video), fourcc, 20, sz, True)

    while True:
        res, frame = camera.read()

        if not res:
            break

        image = detect_image(frame, yolo, all_classes)
        cv2.imshow("detection", image)

        # Save the video frame by frame
        vout.write(image)
```

```
if cv2.waitKey(110) & 0xff == 27:  
    break  
  
vout.release()  
camera.release()
```

In the next section, we will import the YOLO model and start predicting the objects in images and videos.

Importing YOLO

In this section, we will create an instance of YOLO classes. Note that it may take a little time as the YOLO model needs to load up:

```
yolo = YOLO(0.6, 0.5)  
file = 'data/coco_classes.txt'  
all_classes = get_classes(file)
```

In the next section, we will test the implementation of the YOLO model by predicting objects in an image, as well as a video.

Detecting objects in images

In this section, we will predict the objects present in an image using YOLO. In the following code block we will start with importing the image:

```
f = 'image.jpg'  
path = 'images/'+f  
image = cv2.imread(path)
```

The input image looks as follows:



Fig 11.4: Input image

We will perform the prediction using `detect_image` method in the following code:

```
image = detect_image(image, yolo, all_classes)
cv2.imwrite('images/res/' + f, image)
```

This yields the following prediction:



Fig 11.5: Output image

Here, we can observe that it predicted a person with 100% accuracy and the motorbike with 100% accuracy as well. You can take different images and experiment with this yourself!

Detecting objects in videos

Detecting objects in videos may take time. We will import the video name `library.mp4` and perform prediction using `detect_video` method:

```
# # detect videos one at a time in videos/test folder
video = 'library.mp4'
detect_video(video, yolo, all_classes)
```

This yields the following prediction:

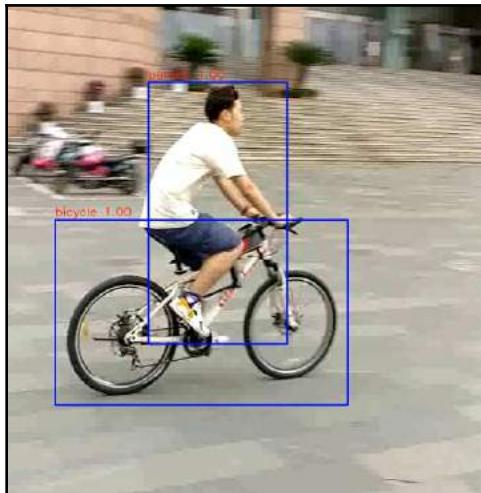


Fig 11.6: Prediction

Here, YOLO predicted a person and a bicycle with 100% confidence!

Summary

In this chapter, we learned about object detection, which is an important aspect of autonomous vehicles. We used a popular pre-trained model called YOLO. Then, we created a software pipeline to perform object predictions on both images and videos and also saw the high quality of YOLO's performance.

This is the last hands-on implementation in this book. In the next chapter, we will read about the next steps to follow in the field of self-driving cars. We will also learn about sensor fusion.

12

Next Steps

We have come to the end of this book. You're now well-versed on the topic of camera sensors, one of the most important sensors of an autonomous vehicle. The great news is that the era of autonomous vehicles has arrived, despite not being widely accepted by the general public. However, major automotive companies are now spending millions on the research and development of autonomous vehicles. Companies are actively exploring autonomous vehicle systems, as well as road-testing autonomous vehicle prototypes. Moreover, many autonomous vehicle systems such as automatic emergency braking, automatic cruise control, lane-departure warning, and automatic parking are already in place.

The primary goal of an autonomous vehicle is to reduce the number of road traffic accidents. Recently, companies have been observed experimenting with SDCs for delivering food, as well as taxi services. These commercial experiments will help bring confidence to the mainstream adoption of autonomous vehicles.

The major companies that are exploring autonomous driving technologies are BMW, Volvo, Tesla, Honda, Toyota, Volkswagen, Ford, General Motors, and Mercedes Benz. Some developments related to these companies are as follows:

- General Motors spent more than \$1 billion to acquire the **self-driving car (SDC)** start-up Cruise Autonomous.
- Toyota is investing around \$2.8 billion (along with two other partners – the Toyota Group companies Aisin and DENSO) into the Toyota Research Institute, specifically to advance developments in autonomous driving.
- In 2018, BMW opened a 248,000 square-foot facility in Munich, Germany for testing autonomous vehicles.

As we already know, autonomous vehicles are driven by various technologies, resulting in safe and efficient driving. The various components of SDCs, such as artificial intelligence, cameras, safety systems, networking infrastructure, LIDAR sensors, and RADAR sensors can be integrated seamlessly with the help of sensor fusion.

Sensor systems are evolving rapidly to meet the demands of these sensors. The three main important sensors of autonomous vehicles are RADAR, LIDAR, and cameras.

These sensors help us achieve the required five levels of autonomy, which we read about in Chapter 1, *The Foundations of Self-Driving Cars*. The five levels of autonomy are as follows:

1. **Level 0 – manual cars:** In Level 0 autonomy, both the steering and the speed of the car is controlled by the driver. Level 0 autonomy may include issuing a warning to the driver, but the car itself may not take any action.
2. **Level 1 – driver support:** In Level 1 autonomy, the car looks into the surrounding environment and monitors acceleration and braking. For example, if the vehicle gets too close to another vehicle, it will apply the brakes automatically, as shown in the following image:

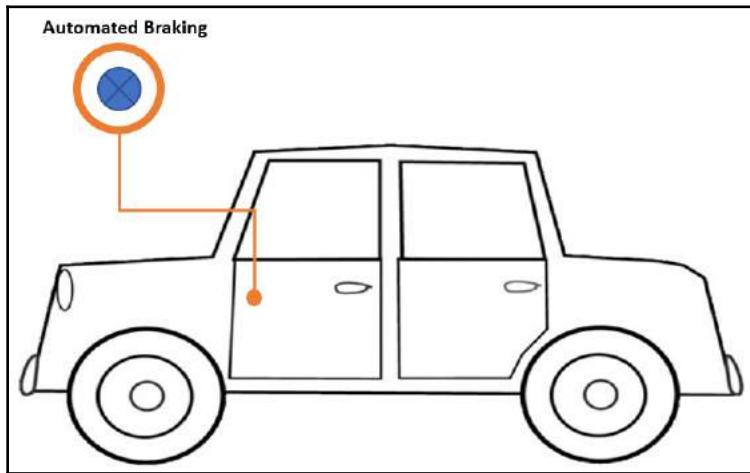


Fig 12.1: Level 1 – driver support

3. **Level 2 – partial automation:** In Level 2 autonomy, the vehicle will be partially automated. The vehicle can take over the steering or acceleration and will try to eliminate the driver from a few tasks. However, the driver is still required in the car to monitor critical safety functions and environmental factors. We can see these features in the following image:

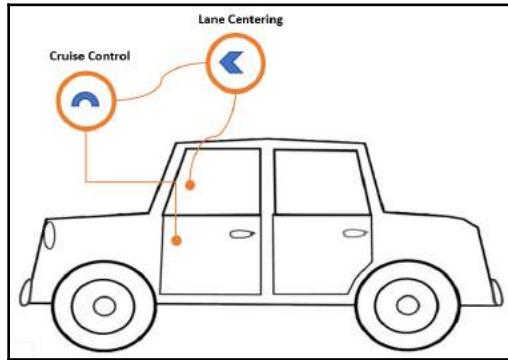


Fig 12.2: Level 2 – partial automation

4. **Level 3 – conditional automation:** From Level 3 autonomy and beyond, the vehicle itself controls all environmental monitoring (using sensors such as LIDAR). At this level, the vehicle can drive in autonomous mode in certain situations, but the driver must be ready to take over when the vehicle is in a situation that may exceed its capacity of comprehension. We can see these features in the following image:

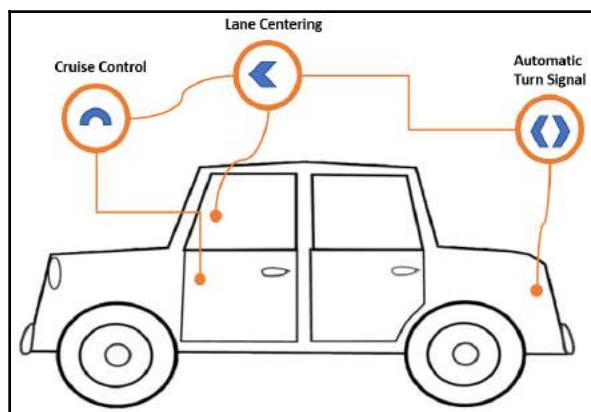


Fig 12.3: Level 3 – conditional automation

5. **Level 4 – high automation:** Level 4 autonomy is only one level below full automation. In this level of autonomy, the vehicle can control the steering, brakes, and acceleration of the car. It can monitor the vehicle, pedestrians, as well as the highway as a whole. Here, the vehicle can drive in autonomous mode for the majority of the time; however, a human is still required to take over in rare situations. These attributes can be seen in the following screenshot:

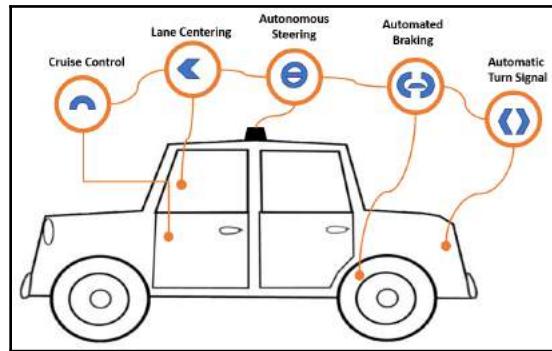


Fig 12.4: Level 4 – high automation

6. **Level 5 – complete automation:** With Level 5 autonomy, the vehicle is completely autonomous. No human driver is required and the vehicle controls all critical tasks, such as steering, braking, and working the pedals. It will even monitor the environment and identify unique driving conditions, such as traffic jams:

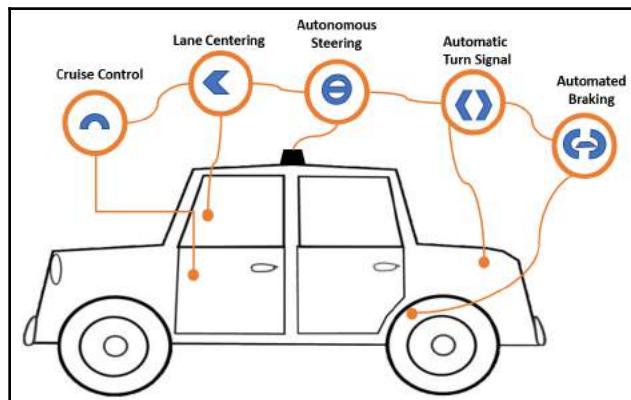


Fig 12.5: Level 5 – complete automation

This book has taught you about the various AI and deep learning concepts that are applied to camera images and can be used in other sensors too.

In the rest of this chapter, we are going to learn more about the five major steps toward realizing SDCs. We looked at these briefly in Chapter 1, *The Foundations of Self-Driving Cars*.

Here is a recap of these five major steps:

- **Computer vision**, which acts as the eyes of the SDC and allows the vehicle to interpret the world around it.
- **Sensor fusion**, which is a way of integrating data from multiple sensors such as RADAR, LIDAR, and lasers to generate a deeper understanding of our surroundings.
- **Localization**, which places our SDC within the world around us and comprehends its location and space.
- **Path planning**, which takes this understanding of what the world looks like and our place within it to chart our course of travel.
- **Control**, which turns the steering wheel and controls the accelerator and brake.

We learned about computer vision in detail, which deals with images and videos. We also learned about deep learning, advanced computer vision techniques, and convolutional neural networks. Furthermore, we implemented real-time projects such as traffic sign detection, object detection using **You Only Look Once (YOLO)**, and semantic segmentation using both E-Net and behavioral cloning using deep learning.

In this chapter, we will cover the following topics:

- SDC sensors
- Introduction to sensor fusion
- Kalman filter

Let's get started!

SDC sensors

Autonomous cars consist of many sensors. The camera is one of the sensors, and there are others such sensors, such as LIDAR, RADAR, ultrasonic sensors, and odometers.

The following image shows the various sensors that are used in an SDC:

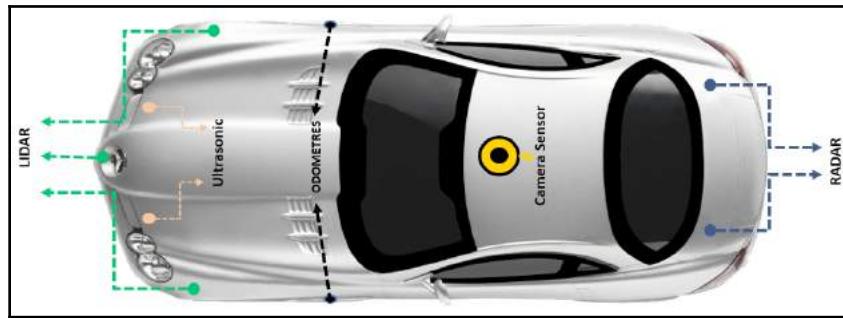


Fig 12.7: SDCs sensors

Knowledge of perceptrons is not enough for a SDC. We must also learn about sensor fusion, localization, path planning, and control.

Sensor fusion is one of the crucial steps when it comes to creating autonomous vehicles. Generally, an autonomous vehicle uses an extensive number of sensors that help it recognize its environment and locate itself.

We will briefly discuss the sensors used in autonomous vehicles in the following sections.

Camera

We have already learned about cameras in this book, which serves as the car's vision. Cameras are used to help the car understand the environment with the help of **artificial intelligence (AI)**. With cameras in place, the car can classify roads, pedestrians, traffic signs, and so on.

RADAR

Radio Detection and Ranging (RADAR) emits radio waves that detect nearby objects. As we discussed in Chapter 1, *The Foundations of Self-Driving Cars*, RADAR has been used in the field of autonomous vehicles for many years. RADAR help cars avoid collisions by detecting vehicles in the car's blind spots. RADAR also performs significantly when detecting moving objects. RADAR uses the Doppler effect to measure the distance between objects, as well as their positioning. The Doppler effect measures the change in waves when an object moves closer or further away. You can read more about the Doppler effect at https://en.wikipedia.org/wiki/Doppler_effect.



RADAR cannot categorize an object, but it is good at detecting its speed and position.

Ultrasonic sensors

In general, ultrasonic sensors are used for estimating the position of static vehicles, such as parked vehicles. They are cheaper than LIDAR and RADAR but only have a range of detection of a few meters.

Odometric sensors

Odometric sensors are a type of camera that help the vehicle estimate its speed by sensing the displacement of the wheels of the car.

LIDAR

The **light detection and ranging (LIDAR)** sensor uses infrared sensors to determine the vehicle's distance from an object. LIDAR has a rotating system that sends reflecting **electromagnetic (EM)** waves and calculates the time taken for them to come back to the LIDAR. LIDAR generates **point clouds**, which are sets of points that describe an object or surface in the environment around the sensor.



The LIDAR sensor can also classify objects as it generates 2 million cloud points per second, which generates the 3D shapes of the objects.

Introduction to sensor fusion

The sensors that are used in autonomous vehicles have various advantages and disadvantages. The main aim of **sensor fusion** is to utilize the various strengths of each sensor used in the vehicle so that the car can understand the environment more precisely.



Camera sensors are great tools for detecting roads, traffic signs, and other objects on the road. LIDAR estimates the position of the vehicle accurately. RADAR estimates the speed of the running vehicle accurately.

Kalman filter

One of the most popular sensor fusion algorithms is the Kalman filter. It is used to merge the data from various autonomous vehicle sensors. The Kalman filter was invented in 1960 by Rudolph Kalman. It is used to track navigation signals, as well as phones and satellites.



The Kalman filter was used during the first manned mission to land on the moon (the **Apollo 11 Mission**) for communication between staff on Earth and the crew on the shuttle/rocket.

The main application of the Kalman filter is data fusion, which is used to estimate the state of a dynamic system in the present, past, and future. It can be used to monitor a moving pedestrian's location and velocity over time, and also to quantify their associated uncertainty. In general, the Kalman filter consists of two iterative steps:

- Predict
- Update

The state of a system is calculated using a Kalman filter and is denoted as x . This vector is composed of a position (p) and a velocity (v), while the measure of uncertainty is " P ".

This results in the following formula:

$$x = (p/v)$$

Kalman filtering, also known as **Linear Quadratic Estimation (LQE)**, is an algorithm that helps us obtain more accurate estimates from the sequence of measurements we've observed.

Now, we will summarize this chapter.

Summary

In this chapter, we learned about sensor fusion. Sensor fusion is the next step after collecting all of the sensor data. This book taught you about one of the most important types of sensors available: cameras. We also learned about deep learning networks that enable camera sensors to function, and are also useful for making predictions from the data generated by other types of sensors. Finally, we learned about Kalman filters.

The overall goal of this book was to introduce you to the field of SDCs and to help you prepare for a future in the industry.

Here is a quick summary of the chapters we covered in this book:

In Chapter 1, *The Foundations of Self-Driving Cars*, we addressed the complicated path of how SDCs are becoming a reality. We discovered that SDC technology has existed for decades. We learned how it has evolved and learned about advanced research on the topic as a result of modern computational power. We also learned about the advantages and disadvantages of SDCs, the challenges in creating them, as well as the levels of autonomy of an SDC.

In Chapter 2, *Dive Deep into Deep Neural Networks*, we covered the concept of deep learning in detail. This is the most interesting and important topic of this book as it is the first step toward creating AI for SDCs.

In Chapter 3, *Implementing a Deep Learning Model Using Keras*, we studied the basics of Keras and implemented a deep learning model using the Auto-Mpg dataset. We also learned why Keras is useful.

In Chapter 4, *Computer Vision for Self-Driving Cars*, we learned about the importance of computer vision and the challenges in computer vision, as well as color spaces, edge detection, and the different types of image transformation. We also saw lots of examples using the OpenCV library.

In Chapter 5, *Finding Road Markings Using OpenCV*, we found lane lines in an image and a video by using OpenCV techniques.

In Chapter 6, *Improving the Image Classifier with CNN*, we learned about convolution neural networks and different ways of tweaking convolutional neural networks. We also implemented a handwritten digit (MNIST dataset) recognition model using Keras.

In Chapter 7, *Road Sign Detection Using Deep Learning*, we created a traffic sign detector with 95% accuracy. This project was one of the important steps we took toward realizing autonomous vehicles. We created a model that reliably classified traffic signs and learned to identify their most appropriate features independently.

In Chapter 8, *The Principles and Foundations of Semantic Segmentation*, we learned about the importance of semantic segmentation in the field of SDCs. We also learned about semantic segmentation at a high level and gained an overview of a few popular deep learning architectures related to semantic segmentation.

In Chapter 9, *Implementation of Semantic Segmentation*, we learned about the techniques we can apply to semantic segmentation using OpenCV, deep learning, and the E-Net architecture. We used the pre-trained ENet model on the Cityscapes dataset and performed semantic segmentation on both images and video streams. There were 20 classes of different objects in the context of SDCs and road scene segmentation, which included vehicles, pedestrians, and buildings.

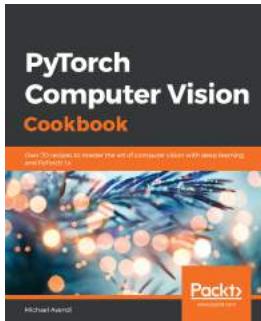
In Chapter 10, *Behavioral Cloning Using Deep Learning*, we focused on a useful technique called behavioral cloning. This chapter was relatively intense and combined all of the previous techniques we dealt with in this book. This included deep neural network feature extraction with convolutional neural networks, as well as continuous regression.

In Chapter 11, *Vehicle Detection Using OpenCV and Deep Learning*, we learned about object detection. We used one of the most popular pre-trained object detection models available (YOLO). Using this model, we created a software pipeline to perform object prediction on both images and videos.

I hope you enjoyed reading this book. I wish you great success in your work and contributions to the field of SDCs!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

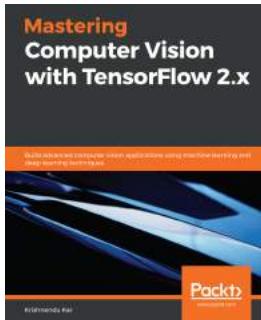


PyTorch Computer Vision Cookbook

Michael Avendi

ISBN: 978-1-83864-483-3

- Solve the trickiest of problems in computer vision by combining the power of deep learning and neural networks
- Leverage PyTorch 1.x capabilities to perform image classification, object detection, and more
- Train and deploy enterprise-grade, deep learning models for computer vision applications



Mastering Computer Vision with TensorFlow 2.x

Krishnendu Kar

ISBN: 978-1-83882-706-9

- Gain a fundamental understanding of advanced computer vision and neural network models in use today
- Cover tasks such as low-level vision, image classification, and object detection
- Develop deep learning models on cloud platforms and optimize them using TensorFlow Lite and the OpenVINO toolkit

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

3

3D layers
need for 176

A

accuracy 201
activation functions
about 40
hyperbolic tangent activation function (tanh) 42, 43
rectifier linear function (ReLU) 42
sigmoid function 41
threshold function 40, 41
Advanced Driving Assistance System (ADAS) 7, 220
affine transformation
about 127
reference link 127
AlexNet architecture 61
artificial intelligence (AI) 10, 294
Artificial Neural Networks (ANNs) 39
working 39
Auto-Mpg dataset 64, 65
autonomous driving
computer and hardware architecture 18, 19, 20, 21, 22, 23
fast internet 24
safe systems, building 17, 18
software programming 24
autonomous mode 250
axon 34

B

backpropagation 45
batch size 59
behavioral cloning, with deep learning

data collection 249, 250, 251, 252, 253, 254
data preparation 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267
model development 268, 269, 270, 271
simulator, evaluating 271, 272, 273, 274
behavioral cloning
with deep learning 249
blurring kernel 111

C

camera sensors 84
Canny edge detection
about 121, 122, 123, 124, 125, 126, 158, 159
reference link 126
color selection techniques
challenges 94
color space
HSV space 96, 97
manipulation 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107
RGB space 95, 96
techniques 94, 95
Comma Separated Value (CSV) 204
Common objects in content (COCO) 280
compound annual growth rate (CAGR) 15
computer format
images 174
computer vision, challenges
camera limitation 82
lighting 82
object variation 82, 83
scaling 82
viewpoints 81
computer vision
about 11, 80, 81
approaches, for SDC 26, 27
artificial eyes, versus human eyes 83, 84

challenges 81
for SDC vision 28
confusion matrix 199
control 12
convolution layer 177
convolution
about 107, 108, 109, 110
blurring 110, 111, 112, 113, 114, 115, 116,
117
sharpening 110, 111, 112, 113, 114, 115, 116,
117
working 178
convolutional neural network (CNNs)
about 61, 176
convolution layer 177
depth 181
fully connected layers 185
intuition 175
layers 176
need for 174
padding 181
Rectified Linear Units (ReLU) 184
stride 181, 182
zero-padding 183
cost function
of neural network 43, 44
cross-entropy function 44

D

data preparation 208, 209
data
loading 205, 206
dataset
image format 204, 205
overview 204
structure 204
deep learning model, building with Keras
about 64, 69, 70
Auto-Mpg dataset 64, 65
data, importing 66, 67
data, splitting 68
data, standardizing 68, 69
deep learning
approaches, for SDC 26, 27
Keras, used for 61, 62, 63, 64

used, for behavioral cloning 249
deep network, building with Keras
pipeline 63
deep networks 64
deep neural networks (DNNs) 31
DeepLabv3+
about 227, 228
Decoder 228
Encoder 228
Defense Advanced Research Projects Agency
(DARPA) 9, 14
deferred (symbolic) execution 60
dendrites 34
domain adaption 60
draw box function 282, 283
dropout 51

E

eager (imperative) execution 60
edge detection 118
Efficient Neural Network (E-Net)
about 229, 231, 233
architecture 229
bottleneck block 230
initial block 230
electromagnetic (EM) 295
epoch 59

F

F1 score 201
fast YOLO 279
feature map
about 178
creating 178, 179, 180
functional model 59, 60

G

Generic Neurotransmitter system
about 35
reference link 34
get class function 282
Global Positioning System (GPS) 84
gradient calculation 118

H

handwritten digit recognition
about 185
accuracy report 201
aim 186
confusion matrix 199
data transformation 190
data, loading 187, 188, 189
data, reshaping 190
model architecture, visualizing 197
model, building 192
model, compiling 192, 193
model, saving 197
model, training 194
output, one-hot encoding 191
problem 186
validation, versus test accuracy 196
validation, versus train loss 195

Hough transform

about 146, 147, 148, 149, 150, 151, 152, 153
applying 163, 164, 165, 166
reference link 150
HSV space 96, 97
hue saturation value (HSV) 95
hyperbolic tangent activation (tanh) function 42, 43
hyperparameters
about 46
model training-specific hyperparameters 47

I

image cropping 139
image exploration 207, 208
image features
examples 177
image format 204
image transformation
about 126
affine transformation 127
Hough transform 146, 147, 148, 149, 150, 151, 152, 153
image cropping 139, 140, 141, 142, 143
image dilating 139, 140, 141, 142, 143
image eroding 139, 140, 141, 142, 143
image rotation 128, 129, 130

image translation 130, 131, 132
image, resizing 133, 134
perspective transformation 135, 136, 137, 138, 139
projective transformation 128
regions of interest, masking 143, 144, 145, 146
types 127
images
blocks, building 84
converting, from RGB to grayscale 87, 88, 89
digital representation 84, 85, 86, 87
in computer format 173
road-marking detection 90
semantic segmentation 233, 234, 235, 236, 237, 238
intersection over union (IoU) 277

K

Kalman filter 296
Keras execution, types
deferred (symbolic) execution 60
eager (imperative) execution 60
Keras, deep learning model
compiling 69, 70
loading 76
new data, predicting 74, 75
performance, evaluating 75, 76
saving 76
training 70, 71, 72, 74
unseen data, predicting 74, 75
Keras, models
building 58
functional model 59, 60
sequential model 58, 59

Keras

advantages 56
URL 55
used, for deep learning 61, 62, 63, 64
versus TensorFlow 52, 53
working principle 56, 58
working with 55
kilometers (KM) 17

L

L1 regularization 51
L2 regularization 51
Laplacian edge detector 120, 121
levels of autonomy
 about 24
 complete automation 26
 conditional automation 25
 driver support 25
 high automation 26
 manual cars 25
 partial automation 25
LIDAR sensor
 for SDC vision 28
Light Detection and Ranging (LIDAR) 84, 295
Linear Quadratic Estimation (LQE) 296
linear regression
 neural networks for 245, 246, 247, 248
localization 11

M

machine learning (ML) 55
Mean Absolute Error (MAE) 69
mean average precision (mAP) 276
Mean Squared Error (MSE) 69, 246
Microsoft Cognitive Toolkit (CNTK) 55
mini-batch gradient descent 46
MNIST dataset
 about 185
 reference link 186
model accuracy 211, 212, 213, 214, 215
model training 210, 211
model training-specific hyperparameters
 about 47
 batch size 49
 learning rate 47, 48, 49
 number of epochs 49

N

National Highway Traffic Safety Administration (NHTSA) 10
network architecture-specific hyperparameters
 about 50
 activation functions, as hyperparameters 52

number of hidden layers 50
regularization 50
neural networks
 about 31, 32
 cost function 43, 44
 for linear regression 245, 246, 247, 248
 neuron 33, 34
 neuron 33, 34, 35, 36, 37, 38, 39

O

odometric sensors 295
Open Neural Network Exchange (ONNX) 62
Open-ended Neuro-Electronic Intelligent Robot Operating System (ONEIROS) 55
optimizers 44, 45

P

path planning 12
perceptron 35, 36, 37, 38, 39
Pomerleau
 reference link 14
precision 201
projective transformation 128
Pyramid Scene Parsing Network (PSPNet)
 about 226, 227
 architecture, reference link 227

R

radar 84
Radio Detection and Ranging (RADAR) 294
recall 201
Rectified Linear Units (ReLU) 59, 184, 223
rectifier linear function 42
red green blue (RGB)
 about 95
 image, converting to grayscale 87
RGB space 95, 96
road markings, detecting in images
 bitwise_and, applying 161, 162, 163
 Canny edge detection 158, 159
 detected road markings, optimizing 167, 168, 169
 Hough transform, applying 163, 164, 165, 166
 image, converting into grayscale 156, 157
 image, loading with OpenCV 155

image, smoothing 157, 158
region of interest, masking 159, 160, 161
road markings
detecting, in images 154, 155
detecting, in video 170, 171, 172
road-marking detection
about 90
with grayscale image 90, 91
with RGB image 92, 93, 94

S

scikit-learn library
reference link 68

SDC sensors
about 293, 294
camera 294
Light Detection and Ranging (LIDAR) 295
odometric sensors 295
Radio Detection and Ranging (RADAR) 294
ultrasonic sensors 295

SDC vision
computer vision, using 28
LIDAR sensor, using 28

SDC, core components
computer vision 11
control 11
localization 11
path planning 11
sensor fusion 11

SegNet
about 224, 225
architecture, reference link 225
decoder 226
encoder 225

self-driving car (SDC)
about 8, 9, 10, 12
advancements 13, 14, 15
advantages 12
approaches 10
computer vision, approaches 26, 27
current deployments, challenges 15, 16, 17
deep learning, approaches 26, 27
disadvantages 13
used, for processing sensory data 27

semantic segmentation, architecture

about 221, 222
DeepLabv3+ 227, 228
E-Net 229, 231
overview 222
PSPNet 226, 227
SegNet 224, 225
U-Net 223, 224

semantic segmentation
about 220, 232
architecture 221, 222
in images 233, 234, 235, 236, 237, 238
in videos 238, 239, 240, 241, 242

sensor fusion 11, 294, 295

sequential model 58, 59

sigmoid function 41

Sobel edge detector 119, 120

Society of Automotive Engineering's (SAE's) 24

softmax function 185

stochastic gradient descent 46

stride
example 182

T

TensorFlow
versus Keras 52, 53

threshold function 40, 41

training mode 250

type, regularization
dropout 51
L1 regularization 51
L2 regularization 51

U

U-Net
achievements 223
ultrasonic sensors 295

V

videos
semantic segmentation 238, 239, 240, 241, 242

Virginia Tech Transportation Institute (VTTI) 10

visual cortex
about 175
reference link 175

Visual Object Classes (VOC) 276

W

Waymo project 14

Y

YOLO architecture 278
YOLO loss function 277
YOLO object detection
 detect image function 283, 284
 detect video function 284, 285
 draw box function 282, 283
 get class function 282
 image function, processing 281
 implementing 280
 importing 285
 in images 285, 286, 287

 in videos 287
 libraries, importing 281

YOLO v2 279

YOLO v3 280

You Only Look Once (YOLO), types
 fast YOLO 279
 YOLO v2 279
 YOLO v3 280

You Only Look Once (YOLO)

 about 276
 classification 277
 reference link 277

Z

zero-padding 183