Hands-On Artificial Intelligence for Android

Understand Machine Learning and Unleash the Power of TensorFlow in Android Applications with Google ML Kit

VASCO CORREIA VELOSO

Hands-On Artificial Intelligence for Android

Understand Machine Learning and Unleash the Power of TensorFlow in Android Applications with Google ML Kit

Vasco Correia Veloso

www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

ISBN: 978-93-55510-242

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

To View Complete BPB Publications Catalogue Scan the QR Code:

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

www.bpbonline.com

Dedicated to

My wife Ana, for all her encouragement and support

About the Author

Vasco Correia Veloso has been developing software for over 20 years. He has developed software in many programming languages and systems: from the assembly, through C, C++, and Prolog to Java, Scala, and Kotlin. On big and small computers, using everything from floppy disks to SSDs, on-premises, and cloud, he's been there, done that, and used it. His academic background includes courses in applied mathematics and a degree in computer and software engineering.

He loves to learn how things work and share his learnings with others. He disassembled his grandfather's watch as a kid and grew up to (dis)assemble not only software but also hardware in the form of embedded systems.

The field of artificial intelligence has spiked his curiosity since HAL 9000 appeared on the silver screen.

Vasco brings teams together to produce well-crafted software. He enjoys sharing knowledge by teaching and learning and continues to design software and connected devices.

In his spare time, he takes up photography and has a keen interest in aviation. He has flown ultralight aircraft and believes that the focus necessary to fly and reach the destination while enjoying the scenery along the way also applies to software engineering.

About the Reviewers

- **Riccardo Mascarenhas** is a passionate and ambitious senior full-stack developer with 10+ years of experience. He likes to work on both the backend and frontend. He has worked extensively with Java, Kotlin, and almost all features of the Spring framework on the backend. His frontend skills are considered good while familiar with its technologies, like Angular and AngularJS, NgRx, Karma, Jest, Codecept.js, Cypress.io, and Jasmine. But his real knowledge lies on the backend side, where he likes many modern architectural approaches, like Domain-Driven Design, CQRS, and microservices architecture.

  He is a perfectionist who genuinely cares about proper testing and code quality - TDD is the way to go! Riccardo has experience with testing tools like Junit, Mockito, REST Assured, and Cucumber. He's worked with multiple databases in his careers, such as MongoDB, PostgreSQL, and Oracle. Besides that, he also is experienced with JDBC, and he knows his way in tools like Maven, Git, and Docker.

- **Sergei Miliaev** graduated from Voronezh State University's Physics Department in 2010 and defended his PhD thesis at the university in 2013. Sergei started working on computer vision and machine learning research in 2009 with the tasks of semantic image segmentation and natural scene text understanding with Moscow Lomonosov State University and Microsoft Research Cambridge. He joined VisionLabs as the Principal Researcher in 2013, and he leads the research teams that work on face recognition and vehicle recognition algorithms there. The face recognition solutions developed by VisionLabs using these algorithms have been demonstrating top performances on the ongoing Face Recognition Vendor Test organized by the National Institute of Standards and Technology. He has authored 20 papers, and his research interests are computer vision, machine learning, and deep learning.

Acknowledgement

It is said that it takes a village to raise a child. Likewise, it takes more than one person to write a book. I want to thank my friend and former colleague Manuel Lopes for being a guinea pig to ensure that the first examples were easy to follow. Riccardo Mascarenhas is a great sparring partner and was kind enough to review my usage of Kotlin in this text. I appreciate Sergei Miliaev's honest feedback on all things related to machine learning and artificial intelligence. In the last few years, Ramon Wieleman was—knowingly or otherwise—a significant influence in getting me out of my comfort zone. My alma mater ISEL in Lisbon, Portugal, will always have a special place in my heart for all the experiences I had during my time there.

Preface

Artificial intelligence and its machine learning discipline are two big topics nowadays. As technology progresses, previously inaccessible algorithms are no longer limited to research laboratories and can be used in everyday electronics. We show how developers can build full-fledged machine learning applications that run on their Android phones and recognize, apply effects on, and classify images, including people's faces.

We begin with a quick introduction to Android application development basics with examples in the Java language, using the Android Studio integrated development environment. The next step is to build an Android application in Kotlin because this is becoming the de facto language for Android application development. All further examples in the book are written in Kotlin.

We will provide a brief description of artificial intelligence and some fundaments of machine learning. We then go over the history of artificial intelligence and describe some of the most common machine learning algorithms, with examples.

We then show the development of Android applications, which can use machine learning to recognize images, identify faces, apply effects to photographs, and more. These applications are based on TensorFlow models—some of them are built and trained by the reader—and are converted to TensorFlow Lite for mobile applications.

Chapter 1 shows how to get started with Android Studio and build a simple Android application using Java. We also walk through the structure of an Android application project in Android Studio. Finally, we explore how the different classes and resources in the application work together, including the Android permissions model, and discuss how the traditional Model-View-Controller pattern can be applied to the development of Android applications.

Chapter 2 intends to show how to receive and process user events in an Android application. Android applications may serve a single purpose, but they usually do not have a single screen. Multiple screens can be implemented with multiple activities. We explain how different activities are built and how their communication mechanism works via intents.

Chapter 3 shows how to build the application that serves as the basis for examples in the following chapters. We also build a data persistence layer to be used by an

application and show how it is used to create an application database, supported by the SQLite engine and the Room ORM.

Chapter 4 looks at the term artificial intelligence and how it became an extensive study discipline from a historical and philosophical perspective. Starting from this high-level perspective, we isolate the field of machine learning from this extensive corpus of research to understand the challenges researchers aim to overcome.

Chapter 5 helps readers understand TensorFlow's fundamental concepts. It is one of the most popular frameworks that can be used to implement and train machine learning models. The chapter also covers TensorFlow's installation.

Chapter 6 is about training classification models for image recognition. We use two well-known datasets to create TensorFlow classification models for this purpose: one geared toward handwritten digits and another toward clothing.

Chapter 7 covers one of our image application building blocks—capturing live images using the device's camera. We show how the CameraX library can be used to streamline the entire process.

Chapter 8 shows how to use the machine learning models created in Chapter 6 in an Android application using TensorFlow Lite. An application is also created to analyze the images captured by the camera, building upon the techniques shown in chapter 7.

Chapter 9 covers the usage of the Google ML Kit library in an Android application to process the images captured by the phone camera and detect faces that may be present in the images.

Chapter 10 explores using a TensorFlow Lite model in an Android application to recognize the most prominent face in images previously captured and analyzed.

Chapter 11 shows how we can build an Android application for face recognition capable of extending its set of recognizable faces at runtime. It integrates the concept of persistently adding a face to the application, making it recognizable between application runs.

Chapter 12 demonstrates the possible usages of Generative Adversarial Networks (GANs). We show how GANs can be used to automatically generate realistic images. GANs can also process authentic images, which are presented as a GAN that applies a specific transformation effect to photographs.

Chapter 13 briefly touches the world of natural language processing, a task for which recurrent neural networks are particularly well suited. We showcase a model that attempts to describe the scenes depicted in images in plain English. A sample application is built to use this model to provide descriptions of photographs.

Code Bundle and Coloured Images

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

https://rebrand.ly/p47kbua

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Hands-On-Artificial-Intelligence-for-Android**. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

Errata

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

errata@bpbonline.com

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

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

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

Piracy

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

If you are interested in becoming an author

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

Reviews

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

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

Table of Contents

1. Building an Application with Android Studio and Java **1**

Introduction 1

Structure 1

Objectives 2

Setting up Android Studio 2

Creating a simple Android application 4

Android project structure 6

Running the application 9

Using an Android emulator *9*

Using a real device *12*

The Android application lifecycle 16

Android application resources 20

Resource qualifiers *21*

Identifying resources in application code *22*

Localization *23*

The Android application manifest 26

Application permissions *29*

Describing the features required by the application *31*

Applying the Model-View-Controller pattern 32

An alternative to MVC is the Model-View-Presenter *33*

Data binding allows a new pattern – Model-View-ViewModel *34*

Which pattern should be used? *35*

Conclusion 35

2. Event Handling and Intents in Android **37**

Introduction 37

Structure 37

Objectives 38

User events 38

Modifying the layout of an activity 38

Using the Android Studio layout editor 39

The ConstraintLayout 41

Placing a new component on the layout 42

Using string resources in components 44

Changing the component identifier 45

The XML version of the button 45

Registering to receive events 46

Reacting to the event 47

Adding an activity to our application 48

Intents in Android applications 53

Creating an intent 54

Showing our about activity with an intent 54

Returning a value from the target activity 56

Using Intents across Android applications 63

Intent filters in the application manifest 64

Starting another application with an implicit intent 65

Conclusion 68

3. Building Our Base Application with Kotlin and SQLite 69

Introduction 69

Structure 69

Objectives 70

A first glance at the Kotlin language 70

Fundamentals of the Kotlin language 72

Packages 72

Code organization 73

Visibility 73

Data types 73

Variables and properties 74

Type inference 76

Functions 77

Lambda expressions ... 79
Nullability ... 82
Classes and objects ... 84
Using a SQLite database in our application ... 87
Implementing SQLite in Android ... 87
Testing SQLite-based databases ... 92
Consequences of using SQLite directly ... 93
Understanding object-relational mapping ... 93
Working with a database with Room ORM ... 94
Getting started ... 94
Defining the data entities ... 96
Creating Data Access Objects (DAO) ... 97
Creating the database ... 101
Using database views ... 103
Object references are not supported ... 104
Converting object references to database types ... 104
Using database migrations ... 106
Running queries outside of the main thread ... 108
Testing Room-based databases ... 109
Conclusion ... 112

4. An Overview of Artificial Intelligence and Machine Learning ... 113
Introduction ... 113
Structure ... 113
Objectives ... 113
The past and the future of artificial intelligence ... 114
Depictions of artificial intelligence in literature ... 114
Artificial intelligence is not a new idea ... 116
Thinking about intelligent systems ... 119
Defining machine learning ... 120
Linear regression algorithm ... 123
Clustering methods ... 124

Neural networks *125*

Deep neural networks *126*

Opportunities to improve machine learning 127

Improving performance and accuracy *127*

Working toward explainability *129*

Avoiding bias *129*

Thinking about security *130*

Conclusion 131

5. Introduction to TensorFlow 133

Introduction 133

Structure 133

Objectives 134

Installing TensorFlow 134

Preparing Python *135*

Creating virtual environments for TensorFlow *137*

Installing TensorFlow in the virtual environments *138*

Verifying the TensorFlow version installed *139*

Tensors 140

Creating tensors with the constant() function *142*

Creating tensors with generated data *143*

Operations with tensors *145*

Variables 151

Graphs 152

Simple model training 154

Loading and preparing the dataset *156*

Training a linear regression model with one feature *158*

Training a linear regression model with all features *163*

Using a deep neural network for regression *164*

Conclusion 168

6. Training a Model for Image Recognition with TensorFlow 169
Introduction 169
Structure 169
Objectives 169
Recognizing handwritten digits 170
Preparing and loading the MNIST dataset 170
Building the model 173
Saving the model 176
Testing the saved model with an image loaded from the disk 176
Recognizing simple clothing items 179
Preparing and loading the Fashion-MNIST dataset 179
Building the model 180
Finding a better model 182
Evaluating both models with a realistic image 186
Conclusion 188

7. Android Camera Image Capture with CameraX 189
Introduction 189
Structure 189
Objectives 190
Ways of working with cameras on Android 190
Intents 190
Specialized camera APIs 190
Getting started with the CameraX API 192
Requesting permissions to use the device's cameras 194
Setting up the camera preview 196
Taking a picture 201
Configuring CameraX for image capture 201
Saving the captured image as a JPEG file 202
Adding a trigger button 203
Displaying the captured image 205
Conclusion 207

8. Using the Image Recognition Model in an Android Application 209

Introduction 209

Structure 209

Objectives 210

Fundamentals of TensorFlow Lite 210

Converting TensorFlow models into TensorFlow Lite 211

Training an existing TensorFlow Lite model 212

Setting up image analysis in the Android application 214

Cropping captured images 217

Converting captured YUV images to bitmaps 219

Using TensorFlow Lite in the application 222

Creating modules for TensorFlow Lite APIs 222

Working with a converted model 224

Working with a trained existing TensorFlow Lite model 231

Generating code for working with a model with metadata 235

Running TensorFlow Lite on dedicated hardware 236

Graphical Processing Units 236

More than just GPUs 238

Faster performance is not guaranteed 238

Conclusion 239

9. Detecting Faces with the Google ML Kit 241

Structure 241

Objectives 242

Understanding the Google ML Kit 242

Looking at face detection with the Google ML Kit 243

Including Google ML Kit in our Android application 244

Enabling view binding 244

Configuring the project's dependencies 245

Adding metadata for the Google Play Services 246

Preparing the user interface 246

Configuring the CameraX use cases 248

Creating the image analysis class 248
Scanning images for faces in real-time 252
Configuring the minimum image resolution 252
Creating the face detector object 252
Analyzing an image 255
Using the face detection results 256
Conclusion 260

10. Verifying Faces in Android with TensorFlow Lite **261**
Introduction 261
Structure 261
Objectives 262
Understanding face recognition 262
Understanding normalization 264
Normalizing a ratio scale 264
Using the Euclidean norm for normalization 266
Looking at the FaceNet model 267
Converting Hiroki Taniai's implementation of FaceNet to TensorFlow Lite . 269
Working with the MobileFaceNets model 273
Using the Euclidean distance to identify a face 274
Incorporating the MobileFaceNets model in an Android application 275
Adjusting the user interface 275
Extracting embeddings from face images 277
Writing a new image analyzer for face verification 279
Displaying the distance between two face embeddings vectors 283
Obtaining face embeddings for testing 284
Conclusion 286

11. Registering Faces in the Application **287**
Introduction 287
Structure 287
Objectives 288

Building the identity store 288
Implementing a view model to interface with the identity store 291
Processing multiple faces from camera images 294
Designing the user interface 298
Adding new faces to the application 298
Recognizing faces 305
Listing the face detection results *306*
Putting everything together *308*
Running the Android face recognition application *312*
Conclusion 313

12. Image Processing with Generative Adversarial Networks 315
Introduction 315
Structure 315
Objectives 315
Understanding Generative Adversarial Networks 316
Training a simple GAN 318
Applying a special effect to photographs 326
Transforming images into anime-style pictures *326*
Converting the GAN to the TensorFlow Lite format *327*
Trying the GAN in the computer *328*
Taking anime-styled pictures in Android 330
Repurposing the image capture application *330*
Capturing an image into memory *331*
Transforming the captured image *332*
Saving the modified image in the device's gallery *337*
Putting it all together *338*
Conclusion 342

13. Describing Images with NLP 343
Introduction 343
Structure 343
Objectives 344

Understanding recurrent neural networks 344
Evolving into long short-term memory networks 345
Performing automatic image captioning 346
Understanding the Show and Tell model *347*
Converting the Show and Tell model to TensorFlow Lite *349*
Implementing automatic image captioning in Android 353
Repurposing the image capture application *353*
Incorporating the TensorFlow Lite model *353*
Implementing beam search *356*
Saving metadata in JPEG image files *361*
Putting it all together *364*
Conclusion 365

Index 367-374

Chapter 1
Building an Application with Android Studio and Java

Introduction

This chapter shows how to get started with Android Studio and build a straightforward Android application using Java. We explain how the application's different classes and resources work together, including the Android permissions model. Finally, we discuss how the traditional Model-View-Controller pattern can be applied to the development of Android applications.

Structure

We cover the following topics in this chapter:

- Setting up Android Studio
- Creating a simple Android application
- Android project structure
- Running the application
- The Android application lifecycle
- Android application resources
- The Android application manifest
- Applying the Model-View-Controller pattern

Objectives

By the end of the chapter, you have a sample application running both in an emulator and a real device. You are also able to reason about the basics of Android application development.

Setting up Android Studio

Welcome to this exciting journey, where we put in practice the most recent developments in artificial intelligence under the form of Android applications.

All journeys must have a starting point. We begin ours by familiarizing ourselves with the preferred development environment for building Android applications—the Android Studio IDE.

Delving into this IDE's details is not the purpose of this book, but we wish that those who are not so familiar with Android Studio can also follow the applications we build throughout this book. We provide as many references as possible so that readers of all backgrounds feel supported.

The very first step is to download the installation package. Open the browser and navigate to the Android Studio website at **https://developer.android.com/studio**.

The correct installation package should have been automatically detected for you. Download and use it as appropriate for your operating system. At the time of writing, packages were available for Windows, Linux, mac, and Chrome OS. All screenshots and examples in this book have been made on a Linux computer, so you may notice some small differences if you are using Windows or macOS. Do not worry; these differences in Android Studio are purely cosmetic, and functionality is identical in all platforms.

If you have questions specific to your environment, the Android Studio user guide contains an installation section with details for each of the aforementioned platforms. It is available at **https://developer.android.com/studio/install**.

Once presented with the configuration wizard, you may proceed with the standard installation. This installs the IDE itself, an emulator, the latest version of the Android SDK, and local copies of additional libraries to be used in the development of Android applications.

Figure 1.1 illustrates an example of what options are selected with this kind of installation:

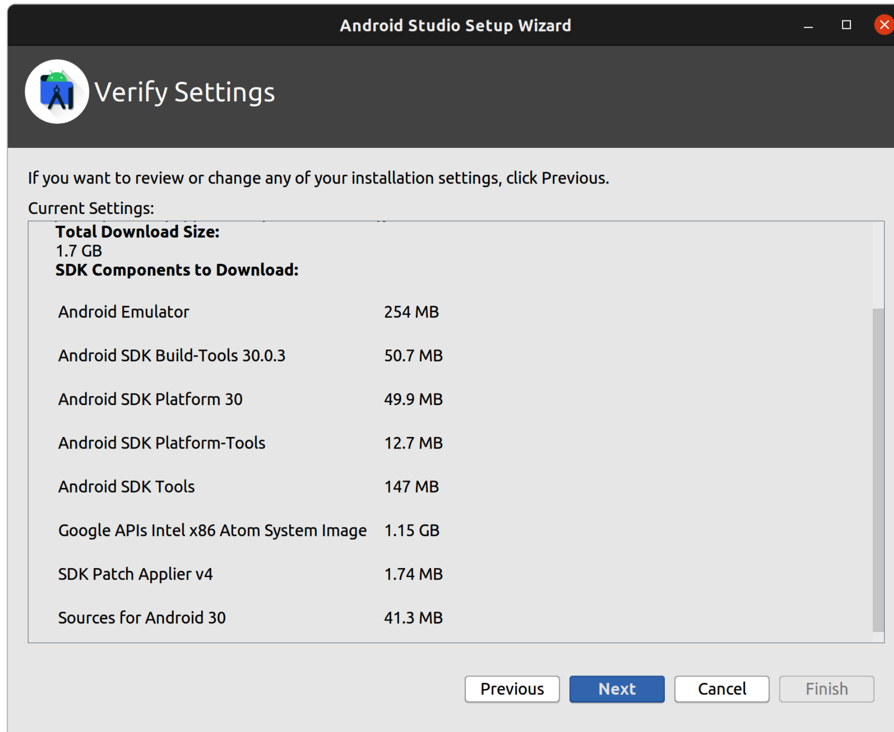

***Figure 1.1**: Standard Android Studio installation options*

We are presented with a window identical to the one in *figure 1.2* once the installation is complete and Android Studio is ready:

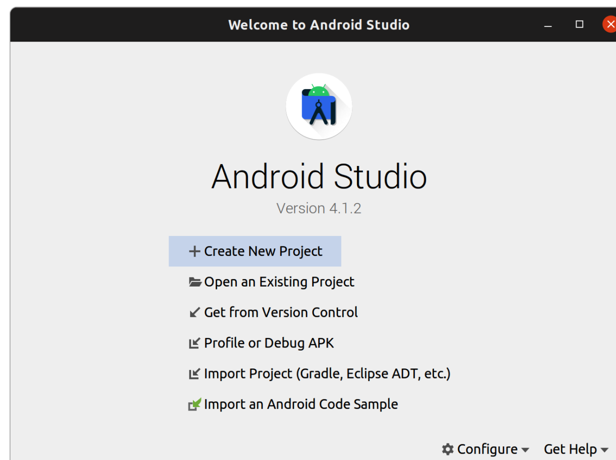

***Figure 1.2**: Android Studio welcome window*

Creating a simple Android application

Android Studio can create a simple Android application for us. This feature comes in handy because it frees us from memorizing all the folders and configuration files that must be in place before a proper Android application can be built.

Click on the "Create New Project" button in the welcome window to get started. You are then presented with a wizard that aims to gather a set of minimal information, which is dependent on the purpose and name of the application being created.

The first screen serves two purposes. On the one hand, it allows us to choose the target platform for our application. On the other hand, it offers us a set of predefined activities to get started. We want to build an Android phone application. We discuss activities later in this book, so let's choose the **`Empty Activity`** template for Phone and Tablet applications (see *figure 1.3*):

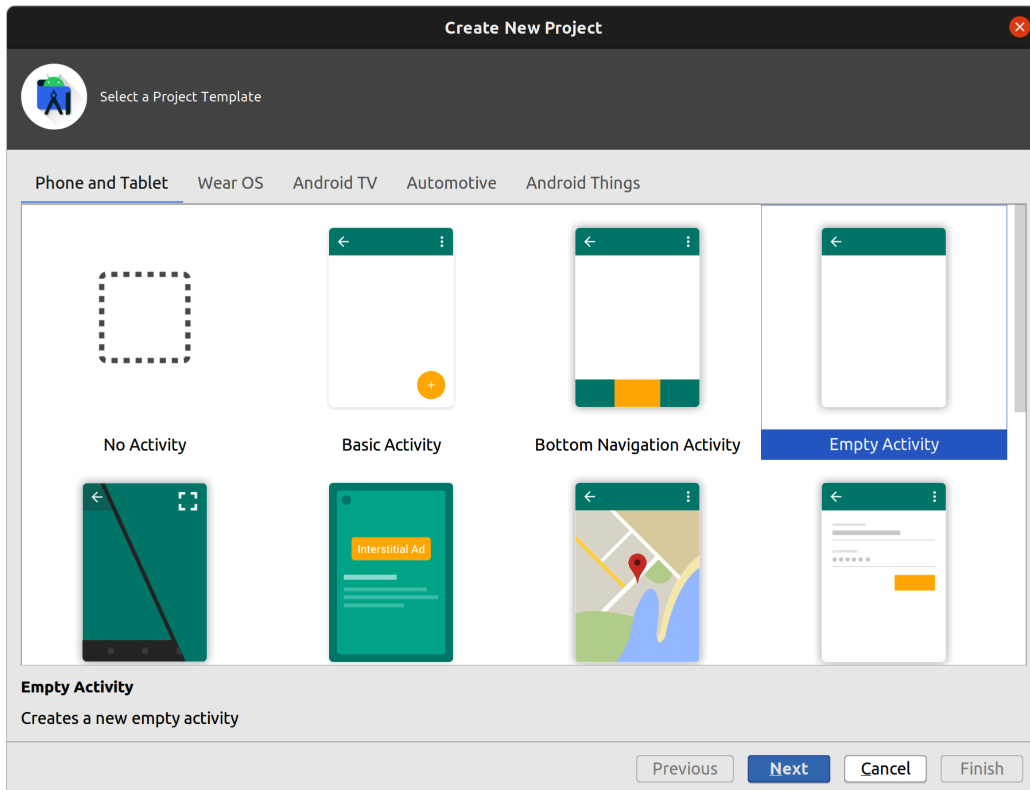

Figure 1.3*: New Project Wizard: The choice of an Activity*

Then, we need to decide on a package name and choose where our application is located so that Android Studio knows where and how to generate the project structure, configuration files, and minimal code.

Our first application aims to demonstrate the basics of Android applications, so let's call it a **`Demonstration Application`**.

You can then choose to customize the place where the project is generated instead of the default location.

As the title of this chapter implies, we are starting our work with Java. Let's select the **`Java`** language in the **`Language`** drop box.

Take a look at *figure 1.4* for an example of a possible project configuration:

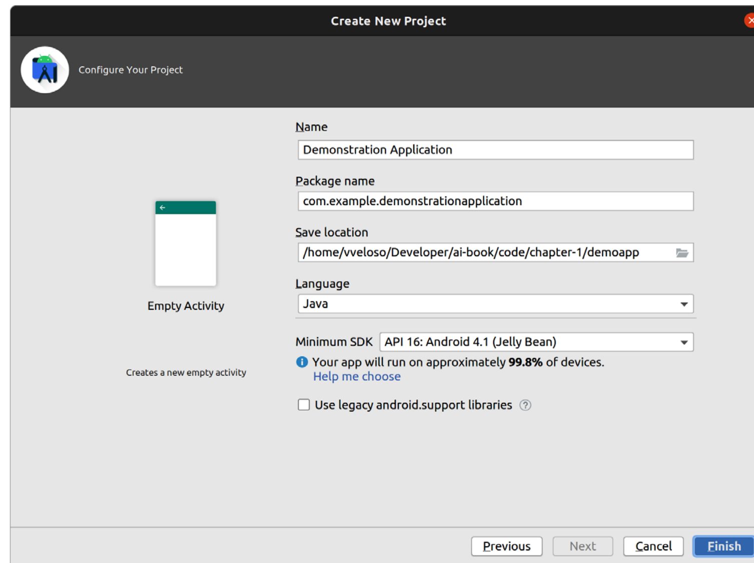

***Figure 1.4**: New Project Wizard: Application details*

Android Studio creates our template application after we click on the **`Finish`** button, loads the project to its main window, and proceeds to create an index of the code. The primary IDE window appears on screen with the project contents once everything is ready. Its most exciting part (for now) should look like *figure 1.5.*

Our application is now ready to run. We do not need to make any changes. Android Studio has generated an application on the lines of Hello World for us.

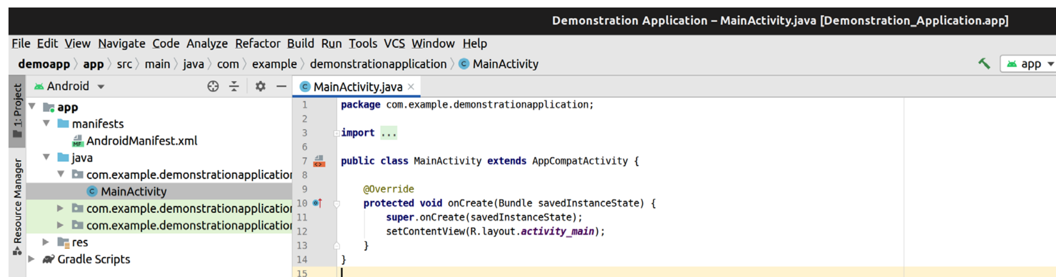

***Figure 1.5**: Android Studio window after creating the project*

Android project structure

As with all software development projects, Android applications have a predefined project structure that must be respected. Part of this structure is imposed by the build tool in use, which, in the case of the project we generated, is Gradle. However, the set of platform-specific files defines the most important part. Some of these files are mandatory for all Android applications, while others become mandatory as we add features to the application.

Let's take a quick look at the project structure automatically generated for us by Android Studio. *Figure 1.6* provides an expanded view of the latter:

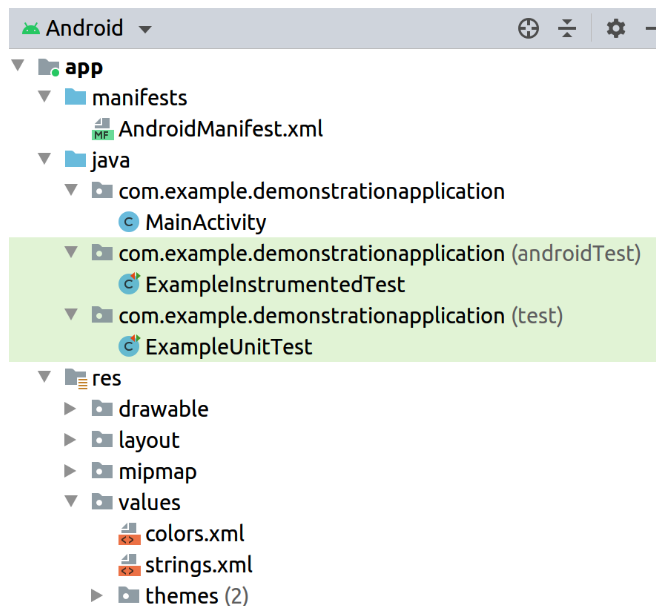

***Figure 1.6**: Expanded view of the demonstration application Android project structure*

The Android project tree view allows us to focus on the essential project items. From top to bottom, we can find the following types of files:

- The application manifest - This file contains critical information about the application and is used by various build and deployment tools, the Android operating system, and Google Play Store.
- The application package name - This is usually identical to the primary Java package name, which is the best practice. In our example, that would be **`com.example.demonstrationapplication`**. The application package name is important because it is used to uniquely identify the application in several places throughout the Android ecosystem, namely, the operating system, and Google Play Store.

- The different application components - These would be the application activities and the services, content providers, and broadcast providers it exposes. We define these later on.
- The application permissions - You must have installed Android applications that explicitly ask for your permission to access specific resources, such as the device location, file storage or SD card, or microphone. These requests were made by the operating system based on the requirements described in the permissions section of the application manifest.
- The application requirements, in terms of software and hardware. Constraints like the minimum Android API version that the application needs to run, or the need for a camera, are described in this section. Google Play Store uses it to determine which devices can run the application.
- The application Java code - It goes without saying that an application without code is useless. This section has the entire application code, including the different testing scenarios.
- The application resources - An Android application can use several kinds of resources. The Android Studio template generated quite a few of them:
    - **Drawables**: As the name implies, these resource files describe items that can be drawn on the screen. There are different kinds of drawables, depending on their purpose. Not all drawables are bitmaps (or image files); some are collections of vectors in an XML format.
    - **Layouts**: These are the layouts of the application's screens. In other words, they define the structure of the different sections of the User Interface (UI). These files describe the different UI elements' locations and their types, names, attributes, and placement constraints.
    - **Something called mipmaps**: These are a specific type of drawables. This directory must only contain the application launcher icons, and they are separated from the other drawables because we can provide versions of the images with different resolutions, selected automatically for the best experience in different devices.
    - **Value files**: The best practice concerning the use of any kind of values in software development is to avoid specifying them directly in the code (that practice is known as hardcoding). Instead, we should strive to store them as constant values. The advantage is that they get descriptive names this way and can be easily changed in a single location. The Android build platform allows us to use value files to store several kinds of values, like colors, numbers, identifiers, Boolean

values, and strings, among others. These values can be accessed from our code or other resources, depending on their type.

The project structure shown in *figure 1.6* is known as the Android project tree. It does not directly represent the actual project structure as it is stored in the file system. We need to use the Project tree to see the real file system tree.

Here, you will notice that a dropdown menu is present right above the project structure. The Android option is selected in this figure, so it is showing us the Android project tree. A tree similar to the illustration in *figure 1.7* is displayed if you select the **`Project`** option instead.

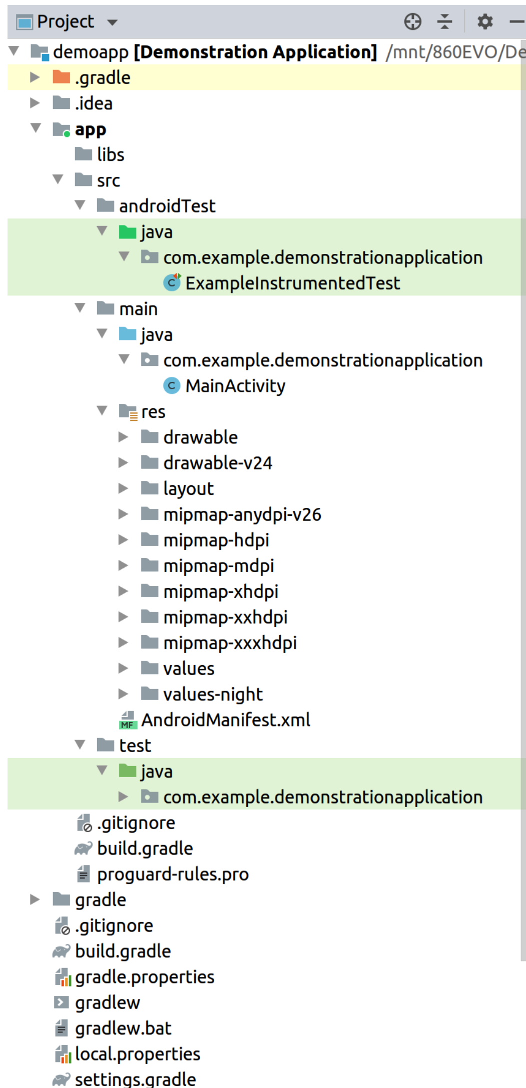

***Figure 1.7**: The Project tree showing the project structure as on disk*

It contains much more detail as far as the actual file system structure is concerned. It also allows access to files that are not shown in the Android view, like the raw Gradle configuration files.

Unfortunately, it also adds quite a bit of visual clutter, so the Android project tree is preferable for most work. Also, most of the Gradle configuration settings are accessible from the IDE, so you may not need to edit the files directly.

Of course, the choice of the project tree to use depends on technical factors as well as personal preferences. Some developers prefer to edit as much as possible inside files or customize their build environment. There are also other project trees available in Android Studio. Just choose the one you need from the `Project` dropdown box. You are free to explore and choose the way of working you prefer.

Tip: The launcher is the name given to the device home screen. The home screen is managed by a special type of application called a launcher because its main objective is to allow the user to start (launch) the different applications installed in the device. A launcher can use whichever method it wants to display the applications to the user, so the Android community has adopted the term “launcher” to refer both to this application and to its user interface.

Running the application

Applications, like greyhounds, are meant to run. We don’t build them so they can lie around doing nothing. So the question begs to be asked: how do we run an Android application?

Well, an application that targets a specific kind of device (a phone in our case) requires such a device to run. Generally speaking, we cannot run Android applications directly in our computer’s primary operating system.

So, there are two choices to run an Android application: it runs on one physical device like a phone or it runs on something called an emulator.

Using an Android emulator

We know that there are several kinds of Android phones in the market. They run different versions of Android, and they offer different combinations of hardware features. For example:

- Different phones have different screen sizes and resolutions.
- Some older phones may not have a GPS or a front camera.
- Most tablets run primarily in landscape (horizontal) orientation, whereas most phones are used in portrait (vertical) orientation.

While it is true that the majority of modern devices have similar hardware, there are many older devices in use which do not.

If our application can use a feature but it is not critical, we want to know if it can handle the feature's absence gracefully.

One way of handling this is to buy as many devices as possible and test the application in each one of them.

This approach becomes expensive and difficult to manage quickly. Some companies offer an application testing service on a fleet of physical devices, and this helps, but we don't want to ship our application to a third-party service every time we change a few lines of code, do we? These services can be useful at the end of the application development cycle, not at the beginning.

Emulators come to the rescue. An emulator pretends to be a physical device, but it really is virtual. Just like we can run different operating systems inside a virtual machine, we can run Android inside a virtual machine.

These virtual machines are different because they allow us to turn off and on many hardware options, and they can also emulate some hardware. For example, we can send GPS coordinates to our application without an actual GPS device.

These virtual machines allow us to run the Android operating system as if it were running on a physical device, so they are called **Android Virtual Devices** (**AVD**) in the tooling. Most developers also use the term "Android emulator" or only "emulator" when the context is clear.

How does Android Studio work with the emulator? The emulator is part of the Android SDK Tools, a different package. Fortunately, this package is installed by default when we install Android Studio. The IDE also has special bindings that allow us to see the available emulators from its user interface.

You find two dropdown boxes roughly around the middle of the Android Studio window, in the toolbar. This is illustrated in *figure 1.8:*

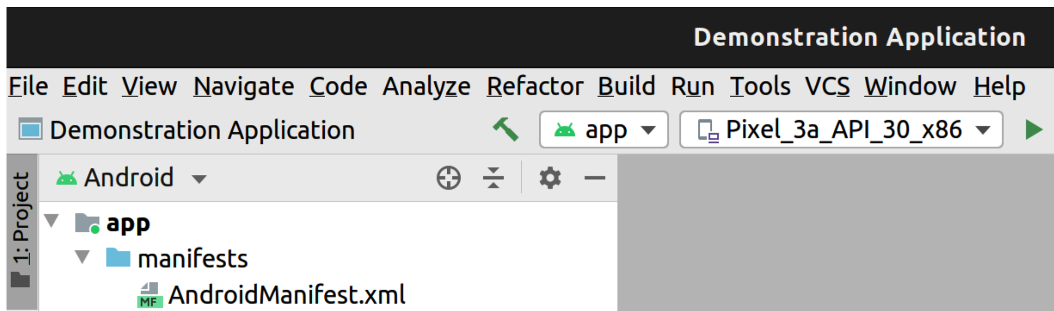

Figure 1.8: *Target Android device drop box*

The first dropdown box is called the **`Run Configurations`** dropdown and should have been populated automatically with an entry that refers to our application.

The second is the one we are interested in—the AVD dropdown. It should also have been populated already, but this time it contains the name of the emulator created during the Android Studio installation.

The emulator name looks strange, but it's logical. Let's dissect it. It comprises three sections separated by an underscore (**`_`**). In the case of the name shown in *figure 1.8:*

- The AVD skin, which mimics the looks of a physical device, is **`Pixel_3a`**. So, it resembles a Google Pixel 3 phone.
- The Android API level describes the software capabilities and directly relates to the Android operating system version. The emulator runs API level 30 in this case, which corresponds to Android 11.
- The image architecture identifies the underlying processor architecture. Our AVD has an x86 image architecture, which means it is meant to run on Intel-compatible systems.

We can change this predefined AVD name. We wanted to discuss the default name because it introduces some essential concepts relevant to the following section, where we create a different AVD.

At this point, we have already selected the application and the AVD. Now, we need to press the green "play" button to the right of the AVD dropdown box. This is the **`Run`** button that builds the application, automatically starts the emulator, and loads the application into it.

This process takes some time. The actual time depends on your computer and the complexity of the application being prepared.

The progress of all tasks is displayed at the bottom of the Android Studio window. You can find the name of the current task and a progress bar there.

Once the process is complete, you see a window similar to the one shown in *figure 1.9* on your screen.

***Figure 1.9**: Emulator running the generated application*

It presents a particular border designed to mimic the looks of an actual phone. You find the usual Android screen inside it. Our application will be running in the foreground.

Our application screen may not appear right away, especially if this is the first time the emulator is launched. We see this delay because the application must first be transferred to the emulator, which can take a few moments. Also, the emulator is not a real device, so it is, potentially, a lot slower.

The bottom line is to use as many emulators as necessary to test your applications in different Android versions and different screen sizes. Once you've confirmed that your application is behaving as you want, take the next step and test it on one or more real devices.

Using a real device

Granted, emulators are fun, but nothing beats the warm feeling one gets when their application runs on a real device for the first time!

Configuring the device

The first thing you need to do is prepare your device for connecting to the computer you're using to develop the application:

1. Enable the **`Developer Options`** in the device. Open the **`Settings`** app, go to the **`About Phone`** screen and tap on the **`Build Number`** seven times in a row (the **`About Phone`** screen may be inside the **`System`** section in some Android versions).
2. In your device, open the **`Settings`** app, select **`Developer Options`**, and enable **`USB Debugging`** (**`Developer Options`** may be inside the **`System`** section in some Android versions).

Configuring macOS or Chrome OS

You don't need to configure anything if your development environment runs macOS or Chrome OS.

Configuring (Ubuntu) Linux

If you're running Ubuntu Linux, you need to ensure that your user account belongs to the plugdev group and that the udev rules for Android devices are installed.

Run the **`id`** command to check if your user account is already in the plugdev group. You don't need to change anything if the output of the command already includes this group. Consider this example:

```
1. > id
2. uid=1000(vveloso) gid=1000(vveloso)
   groups=1000(vveloso),4(adm),24(cdrom),27(sudo),30(dip),
   46(plugdev),116(lpadmin),126(sambashare),
   136(kvm),140(libvirt),999(docker)
```

Otherwise, you need to run the following command to add yourself to the plugdev group:

```
1. sudo usermod -aG plugdev $LOGNAME
```

This change only takes effect at the next login, so it is a good idea to log out and log in again before continuing.

Finally, install the udev rules for Android devices with the following command to ensure that they are in place in Ubuntu Linux:

```
1. sudo apt-get install android-sdk-platform-tools-common
```

Configuring Windows

You need to download the correct drivers for your device so that Windows can recognize it as an Android development device when you plug it in.

You can download the Google driver package from **https://developer.android.com/studio/run/win-usb** if you're using a Google device.

For any other devices, you need the driver package published by the manufacturer. At the time of writing, a comprehensive list of these packages is available at **https://developer.android.com/studio/run/oem-usb**.

You need to download the correct package for your device and follow the manufacturer's instructions to install it.

If the manufacturer provides no installation instructions, the pages at the preceding addresses contain generic instructions that may also apply to your device. Unfortunately, the numerous makes and models of Android devices makes it difficult to provide instructions for each one.

Connecting the device to the computer

Now that the device and the computer are configured, it's time to use a USB cable to connect the device to the computer.

Some devices may ask you if you want to use the USB connection as **`Charging only`** or for **`File transfer`** (or similar wording). Ensure that you select **`File transfer`** for a proper connection to the computer.

Recent Android versions ask you for permission before communicating with the computer when the **`Debugging Mode`** is active. Ensure that you authorize the connection if this happens.

Running the application on the device

Once the device is connected to the computer and is properly recognized, a new entry appears in the same target Android device dropdown box we used when discussing the emulator (see *figure 1.8*).

The following figure shows how this dropdown box looks after the author's device is connected to the computer. Note that the virtual device also appears in the list.

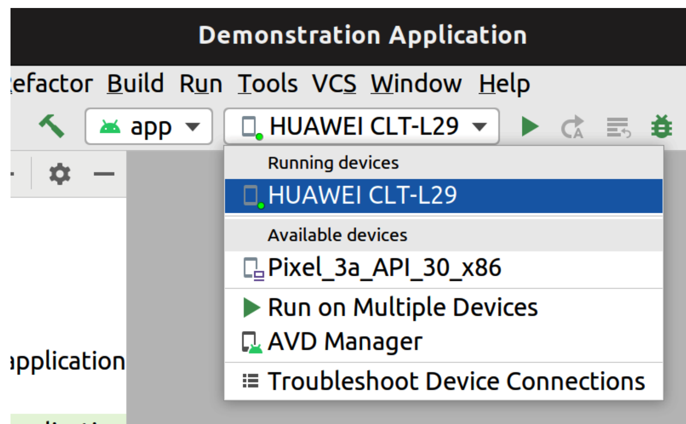

***Figure 1.10**: Target Android device dropdown with a physical device*

Now that the device is connected and selected in the dropdown list, we can run the application as we did before—by clicking on the green **Run** button.

Android Studio executes the same process: the application is built, downloaded to the device, installed, and finally executed. So, it is expected that the process takes some time. Remember that you can follow the steps of the process by looking at the Android Studio window's bottom. The current step and its progress appear there. Don't forget to unlock your phone to use the app.

Figure 1.11 shows what the application looks like on the author's phone. Note that the application background is black in this device, while it was white in the emulator, as shown in *figure 1.9*.

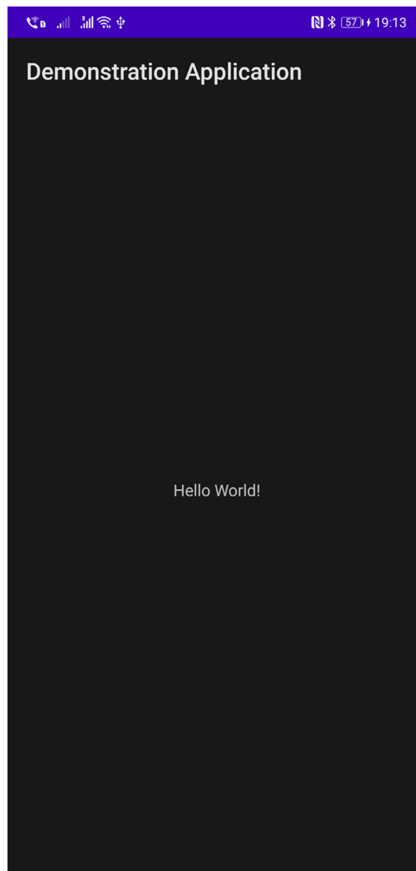

Figure 1.11*: Application running on the author's phone*

The dark colors result from the author's phone being configured to use dark mode, a setting that became available in Android 10. As the name implies, it asks applications to adopt a user interface based on dark colors. In the case of our demonstration application, this was supported automatically with the color scheme shown.

We have chosen to maintain this difference to stress that it is essential to test our application on different devices with different configurations.

Dark mode, for example, can have a dramatic impact on how an application is presented to the user. In some cases, the application's user interface may need improvements to look good in both light and dark modes.

The same applies to all other potential differences between devices.

The Android application lifecycle

We start applications as needed when we use them on our computers, and we close them once we are done using them. Quite simple.

Mobile devices like phones and tablets, watches, TVs, automobile dashboards, or **Internet of Things** (**IoT**) devices are usually resource-constrained, which means some shared resources are limited and must be used sparingly.

Examples of such resources are memory, and the battery power in the case of mobile devices.

Only a few applications can run simultaneously when memory is limited. Maybe not even one whole application can fit in the memory at once, so special care must be taken to avoid exhausting the device memory so that it can run such applications and continue to operate regularly.

Likewise, the amount of power a battery can provide until it discharges is limited. There is a direct relationship between the work the device is doing and the amount of power it needs to function. The more applications are running, the more power is consumed. So, specific battery power management techniques must be used to minimize power consumption and increase the amount of time between charges. One of the techniques used to achieve these goals is to close unnecessary or unused applications.

To solve these and other similar problems, application management in the Android operating system is more complicated than in our computers.

Android applications don't just run and stop running. They can be suspended, resumed, and invoked in different manners. We look at how applications can be invoked in the following chapters.

For now, let's take a look at the different lifecycle stages of an Android application.

We are in a bit of a chicken-and-egg situation. You see, most Android applications are built around the concept of activity. The lifecycle stages we are discussing apply to all the activities that belong to an application, so we need to discuss an activity's lifecycle to fully understand what an activity is. But we need to talk about activities to discuss their lifecycle.

We have chosen to wait until the next chapter to delve deeper into activities. At this point, we can think of an activity as an application screen. One activity is left behind and another is started every time the user navigates from one screen to another.

For example, the screen showing the list of emails would be implemented as an activity in an email application. The screen showing one email message would be a different activity.

Take a look at *figure 1.12*. It shows a state diagram representing the Android activity lifecycle.

An activity goes through a sequence of states from the moment it is launched until the moment it is completely removed from memory. Each of these states corresponds to a lifecycle event with well-defined responsibilities.

Launch

The launch stage is not an event per se. It corresponds to an action by the user, which results in the activity being launched by the operating system; for example, when the user taps on the application's icon on the home screen. The operating system has loaded the application into memory and is now ready to start working with the activity.

Create

The system has just finished creating the activity.

It is now time for the activity to prepare its essential internal state. The activity only goes through this stage once during its lifetime, so it is at this point that the activity runs its one-time preparation logic; for example, creating its user interface or restoring its state from a previous execution.

Start

The activity is now created, but it is still in the background.

It may take advantage of this state to initialize the user interface created in the Create stage. Note that the Start stage can be called from the Create stage and the Restart stage, so this is the best location to initialize the user interface state.

Both the Create and Start stages can deal with the user interface. The difference is that Start can be called multiple times but Create cannot. Create is responsible for

creating the user interface, while Start is responsible for setting up the user interface state (that is, what is being displayed).

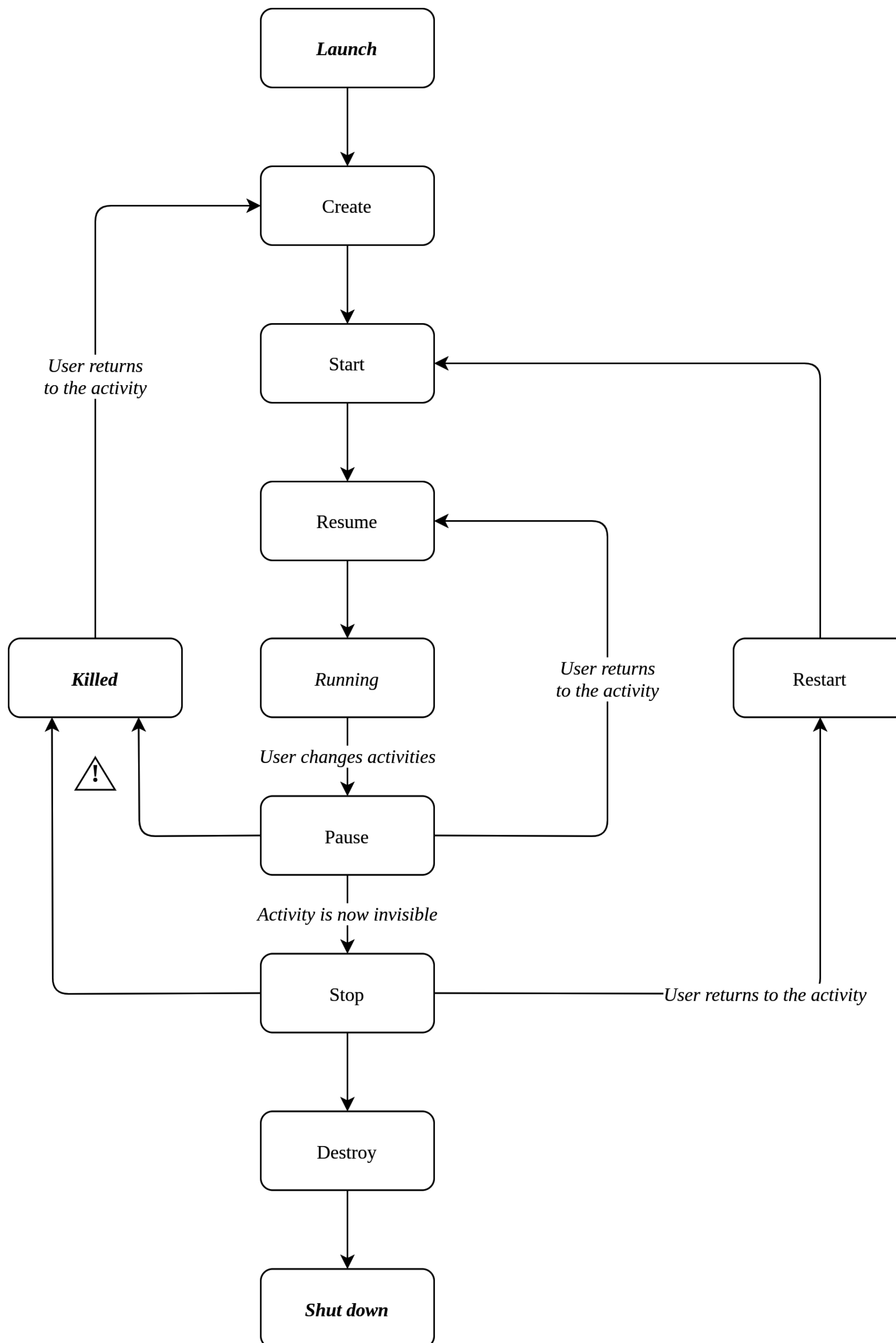

***Figure 1.12**: Android activity lifecycle*

Resume

Finally, the activity has been placed in the foreground. The user can now interact with it.

Now is the time for the activity to enable any features that only make sense when it is visible and in the foreground (interacting with the user), such as a live camera preview.

Running

The activity is in the foreground, fully visible on the screen, and the user interacts with it. It corresponds to our usual notion of an application running.

Pause

The activity is paused by the system when it is no longer in the foreground.

An activity left the foreground after something happened, indicating that the user will no longer interact with it immediately. For example, the user starts interacting with another activity or application, including taking a phone call, or the device's screen turns off.

This stage is the opposite of the Resume stage. Generally speaking, it should be used to release resources or suspend procedures started in the Resume stage.

The system places the activity back in the Resume stage if the user returns to it.

Stop

The system stops the activity when it is no longer visible to the user.

An activity stopped being visible when another activity covered the entire screen or when it finished running and is about to be terminated.

Resources that are not required when the activity is not visible should be released at this point. It is also the last appropriate stage to save the activity state.

Restart

The user has returned to the activity.

The system now brings the activity back to the foreground and back to the Start stage.

Destroy

The system is destroying the activity.

It can happen when the activity is finishing its regular operation and the application is terminating, or when the system needs to recreate the activity due to a configuration

change. For example, rotating the device is a configuration change that causes activities to be destroyed and recreated due to changes in the screen orientation.

All resources that were not released earlier should be released at this point.

Shut down

The activity is no longer in memory.

Killed

This stage also means that the activity is no longer in memory, but it is different from the shutdown stage because it is triggered when the operating system needs to free memory up.

Note that this removal of the activity from memory may occur after either the Pause or Stop stage. The activity has had the opportunity to release some of its resources and save most or all of its state.

There is a call for attention about this stage in *figure 1.12* because it breaks the usual lifecycle flow. Even when handled properly, it may cause some mismatch between the user's expectations and the actual activity state when it is recreated.

The next chapter discusses in detail the different ways how activities can store their state.

Android application resources

We presented an overview of the different application resources in the *Android project structure* section. In this section, we look at the details of resource usage in an Android application.

As we saw in *figure 1.6* and *figure 1.7* in the *Android project structure* section, our demonstration project was generated with different kinds of resources already provisioned.

Resources are placed in subdirectories inside the **`res/`** project directory. The most common resource types were already generated and defined in the preceding section.

Android application projects support a fixed set of resources, each with its purpose, and the resource subdirectory names reflect the type of resources that it contains.

The following subdirectory names and the corresponding resource types are supported inside the **`res/`** directory:

Directory	Resource type
`Animator`	Property animations defined as XML files. A property animation is a definition of how an object's field (property) value changes over time. It may or may not have visual consequences.
`anim`	Tween animations defined as XML files. Tween animations are view object animations altering a view object's position, size, rotation, or transparency. A view object is an object drawn on the screen.
`color`	State list of colors defined as XML files. Not the same as color values because a color state list defines the colors applied to a view object depending on its state, while a color value defines a single color as a constant value.
`drawable`	Resources that can be drawn on the user interface. Can be defined as bitmap files like PNG, JPG, or GIF or as XML files. The file format depends on the specific type of resource.
`mipmap`	Drawable files representing launcher icons.
`layout`	User interface layouts defined as XML files.
`menu`	Application menus defined as XML files.
`Raw`	Arbitrary files that are not processed in any way.
`values`	Simple values like strings, integers, and colors defined as XML files. You may mix resources in the same file, but the following file names are commonly defined by convention and to make it easier for you: • `strings.xml` for string values • `colors.xml` for color values • `arrays.xml` for resource arrays • `dimens.xml` for dimension values • `styles.xml` for styles
`xml`	Arbitrary XML files.
`font`	Font files. XML, TTF, TTC, or OTF file formats are supported.

***Table 1.1**: Supported resource directory names in Android projects*

The preceding table should give you a good idea of the different types of resources that can be used in Android applications.

Resource qualifiers

We would like to point out that the preceding table's directory names describe the default resources. The Android operating system can automatically choose the most appropriate resource for the current context.

For example, we can define different versions of the same layout resource for different screen sizes, resolutions, or screen orientations. Think about defining a friendly user interface, considering whether the screen is in portrait or landscape mode—the system automatically chooses the correct definition.

We can take advantage of this feature by applying qualifiers to the directory names.

In the case of our layout example, the directory structure could be as follows:

```
res/
  layout/
    our-screen.xml
  layout-land/
    our-screen.xml
```

The default portrait layout is located in the layout directory, and the alternative, that is, landscape layout, is in the layout-land directory. Both definitions have the same name, so the system knows that they are different definitions for the same resource.

Android Studio comes with some tools that free us from having to manually create these directories, but different tools are used for different resources. Drawable resources can be managed with the Image Asset Studio, layouts with the Activity Design, and strings with the Translations Editor.

We do not discuss all resource types, qualifiers, or editing tools in detail as this is out of the scope of this book. We discuss only the necessary resource types for our application in this and the following chapters. You may, of course, find further information in the Android documentation for developers at **https://developer.android.com/guide/topics/resources**.

Identifying resources in application code

So far, we have discussed the storage of resources in XML files. How are those resources accessed at different points of our application? How do we load a string, for example?

A resource identifier (ID) is always available for each resource. This identifier is composed of the resource type and name. The resource name can be the resource's file name or the value of the **`android:name`** XML attribute if the resource is a simple XML value.

We can use the special **`R`** class from the Java or Kotlin application code to refer to our application resources. This class is automatically generated and contains references to all the resources. The class members are generated following the composition of the resource identifiers.

Suppose we have the following string resources defined in their XML file:

```
1. <resources>
2.     <string name="app_name">Demonstration Application</string>
3. </resources>
```

We then take advantage of the **R** class in our code to access this resource of string type:

```
1. R.string.app_name
```

Another example is available in our demonstration application. The **MainActivity** class contains the following excerpt, where a layout is being attached to the content view (we discuss what this means in the next chapter, *Event Handling and Intents in Android*) in line 3:

```
1.     protected void onCreate(Bundle savedInstanceState) {
2.         super.onCreate(savedInstanceState);
3.         setContentView(R.layout.activity_main);
4.     }
```

The syntax is a little different if we need to refer to a resource from within another resource's XML file. It is something like the following:

```
1. @string/app_name
```

For example, we are referencing a string resource called **hello_world** in line 4 of the following XML fragment:

```
1.     <TextView
2.         android:layout_width="wrap_content"
3.         android:layout_height="wrap_content"
4.         android:text="@string/hello_world"
5.         app:layout_constraintBottom_toBottomOf="parent"
6.         app:layout_constraintLeft_toLeftOf="parent"
7.         app:layout_constraintRight_toRightOf="parent"
8.         app:layout_constraintTop_toTopOf="parent" />
```

Localization

When we develop an Android application, we must consider that our potential users do not all understand the same language.

We should then consider making our application accessible to users speaking other languages. The process of doing so is called localization because it is all about making our application suitable for a different locale. It goes beyond translation because it should also consider cultural differences.

We can accomplish this by taking advantage of the same resource qualifiers.

> **Tip: A locale defines several aspects that describe a user's experience related to the language. This combination is relevant because some details, like the date and time formats or the spelling, differ among regions that use the same language. In Android, for example, we use the qualifier en-US to represent English as used in the United States.**

One of the many resource qualifiers that can be used is the locale name, like **`fr`** for French, **`pt`** for Portuguese, and so on.

The language qualifier may be applied to drawable resources in addition to string resources.

So, when a specific user's device is set to a locale that is not the default language, the system selects the correct resources from all the variants in the application.

All we need to do is include those variants in the application's resources.

String resources

When we open the **`strings.xml`** file from the **`res/values`** directory, which by convention is an XML file containing only string resources, Android Studio gives us the opportunity to open the Translations Editor.

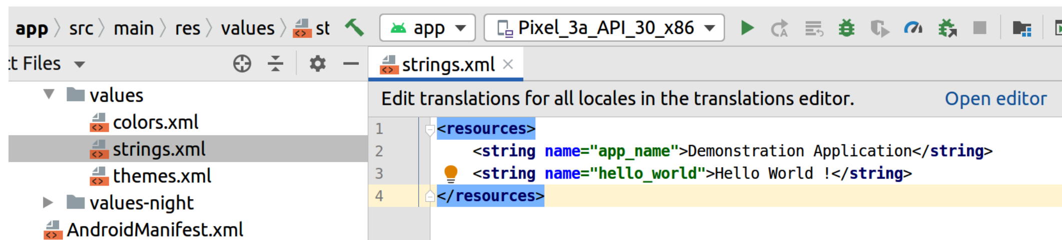

***Figure 1.13**: Opening the translations editor in Android Studio*

The user interface of the translations editor is quite simple. It consists of a grid layout with columns for the resource name, its file location, the default resource value, and the values for each locale.

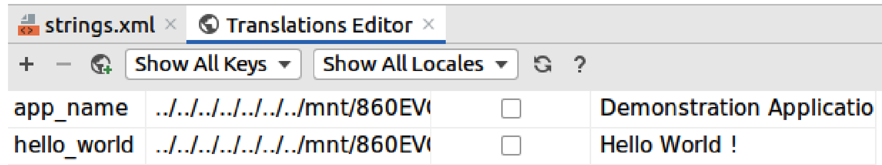

Figure 1.14*: The translations editor*

Naturally, there are no locales and no translations when the editor is opened for the first time.

We can add a locale by clicking on the icon highlighted in *figure 1.14*. A list of supported locales appears, so we can select the appropriate language. Further locales can be added by repeating this procedure.

Once locales have been added, they appear in the translations editor as new columns. They also appear as qualified resources in the Android project tree.

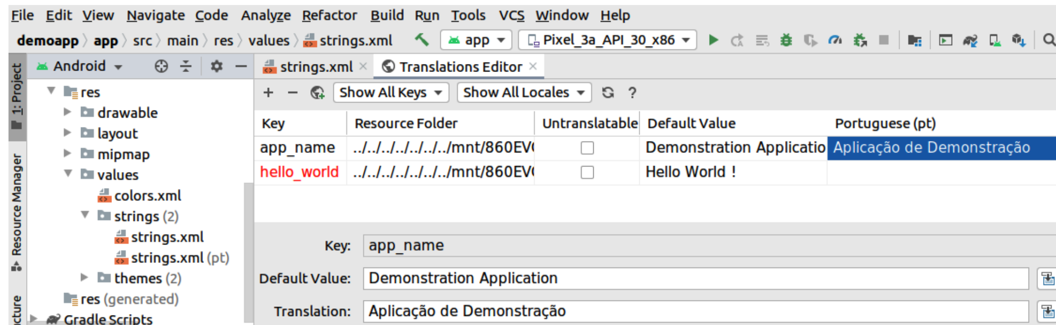

Figure 1.15*: The translations editor with one locale set*

The translation process is simple.

To insert a new translation, click on its cell and type the translation at the bottom of the screen. Resources that have not been fully translated yet have their names painted red to make it easier to find them. One of the two resources in *figure 1.15* is being translated, and the other has no translation yet.

We should check the corresponding checkbox in the **`Untranslatable`** column if a string resource is impossible to translate. This tells the linter that the absence of a translation is not a problem.

As soon as a locale is defined, the system chooses the best language at runtime and automatically uses its text.

Other types of resources

Unfortunately, there is no built-in support for localizing other types of resources.

This lack of support means that you need to create the appropriate directories within the **res/** directory hierarchy, including the language qualifier.

For example, if you have images that should be adjusted depending on the locale stored as drawables, you would need to create another drawable folder as follows:

```
res/
  drawable/
  drawable-pt/
```

You may ask, when does an image need to be localized? Some cultures assign different meanings to colors, for example, or some iconography may not be appropriate in some regions of the globe.

The kind of resources that should be localized depends on their contents and the regions of the world where you intend to distribute the application. It's always good to seek advice when you want to launch your application in regions you're not familiar with.

The Android application manifest

We have touched the subject of application manifest briefly earlier. In a few words, the application manifest aims to inform the system about the application's components, permission, and hardware requirements.

The application manifest is a file named **AndroidManifest.xml** placed in the manifests directory.

Let's get started by looking at the manifest automatically generated by Android Studio for our demonstration application.

```
1.  <?xml version="1.0" encoding="utf-8"?>
2. <manifest xmlns:android=”http://schemas.android.com/apk/res/
   android”
3.    package="com.example.demonstrationapplication">
4.
5.    <application
6.        android:icon="@mipmap/ic_launcher"
7.        android:label="@string/app_name"
8.        android:roundIcon="@mipmap/ic_launcher_round"
9.        android:theme="@style/Theme.DemonstrationApplication">
10.       <activity android:name=".MainActivity">
```

```
11.         <intent-filter>
12.           <action android:name="android.intent.action.MAIN" />
13.           <category android:name="android.intent.category.
                     LAUNCHER" />
14.         </intent-filter>
15.      </activity>
16.   </application>
17.
18. </manifest>
```

Lines 2 and 3 begin the manifest. Like most XML files, the applicable namespace is imported and, more importantly, the base application package is set. Usually, this package's name is the same as the base package for the application's Java or Kotlin code. It is essential to know that this package name becomes the application's unique identifier once it's compiled, so choose it carefully.

The actual application definition begins in line 5 with the application element.

Lines 6 to 9 define basic user interface settings like the application's theme, launcher icons, and label.

Activities are application components. As such, they must appear in the manifest; otherwise, the system does not know about them.

The main (and only) activity in our demonstration application is described in lines 10 to 15 in the **`activity`** element.

Note that the name of the activity is its Java class name. It needs to be fully qualified; however, it starts with a period, so the system automatically prefixes it with the application's package name. So, **`.MainActivity`** becomes **`com.example.demonstrationapplication.MainActivity`**.

We find the **`intent-filter`** element inside this **`activity`** element. We discuss intents in detail in the next chapter. For the time being, know that this definition allows the launcher to start our application by launching the **`MainActivity`** activity.

This manifest represents the bare minimum required to build and launch an Android application.

The build system configuration defines other aspects of the application, such as its version number and the minimum Android SDK version that it requires. These are set in the **`build.gradle`** file for the application and injected in the manifest at build time.

Here's an excerpt of this file as used by our demonstration application:

```
1.  plugins {
2.  id 'com.android.application'
3.  }
4.  android {
5.  compileSdkVersion 30
6.  buildToolsVersion "30.0.3"
7.
8.  defaultConfig {
9.     applicationId "com.example.demonstrationapplication"
10.    minSdkVersion 16
11.    targetSdkVersion 30
12.    versionCode 1
13.    versionName "1.0"
14.
15.    testInstrumentationRunner "androidx.test.runner.
         AndroidJUnitRunner"
16. }
17.
18. // (other build settings)
19. }
20. dependencies {
21. // (required libraries)
22. }
```

The **android** section in line 4 states that the application is compiled with version 30 of the Android SDK and that it targets version 30; however, it can run with a minimum version as low as 16. The latter version defines the lowest Android version that is compatible with your application. In this case, SDK version 16 corresponds to Android 4.1, code name Jelly Bean. If your application requires Android features that became available in later versions, you must update the minimum SDK version accordingly.

The same section states that the current version number is 1 and that the version name is 1.0. These must be increased every time you release a new application so that users see that an update is available.

You don't need to manually edit the build configuration file. These settings can be changed using the **`Module Settings`** dialog box in Android Studio if you prefer.

To open the **`Module Settings`**, select the app module in the tree and press the *F4* key, or right-click on it and select the **`Open Module Settings`** option from the pop-up menu.

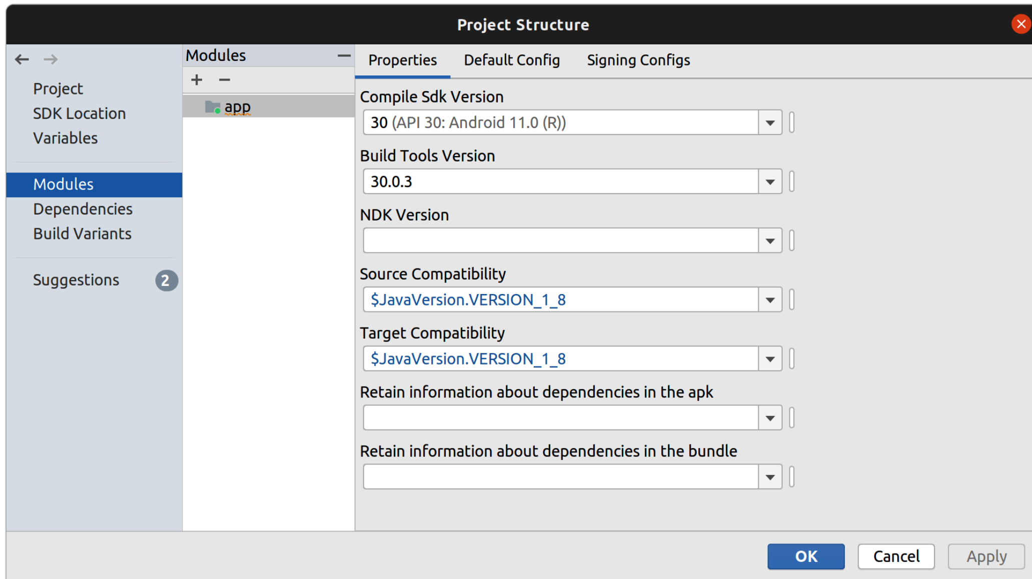

Figure 1.16: *Our demonstration application module settings*

We can include other elements to describe different requirements of the application. Let's look at the elements that we use in this book's applications.

Application permissions

By design, Android applications live in a sandbox. They have limited access to the system and the device's features.

Whenever an application exposes some functionality that requires a usually blocked feature by the sandbox, the application needs to ask the user for permission to use that feature.

Of course, applications should not request permissions that they do not need. Users are, thankfully, more aware of privacy issues. We must always ensure that the features we wish to implement require additional permissions to be granted.

Let's imagine that our application needs to establish a connection to the Internet for some reason. For example, it might need to update its machine learning model periodically.

By default, Android applications are not allowed to open sockets, so no network connections are allowed.

In order to overcome this limitation, we need to declare the permission request in the manifest file. We use the **users-permission** element to do so, as shown in line 5 in the following excerpt:

```
1.  <?xml version="1.0" encoding="utf-8"?>
2. <manifest xmlns:android=”http://schemas.android.com/apk/res/
   android”
3.     package="com.example.demonstrationapplication">
4.
5.     <uses-permission android:name="android.permission.INTERNET"
   />
6.
7.     <application>
8.         <!-- normal application configuration -->
9.     </application>
10. </manifest>
```

From now on, the system is aware that our application needs to make network connections.

This specific permission is classified as normal in Android. These permissions are granted automatically during installation. However, some other permissions are classified as dangerous. Think about permissions that give the application access to personal information, calls, text messages, external storage, and such.

These permissions must be declared in the manifest as well, but they are not granted during installation since Android 6.0 (API 23). If the application is installed in a device running this version or a more recent one, it must ask the user for permissions at runtime.

Asking for permissions at runtime should be done programmatically and only when the application needs the permission. For example, if the application needs camera access, it should only ask for permission when the user wants to activate the camera and not before that.

There is a specific API to ask for permissions at runtime, which we demonstrate in full in the next chapter. For now, it is essential to know that specific code needs to be written to check permissions and that it can take advantage of the following:

- The `ContextCompat` class to check if the permission has already been granted
- The `RequestPermission` contract, from the AndroidX library, to leverage permission requests already implemented in the system

Many permissions are available; you can find the full list at **https://developer.android.com/reference/android/Manifest.permission**. The classification assigned to each permission is also listed on this page, so we can know beforehand if the runtime permission process is required.

Describing the features required by the application

We discussed how our application could be allowed to use device and system features in the previous section. What if the feature we need is not present in the users' device?

Suppose the application cannot operate without the feature being present in the device, regardless of permissions. In that case, we want to tell the user that there is a compatibility problem and maybe not allow the application's installation.

Google Play can perform this filtering and inform the user when the application is not compatible with their device.

Our application must declare the hardware or software features it needs in its manifest for this mechanism to work. At the same time, it should state whether the feature is required, that is, whether the application can function when the feature is not present.

This declaration is made by using the **uses-feature** element in the manifest file. The following excerpt shows a possible declaration for an application that needs to use the device's camera:

```
1.  <?xml version="1.0" encoding="utf-8"?>
2. <manifest xmlns:android=”http://schemas.android.com/apk/res/
   android”
3.   package="com.example.demonstrationapplication">
4.
5.     <uses-feature
6.          android:name="android.hardware.camera"
7.          android:required="true" />
8.     <uses-feature
```

```
9.            android:name="android.hardware.camera.autofocus"
10.           android:required="true" />
11.
12.     <application>
13.         <!-- normal application configuration -->
14.     </application>
15. </manifest>
```

Note that the application is declaring that both features are required. This means Google Play does not allow installation of the application in devices without a camera or with a camera lacking an autofocus feature. If the application functions without a camera, the `android:required` attribute should be set to false.

Some permissions imply that the corresponding feature usage is required, so it is always a good idea to explicitly declare the features. This way, there are no surprises at installation or runtime.

The full list of features supported by this manifest element is documented at **https://developer.android.com/guide/topics/manifest/uses-feature-element**, along with their relationship with permissions.

Applying the Model-View-Controller pattern

Generally speaking, the job of the vast majority of applications is to gather and process some data, display that data to the user, and allow the user's actions to modify said data.

It is easy to identify three distinct concerns from this sentence:

- To model, store, and process data
- To present data to users and allow them to act on this data
- To effect changes on data based on user actions

A vastly accepted paradigm of software development is that different parts of the program should handle different concerns; the same unit of code must have one well-defined responsibility. This principle is known as the principle of separation of concerns.

For quite some time, software development's golden architectural pattern was the **Model-View-Controller** (**MVC**) pattern. This pattern allows for easier separation of concerns between the different parts of the application:

- The model handles the design of data in the application and related tasks
- The view is responsible for displaying the data to the user and allowing them to interact with the data
- The controller receives requests from the view and translates them into model updates

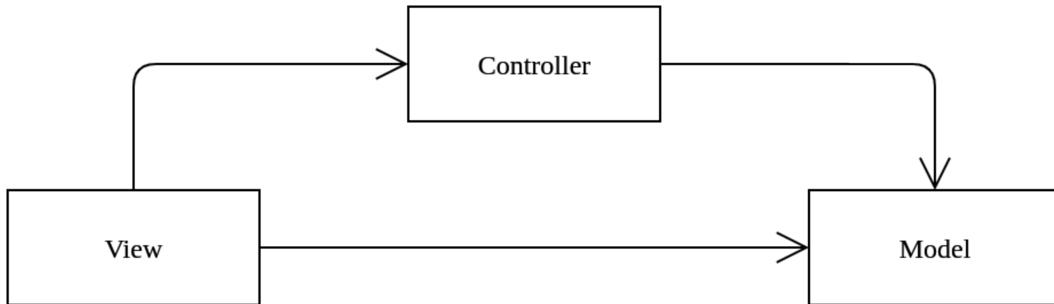

***Figure 1.17**: MVC pattern dependencies*

After being translated into the Android world, this pattern is often implemented with the view layout as the view, the activity as the controller, and the application's data model as the model.

The disadvantages of this application of the MVC pattern in Android applications are often:

- Business logic easily creeps into the controller because of its dependencies on both the view and model. The model may become anemic.
- The controller is difficult to test because it is tightly coupled to the Android API.

An alternative to MVC is the Model-View-Presenter

A slightly different pattern was introduced to facilitate testing and reduce coupling between the view and the controller—the **Model-View-Presenter** (**MVP**).

There are small but relevant differences between MVP and MVC:

- The model remains the same
- The view is now not only the view layout but also the activity

- The presenter replaces the controller and is responsible for updating the view and modifying the model

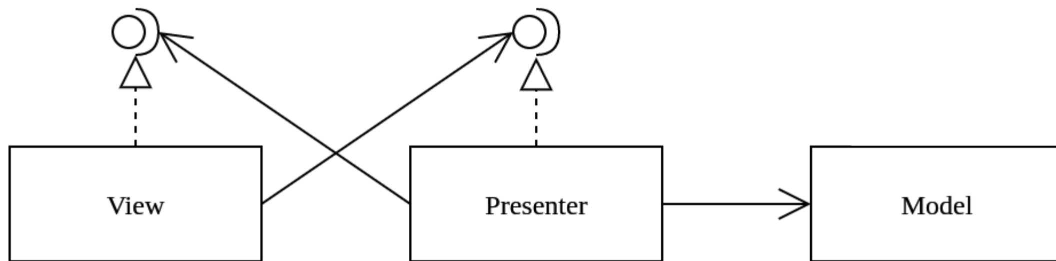

***Figure 1.18**: MVP pattern dependencies*

Testability is improved when using this pattern because the presenter is not tightly coupled to the Android API. Ideally, it should not have any relationship to it whatsoever.

Some disadvantages include:

- Two new interfaces need to be introduced to prevent a circular dependency between the view and the presenter
- The danger of having too much business logic in the presenter remains, and it can only be mitigated by developer discipline

Data binding allows a new pattern – Model-View-ViewModel

The introduction of the Data Binding Library for Android allows us to use a new pattern—**Model-View-ViewModel** (**MVVM**).

Again, the differences are small when it is compared to MVP:

- The model is still the same
- The ViewModel replaces the presenter and is responsible for supplying a mechanism through which components can be notified of model changes (events, or an observable) and invoke actions to update the model
- The view binds directly to the events exposed and accepted by the ViewModel

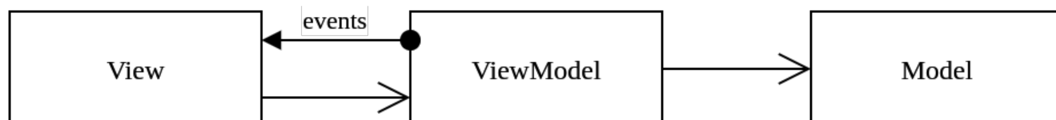

***Figure 1.19**: MVVM pattern dependencies*

Testability remains good because the ViewModel is not tightly coupled to the Android API. There is no need to implement additional interfaces because the ViewModel never needs to know who the consumer of its data events is.

An alternative to the Data Binding Library is a reactive programming model to transmit the model updates to the view.

Its disadvantages include:

- Complexity is added through the introduction of an additional library
- There is now the danger of allowing logic to appear in the view; again, only developer discipline can prevent this

Which pattern should be used?

It depends on the complexity of the application and personal taste.

Minimal applications with a simple model and little logic may get away with an MVC implementation. MVP or MVVM become better choices when models become non-trivial and more logic is required.

Note that the fundamental principle of both MVP and MVVM is the same—an intermediate entity isolates the view from the model. Only the mechanism by which events and commands are exchanged is different.

The choice ultimately boils down to personal preference.

Conclusion

Android Studio is a powerful tool for building and running an Android application. It allows us to browse the project's structure easily and identify and edit the different files and resources that compose the application.

Android applications have a well-defined lifecycle, whose primary purpose is to allow the operating system to optimize resource usage on the device (mainly battery and memory).

The way they are built lends itself to the application of well-defined patterns for segregation of concerns like the Model-View-Controller, the Model-View-Presenter, or the Model-View-ViewModel. It is up to the developer to choose the most appropriate pattern for the application at hand.

Android applications do not exist in isolation, so the next chapter shows how different activities can communicate. It also addresses the usage of services exposed by third-party applications for the benefit of our application.

CHAPTER 2

Event Handling and Intents in Android

Introduction

This chapter intends to show how to receive and process user events in an Android application. We cannot interact with our users without this mechanism. Android applications may also serve a single purpose, but they usually do not have a single screen. Multiple screens can be implemented with multiple activities. We can also use another application's features, which requires communication between the participating applications. We explain how the different activities are built and how this communication mechanism works via intents.

Structure

We cover the following topics in this chapter:

- User events
- Modifying the layout of an activity
- Registering to receive events
- Adding an activity to our application
- Intents in Android applications
- Using Intents across Android applications

Objectives

By the end of this chapter, you can build an Android application with multiple activities that respond to user interaction. You also understand what intents are and how they are used to communicate between activities, regardless of their respective application.

User events

Simply put, user events are how the user's interactions are communicated to the application.

The user may interact with the application in several different ways, including:

- Tapping on buttons
- Typing on an input box
- Choosing an option from a dropdown list
- Making a gesture on the screen

As covered in the previous chapter, Android screens are managed by activities. So, it is the activity's responsibility to register its interest in any events coming from its user interface elements.

Note: Although it is true that an activity manages each screen, there are different ways of implementing user experiences with different screens. An alternative and possibly more efficient way to do so involves using a single activity that displays different fragments as appropriate for the user journey. A fragment can be seen as a set of user interface elements grouped in a component that an activity can display. For the sake of simplicity, we keep using activities for now.

Modifying the layout of an activity

We have generated a new application for this chapter using the steps outlined in the previous chapter. As you know, the generated layout for the main activity is almost empty. It only contains a hello world user interface element, which is not very useful for us.

So, we want to add a new user interface element to this screen. Something that makes it clear to the user that they can interact with it.

Let's add a button to our main activity.

Using the Android Studio layout editor

Activity layouts can be modified by directly editing the XML file, but that is cumbersome and error-prone. It is much better to use the layout editor from Android Studio.

Open the project in Android Studio and locate the **`activity_main.xml`** file inside the **`res/layout`** directory in the Android project tree to open the layout editor to change our main activity's layout.

A new tab containing the layout editor should open.

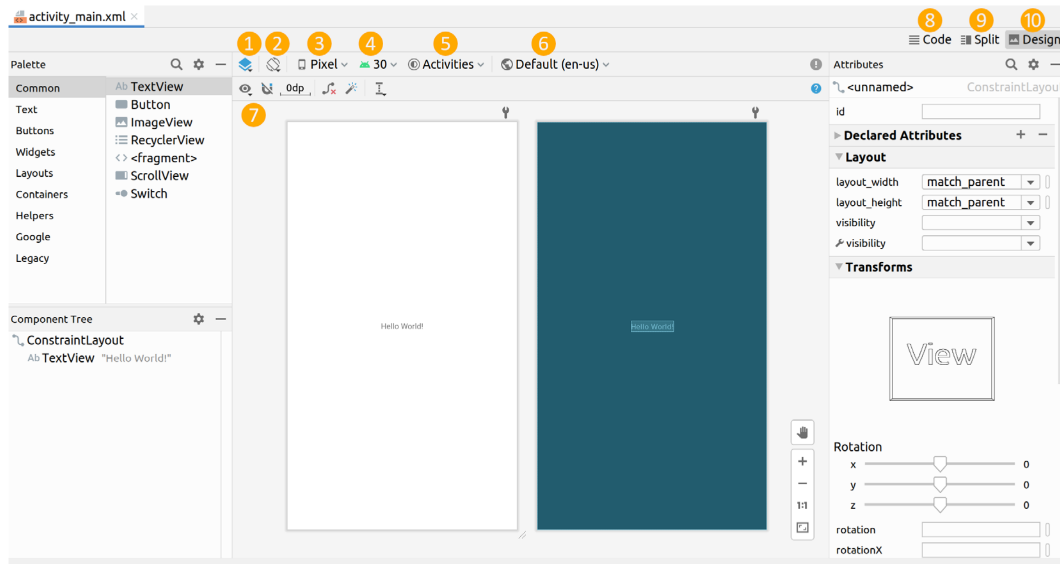

Figure 2.1: *The layout editor*

The layout editor is divided into four significant sections:

- The component palette and tree on the left-hand side. It lists the different user interface component types known to the layout editor.
- The layout design section is in the middle.
- The attributes editor on the right-hand side. It displays the available attributes for the selected component. There is no component selected in *figure 2.1*, so it displays the layout properties.
- The editing mode toolbar at the top. It allows us to switch between the three editing modes available.

The layout design section contains a toolbar with several tools. Some of them are highlighted in *figure 2.1* and described here:

1. **The design surface selector**: It allows us to see the design, the blueprint, or both. The figure shows both design and blueprint surfaces. The blueprint surface (in blue) draws only the outlines of components and is useful for advanced layouts, mainly when using invisible components in the design surface, such as guidelines.
2. **The orientation selector**: We can preview the layout in portrait or landscape orientations using this selector. It also allows us to work in night or day mode and automatically creates variants of our current layout (for example, it can create a landscape variant if the layout is in portrait). These variants are useful to fine-tune the layout to specific orientations or user interface types.
3. **The preview device selector**: The preview shown is based on particular device screen characteristics, like size and resolution. We can select different devices from this list and visualize how our layout adapts to their different characteristics.
4. **The preview API version selector**: Some features of the layout may look different when rendered in different API versions.
5. **The theme selector**: Themes are collections of attributes applied to all the views in the application. For example, a theme can be used to ensure that all views have a specific background color. This selector comes in handy to see how the layout looks when a particular theme is applied.
6. **The language selector**: As mentioned in the previous chapter, it is a good practice to translate the application's strings into different languages. However, that results in strings with different lengths and characters. We can use this selector to check if the layout needs to be adjusted to accommodate all the languages we use in the application.
7. **The view options**: We can use this selector to toggle the display of additional elements in the layout preview. Some allow us to understand how the layout appears on the device, such as a simulation of the system user interface. We may find others useful while working, such as for visualizing margins and constraints.
8. **The code view**: Only the XML layout file is available for editing in this view mode.
9. **The split view**: The tab is divided into two sections when this view mode is active: one contains the XML code, and the other contains the design view. It is useful to see how changes in the XML affect the design and vice-versa.
10. **The design view**: The tab only contains the designer. This view mode is the active mode in *figure 2.1*.

The ConstraintLayout

When we look at the component palette section, we find a component tree that shows the parent-child relationship between the layout components.

The root component in this tree is the class responsible for arranging the layout components. Our layout uses the **`ConstraintLayout`** class, which happens to be the one used by default in the newly generated code.

This type of layout is powerful and flexible. It allows us to create responsive layouts that rearrange themselves when the screen characteristics change. The feature that allows such behavior is the definition of relationships, called constraints, between the different views in the layout.

Let's click on the **`Hello World`** view in the layout editor. Note that it is centered on the screen with the help of four constraints. The arrows with wavy lines shown in *figure 2.2* represent the constraints.

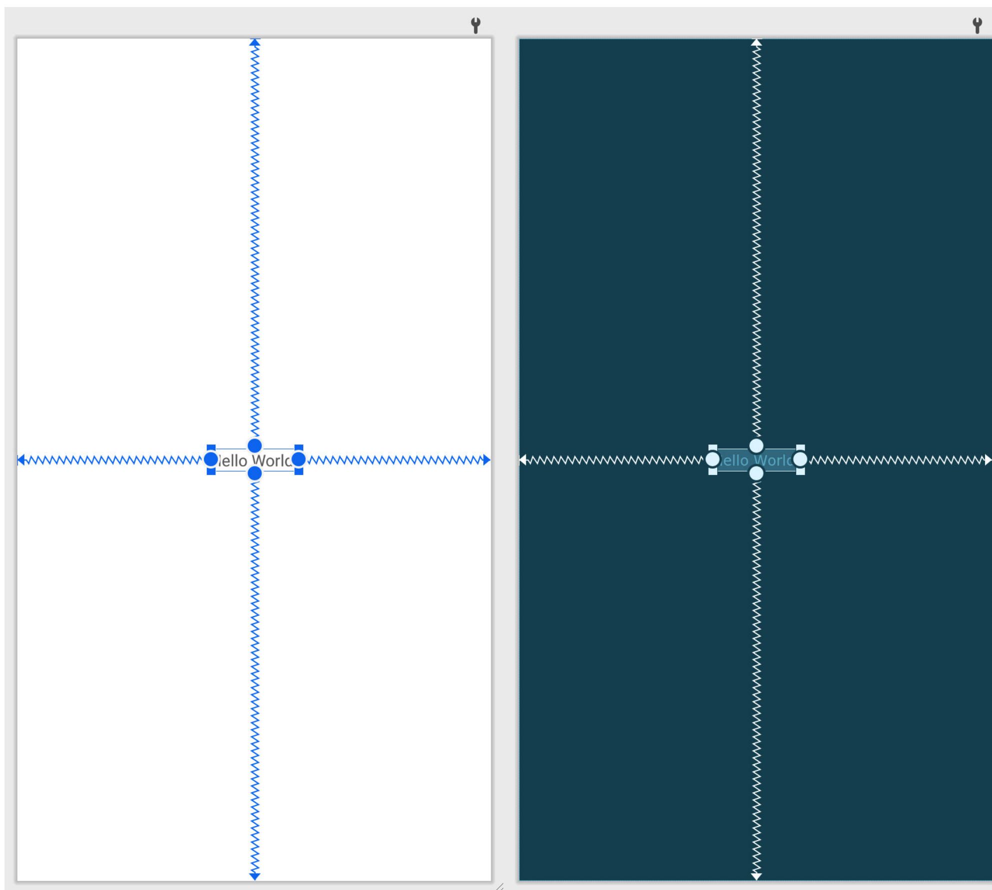

***Figure 2.2**: Visualizing layout constraints*

In this case, the **`Hello World`** text view is connected (or aligned) to all four corners of the parent view, which is the layout view. These connections ensure that it remains horizontally and vertically centered in the view, regardless of its dimensions.

It is not mandatory to define all four constraints, but every component must have at least one horizontal and one vertical constraint. Otherwise, it is not rendered at the expected location when the application runs on the device.

Placing a new component on the layout

We are finally ready to place our first button!

Drag the **`Button`** view item from the palette into the design view. Drop it horizontally centered in the lower third of the layout. A new component is created with a default text and identifier.

You should now see something similar to *figure 2.3:*

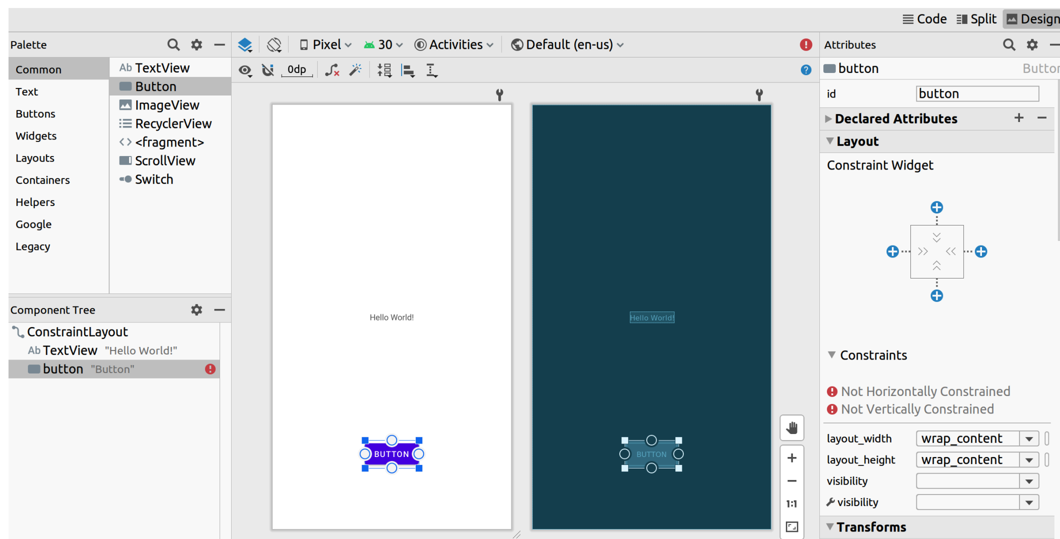

***Figure 2.3**: A freshly placed button*

Note that there are exclamation points on a red background in all three sections: the component tree, the toolbar, and the attributes pane.

They provide a visual indication of problems with the current layout. In this case, the required horizontal and vertical constraints for the button are missing. There's a nice tip to this effect in the attributes pane.

Adding the missing constraints is simple. Note that the selection box around the button has square and round handles. The round handles are used to set constraints.

Let's place our button horizontally centered on the layout and slightly above the bottom of the layout:

1. Drag the left round handle. An arrow is drawn as you drag; connect the arrow to the left border of the layout. Note that the button is placed at the border when you let go of the arrow tip. Do not worry; this happens because there are no other constraints to pull the component in another direction.

2. Drag the right round handle to create a constraint connected to the right border of the layout. The button is placed in the middle once you let go because there are now two constraints pulling in opposite directions. The lines become wabbly or springy to indicate this opposition.

3. Drag the bottom round handle to create a constraint connected to the bottom of the layout. The button is now placed against the bottom edge.

4. We want the button to be placed at some distance from the edge. Drag the button a little upward. Note that the vertical constraint now shows a number next to it. We just defined a margin.

Your layout probably looks like the one in *figure 2.4* now.

Yellow triangles with an exclamation mark replaced all red exclamation marks. This means the layout is now technically correct, but some best practices are not being followed.

If you click on the yellow triangle, a new panel appears at the bottom of the view explaining the issues found. In our case, it reports that the text is hardcoded and that we should use string resources.

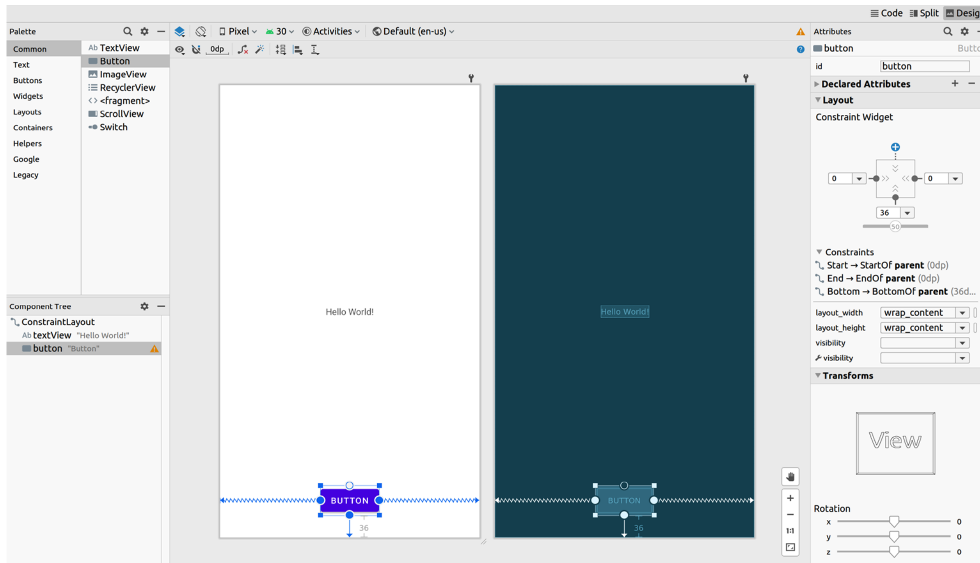

Figure 2.4: *The button with constraints defined*

Let's fix the resources problem.

Using string resources in components

Follow these steps to replace a selected component's text with a string resource:

1. Expand the **Declared Attributes** section in the attributes pane (if not expanded already).
2. Find the text attribute in this section. Click on the **pick a resource** button next to the text box to pick a resource.
3. Click on the plus button in the dialog box to add a new resource.
4. Fill in the resource details. We suggest using **about_button** as the string identifier and **About** as our new button's value. Leave **main** as the source set and **strings.xml** in the **values** folder as the target file.

Figure 2.5 illustrates this flow:

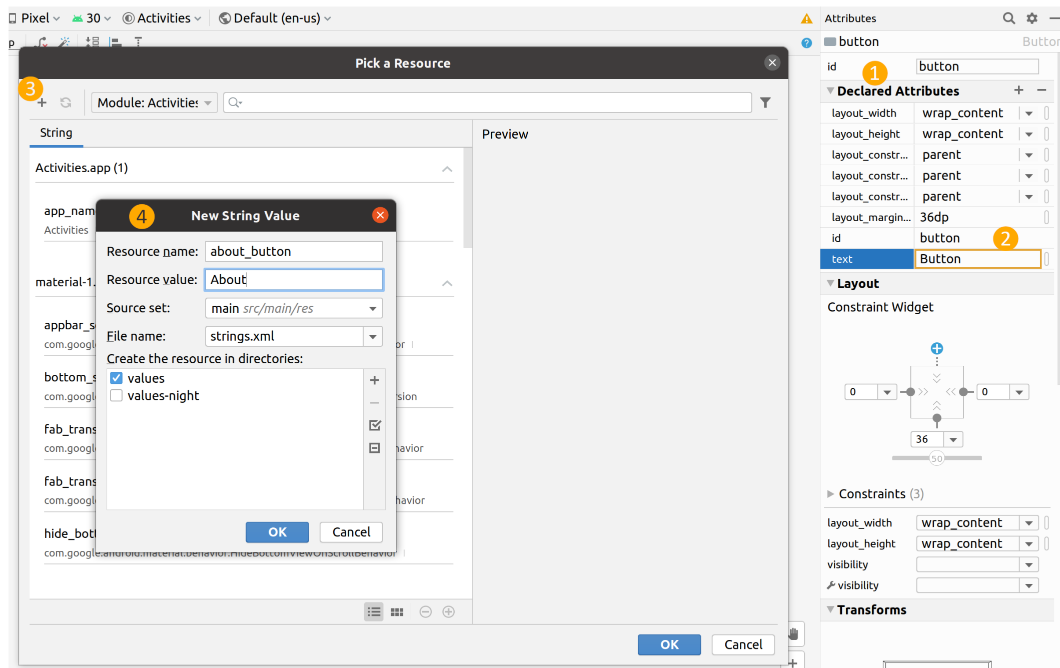

Figure 2.5: Adding a string resource from the layout editor

Steps 3 and 4 are replaced by a simple selection of the existing resource from the dialog box if you have already created the string resources you want to use.

We recommend choosing string identifiers that help understand the context where the string is used. This practice improves the maintainability of the resources and

helps choose the best translation later on, hence our suggestion of **about_button** for the new string identifier.

Changing the component identifier

A default identifier is assigned to the component when we place it in the layout editor. This identifier is often not the most suitable.

The button was assigned the **button** identifier in our example. We have designed the button to look like it describes something by choosing **About** as its text. Although it is true that it is a button (and the only one, for that matter), let's choose a better identifier.

Change the value of the **id** field in the attributes pane, for example, to **btn_about**, and press enter. A renaming dialog appears; confirm the refactoring operation.

There is no rigid rule about the composition of the identifier names. Most developers agree that it is good to use a descriptive name to convey the type of component and semantics. It helps identify the component when we refer to it in the code. Again, it's about improving maintainability, hence our suggestion of **btn_about**, which tries to refer to a button used for a feature called **about**.

The XML version of the button

All the steps we took to add and customize the button in the layout editor could have been performed directly in the XML layout file.

If we switch to the layout editor's code or split-view mode, we can see what the actual button code looks like in the XML file. The new button element is located inside the full layout file, next to the existing elements.

```
1.      <Button
2.          android:id="@+id/btn_about"
3.          android:layout_width="wrap_content"
4.          android:layout_height="wrap_content"
5.          android:layout_marginBottom="36dp"
6.          android:text="@string/about_button"
7.          app:layout_constraintBottom_toBottomOf="parent"
8.          app:layout_constraintEnd_toEndOf="parent"
9.          app:layout_constraintStart_toStartOf="parent" />
```

We can see all the changes we made in this button element: the string resource in line 6, the constraints in lines 7 to 9, and the identifier in line 2.

Note that the identifier in line 2 does not have a resource type classification prefix like the string resource identifier in line 6 has. The recommendation to choose descriptive names is especially relevant for component identifiers.

Registering to receive events

A button in our user interface is not useful if our application does nothing when the user clicks on it.

Our application expresses its desire to be informed of an event by registering an event listener through the existing **`View`** class methods.

Views expose a few events:

- **`onClick()`**: Fired when the user taps or clicks on a component
- **`onLongClick()`**: Fired when the user touches and holds or clicks on and holds a component
- **`onKey()`**: Fired when a hardware key is pressed while a component is focused
- **`onFocusChange()`**: Fired when the user navigates away from or onto a component
- **`onTouch()`**: Fired when the user performs a touch action, which can also include gestures
- **`onCreateContextMenu()`**: Fired when a context menu is being built, usually due to a long click

We are interested only in the **`onClick()`** event now. This event is fired when the user taps on our button.

Event registration is a task that is performed only once. So, keeping in mind the activity lifecycle diagram from the previous chapter, the best place to register for events is during the activity's creation. We do it in the activity's **`onCreate()`** method.

> **Note: Some events are not available on all devices. Note that the brief description we provided mentions hardware keys. Some devices have touchscreens, others have physical keys, trackpads, or keyboards, and yet others have both. So, events may have different physical sources and, for that reason, may not be available on all devices. For example, onKey() is only available on devices with physical keys.**

We need to find the view we are interested in and then implement the listener interface whose method is invoked when the event is triggered. In Java, we can do it easily with a method reference:

```
1.     @Override
2.     protected void onCreate(Bundle savedInstanceState) {
3.         super.onCreate(savedInstanceState);
4.         setContentView(R.layout.activity_main);
5.
6.         findViewById(R.id.btn_about)
7.             .setOnClickListener(this::onAboutButtonClicked);
8.     }
9.
10.    private void onAboutButtonClicked(View view) {
11.
12.    }
```

The preceding code snippet demonstrates this procedure.

In line 6, we searched for the view emitting the event we are interested in through the **`findViewById()`** method. This identifier needs to be the same identifier we specified in the layout file. We use the automatically generated **`R`** class to avoid mismatches. Remember the best practice of choosing descriptive identifiers? It is useful because we can now just read the code and know that it refers to our **`about`** button.

Then, we registered for the event in line 7. This registration is only about providing a callback that is called when the event is produced.

Reacting to the event

We have registered for the event, but we have not done anything useful with it so far.

Let's start by displaying a simple dialog box with information about our application.

Naturally, we need to have the information text ready. The dialogue box also needs a title and some text for the button that the user uses to dismiss it. So, we begin by adding all of them to our string resources file:

```
1. <string name="about_text">I\'m a demonstration application meant
   to showcase some essential techniques used for building Android
   applications.</string>

2. <string name="about_title">About myself</string>

3. <string name="done_button">Done</string>
```

The Android SDK provides an implementation of a dialog box suitable for our purpose in the **AlertDialog** class. In addition to displaying a message, it supports up to three buttons and a list of selectable items or a custom layout, but we do not need all of that to display a small message.

Since the **onAboutButtonClicked()** method is invoked when the **onClick()** event is fired, it is the natural place to create our dialog box:

```
1.  private void onAboutButtonClicked(View view) {
2.      new AlertDialog.Builder(this)
3.          .setTitle(R.string.about_title)
4.          .setMessage(R.string.about_text)
5.          .setNeutralButton(R.string.done_button, null)
6.          .show();
7. }
```

Note that we can provide an event listener as well in line 5. This listener is called when the user taps on the button. We have specified a null listener because we do not need to execute any action when that happens; we can allow the dialog to be dismissed.

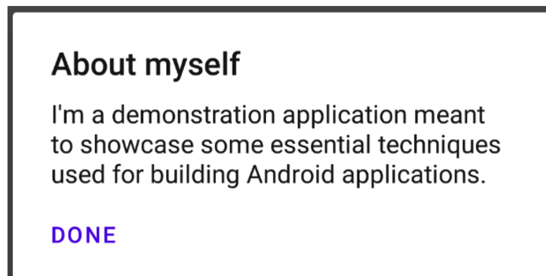

Figure 2.6*: A simple dialog box*

Like with most other Android user interface elements, this dialog box is dismissed when the user touches outside or clicks on the **Back** button.

Adding an activity to our application

As we mentioned earlier, we can have multiple activities in the same Android application. Each activity manages its layout, or screen if you prefer.

Let's replace the about dialog with an activity.

The first step is the creation of the activity and its layout. We take advantage of Android Studio's tools to do this seamlessly.

Start by right-clicking on the application root in the Android project layout. Then, choose **New | Activity | Empty Activity** from the context menu.

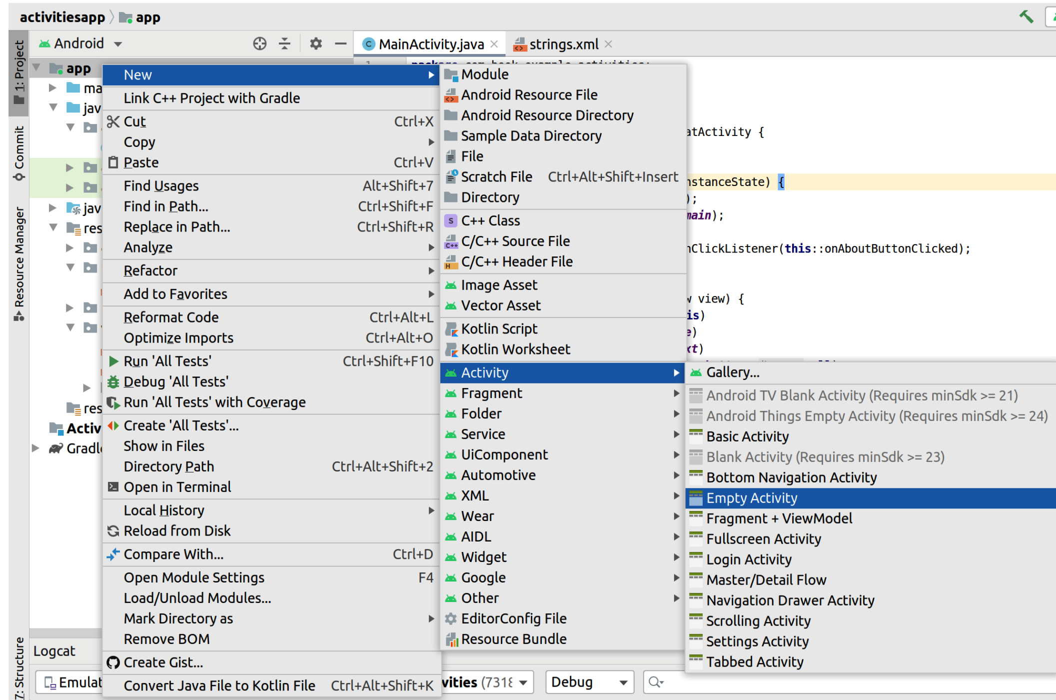

Figure 2.7: Creating an activity using Android Studio's tools

This option brings up the **New Android Activity** dialog box. Let's name this activity **AboutActivity**:

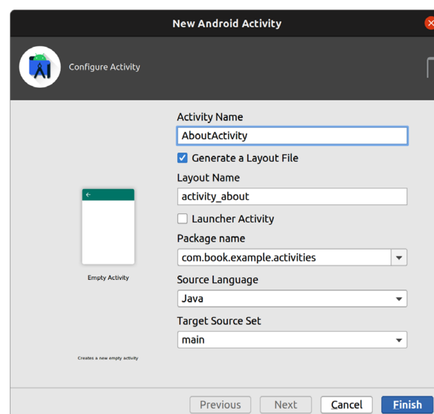

Figure 2.8: Details for our new About activity

An empty layout is generated, along with its activity class, when we click on the **Finish** button.

We need to build the layout now. We are creating a screen with information about the application, so we can add two text views: one for the title and one for the description message.

The process is identical to the one we followed earlier:

1. Open the layout editor by double-clicking on the **activity_about.xml** layout file in the **res/layout** directory.
2. This step is not mandatory, but try to enable the option to show all constraints in the view options menu (number 7 in *figure 2.1*). See the difference in the blueprint area and decide for yourself if and when this feature is useful.
3. Drag one **TextView** from the palette onto the layout design area. We use this view to display a title.
   - Set its constraints so that it is placed at some distance from the top and centered horizontally
   - Define its identifier
   - Set its text to reuse the string resource we created earlier for the dialog box's title
   - It may be fun to change the text view appearance. Select one of the existing styles for the **textAppearance** attribute
4. Drag another **TextView** from the palette. This one will display the application description.
   - Set its constraints so that it is placed horizontally and vertically centered
   - Define its identifier
   - Set its text to reuse the string resource we created earlier for the dialog box's message
   - Note that the text view automatically resizes to fit the text width up to its parent's limits. In this case, it touches the layout's outer edges without any margins
   - Set the text view's **paddingLeft** and **paddingRight** attributes for visual improvement. For example, try a setting of 32dp (device pixels) on both.

Upon completing these steps, you should have a layout design similar to the one in *figure 2.9*. If you have activated the option to show all constraints, the blueprint view always displays them.

Even with such a simple layout, we can see how the blueprint view on the left differs from the layout design view on the right. As we mentioned previously, the blueprint view's purpose is to focus only on the layout's structure.

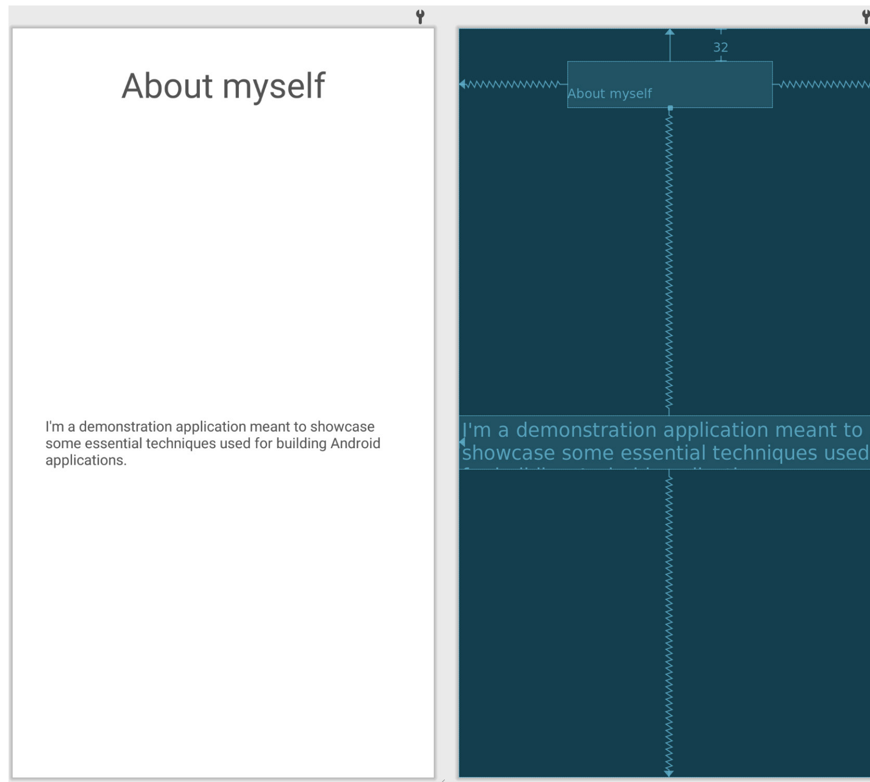

***Figure 2.9**: A possible layout for the activity describing the application*

The new **`TextView`** elements in the **`activity_about.xml`** file should be similar to the following excerpt. Pay attention to the usage of element, string, and style identifiers to prevent hardcoding these resources' values in the layout itself.

```
1. <TextView
2.     android:id="@+id/txt_about_title"
3.     android:layout_width="wrap_content"
4.     android:layout_height="wrap_content"
5.     android:layout_marginTop="32dp"
6.     android:text="@string/about_title"
7.     android:textAppearance="@style/TextAppearance.AppCompat.
         Display1"
```

```
8.      app:layout_constraintEnd_toEndOf="parent"
9.      app:layout_constraintStart_toStartOf="parent"
10.     app:layout_constraintTop_toTopOf="parent" />
11.
12. <TextView
13.     android:id="@+id/txt_about_message"
14.     android:layout_width="wrap_content"
15.     android:layout_height="wrap_content"
16.     android:paddingLeft="32dp"
17.     android:paddingRight="32dp"
18.     android:text="@string/about_text"
19.     app:layout_constraintBottom_toBottomOf="parent"
20.     app:layout_constraintEnd_toEndOf="parent"
21.     app:layout_constraintStart_toStartOf="parent"
22.     app:layout_constraintTop_toBottomOf="@+id/txt_about_title" />
```

Remember our earlier discussion of the application manifest? The new activity was also added to the manifest automatically. Pay attention to line 8 in the following excerpt:

```
1. <application
2.      android:allowBackup="true"
3.      android:icon="@mipmap/ic_launcher"
4.      android:label="@string/app_name"
5.      android:roundIcon="@mipmap/ic_launcher_round"
6.      android:supportsRtl="true"
7.      android:theme="@style/Theme.Activities">
8.      <activity android:name=".AboutActivity"></activity>
9.      <activity android:name=".MainActivity">
10.         <intent-filter>
11.             <action android:name="android.intent.action.MAIN" />
12.             <category android:name="android.intent.category.
                  LAUNCHER" />
13.         </intent-filter>
```

```
14.     </activity>
15.</application>
```

Now that we have completed our new activity, how do we display it once the user clicks on the about button? We do this using something called an intent.

Intents in Android applications

Android activities do not communicate through method calls. It is not possible to, for example, somehow call a method in a different activity from the activity in the foreground. Activities communicate asynchronously through a form of message passing.

Intents are the materialization of these messages.

They are mainly used for three purposes:

- To start an activity
- To start a service
- To send a broadcast message

A service is a component that runs in the background and does not have a user interface. It may be started once to execute a specific task, at a scheduled time, or at intervals. Services are out of the scope of this book; you can visit **https://developer.android.com/guide/components/services** for more information about them.

As the name implies, a broadcast message is a message that any application can receive. They can be sent by applications or by the system. Charging or startup messages are examples of broadcasts sent by the system. An application can send a broadcast just by using the **`sendBroadcast()`** method in its **`Context`** object. Do not confuse a broadcast with a user event. The first comes from outside of the application and is based on intents, while the second is exclusively internal and is based on callbacks.

Intents may be explicit or implicit.

An explicit intent specifies which application is meant to receive it. The target application's package name or a fully qualified class name is used in an explicit intent.

On the other hand, an implicit intent is generic and does not specify its target application. Often, the sender application does not even know which application receives it. Instead of naming a specific component, implicit intents declare an action to perform; for example, sending an email or showing a location on a map.

Creating an intent

An intent is an instance of the **Intent** class.

The following are the most important pieces of information that can be present in an intent object:

- The name of the component to be started. It is required only when building an explicit intent because it identifies the target application or component. As mentioned earlier, this needs to be a fully qualified class name or a package name.
- The name of the action to be performed. This name is optional and is a string understood by the target application. There are predefined action names in the **Intent** class and elsewhere. We can define any other name, as long as the receiving activity understands it.
- The category of the intent. It is not required and is used to determine the kind of component that should handle the intent. The description of the **Intent** class contains the list of categories available.
- The data that makes up the intent's payload, if any. This data is transferred as a URI and, optionally, an MIME type. It is not necessary if the intent's action does not require data transfer.

The system can determine an intent's target component by looking at the combination of all these attributes.

Showing our about activity with an intent

It is now clear that the best way to start our new activity is using an explicit intent.

We begin by changing the contents of the **onAboutButtonClicked()** method that we implemented earlier in the **MainActivity** class:

```
1.  private void onAboutButtonClicked(View view) {
2.     final Intent aboutIntent = new Intent(
3.             this,
4.             AboutActivity.class);
5.     startActivity(aboutIntent);
6. }
```

Remember that we know which component we want to target with this intent, so we can use this shorthand constructor to specify the target component while creating the intent.

Its first parameter is a reference to the context, that is, the application package that implements the target class. We supply a reference to the activity's object because it is also a context object.

The second parameter is the class that receives the intent.

We can also specify the component separately. Although it is not as convenient for this specific usage, the following implementation is equivalent:

```
1.  private void onAboutButtonClicked(View view) {
2.      final Intent aboutIntent = new Intent();
3.      aboutIntent.setComponent(
4.          new ComponentName(this, AboutActivity.class));
5.      startActivity(aboutIntent);
6. }
```

Note: It was not necessary to specify any action or category names in this case. The precise set of intent attributes that are necessary varies depending on the activity being launched.

With these changes in place, our user sees **`AboutActivity`** on the screen when they tap on the **`About`** button.

The Android system keeps a stack of the previous activities in memory for navigation purposes. A bit like the history in an internet browser, the activity history allows users to return to the previously displayed activities using the device's **`Back`** button.

So, we do not need any button in our **`AboutActivity`** to switch back to **`MainActivity`**; the system takes care of this navigation. All we need to do is to click on the device's **`Back`** button. It causes the previous activity to be displayed, and that is the **`MainActivity`**.

Figure 2.10 illustrates the flow we just implemented:

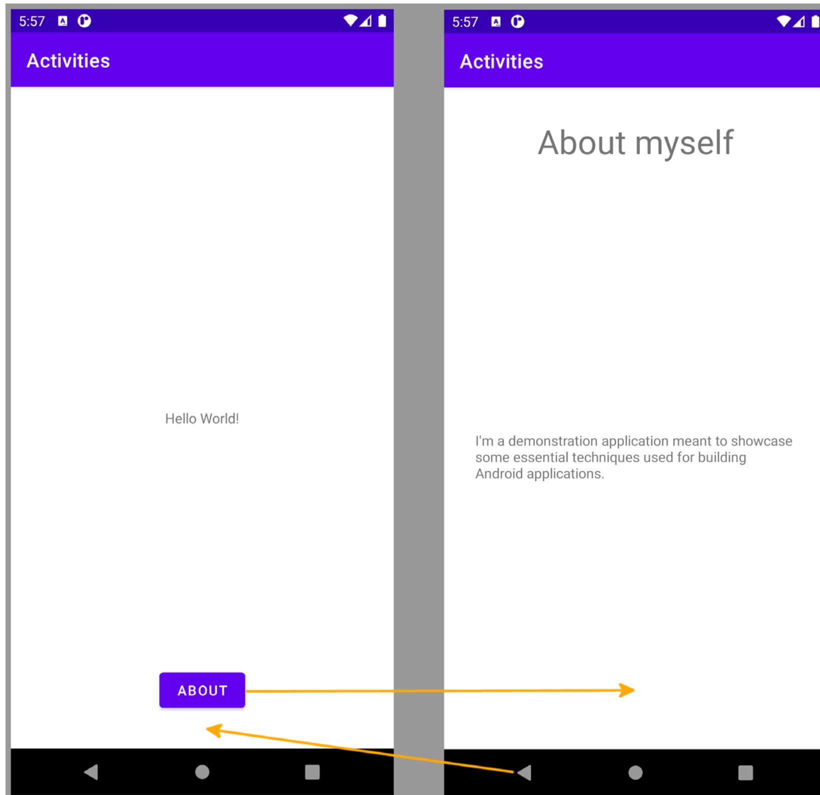

Figure 2.10*: Getting to the About activity from the main activity and back*

Returning a value from the target activity

We have seen how we can use an intent to show another activity. What if it becomes necessary to have data sent back from the activity we have invoked?

It is a three-step process:

1. The source activity creates an intent as earlier but uses the **`startActivityForResult()`** method to start the target activity this time.

   This method requires code to be provided, which identifies the result's source once it arrives. It is known only to the source activity.

2. The target activity creates an intent with the result, stores it using the **`setResult()`** method, and terminates it by calling the **`finish()`** method.

   The first parameter to **`setResult()`** is the result code. The standard code is **`RESULT_OK`** and **`RESULT_CANCELED`**.

3. The source activity gets its **onActivityResult()** method called with the result.

   The activity needs to use the request code and the result code to determine what to do with the information.

Relatively straightforward, right?

We have modified our sample application to contain a new activity with a yes/no question. This activity contains the question's text and two buttons so that the user can answer. *Figure 2.11* illustrates this new activity.

The following components were placed in it, repeating the same procedures illustrated earlier. The activity was also added to the manifest file.

```
1. <TextView
2.     android:id="@+id/txt_question"
3.     android:layout_width="wrap_content"
4.     android:layout_height="wrap_content"
5.     android:text="@string/question"
6.     app:layout_constraintBottom_toBottomOf="parent"
7.     app:layout_constraintEnd_toEndOf="parent"
8.     app:layout_constraintStart_toStartOf="parent"
9.     app:layout_constraintTop_toTopOf="parent" />
10.
11.<Button
12.     android:id="@+id/btn_yes"
13.     android:layout_width="wrap_content"
14.     android:layout_height="wrap_content"
15.     android:layout_marginBottom="32dp"
16.     android:text="@string/yes"
17.     app:layout_constraintBottom_toBottomOf="parent"
18.     app:layout_constraintEnd_toStartOf="@+id/guideline"
19.     app:layout_constraintStart_toStartOf="parent" />
20.
21.<Button
22.     android:id="@+id/btn_no"
```

```
23.     android:layout_width="wrap_content"
24.     android:layout_height="wrap_content"
25.     android:layout_marginBottom="32dp"
26.     android:text="@string/no"
27.     app:layout_constraintBottom_toBottomOf="parent"
28.     app:layout_constraintEnd_toEndOf="parent"
29.     app:layout_constraintStart_toStartOf="@+id/guideline" />
30.
31. <androidx.constraintlayout.widget.Guideline
32.     android:id="@+id/guideline"
33.     android:layout_width="wrap_content"
34.     android:layout_height="wrap_content"
35.     android:orientation="vertical"
36.     app:layout_constraintGuide_percent="0.5" />
```

Note: We have used a vertical guideline to have the two buttons equally separated from each other horizontally.

Naturally, the companion activity class also needs to be modified to work with the new buttons and return the result.

The following changes have been made to the **onCreate()** method, and new methods have been added as necessary:

```
1.  @Override
2.  protected void onCreate(Bundle savedInstanceState) {
3.      super.onCreate(savedInstanceState);
4.      setContentView(R.layout.activity_question);
5.
6.      findViewById(R.id.btn_yes)
7.          .setOnClickListener(this::onYesButtonClicked);
8.      findViewById(R.id.btn_no)
9.          .setOnClickListener(this::onNoButtonClicked);
10. }
11.
```

```
12. private void onNoButtonClicked(View view) {
13.     returnAnswer("no");
14. }
15.
16. private void onYesButtonClicked(View view) {
17.     returnAnswer("yes");
18. }
19.
20. private void returnAnswer(String answer) {
21.     final Intent result = new Intent();
22.     result.putExtra("answer", answer);
23.     setResult(RESULT_OK, result);
24.     finish();
25. }
```

The code responsible for capturing and handling user events should be familiar. The interesting part is the **returnAnswer()** method. As we described earlier, it creates an intent, sets it as the activity's result with **setResult()**, and terminates the activity by calling **finish()**.

The actual result data is sent within the intent as a piece of extended data. The URI and MIME type can also be set if the desired result data fits this pattern. In a nutshell, any data that can be sent in an intent to start an activity can also be sent in an intent as a result.

Finally, we need to invoke this new activity and consume its result.

We have modified the main activity, so it has another button to ask the question now. The identifier of the hello world text view has also been changed. We use this view to display the answer.

```
1. <TextView
2.     android:id="@+id/txt_mainMessage"
3.     android:layout_width="wrap_content"
4.     android:layout_height="wrap_content"
5.     android:text="@string/hello_world"
6.     app:layout_constraintBottom_toBottomOf="parent"
7.     app:layout_constraintLeft_toLeftOf="parent"
```

```
8.      app:layout_constraintRight_toRightOf="parent"
9.      app:layout_constraintTop_toTopOf="parent" />
10.
11.<Button
12.     android:id="@+id/btn_about"
13.     android:layout_width="wrap_content"
14.     android:layout_height="wrap_content"
15.     android:layout_marginBottom="36dp"
16.     android:text="@string/about_button"
17.     app:layout_constraintBottom_toBottomOf="parent"
18.     app:layout_constraintEnd_toStartOf="@+id/guideline1"
19.     app:layout_constraintStart_toStartOf="parent" />
20.
21.<androidx.constraintlayout.widget.Guideline
22.     android:id="@+id/guideline1"
23.     android:layout_width="wrap_content"
24.     android:layout_height="wrap_content"
25.     android:orientation="vertical"
26.     app:layout_constraintGuide_percent="0.33" />
27.
28.<androidx.constraintlayout.widget.Guideline
29.     android:id="@+id/guideline2"
30.     android:layout_width="wrap_content"
31.     android:layout_height="wrap_content"
32.     android:orientation="vertical"
33.     app:layout_constraintGuide_percent="0.67" />
34.
35.<Button
36.     android:id="@+id/btn_question"
37.     android:layout_width="wrap_content"
38.     android:layout_height="wrap_content"
```

```
39.     android:layout_marginBottom="36dp"
40.     android:text="@string/question_button"
41.     app:layout_constraintBottom_toBottomOf="parent"
42.     app:layout_constraintEnd_toEndOf="parent"
43.     app:layout_constraintStart_toStartOf="@+id/guideline2" />
```

Figure 2.11 depicts the new layout of the main activity.

The necessary modifications to the **MainActivity** class include:

- Registering for a new user event and handling it by calling **QuestionActivity** expecting a result
- Handling the result returned by **QuestionActivity**

```
1.  @Override
2.  protected void onCreate(Bundle savedInstanceState) {
3.      super.onCreate(savedInstanceState);
4.      setContentView(R.layout.activity_main);
5.
6.      findViewById(R.id.btn_about)
7.          .setOnClickListener(this::onAboutButtonClicked);
8.      findViewById(R.id.btn_question)
9.          .setOnClickListener(this::onQuestionButtonClicked);
10. }
11.
12. private void onQuestionButtonClicked(View view) {
13.     final Intent questionIntent = new Intent(this,
14.             QuestionActivity.class);
15.     startActivityForResult(questionIntent, REQUEST_AN_
        ANSWER);
16. }
17.
18. private void onAboutButtonClicked(View view) {
19.     final Intent aboutIntent = new Intent(this,
20.             AboutActivity.class);
```

```
21.     startActivity(aboutIntent);
22. }
23.
24.   @Override
25.  protected void onActivityResult(int requestCode, int
    resultCode, @Nullable Intent data) {
26.
27.     if (requestCode == REQUEST_AN_ANSWER) {
28.         setMessage(
29.             resultCode == RESULT_OK
30.                 ? getString(R.string.answer_is,
31.                      data.getStringExtra("answer"))
32.                 : getString(R.string.hello_world)
33.         );
34.     } else {
35.         super.onActivityResult(requestCode, resultCode,
              data);
36.     }
37. }
38.
39.  private void setMessage(String text) {
40.     ((TextView) findViewById(R.id.txt_mainMessage))
41.             .setText(text);
42. }
```

Note that the implementation of **`onActivityResult()`** needs to check if:

- The request code is the one it knows how to handle
- The result code indicates a valid result

> **Note: The activity is closed by the system when the user presses the Back button on the question activity. In this case, no result was set. The `onActivityResult()` method is still called by the system, but with the standard `RESULT_CANCELED` result code.**

The main activity changes its message to display the result received when it receives a valid result. Lines 30 to 32 from the previous excerpt show how we can use and format string resources.

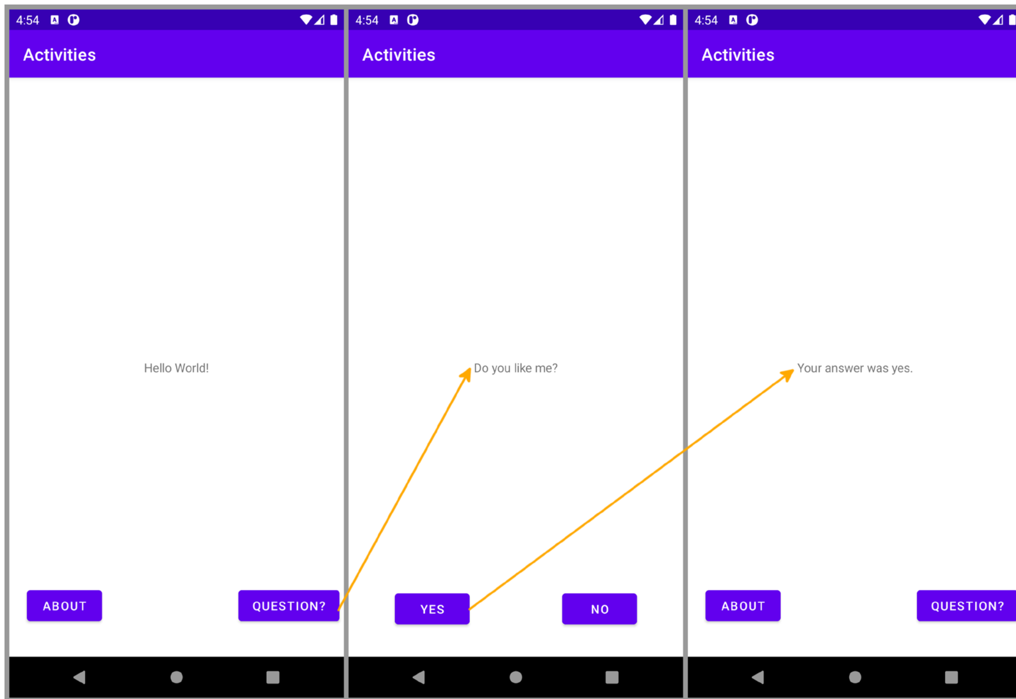

***Figure 2.11**: The question activity flow*

Figure 2.11 illustrates the end state of all the activities we have created and modified in this section, along with the expected flow:

1. First, the user taps on the **QUESTION** button
2. Then, the question activity is displayed
3. The main activity is updated when the user taps on one of the answers

Using Intents across Android applications

When we described intents in the previous section, we mentioned that the Android system uses a combination of all fundamental intent attribute values to determine the activity that must be started: the component and action names, the category, and the data type.

It follows that we can start activities from any application in the device, provided that the target activity allows it.

Intent filters in the application manifest

Remember the difference between explicit and implicit intents? An implicit intent does not specify an activity name and contains only an action and category name and, optionally, a data type.

To advertise the implicit intents that an activity can receive, they must be included in the application's manifest file under the form of intent filters.

Each intent filter declares the combination of the action name, category name, and data type that the activity can accept.

Let's look at our declaration of **MainActivity** in the manifest file again:

```
1. <activity android:name=".MainActivity">
2.     <intent-filter>
3.         <action android:name="android.intent.action.MAIN" />
4.         <category android:name="android.intent.category.LAUNCHER"
             />
5.     </intent-filter>
6. </activity>
```

In the previous chapter, we stated that the **intent-filter** element was meant to allow the application to be started by the launcher. It now makes more sense—it means that the **MainActivity** activity can accept intents with the given action and category names.

Let's pretend that our **MainActivity** was also able to accept some text to send somewhere. We would then add a second intent filter, which would look like this:

```
1. <intent-filter>
2.     <action android:name="android.intent.action.SEND" />
3.     <category android:name="android.intent.category.DEFAULT" />
4.     <data android:mimeType="text/plain" />
5. </intent-filter>
```

Note that the category name is **android.intent.category.DEFAULT** in this case. You must declare this category name to receive implicit intents because it is used to mark the default action to be performed on some data type. This category is not used when creating the intent, but **startActivity()** and **startActivityForResult()** handle the implicit intent as if it had been specified.

Note: Intent filters do not affect explicit intents. Do not use intent filters if you must prevent other applications from starting your activities with explicit intents. Set the "exported" attribute to false for that activity instead. Consider this example:

```
1. <activity android:name=".AboutActivity"
2.           android:exported="false" />
```

Starting another application with an implicit intent

Now that we know how to advertise our activities to other applications, we can see how we can start other applications from our demonstration application.

You have probably already realized that the approach is similar to the one we chose regarding explicit intents.

Let's send the user to a web page using a web browser.

We first add a new button to our `MainActivity` so that the user can trigger the new feature. Refer to *figure 2.12* for an illustration of the main activity's new layout.

```
1. <Button
2.     android:id="@+id/btn_web"
3.     android:layout_width="wrap_content"
4.     android:layout_height="wrap_content"
5.     android:text="@string/show_web_page"
6.     app:layout_constraintBottom_toBottomOf="@+id/btn_about"
7.     app:layout_constraintEnd_toStartOf="@+id/guideline2"
8.     app:layout_constraintStart_toStartOf="@+id/guideline1" />
```

Then, we register to be informed when the user taps on the button:

```
1.  @Override
2.  protected void onCreate(Bundle savedInstanceState) {
3.      super.onCreate(savedInstanceState);
4.      setContentView(R.layout.activity_main);
5.
6.      // (existing event registrations come here)
7.
```

```
8.     findViewById(R.id.btn_web)
9.         .setOnClickListener(this::onWebButtonClicked);
10. }
```

Finally, we create an implicit intent and start the target activity:

```
11. private void onWebButtonClicked(View view) {
12.     final Intent webIntent = new Intent(
13.             Intent.ACTION_VIEW,
14.             Uri.parse("https://www.bpbonline.com"));
15.     startActivity(webIntent);
16. }
```

Note that this new intent only contains an action name and the data URI. It does not name any activity or package; the system finds the correct activity to start.

However, this approach has a problem— our application crashes if no activity can handle the intent. The **startActivity()** method throws **android.content.ActivityNotFoundException** if the system cannot resolve the intent to an activity.

Naturally, the first option to prevent the crash is to catch the exception:

```
1.  try {
2.      startActivity(webIntent);
3.  } catch (ActivityNotFoundException ex) {
4.      Log.d("MainActivity", "Can't show the web page.", ex);
5.  }
```

Suppose you do not want to deal with exception handling in this situation or that you want to use the intent to determine beforehand if an application capable of viewing web pages is available.

In that case, the intent object exposes the **resolveActivity()** method, which can be invoked to determine if a target activity can be found:

```
1.  if (webIntent.resolveActivity(getPackageManager()) != null) {
2.      startActivity(webIntent);
3.  }
```

However, there is a catch.

Calling this method means the application is querying the system's list of installed applications, albeit indirectly. Starting with Android 11, applications need to declare these queries in the application manifest in most cases.

It is, undoubtedly, required for this use case.

Our little application's manifest needs to be extended with a new queries section, so the **resolveActivity()** method works with our web intent in Android 11:

```
1. <queries>
2.     <intent>
3.         <action android:name="android.intent.action.VIEW" />
4.         <category android:name="android.intent.category.
             BROWSABLE" />
5.         <data android:scheme="https" />
6.     </intent>
7. </queries>
```

Best practice: Applications are only supposed to ask for strictly necessary permissions. The capability to query the list of installed applications is a type of permission.

Therefore, unless your application has a compelling use case for resolving activities for implicit intents before submitting them to the system, the best practice, in this case, is to handle the exception.

Once everything is in place, we should have an application that can execute the flow illustrated in *figure 2.12*:

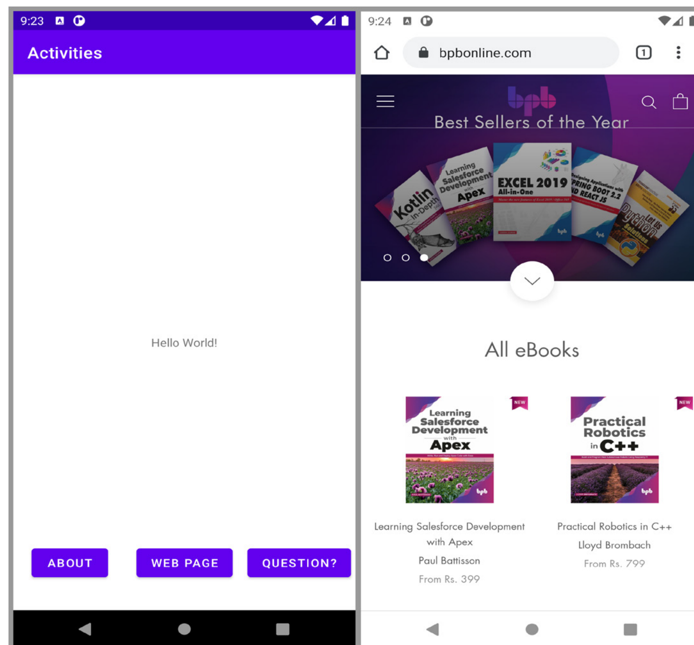

***Figure 2.12**: The web activity flow*

As usual, the user can return from the browser to our main activity by clicking on the **`Back`** button.

The flow for starting an activity in another application and receiving a result is also identical to the one we demonstrated earlier. The other activity is started with **`startActivityForResult()`**, and the source activity handles the result in **`onActivityResult()`**.

Of course, the intent needs to be specified according to the other activity's expectations when starting third-party activities. A list of some common intents is available at **https://developer.android.com/guide/components/intents-common.html**. If the desired action is not present in this list, you must contact the application's developers.

Conclusion

In this chapter, we saw how to work with activities—modifying and creating them and handling the user's interactions with their components. We then explained how intents are used to help activities start other activities and transfer data back and forth. Finally, we took advantage of the features implemented in other applications to add functionality to our application.

The Kotlin language is becoming the go-to language for writing Android applications, so the next chapter presents its fundamental features and syntax. It also shows how an Android application can take advantage of an embedded database and the available options for Android developers.

CHAPTER 3
Building Our Base Application with Kotlin and SQLite

Introduction

This chapter explores how to build the application that serves as the basis for examples in the following chapters. The big difference is that it is written in the Kotlin language instead of Java. Kotlin is a relatively new language that was adopted as an alternative to Java in Android Studio.

Some example applications need to store user data permanently. We build a data persistence layer to be used by an application and show how this is used to create an application database, supported by the SQLite engine and the Room ORM.

Structure

We cover the following topics in this chapter:

- A first glance at the Kotlin language
- Fundamentals of the Kotlin language
- Using a SQLite database in our application
- Object-relational mapping
- Working with a database with Room ORM

Objectives

By the end of this chapter, you are familiarized with the most significant differences between Java and Kotlin. You can read the Kotlin code used in the examples and explanations and are familiar with the advantages of Kotlin over Java.

You can also understand how to store different application data types in the device and interact with the Android system for that purpose.

A first glance at the Kotlin language

How do we use Kotlin to write our Android application? We can mix Kotlin and Java in the same project, but the easiest and simplest way is to generate an application from scratch already configured for the Kotlin language.

Let's do it then. We create a new Android application project, similar to what we did in the previous chapters, with the difference being that we ensure that we select the Kotlin language in the project configuration dialog box. Take a look at *figure 3.1:*

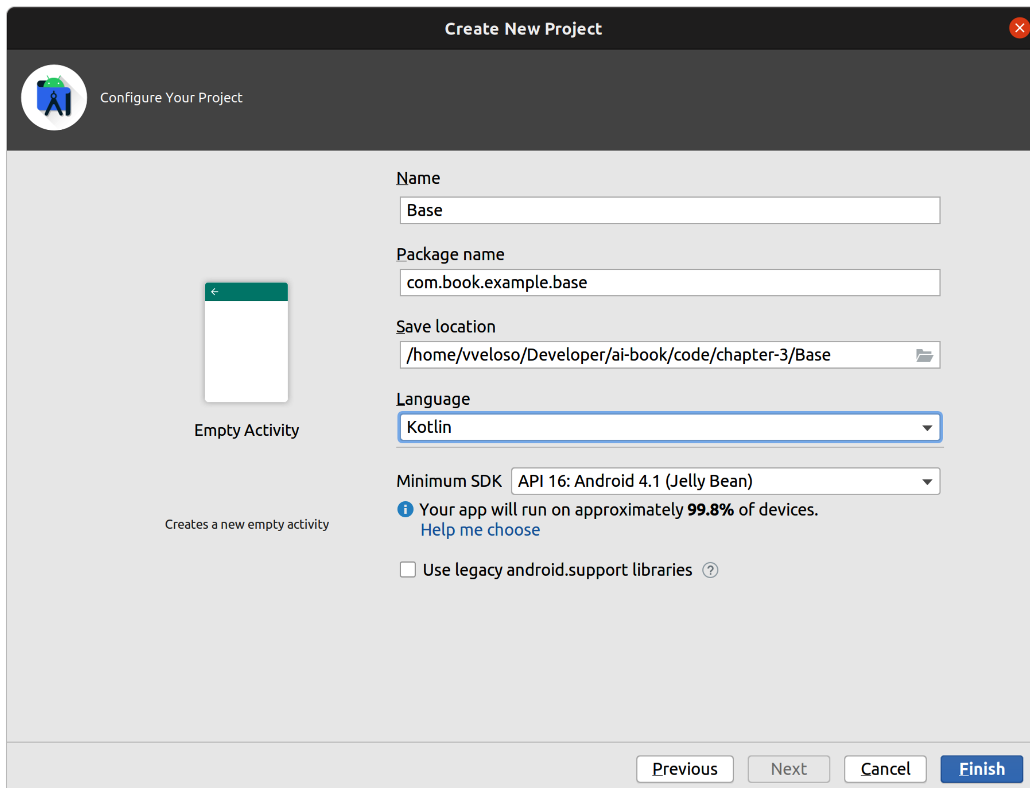

***Figure 3.1**: Creating a Kotlin-based project*

That's all it takes. The code is generated in the Kotlin language now. The location is the same as for the Java files to make it easier to mix languages, but the file extension is now **`.kt`** instead of **`.java`**.

Best practices: Languages can be mixed in the same project, but we do not recommend it. It is a lot easier to read and understand a codebase if it is written in the same language.

Now that the empty activity has been generated, let's see how the Kotlin version is different from the Java one.

First, recall the old Java code:

```
1.  package com.book.example.activities;
2.
3.  import androidx.appcompat.app.AppCompatActivity;
4.  import android.os.Bundle;
5.
6.  public class MainActivity extends AppCompatActivity {
7.     @Override
8.     protected void onCreate(Bundle savedInstanceState) {
9.         super.onCreate(savedInstanceState);
10.        setContentView(R.layout.activity_main);
11.    }
12.}
```

Now, look at the new Kotlin code:

```
1.  package com.book.example.base
2.
3.  import androidx.appcompat.app.AppCompatActivity
4.  import android.os.Bundle
5.
6.  class MainActivity : AppCompatActivity() {
7.     override fun onCreate(savedInstanceState: Bundle?) {
8.         super.onCreate(savedInstanceState)
9.         setContentView(R.layout.activity_main)
```

```
10.     }
11. }
```

They're strikingly similar, albeit using different paradigms. We discuss the Kotlin syntax and semantics in the upcoming sections, but we can already make a small list of the most significant differences in these few lines of Kotlin code:

- Semicolons are optional at the end of the statements
- There are no visibility declarations because Kotlin defaults to public visibility
- Kotlin requires an explicit override keyword in overridden methods
- There is no extends keyword, but only a colon
- The base class constructor is invoked explicitly
- Type names come after the variable name
- Nullable variables must be declared explicitly with a question mark on the type name; Kotlin's default is that variables cannot be null

Fundamentals of the Kotlin language

Kotlin is quite intuitive for a Java developer, but it's better to take some time to discuss the basics of this new language.

One of the arguments in favor of Kotlin is that it has the most recognizable aspects of functional languages like Scala and does away with the extreme verbosity that we all recognize in Java.

Although Java got a lot better after streams, lambdas, method references, and default interface methods were introduced in version 8, Kotlin can still make a difference. Let's see how.

Note: What follows is by no means a comprehensive description of the Kotlin language. It should be enough for you to understand our usage of Kotlin in this book, but there is much more to explore.

Packages

Kotlin code files may start with a package declaration, like Java.

Unlike Java, however, there is no connection between a package declaration and the file's location in the file system. It is recommended that they match to ease the developer's cognitive burden (a fancy way of saying that it saves you from memorizing their locations), but it is not required.

A package declaration is optional. All declarations inside that file are placed in the default package when it is absent.

Code organization

Unlike Java, Kotlin does not require a function to be inside a class. Additionally, it does not require variable declarations to be inside a class.

Visibility

Just like Java, Kotlin does not require a declaration's visibility to be explicitly modified. When not modified, that symbol (variable, function, class, and so on) is public.

Kotlin supports the following visibility modifiers:

- Private
- Protected
- Internal
- Public

Generally speaking, private declarations are only visible within the same scope—file for top-level declarations or class for member declarations. Protected declarations are the same as private, except for subclasses, where they are also visible. Internal declarations are visible only inside the same module. Finally, public declarations are the default and are visible everywhere.

Data types

Kotlin supports the same basic data types as Java, except that it does not distinguish between primitive and boxed types. All types look like objects to the user, regardless of their internal representation.

The following numeric types are supported:

- Byte
- Short
- Int
- Long
- Float
- Double

Floating-point properties are initialized with numeric literal constants with a decimal part (for example, 1.0) and default to the **Double** data type. To specify the **Float** type explicitly, add an f suffix (for example, 1.0f).

There are also types dealing with Boolean values, characters and strings:

- Boolean
- Char
- String

An interesting deviation from Java and other languages is that arrays are represented as **Array** class instances. The **Array** class receives a type parameter that defines the type of its elements. Arrays are created using the **arrayOf()** function or the constructor that takes a size and an initializer function:

```
1.  val lettersArray: Array<String> = arrayOf("a", "b", "c")
2.  val stringNumbersArray: Array<String> =
3.      Array(10) { index -> index.toString() }
```

For efficiency purposes, specific array classes are defined for types that can be represented internally as primitives: integers, shorts, and so on. These have their own types, such as **IntArray**, and builder methods, such as **intArrayOf()**.

```
1.  val anArray: IntArray = intArrayOf(1, 2, 3, 4)
```

Items in the array can still be accessed in the usual way with square brackets:

```
1. println(anArray[1])
```

Although an array is an instance of the **Array** class or one of the specialized classes, item indexing with square brackets works, thanks to a Kotlin feature called operator overloading. We are not covering this topic in this book, but it is a powerful feature that should be used cautiously. Readers with a background in C++ are probably familiar with a similar implementation in that language.

Variables and properties

A property is declared using the **var** or **val** keyword, followed by the variable name, a colon, the variable type, and an initializer.

The property is mutable when using the **var** keyword, and the property becomes immutable (that is, read-only) when using the **val** keyword.

The following is a declaration of a mutable property:

```
1.  var age: Int = 42
```

It can be modified in a function in the usual way:

```
1.  fun birthday() {
2.      age = age + 1
3.  }
```

An immutable property is defined just by using the **val** keyword:

```
1.  val age: Int = 42
```

In Java, we would call it a **final** variable. We get a compilation error if we attempt to change its value:

```
1.  fun birthday() {
2.      age = age + 1 // ERROR: Val cannot be reassigned
3.  }
```

The initialization of a property is mandatory: there are no default property values. When the property is declared inside a class, its initialization may be done through the class constructor.

When declared inside a function, properties are just variables. However, they gain powerful additional features and become real properties in the object-oriented sense when declared in global or class scope.

Remember the simple declaration of an age property?

```
1.  var age: Int = 42
```

The following would be a possible equivalent implementation of the same declaration, but in Java:

```
1.  private int age = 42;
2.
3.  public int getAge() {
4.      return age;
5.  }
6.
7.  public void setAge(int value) {
8.      age = value;
9.  }
```

That's right. As a rule of thumb, mutable Kotlin properties are composed of a backing field, a setter, and a getter. Naturally, immutable properties cannot have a setter.

Because of this, property declarations also allow us to redefine their setters and getters. This flexibility has many uses.

For example, we can redefine the accessor's visibility. In the following excerpt, the mutable property can be modified only within the current scope. Other users can only read it.

```
1.  var age: Int = 42
2.   private set
```

Or we can manipulate the value returned by the getter. The following excerpt manipulates the property's backing field:

```
1.  var name: String = "mary"
2.   get() = field.capitalize()
```

We can also have an immutable property that is backed only by some other function:

```
1.  val isEmpty: Boolean
2.   get() = this.size() == 0
```

The power behind this automatic generation of getters or setters makes even more sense once we realize that the need for writing those accessors as disjoint functions in classes has disappeared.

Tip: Always define a property or variable as immutable, using the val keyword. This way, you avoid having mutable state in your application, which can be a problem with parallel programming. If you decide later that you must change the property's contents, you can always change it into a var.

Type inference

Surely you have noticed that all declarations presented so far have included the type being used in the property.

Kotlin supports type inference, which means it is not necessary to specify the type of a property or the return type of a function as long as the compiler can figure it out automatically from the code.

For example, given the following declarations:

```
1.  val longWithType: Long = 1234
2.  val intWithType: Int = 987
```

The following ones are equivalent:

```
1.  val longWithoutType = 1234L
2.  val intWithoutType = 987
```

Note that we did not specify the property types in the second excerpt. Due to their initializers, the compiler can determine the correct type to use.

Pay particular attention to line 1 on both excerpts. We need to ensure that the correct suffix is used when a type is not specified and we use numeric literal constants as initializers, hence the L suffix to denote a long integer in the second version.

It may be tempting – and scary – to think that this kind of type inference means no type safety in Kotlin. This thought is not true: Kotlin remains a strongly typed language. The feature is there to save us some typing. If the compiler is faced with an ambiguous expression, it refuses to compile the declaration.

Tip: Be careful with type inference, especially when exposing public functions or properties. The lack of a type declaration may make it difficult for a human reader to fully understand the function or property's semantics. Consider including the type when it is not clear from the full declaration.

Functions

We already know that some packages and static functions are always imported automatically in Kotlin code files. Thus, the following is a perfectly valid Kotlin program:

```
1.  fun main() {
2.      println("here I am!")
3.  }
```

Let's dissect these three lines, shall we?

The function declaration begins immediately without the need for an enclosing class. It starts with the `fun` keyword, which is shorthand for function. There is no visibility declaration, which means the function is public. Finally, there is no return value type, which means the function does not return anything.

One function call lies inside the function body. Note that there is no import statement for this function because many static functions are imported automatically by the Kotlin compiler. There is no semicolon; it is optional.

We add the return type after the argument list, preceded by a colon, when we need to return a value from our function.

```
1.  fun add(x: Int, y: Int): Int {
2.      return x + y
3.  }
```

This new function is then invoked like we expect to do in Java:

```
1.  fun main() {
2.      println("I can add 2 and 2 together: " + add(2, 2))
3.  }
```

When the function body can be reduced to a single expression, we can even omit the curly braces and use an equals sign, as follows:

```
1.  fun add(x: Int, y: Int): Int = x + y
```

Nice, right? Almost like a property declaration!

As discussed in the previous section, type inference also applies to functions. So, the following is an equally correct declaration of the **add()** function:

```
1.  fun add(x: Int, y: Int) = x + y
```

It is expected that the function returns an integer because it adds two integers. Its expression is clear, so this is a case when omitting the type from the declaration does not harm the understanding of the function's semantics.

In Java, we can talk about functions and methods. The difference is that the former return some value while the latter do not return anything. This distinction does not exist in Kotlin.

Kotlin always declares functions, which means that they always have a return value. What about **void** functions or functions that do not return anything, like the preceding **main()** function?

There is a particular type in Kotlin used when a function does not return anything useful—Unit. The **Unit** type cannot be instantiated, a bit like the **Void** type in Java. Unlike Java's Void type, however, it is impossible to convert the null value to an instance of Unit.

The compiler uses type inference, so we do not need to specify **Unit** as the return type of a function with no return statement. Therefore, the following declaration of the **main()** function is equivalent to the one we presented initially.

```
1.  fun main(): Unit {
2.      println("here I am!")
3.  }
```

Of course, it is preferable to omit the return type declaration for functions like this.

Given that these functions do return a value (**Unit**), the following is also legal Kotlin code, albeit not idiomatic and not very useful:

```
1.  fun main(): Unit /* redundant type */ {
2.      // I do nothing
3.      return Unit // this statement is unnecessary
4.  }
5.  val result: Unit = main() // why would we do this?
```

Nobody should write code like this.

Lambda expressions

Lambda expressions in Kotlin have a syntax similar to Java, albeit with some differences.

The first difference is that we do not need to define a functional interface to represent a function type. Function types exist in Kotlin.

```
1.  val add: (Int, Int) -> Int =
2.          { a: Int, b: Int -> a + b }
```

The preceding excerpt defines a property called **add**. This property has a type like any other, which, in this case, is a function type. The property becomes a reference to a function that takes two integer arguments and returns an integer value.

It is initialized using the usual pattern, but in this case, we declare a lambda using curly braces. Inside this body, we declare the function's arguments on the arrow operator's left-hand side and its body on the right-hand side.

Note that the function arguments must be qualified with their types. This is necessary because type inference works from right to left, that is, it needs a fully qualified type on the right side operand to infer the type of the left side operand.

Taking advantage of type inference, we can rewrite the property declaration as follows:

```
1.  val add = { a: Int, b: Int -> a + b }
```

Then we can use it as a function declared with the **fun** keyword:

```
1.  fun main() {
2.      println("I can add 2 and 2 together: " + add(2, 2))
3.  }
```

It is seldom useful to declare properties that hold functions. However, knowing how to declare a property (or variable) of a function type, we can now introduce the concept of a higher-order function.

A higher-order function is a function that takes another function as an argument.

We already have a function that adds two integers. We can write the following if we want to define a function that doubles the result of an operation on two integers:

```
1.  fun twice(a: Int, b: Int, operation: (Int, Int) -> Int): Int =
2.          2 * operation(a, b)
```

Note that the last argument of the **twice** function is a function type. It accepts any function that takes two integers as arguments and returns an integer value.

We can now write this:

```
1.  val add = { a: Int, b: Int -> a + b }
2.
3.  fun twice(a: Int, b: Int, operation: (Int, Int) -> Int) =
4.          2 * operation(a, b)
5.
6.  fun main() {
7.      println("I can double 2+2: " + twice(2, 2, add))
8. }
```

In this specific situation, we don't gain anything from having the **add()** function defined as a function type property. We can rewrite this excerpt using a method reference:

```
1.  fun add(a: Int, b: Int) = a + b
2.
3.  fun twice(a: Int, b: Int, operation: (Int, Int) -> Int) =
4.          2 * operation(a, b)
5.
6.  fun main() {
7.      println("I can double 2+2: " + twice(2, 2, ::add))
8. }
```

Note the two colons before the name of the **add()** function in this version. They indicate that we do not want to invoke it but rather take its reference and assign it a variable, in this case, the **twice()** function's last argument.

If we want to write a lambda function directly as the **twice()** function's operation argument, we can:

```
1. twice(2, 2, { a,b -> a + b })
```

It's simple and, again, quite similar to Java.

Our **twice()** function illustrates a particularity with Kotlin's syntax regarding lambda functions. When the function type argument is the last in the function's argument list, the recommended way of writing the lambda function is outside of the argument list:

```
1. twice(2, 2) { a,b -> a + b }
```

This syntax takes some getting used to. Its benefit becomes apparent when chaining functions that take lambdas as their only argument. Let's take the **IntArray** class as an example. We can operate on the array's elements by chaining some of this class's functions:

```
1.  val result = intArrayOf(1, 2, 3)
2.      .filter {
3.          it % 2 == 0
4.      }
5.      .map {
6.          it * 2
7.      }
8. println(result) // prints [4]
```

In this excerpt, we take an array of integers, filter it so that only even numbers are present, and double these numbers. Both the **filter()** and **map()** functions take a single argument, which is a function of **(Int) -> Int** type. You can now see how Kotlin's syntax of favoring placing the last lambda argument outside of the function's argument list improves legibility in situations like this.

This excerpt also showcases another bit of syntactic sugar that Kotlin applies to lambda functions. When the function receives only one argument, it is unnecessary to use the arrow operator to declare it. Instead, use the predefined variable **it** to refer to the single argument inside the lambda's body. Short lambdas then kind of read like an English sentence: *map it (times) 2*.

All these details may look a bit overwhelming at first, but it's only a matter of practice before they become second nature.

Nullability

In Java, every variable of a non-primitive type can hold a null reference. So, it is relatively easy to cause **`NullPointerExceptions`** if we are not careful. Many good applications' code is littered with null checks:

```
1.  if ( /* something is not null */ ) {
2.      // use something
3.  }
```

And yet, the danger of trying to dereference a null reference exists. All these checks add unnecessary cognitive and runtime complexity to the application. Developers are often left wondering if a function's result can ever be null when the code's original writer did not document that.

Kotlin's type system's goal is to eliminate that uncertainty and, with it, the danger of null references; no usages of types allow null values unless expressly stated otherwise.

However, many functions return nullable values when interacting with Java code. We need to be able to receive these nullable values from Java and handle them properly.

On the other hand, we may need to write a Kotlin function that does not always return a value. In this case, we are forced to return a null value because the Kotlin language does not have an equivalent to Java's Optional class.

When a property or argument must be allowed to contain a null value, the developer must suffix the type name with a question mark:

```
1.  val name: String = null
2. // Error: null cannot be a value of non-null type String
3.
4.  val name: String? = null
5. // OK
```

The same principle applies to function return values.

This way, we can be sure that a given value can never be null unless explicitly declared.

What if the value is null and we forget to check it? When working with nullable values, the compiler forces us to perform null checks:

```
1.  val name: String? = "john"
2.  println(name.capitalize()) // COMPILATION ERROR
```

We can obtain compilable code using a standard if statement:

```
1.  val name: String? = "john"
2.  if (name != null) {
3.      println(name.capitalize())
4. }
```

Another option is to use the safe call operator (**?.**). This operator results in a null check that executes the call and returns its result if the value is not null, returning null otherwise.

```
1.  val first: String? = "john"
2.  val last: String? = null
3.  println(first?.capitalize()) // prints "John"
4.  println(last?.capitalize())  // prints "null"
```

You can combine the safe call operator with the **let()** or **run()** functions if you want to do something only if a value is not null:

```
1.  val first: String? = "john"
2.  val last: String? = null
3. first?.let { // prints "john"
4.      println(it)
5. }
6. last?.run { // does not print anything
7.      println("no")
8. }
```

It is also possible to use the Elvis operator (**?:**), which bears some resemblance to the ternary operator. When given a nullable reference, it returns the reference's value if it is not null; otherwise, it returns another value.

```
1.  val name: String? = null
2.  val trueName = name ?: "unknown"
```

We can also combine it with the safe call operator:

```
1.  val name: String? = null
2.  val nameLength = name?.length ?: -1
```

In the previous excerpt, the **nameLength** variable contains -1 if name was null.

Finally, for completeness, we can force a **NullPointerException** by using the not-null assertion operator (**!!.**). It returns the reference if it is not null and throws a **NullPointerException** otherwise:

```
1.  fun changeCase(name: String?) =
2.      name!!.capitalize()
```

The preceding function throws a **NullPointerException** if the name parameter is null. Otherwise, it returns the capitalized version of the parameters' contents.

Tip: Generally speaking, using the not-null assertion operator is a bad practice. Try to avoid null values as much as possible in your program, handling them at the earliest. You may even consider using external libraries like Arrow Core (https://arrow-kt.io), which add popular functional programming abstractions to Kotlin, such as Option, Try, or Either.

Classes and objects

It should not come as a surprise that Kotlin also supports classes, once again, with some improvements and extra features when compared to Java.

We start with the class declaration. Like in Java, a class is declared using the class keyword followed by its name and body:

```
1.  class Animal {
2.      /* ... */
3. }
```

Kotlin's syntax diverges from Java's at this point. Kotlin classes have a primary constructor and can have several secondary constructors. A default constructor that takes no parameters is generated if no constructor is defined.

The definition of the primary constructor follows the class name. The simplest way to declare it is to provide the constructor's argument list. This can then be used for property initialization.

Classes are instantiated by calling the class name as if it was a function. There is no "new" keyword in Kotlin.

```
1.  class Pet(birth: LocalDate) {
2.      val age = YEARS.between(birth, LocalDate.now())
3. }
4.
5.  val p = Pet(LocalDate.of(2005, Month.FEBRUARY, 2))
```

```
6.
7. println(p.age) // today, prints 16
```

Suppose we want to have class properties initialized directly from constructor arguments. In that case, Kotlin supports an abbreviated syntax—use the **val** keyword before the argument declaration (or **var** if the property must be mutable).

```
1.  class Pet(birth: LocalDate, val name: String) {
2.      val age = YEARS.between(birth, LocalDate.now())
3. }
4.
5.  val p = Pet(LocalDate.of(2005, Month.FEBRUARY, 2), "Scruffy")
6.
7. println("${p.name} is ${p.age} years old.")
8. // prints: Scruffy is 16 years old.
```

We must use the **constructor** keyword if we need to change the constructor's visibility or add annotations to it:

```
1.  class Application @Inject constructor(customers: Repository) {
2.      /* ... */
3. }
```

Secondary constructors are simple to declare with the **constructor** keyword. They must delegate to the primary constructor and always run after all initializations have completed. The delegation is accomplished by placing a colon after the secondary constructor's declaration and invoking the primary constructor using the **this** keyword.

Suppose we want to specify a default name if the pet's name is not known. We can do it using a secondary constructor:

```
1.  class Pet(birth: LocalDate, val name: String) {
2.      val age = YEARS.between(birth, LocalDate.now())
3.
4.      constructor(birth: LocalDate) : this(birth, "Unknown") {
5.
6.      }
7. }
```

We can also do it by providing a default value to the name argument in the primary constructor. Like Java, Kotlin supports default argument values.

```
1.  class Pet(birth: LocalDate, val name: String = "Unknown") {
2.      val age = YEARS.between(birth, LocalDate.now())
3. }
```

There is also a particular type of class in Kotlin—the data class:

```
1.  data class User(val id: Int, val name: String)
```

A Kotlin data class is a class that is meant to be used as a vehicle for transporting data. It is very convenient because the compiler automatically generates the **toString()**, **hashCode()**, and **equals()** functions, among others, by using the data class declaration.

Kotlin classes are final by default. They must be declared with the **open** modifier to allow inheritance. The same applies to any functions inside the class—they must be declared open to be overridden.

Derived classes declare their ancestor by following their declaration with a colon and the ancestor's constructor invocation:

```
1.  open class Pet(birth: LocalDate, val name: String = "Unknown") {
2.      val age = YEARS.between(birth, LocalDate.now())
3. }
4.
5.  class Dog(birth: LocalDate, name: String = "Rufus"): Pet(birth,
   name)
6.
7.  val p = Dog(LocalDate.of(2005, Month.FEBRUARY, 2))
8.
9. println("${p.name} is ${p.age} years old.")
10. // prints: Rufus is 16 years old.
```

In the preceding excerpt, we declared a new **Dog** class that extends from **Pet**. At the same time, we changed the default pet name. Note that **Dog** does not have any new properties or functions, so there is no need to declare the class's body.

The implementation of an interface is done similarly, except that interfaces do not have constructors.

In addition to constructors and properties, Kotlin classes can, of course, also contain functions. Functions declared in interfaces or open functions declared in the base class are implemented in the derived class with the override modifier, as seen in the **`MainActivity`** class earlier in this chapter.

Singletons are quickly declared in Kotlin as object declarations:

```
1.  object Directories {
2.
3.      fun findLogDirectory(): String {
4.          /* ... */
5.      }
6.
7. }
```

There is no need to have all the Java boilerplate of static instance variables to hold the object's reference.

This section was a rapid introduction to the way classes are handled in the Kotlin language. You are encouraged to explore other related Kotlin features, like nested classes, inner classes, object expressions, and companion objects.

Using a SQLite database in our application

Sometimes an Android application needs to store variable amounts of records in the device. These could be an optimization, such as a cache of downloaded data, or something related to the application's usage in that device.

A database may be a good option whenever such records have a fixed structure, are repeatable and need to be queried, or have relations between them.

Let's take a quick look at the support that Android offers for these needs.

Implementing SQLite in Android

SQLite is a small and self-contained SQL database engine. This statement means that SQLite allows building an application that takes advantage of an SQL database without a dedicated database server.

In practical terms, the SQLite libraries work with single-file databases or with in-memory databases. The latter is suitable for testing or volatile data, and the former is what most applications use to persist user data under the form of records (rows in relational database terminology).

SQLite is used as any other database engine. The application issues SQL statements to create, alter, and delete tables and insert, update, and delete rows in those tables.

The Android system includes APIs for handling SQLite databases, so no extra library is required in an Android application.

Like any other database, the first task is to design the tables. In other words, we need to define the structure of the data that we want to store.

Suppose we want to store some basic user information. We can come up with a table that contains the following data:

Column name	Data type	Notes
ID	Integer	User ID number: must be unique
NAME	String	The user name

***Table 3.1**: A possible structure for a table containing user information*

The set of information about the database tables and their columns is called the database schema. We can then translate part of this schema into our code as an object.

```
1.  object UserTable {
2.      const val TABLE_NAME = "USER"
3.      const val COLUMN_ID = "ID"
4.      const val COLUMN_NAME = "NAME"
5. }
```

Once in possession of the structure and names of the tables we need in our database, we can begin to use the Android SQLite API. We start by defining a class that extends from **`SQLiteOpenHelper`**.

```
1.  class SQLiteUserDatabase(context: Context)
2.     : SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_
          VERSION) {
3.   /* ... */
4. }
```

`SQLiteOpenHelper` provides all the machinery necessary to manage the following responsibilities:

- Storing the database file in a private location. The database file is placed in a location in the device's filesystem accessible only by the application that created it.
- Creating and updating the database only when needed.

- Versioning the database. Our code is then informed if there is a need to change the database schema.

As depicted in the previous excerpt, the class's constructor needs a database name, which is the name of the file that contains the data, a number that identifies the version of the database schema, and an Android context. It is necessary to increment the version number every time a new application is released with changes in the database schema.

It is mandatory to override two functions from the **`SQLiteOpenHelper`** class.

The first is the **`onCreate()`** function, which is responsible for creating the database schema and is executed when the database does not exist yet. Our implementation may just need to execute the necessary SQL to create the table we designed earlier.

```
1.  override fun onCreate(db: SQLiteDatabase) {
2.      db.execSQL("""
3.          CREATE TABLE ${UserTable.TABLE_NAME} (
4.              ${UserTable.COLUMN_ID} INTEGER PRIMARY KEY,
5.              ${UserTable.COLUMN_NAME} TEXT
6.          )
7.      """.trimIndent())
8. }
```

Note that we are taking advantage of Kotlin's string templates to build the SQL statement that creates our table. There is nothing fancy going on, just simple SQL statements.

The second mandatory function is **`onUpgrade()`**. It is called when the system determines that the database had been created previously, but it is an older version. This mismatch happens when we change the database version supplied in the constructor. Our implementation is then responsible for making all adjustments necessary to the existing database, ensuring that its schema is updated.

Many different upgrade strategies are possible. For the sake of demonstration, we assume that the user table is some sort of cache that is rebuilt as necessary. So, data can be discarded, and the table can simply be removed and rebuilt. Other scenarios may need a more complicated approach.

```
1.  override fun onUpgrade(db: SQLiteDatabase,
2.                         oldVersion: Int, newVersion: Int) {
3.      db.execSQL("DROP TABLE IF EXISTS ${UserTable.TABLE_NAME}")
4.      onCreate(db)
5. }
```

Now that the code for creating and updating the database is in place, it is time to work with data. Let's see how we can implement the fundamental create, update, and delete operations.

We need to call the **getWritableDatabase()** method from the **SQLiteOpenHelper** class to get a writable reference to an instance of the actual database. This call is a heavy operation, which is why the **SQLiteOpenHelper** class keeps the reference alive for us until the database is closed or its **getReadableDatabase()** counterpart is called. This way, a second write operation does not incur the high cost of opening and preparing the database.

The insertion of a new record can be implemented as follows:

```
1.  fun insertUser(user: User): Boolean {
2.      val userValues = ContentValues().apply {
3.          put(UserTable.COLUMN_ID, user.id)
4.          put(UserTable.COLUMN_NAME, user.name)
5.      }
6.      return writableDatabase.let {
7.          it.insert(UserTable.TABLE_NAME, null, userValues) != -1L
8.      }
9. }
```

We create an instance of the **ContentValues** class with the column names and the data that each column will contain. Afterward, we call the **insert()** function on the writable database instance obtained from **getWritableDatabase()**. The former only needs the table name and the values, returning the number of rows inserted or -1 if an error occurs.

A replace operation translates into a SQL **UPDATE** statement. This statement is generated by calling the **update()** function on a writable database instance. It also returns the number of rows that were affected by the operation.

```
1.  fun updateUser(user: User): Boolean {
2.      val userValues = ContentValues().apply {
3.          put(UserTable.COLUMN_NAME, user.name)
4.      }
5.      return writableDatabase.update(
6.              UserTable.TABLE_NAME, userValues,
7.              "${UserTable.COLUMN_ID}==?",
```

```
                    arrayOf(user.d.toString())
8.          ) > 0
9. }
```

Similar to its **`insert()`** cousin, it needs the table names and the values being updated. Just like a SQL **`UPDATE`** statement, it also requires the **`WHERE`** clause condition and an array of values to be used as arguments to the condition. This separation prevents the typical SQL injection security vulnerability because the condition arguments are not used directly in the condition string, making it impossible to manipulate the resulting statement.

The **`update()`** function also returns the number of rows affected by the operation.

The removal of rows can be achieved by the **`delete()`** function. Its usage is identical to its cousin **`update()`**.

```
1.  fun removeUser(userId: Int): Int =
2.     writableDatabase
3.         .delete(UserTable.TABLE_NAME, "${UserTable.COLUMN_ID}==?",
4.              arrayOf(userId.toString()))
```

Finally, the only operation missing from the basic set of operations is a reading operation. Like it is necessary to call **`getWritableDatabase()`** to obtain a writable database reference, we also need to call **`getReadableDatabase()`** to obtain a readable database reference. This call results in a heavy operation as well, and the same rules apply.

The well-known **`SELECT`** SQL statement is generated internally by the **`query()`** function. Its most straightforward usage is shown in the following excerpt, where we only need to supply the set of columns we want to read and the name of the target table. It also supports other clauses, such as the **`WHERE`**, **`GROUP`**, **`HAVING`**, and **`ORDER`**, but we do not address them here.

The **`query()`** function returns a **`Cursor`**, which is akin to an iterator because it allows us to access all items in the set of tuples returned by the **`SELECT`** statement. It provides some functions to that end, including **`moveToFirst()`**, **`moveToNext()`**, and **`moveToPosition()`** as well as **`count()`**, to retrieve the number of rows returned.

```
1.  fun listUsers(): List<User> =
2.     readableDatabase.let { db ->
3.        db.query(UserTable.TABLE_NAME, arrayOf(UserTable.COLUMN_ID,
4.                   UserTable.COLUMN_NAME), null, null, null, null,
```

```
                                null)
5.              .use { cursor ->
6.                  List(cursor.count) { index ->
7.                      cursor.moveToPosition(index)
8.                      User(
9.                          cursor.getInt(cursor.getColumnIndex(
10.                                 UserTable.COLUMN_ID)),
11.                         cursor.getString(cursor.getColumnIndex(
12.                                 UserTable.COLUMN_NAME))
13.                     )
14.                 }
15.             }
16.     }
```

The cursor must be closed once it is no longer needed. This operation releases resources associated with it and ensures that the application is free from leaks. To do so, we used the Kotlin function **`use()`** in the previous excerpt.

Testing SQLite-based databases

Proper operation of the database class can be validated using a unit test. The test needs to be instrumented because this class requires a valid Android context.

```
1.  @RunWith(AndroidJUnit4::class)
2.  class SQLiteUserDatabaseTest {
3.  @Test
4.  fun userIsInserted() {
5.   SQLiteUserDatabase(ApplicationProvider.getApplicationContext(),
         null)
6.    .use { database ->
7.      val inserted = database.insertUser(User(1, "test"))
8.      assertTrue(inserted)
9.      val users = database.listUsers()
10.     assertThat(users, contains(User(1, "test")))
11.   }
```

```
12. }
13.}
```

Note that we have created an instance of **`SQLiteUserDatabase`** with a null database name in this test. This absence of a name prevents creating a file, creating an in-memory database that is destroyed once the test finishes instead.

Note: Any class that derives from SQLiteOpenHelper also exposes a close() function. This function must be called to release resources when the database is no longer necessary; it's proper housekeeping. The current advice is to call it from the onDestroy() callback of the calling Activity.

Consequences of using SQLite directly

As we saw in the previous section, the Android API that encapsulates the SQLite libraries is quite powerful. It is also a pretty low-level API.

By using the **`SQLiteOpenHelper`** and its methods, we have the full power of SQLite at our fingertips. Still, our implementation does not have any explicit relationship to the application's data model. We must manually verify whether the conditions we chose for each operation are sound every time we modify the data model.

We need to write code to convert our data model classes to and from the API data model. In other words, we need to manually map the class attributes to table columns.

Schema migrations also pose problems. For example, we must write an update manually if the database schema needs to be modified because a new attribute was added to a data model class and its corresponding table needs a new column. We must be careful and define a migration path for all database versions.

These inconveniences do not mean that the **`SQLiteOpenHelper`** API should not be used. They mean that many applications do not need this added complexity.

Understanding object-relational mapping

Some of the preceding problems can be alleviated using a technique known as **Object-Relational Mapping** (**ORM**).

ORM intends to reduce the developer's workload by taking care of the conversion between the classes representing the data model and the database tables that implement its storage.

Looking back at the code from the SQLite in Android section, an ORM implementation would take care of handling the **`ContentValues`** creation and population, calling the getters in the Cursor, and copying data to and from the **`User`** class in general.

Besides mapping class attributes to and from table columns, an ORM can create SQL queries automatically. It often infers the table column names from the class attribute names and the table name from the class name. With this information in hand, it can build SQL statements. The complexity of these SQL statements depends, of course, on the level of sophistication of the ORM implementation.

Using an ORM has at least three significant advantages over using a database's API directly:

- The ORM can automatically validate and ensure that the database schema always matches the application's data model classes
- The same kind of validation can be applied to SQL queries and statements in general
- It can correlate classes with tables, so an ORM can often build the database schema automatically

Fortunately, there is a small ORM included in the Android Jetpack library collection. It is called Room, and the following section shows how it can be used in an Android application.

Working with a database with Room ORM

The Room library provides some benefits over the Android SQLite API. It is not a full-featured ORM like Hibernate, but it already provides some features that make the developer's life easier:

- Code generation based on annotations
- SQL queries can be validated at compile time
- Friendlier database migrations

Getting started

Naturally, the Room library is not included by default in projects generated by Android Studio because all applications don't need to persist structured data.

To get started with Room, we must check if all the following artefacts and configurations are present in the application module's Gradle build file:

- Enable annotation processing with the kapt compiler plugin. This plugin enables code generation based on annotations by the Room compiler.
- Include the Room library dependencies:
    - The runtime and compiler dependencies are mandatory in all projects, and the room-testing dependency is highly recommended

  - The room-ktx dependency is only necessary when we use Room on Kotlin with coroutines. We show a sample of this usage later on, so we must ensure that it is included.
- Enable exporting the database schemas. This detail makes testing easier later on.

Our app module's Gradle build file has received the modifications shown in the following excerpt.

In our example, the database schema files are stored in a **schemas** directory inside the application module. Don't forget to include this directory when using a version control system.

```
1. plugins {
2.     // ...
3.     id 'kotlin-kapt'
4. }
5.
6. android {
7.     // ...
8.     defaultConfig {
9.         // ...
10.         javaCompileOptions {
11.             annotationProcessorOptions {
12.                 arguments += ["room.schemaLocation":
13.                               "$projectDir/schemas".toString()]
14.             }
15.         }
16.     }
17. }
18.
19. dependencies {
20.
```

```
21.
22.     kapt "androidx.room:room-compiler:2.2.6"
23.     implementation "androidx.room:room-ktx:2.2.6"
24.     androidTestImplementation "androidx.room:room-testing:2.2.6"
25.
26.     // ...
27. }
```

After each modification to the Gradle file, Android Studio shows a notification requesting a project synchronization. This synchronization is necessary for the IDE to remain coherent with the Gradle files.

If the notification is not shown or missed, use the Sync Project option with Gradle Files in the File menu to run the Android Studio's Gradle synchronization procedure.

Defining the data entities

Once all dependencies and plugins are in place, we can start using the Room API in our project.

We start by defining an entity. Remember how we defined a data class in the SQLite example representing the table columns and data types? We do the same thing with Room, with the advantage of not needing to write SQL separately to define the table schema. Instead, we use annotations to instruct the Room compiler on how to generate the schema automatically.

Just like in the SQLite example, let's invent a **`User`** entity:

```
1.  @Entity
2.  data class User (
3.      @PrimaryKey val id: Int,
4.      val name: String,
5.      val level: Int
6.  )
```

Its definition is almost identical to the one in the SQLite example. We added a **`level`** field to help with other examples later on, but that is not the most relevant difference.

There are two requirements to define an entity using the Room library:

- The data class must be annotated with the **`@Entity`** annotation
- The entity must have a primary key

The primary key is a column that contains a value that uniquely identifies each row in the table. We use the **`@PrimaryKey`** annotation to mark that column for the Room compiler.

More complex data definitions may require a primary key composed of more than one column. These are defined differently in the **`@Entity`** annotation. An example of the latter appears later in the chapter.

The definition mentioned earlier tells the Room compiler to generate an entity with table and column names based on the class and property names and that its primary key is the ID property. To infer the database elements' names from the class definition is Room's default behavior.

Several annotation parameters can be used to customize many aspects of the table schema, starting with the table and column names.

The following excerpt changes the table name and one column name. It also instructs Room to generate an index based on the level property to optimize future queries based on this property.

```
1.  @Entity(tableName = "user")
2.  data class User (
3.     @PrimaryKey val id: Int,
4.     @ColumnInfo(name = "user_name") val name: String,
5.     @ColumnInfo(index = true) val level: UserLevel
6. )
```

Several other parameters are available in each annotation. We encourage you to look at the JavaDoc documentation for each annotation to find out more.

Creating Data Access Objects (DAO)

Although the Room library has already made it easier to define entities, its full power is unleashed when generating DAOs. These objects are used to work with the tables' data similar to the SQLite example: inserting, removing, and updating rows.

Room's compiler generates a class implementing each DAO, so it requires the DAO definition to be written as an interface declaration.

We again use annotations to instruct Room to generate code for the DAOs. Some annotations allow us to build the insert, delete, and update methods easily.

The following excerpt shows the full definition of a DAO that exposes insert and delete operations:

```
1.  @Dao
2.  interface UserDao {
3.
4.      @Insert
5.      fun insert(vararg users: User)
6.
7.      @Delete
8.      fun delete(user: User)
9.
10. }
```

Simple, right? The Room's compiler does the hard work of generating the implementation at build time.

These convenience functions can optionally return information about their execution:

- The **insert** function can return the row identifiers of the newly inserted rows. Do not confuse these with the entity's data identifier; the row identifiers are internal to SQLite.
- The **delete** and **update** functions can return the number of rows affected by the respective database operation.

So, we can extend the previous definition to include an **update** function and have the **delete** and **update** functions return the number of rows affected.

```
1.  @Dao
2.  interface UserDao {
3.
4.      @Insert
5.      fun insert(vararg users: User)
6.
7.      @Delete
8.      fun delete(user: User): Int
9.
10.     @Update
11.     fun update(vararg users: User): Int
12.
13. }
```

Our DAO is taking shape, but it is not very useful yet. It needs the ability to query the database.

Queries are also simple to declare: we use the **`@Query`** annotation in a function. This annotation then receives the SQL command as its value.

So we still need to write SQL manually, but the Room library brings added value in two ways:

- The queries are verified at compile time. The table and column names must match the entity's definition; otherwise, an error is raised.
- The code for managing the list of rows and copying data from the columns in the query's result set into the data class's properties is generated automatically.

Two lines are all it takes to generate the simplest query possible:

```
1.  @Query("SELECT * FROM user")
2.  fun findAll(): List<User>
```

We may also use parameterized queries. We declare the query parameters as arguments in the function definition and use the arguments' names preceded by a colon in the SQL:

```
1.  @Query("SELECT * FROM user WHERE level = :level")
2.  fun findAllByLevel(level: UserLevel): List<User>
```

Projections can be implemented automatically as well.

Let's imagine that we only need a list of all the user names in our table, so it is wasteful to retrieve all the other table columns. The SQL change is trivial—we just specify the column name in the statement. What about the result's mapping to a class?

The Room library allows us to declare the mapping easily. Just declare a data class containing the column definitions that we need. They must, of course, match the definitions in the original entity data class.

```
1.  data class UserName (
2.      @ColumnInfo(name = "user_name") val name: String,
3.  )
```

The DAO function declaration then becomes identical to any other query:

```
1.  @Query("SELECT user_name FROM user")
2.  fun findAllUserNames(): List<UserName>
```

Joins are mapped automatically as well.

To demonstrate the usage of queries joining tables, let's suppose that the application keeps an audit log of user actions. This audit log registers a timestamp each time a user does something.

Each user is expected to perform more than one action, so the table's primary key needs to disambiguate between them. This disambiguation is achieved by combining the user identifier with the entry's timestamp. So, we chose to use a composite key to demonstrate how these are implemented with the Room library's help.

The first step is to define the auditing entity. Its only difference from the user entity is the use of a composite primary key. Composite primary keys are not defined with annotations but have their names listed in the **`@Entity`** annotation's **`primaryKeys`** parameter.

The second step is to define the mapping for the results of the join. Just like we did before for the projection, we declare a data class matching the definitions of all columns involved.

```
1.  @Entity(tableName = "audit", primaryKeys = [ "userId", "timestamp"
    ])
2.  data class AuditEntry (
3.      val userId: Int,
4.      val timestamp: Long
5. )
6.
7.  data class FullAuditEntry(
8.      val userId: Int,
9.      val timestamp: Long,
10.     @ColumnInfo(name = "user_name") val userName: String,
11.     @ColumnInfo(name = "level") val userLevel: Int
12. )
```

Note: Note how confusing the definition of the FullAuditEntry data class has become because it contains a mixture of default and explicit column names. We did it on purpose to show that the best practice is to be as consistent as possible. Try not to mix default and explicit column naming in your entity definitions.

Now that all entity definitions are in place, we can write the SQL for the join and the DAO function signature:

```
1.  @Dao
2.  interface AuditDao {
3.
4.      @Insert
5.      fun insert(vararg entries: AuditEntry)
6.
7.      @Query("SELECT * FROM audit " +
8.              "INNER JOIN user ON audit.userId = user.id")
9.      fun findAllWithUserDetails(): List<FullAuditEntry>
10.
11. }
```

Creating the database

When we worked with the Android API for SQLite, we had to extend an abstract class provided by the API. The same principle stands when using the Room library.

Our application's database is declared by extending from the **`RoomDatabase`** class. However, Room is more powerful, so its compiler generates additional code to manage the database and needs more information.

So, we need to provide a **`@Database`** annotation listing the entities managed by this database and the current database version. The functions that provide instances of the DAOs we designed are implemented in this database class.

Our application database class looks like the following excerpt:

```
1.  @Database(
2.      entities = [ User::class, AuditEntry::class ],
3.      version = 1)
4.  abstract class ApplicationDatabase: RoomDatabase() {
5.
6.      abstract fun userDao(): UserDao
7.
8.      abstract fun auditDao(): AuditDao
9.
10. }
```

Again, note that the same concept of versioning exists for databases managed by the Room library. Migrations are not entirely generated automatically. We address the topic of database schema migrations with Room later in this chapter.

The actual database is created using the **databaseBuilder()** function in the Room object. It leverages the builder pattern and requires at least three parameters: an Android context, the type reference of the database class, and the database name.

Creating a database managed by Room is also an expensive operation. The current recommendation is to follow the singleton pattern, reusing a single instance of the database across all application activities. The most common implementation of a thread-safe singleton pattern is shown in the following excerpt:

```
1.  companion object {
2.
3.      private const val DATABASE_NAME = "app.db"
4.
5.      @Volatile private var instance: ApplicationDatabase? = null
6.
7.      fun getDatabase(context: Context): ApplicationDatabase =
8.          instance ?: synchronized(this) {
9.              instance ?:
10.                 Room.databaseBuilder(
11.                         context.applicationContext,
12.                         ApplicationDatabase::class.java,
13.                         DATABASE_NAME)
14.                     .build()
15.                 .also { instance = it }
16.         }
17.
18. }
```

The double-checked locking pattern is used to ensure that only one instance of the database is created in the presence of multiple threads attempting to use it concurrently. This pattern consists of obtaining an explicit lock only if it is necessary to modify the shared resource, rechecking the condition after the lock is obtained to prevent creating the resource twice. It is used in combination with the **@Volatile** annotation, so a memory barrier is placed on write operations to the annotated property. We encourage you to read more on these concurrent programming topics if you are not familiar with them yet.

Using database views

We can replace the simple query with a join that we built earlier with a database view.

The procedure is simple—we place a **@DatabaseView** annotation on the mapping data class and move the SQL query.

```
1.  @DatabaseView(
2.   "SELECT audit.userId, audit.timestamp, user.user_name, user.level " +
3.   "FROM audit " +
4.   "INNER JOIN user ON audit.userId = user.id")
5.  data class FullAuditEntry(
6.     val userId: Int,
7.     val timestamp: Long,
8.     @ColumnInfo(name = "user_name") val userName: String,
9.     @ColumnInfo(name = "level") val userLevel: UserLevel
10. )
```

The DAO function signature remains identical, but its associated SQL query now refers to the view name:

```
1.  @Query("SELECT * FROM fullauditentry")
2.  fun findAllWithUserDetails(): List<FullAuditEntry>
```

A database view is also a database entity, so it must be registered in the **@Database** annotation's **views** parameter as well.

```
1.  @Database(
2.     entities = [ User::class, AuditEntry::class ],
3.     views = [ FullAuditEntry::class ],
4.     version = 1)
5.  abstract class ApplicationDatabase: RoomDatabase() {
6.
7.     abstract fun userDao(): UserDao
8.
9.     abstract fun auditDao(): AuditDao
10.
11. }
```

The advantage of having a database view instead of a longer SQL query is that a view is a database object, so the longer SQL query is processed and optimized only once by the database. In contrast, a traditional SQL query may need to be processed every time it is used. Therefore, a database view may be more efficient.

Object references are not supported

Let's get this out of the way—the Room library does not support object references in the database entities.

Other ORM frameworks allow the embedding of entity relationships in the entity definitions. Hibernate, for example, allows such embeddings. Looking back at the previous **AuditEntry** example, we could use this feature to directly insert a reference to the **User** entity in the **AuditEntry** entity. With the proper annotations, the framework would issue the appropriate query to the database when the embedded **User** reference is queried. It would eliminate the need for the **FullAuditEntry** class and associated view.

However, this technique hides the actual cost of seemingly inoffensive property accesses. In the context of applications with a user interface, these accesses tend to eventually make their way into calls made within the UI logic itself because they make it easier for the UI and database models to merge. But this is a bad practice.

Making database calls within the UI in an Android application is particularly bad because the UI thread is blocked for extended periods, which often causes rendering issues.

The Room developers decided to address both problems by not supporting object references in entities. This way, it is necessary to explicitly declare the entities and their queries, making it clear that a database roundtrip is required while separating the UI and persistence models.

This feature brings a problem. What about the objects that do not represent entity relationships but are helpful, such as enums or time management classes?

Converting object references to database types

Going back to the previous **AuditEntry** example, you may have noted that its timestamp property's type is **Long**. This type allows us, for example, to store the timestamp as the number of milliseconds since the epoch (January 1st 1970).

But then we need to be aware of precisely what that number means every time we access the property. It would be much better if we could use, say, **Instant** as it is then clear that the timestamp property represents a fixed point in time.

```
1.  @Entity(tableName = "audit", primaryKeys = [ "userId", "timestamp"
```

```
   ])
2.  data class AuditEntry (
3.      val userId: Int,
4.      val timestamp: Instant
5. )
```

However, a compile-time error is generated if we just change the type. The message is quite helpful. It reads: *Cannot figure out how to save this field into the database. You can consider adding a type converter for it.*

While it does not directly support object references in entity definitions, the Room library allows us to define type converters so we can have the objects in our classes but store their supported representation in the database instead.

```
1.  class InstantConverters {
2.
3.      @TypeConverter
4.      fun fromTimestamp(value: Long?): Instant? {
5.          return value?.let { Instant.ofEpochMilli(it) }
6.      }
7.
8.      @TypeConverter
9.      fun instantToTimestamp(instant: Instant?): Long? {
10.         return instant?.toEpochMilli()
11.     }
12.
13.}
```

The previous excerpt defines a converter that can convert an Instant into a Long value for database storage and back. This converter now needs to be registered, so the Room library becomes aware of its existence. We register it using the **`@TypeConverters`** annotation in our database class.

```
1.  @Database(
2.      entities = [ User::class, AuditEntry::class ],
3.      views = [ FullAuditEntry::class ],
4.      version = 1)
```

```
5.  @TypeConverters(InstantConverters::class)
6.  abstract class ApplicationDatabase: RoomDatabase() {
7.
8.      abstract fun userDao(): UserDao
9.
10.     abstract fun auditDao(): AuditDao
11.
12. }
```

Different database types can be used in converters; the choice depends on the most suitable type for representing the converted object.

We could replace the **Int** used for the **User** entity's level property for an **enum** and store it in the database as a **String**; enums have nice **name()** and **valueOf()** methods that we could use, and they are string-based.

Using database migrations

We have already addressed the need to have database migration procedures to cope with application upgrades that require different database schemas.

Remember that the **@Database** annotation requires a version number to be defined? Although the Room library does not handle migrations automatically, it does manage migration sequences. We can register migration handlers with their respective version numbers, and they are called as necessary in the correct order.

This way, we ensure that users don't lose data when they upgrade the application.

Suppose we wanted to add a password hash field to our **User** entity.

The first step would be to add the property to the entity:

```
1.  @Entity(tableName = "user")
2.  data class User (
3.      // ...
4.      @ColumnInfo(name = "pwd_hash") val passwordHash: String
5.  )
```

Then, we must change the database version in the **@Database** annotation:

```
1.  @Database(
2.      // ...
3.      version = 2)
```

These two changes are sufficient to define a new database schema with an accompanying version number.

Let's declare a migration path from version 1 to version 2, along with the necessary SQL statements. Note that we are adding a non-null column, so we must define a reasonable default value:

```
1.  abstract class ApplicationDatabase: RoomDatabase() {
2.  // ...
3.   companion object {
4.   // ...
5.   internal val MIGRATION_1_2 = object : Migration(1, 2) {
6.    override fun migrate(database: SupportSQLiteDatabase) {
7.      database.execSQL(
8.       "ALTER TABLE user ADD COLUMN pwd_hash TEXT NOT NULL DEFAULT
              ''")
9.    }
10.  }
11. }
12.}
```

Now, we must register it with the **`Room`** database builder using the **`addMigrations()`** function in the existing **`getDatabase()`** function:

```
1.  fun getDatabase(context: Context): ApplicationDatabase =
2.      instance ?: synchronized(this) {
3.          instance ?:
4.              Room.databaseBuilder(
5.                      context.applicationContext,
6.                      ApplicationDatabase::class.java,
7.                      DATABASE_NAME)
8.                  .addMigrations(MIGRATION_1_2)
9.                  .build()
10.             .also { instance = it }
11.     }
```

Now your application knows how to upgrade a version 1 database to version 2.

Running queries outside of the main thread

If we try to use the Room library from within the UI thread, we get a friendly **`IllegalStateException`**. It tells us that it "cannot access database on the main thread since it may potentially lock the UI for a long period of time."

So, we are forced to move all database operations outside of the main thread.

There are a few ways to do so. One involves using Kotlin coroutines along with the Android Jetpack's **`ViewModel`** library. The **`ViewModel`** class was designed to store UI data independently of the activity's lifecycle, so it survives configuration changes that cause the activity to be destroyed and recreated. It implements the MVVM design pattern we discussed in *Chapter 1, Building an Application with Android Studio and Java*.

As an example, we can declare a view model as follows. The **`findUsers()`** function launches a new coroutine within the context of the main thread. It then runs the DAO function in a dispatcher dedicated to I/O tasks, which means it runs in an I/O thread. The result of the DAO function is passed to the consumer function, back in the main thread.

```
1.  class UsersViewModel(application: Application)
2.          : AndroidViewModel(application) {
3.
4.      private val database = ApplicationDatabase.getDatabase(
5.          getApplication<Application>().applicationContext)
6.
7.      fun findUsers(consumer: (List<User>) -> Unit) {
8.          viewModelScope.launch {
9.              val users = withContext(Dispatchers.IO) {
10.                 database.userDao().findAll()
11.             }
12.             consumer(users)
13.         }
14.     }
15. }
```

One possible usage of our view model in the activity's **onCreate()** function could be as follows:

```
1.  val model: UsersViewModel by viewModels()
2. findViewById<Button>(R.id.someButton).setOnClickListener {
3.     model.findUsers {
4.         // do something
5.     }
6. }
```

The following dependencies are required in our application's Gradle configuration to use these view model functions and classes:

```
1. dependencies {
2.
3.     // ...
4.
5.         implementation  "androidx.lifecycle:lifecycle-viewmodel-
               ktx:2.3.0"
6.     implementation "androidx.activity:activity-ktx:1.2.1"
7. }
```

You can find more information about the **ViewModel** class at **https://developer.android.com/topic/libraries/architecture/viewmodel** and about using Kotlin coroutines in Android at **https://developer.android.com/kotlin/coroutines**.

Testing Room-based databases

We have seen one possible way of testing the SQLite-based implementation of a database in the previous sections. The tests for our Room-based implementation are not significantly different.

The basic principle remains the same. We still need to build an instrumented test because the database code still requires an Android context. After all, the Room library continues to be backed by SQLite.

Afterward, we build our instrumented Android JUnit test and create an in-memory database with the **inMemoryDatabaseBuilder()** method:

```
1. db = Room.inMemoryDatabaseBuilder(context,
2.             ApplicationDatabase::class.java).build()
```

Finally, we use our DAOs to execute operations and verify that the database's data matches our expectations. A possible test of our user DAO is exemplified in the following excerpt:

```
1.  val dao = db.userDao()
2.
3. dao.insert(
4.     User(1, "regular", UserLevel.NORMAL),
5.     User(2, "super", UserLevel.SUPERUSER)
6. )
7.
8.  val users = dao.findAll()
9. assertThat(users, containsInAnyOrder(
10.        User(1, "regular", UserLevel.NORMAL),
11.        User(2, "super", UserLevel.SUPERUSER)
12.))
```

Migrations are another part of our database implementation that must be tested carefully. We must ensure that any existing database remains usable after an application upgrade.

This is made possible by the Room testing library. It provides a helper class that can recreate older schemas to run the migrations and verify that the database remains stable afterward.

The first step is to ensure that each version's schemas are available and that the testing library has been added to our dependencies. We did that already while setting up our Gradle definitions in the *Getting started* section. With those settings, the Room library generates the schemas automatically every time the application is compiled.

The second step is to make the saved schema definitions available for tests. The following Gradle configuration excerpt adds the schema directory to the source directories used for Android tests:

```
1. android {
2.     // ...
3.     sourceSets {
4.         androidTest.assets.srcDirs +=
5.                 files("$projectDir/schemas".toString())
6.     }
7. }
```

The final step is to write the testing code. We use the **`MigrationTestHelper`** class to do the following:

- Create the older schema version
- Insert test data in the database according to the older schema version
- Run the appropriate migration paths
- Validate that the test data was converted correctly, if applicable

We need to register our own application database class with the **`MigrationTestHelper`** class, which is typically used in conjunction with a JUnit **`@Rule`**:

```
1.  @get:Rule
2.  val helper: MigrationTestHelper = MigrationTestHelper(
3.      InstrumentationRegistry.getInstrumentation(),
4.      ApplicationDatabase::class.java.canonicalName,
5.      FrameworkSQLiteOpenHelperFactory()
6. )
```

The test database with the old schema is created with the **`createDatabase()`** method, and we can take the opportunity to insert the test data at the same time.

Note that the automatically generated DAOs cannot be used because we are working with different database schema versions. The DAOs always expect the latest schema. The preceding excerpt demonstrates a test that validates only one migration path, but we might need to verify the intermediate migration steps. For example, when migrating to version 3 from version 1 through version 2.

```
1. helper.createDatabase(databaseName, 1)
2.     .apply {
3.         insert("user", SQLiteDatabase.CONFLICT_FAIL,
4.             ContentValues().apply {
5.                 put("id", 1)
6.                 put("user_name", "previous")
7.                 put("level", "NORMAL")
8.             }
9.         )
10.        close()
11.    }
```

The migration paths under test can then be executed with **runMigrationsAndValidate()**. This function also verifies that the resulting schema matches the structure extracted from the entity declarations.

```
1.  val db = helper.runMigrationsAndValidate(databaseName, 2, true,
2.      ApplicationDatabase.MIGRATION_1_2)
```

After this step, we can use the database instance to verify that the test data has been migrated correctly using the SQLite API, as described in the corresponding section.

Testing database implementations may be tedious, but it is essential to ensure that the application's data remains available and coherent.

Conclusion

The fundamentals of the Kotlin language were explained in this chapter. You should now be able to read the code examples presented in this chapter and other sources.

We presented the main features of the two ways of implementing an embedded SQL database in an Android application, along with their advantages and disadvantages, and introduced the Object-Relational Mapping concept. This presentation's goal was to allow you to choose the best approach for your application. We also stressed the importance of testing the application's code with examples applied to the specific case of databases.

The next chapter explores the subject of artificial intelligence. It highlights some aspects of this field of research's history, along with some of its philosophical and practical challenges. It also intuitively explains some common algorithms used with machine learning and how machine learning is not the same as artificial intelligence but a part of it.

CHAPTER 4
An Overview of Artificial Intelligence and Machine Learning

Introduction

Let's look at the term artificial intelligence and how it became an extensive study discipline, accompanied by a historical and philosophical perspective. Starting from this high-level perspective, we then isolate the field of machine learning from this extensive corpus of research to understand the challenges researchers aim to overcome.

Structure

- The past and future of artificial intelligence
- Intelligent systems
- Machine learning
- Problems affecting machine learning

Objectives

We aim to offer a good foundation to understand the history behind today's research. We also intend to clarify that this work is not without challenges, and it is not only

technical but also philosophical and practical. Some pitfalls must be kept in mind so that practitioners can avoid them.

The past and the future of artificial intelligence

We believe that humans have dreamed about machines ever since they figured out how to use tools. These tools, after all, made many tasks easier and faster. Other tasks were impossible to accomplish without tools, so these technological advancements expanded humanity's horizons and increased its desire for further innovations.

Some of these desired innovations were dreamed of under the form of complex machinery with varying degrees of autonomy, like today's concept of robots.

Depictions of artificial intelligence in literature

How far back does the dream of intelligent or autonomous machines go? Literature is a good medium to understand the thought patterns and desires of those who came before us.

Jules Verne's 1880 steam elephant from *The Steam House* is a beautiful description of a powerful four-legged all-terrain machine, but it still needed an operator. We may need to look for intelligent machines elsewhere.

It is thought that Homer's Illiad was written circa 8th century BC. This classical Greek poem describes the automated gates that grant access to the god's heavenly citadel, the devices controlling the Hephaestus god's furnaces, and the wheeled tripods that can travel to the assembly of gods and return to Hephaestus's house at his bidding.

In their paper titled Homer's Intelligent Machines (2019), Genevieve Liveley and Sam Thomas argue that these devices already display some weak autonomy: Homer describes them as being ordered to work instead of being operated. They also point out that Homer already described autonomous ships in his Odyssey through Alcinous's speech:

Tell me also your country, nation, and city, that our ships may shape their purpose accordingly and take you there. For the Phaeacians have no pilots; their vessels have no rudders as those of other nations have, but the ships themselves understand what it is that we are thinking about and want; they know all the cities and countries in the whole world, and can traverse the sea just as well even when it is covered with mist and cloud, so that there is no danger of being wrecked or coming to any harm.
– Odyssey 8:521-585, translation by Samuel Butler (via Project Gutenberg), emphasis ours.

The Greeks were not the only ones to dream of autonomous machines, as other mythologies, like the Hindu, also have accounts of automatons.

Moving closer to our time, the 18th-century French philosopher Étienne Bonnot de Condillac argued in his book *Traité des sensations* that all human ability and knowledge result from transformed sensations. For the sake of his argument, he imagined a statue animated by an empty soul that is fed individual sensations—pieces of knowledge—one at a time until it obtains, or learns, the possible human knowledge and passions. We can, of course, discuss what he meant by an empty soul: did he mean the *ghost in the machine*, as Gilbert Ryle (1949) defined it? Either way, he was already postulating the ways a humanoid object might learn. The similarities with the training process of machine learning models are striking.

There is a myriad of examples of semi-autonomous, fully autonomous, and even intelligent robots in literature as we progress through the 20th century.

Some well-known examples come from Isaac Asimov's positronic robots. He started writing short stories with such characters in 1939, condensing some in a narrative published as his book *I, Robot* in 1950. What makes Asimov's robots unique are the Three Laws of Robotics: dogmas that every positronic robot must obey, that are designed to prevent them from harming humans. Asimov clearly had in mind the possibility of robots being ordered—or even deciding on their own—to cause harm and so, built a set of rules to prevent it. Even today, many have expressed the concern that artificial intelligence might, one day, turn against its human creators. The theme of artificial beings causing humanity's demise has since been explored at length in many science-fiction books and Hollywood movies; the most marketed might well be the Terminator series.

We could not end our series of examples of artificial intelligence in literature without mentioning two of the most well-known contemporary intelligent artificial beings: Marvin The Paranoid Android from Douglas Adams' 1978 radio comedy *The Hitchhiker's Guide to the Galaxy* and HAL-9000 from Arthur C. Clarke's 1968 book *2001: A Space Odyssey*.

HAL is not a robot but an intelligent operating system that controls a spaceship and can learn from its human interactions. Unfortunately for the crew, it learned about human emotions in the worst possible way: facing its upcoming disconnection because of some malfunctions, it decides to defend itself by killing the crew. It became known for the phrase, *I'm sorry, Dave. I can't do that*.

Marvin is an extremely intelligent humanoid robot (thus an Android) that becomes depressed because it has never received any task that would require it to use more than a fraction of its brainpower to accomplish. According to itself: *The best conversation I had was over forty million years ago and that was with a coffee machine*. Quite a coffee machine that must have been!

Artificial intelligence is not a new idea

We have looked at just a few examples demonstrating that humans have been dreaming of creating autonomous and intelligent beings. With all these dreams, it would only be a matter of time before people would attempt to go from theory to practice.

And so they did. One exciting example of a contraption that fell halfway between dream and reality was The Turk, also known as the Mechanical Turk or Automaton Chess Player. Built in 1770 by Wolfgang von Kempelen, it depicted a human torso dressed in oriental clothes at a table with a chessboard. It was supposedly able to win a game of chess against any human opponent. Although many praised its mechanical abilities at chess, it was an elaborated hoax. A human would sit inside the table and operate the machine to play the game.

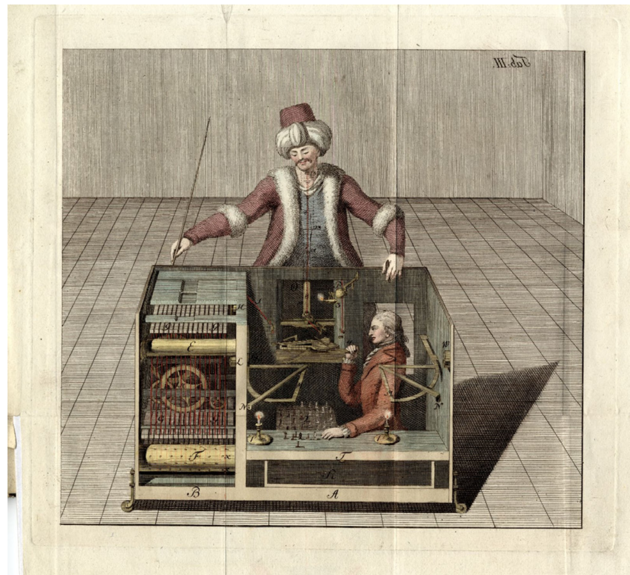

Figure 4.1: *An illustration of how the Turk might work.*
© UB der HU zu Berlin, Historische Sammlungen: 3639 v:F8

Choosing chess to demonstrate The Turk's alleged capabilities makes sense because it is believed that the game requires thinking or intelligence to be played skillfully.

In 1912, a real chess-playing automaton was built by Leonardo Torres y Quevedo. It played an end game of king and rook against king. It always played the side with the king and rook and would always win against its human opponent. It was pretty advanced for its period (Vigneron, 1914).

Chess was also used as a study vehicle in early AI work. Claude Shannon pointed out in his paper *Programming a Computer for Playing Chess* (1950) that it's an interesting

problem as it is very well defined in terms of the rules and outcomes. Although it can be solved using classic dictionary or game theory algorithms (the minimax), the number of possible moves in the game—estimated at 10^{43}—makes it impractical to solve using only those algorithms. No chess-playing program was created at that time, but Shannon demonstrated that a computer could be programmed to play a satisfactory game of chess.

The 1950s and 1960s brought along better tools (languages like Lisp) and more computational power to further research efforts. A checkers-playing program was written by Arthur Samuel in the 1950s. It learned through experience and improved its abilities from playing against humans and other computers.

Note: An artificial chess player, IBM's Deep Blue, eventually beat Grandmaster Gary Kasparov in 1997.

One of the first approaches to building intelligent programs was the so-called expert systems. These systems' goal was to solve problems by emulating a human expert's reasoning over a body of knowledge. The systems were initially based on rules formulated using conditional or predicate logic, depending on the programming language in use (Lisp vs Prolog, for example). There were great expectations regarding these systems' capabilities, and many performed exceptionally well in their specialized domains, but not without some criticism. An example is the 1984 paper by John McCarthy titled *Some Expert Systems Need Common Sense*. Research in this field still continues, refining the way these systems learn and infer their conclusions.

An algorithm called the perceptron algorithm was invented earlier, in 1958, and was implemented in hardware to recognize images. Unfortunately, it failed to meet the high expectations at the time because it could not be trained to recognize many classes of patterns.

The disappointments related to the perceptron, expert systems, and other techniques led to two periods that came to be known as the AI winters, when machine learning research stagnated: in the late 70s and between the late 80s and early 90s. Today, we would recognize these times as hype and disappointment cycles.

It is said that the 21st century brought about the digital age, and with it, new means of digital storage. The digital age also brought with it digital services, which meant enormous amounts of data were being generated and collected by and from worldwide users.

The term Big Data was coined by the end of the 20th century or the start of the 21st due to this data generation growth.

Note: Big Data is a term used to designate a field of work specializing in processing large amounts of data or highly complex data. Big Data systems come into play when traditional data processing software can no longer handle data. By large amounts, we mean amounts in the zettabyte scale (1000^7 bytes), according to Sarah Everts in her article Information Overload (2016).

Research in artificial intelligence saw a big boost with Big Data's birth because datasets could now be larger and more relevant. It also became easier to ask users to produce data by employing digital questionnaires and surveys. In addition, technological advancements allowed the creation of dedicated **graphical processing units** (**GPUs**). Since most graphical processing is inherently mathematics, the rise of GPUs added a significant amount of computational power to accelerate calculations.

ImageNet is a brilliant example of these new capabilities. In 2006, the AI researcher Fei-Fei Li struggled with one of the biggest machine learning problems—overfitting and underfitting or generalization (we discuss these problems later in the chapter). She realized that part of the answer resided in the size of the training dataset. The bigger and more varied the dataset, the better.

The Princeton psychologist George Miller had started a project called WordNet in the late 1980s. WordNet was intended to be like a dictionary, but the words would be related to each other hierarchically instead of alphabetically. So dog was a specialization of canine, which, in turn, was a specialization of mammal, and so on. Given that WordNet's structure made it a good candidate to be read by machines, Li started the ImageNet project to use WordNet's definitions as classification labels for the most significant image database they could build. Images would be extracted from the Web and given to humans for classification.

Due to the project's sheer scale, they eventually decided to use the Amazon Mechanical Turk service for image classification. Thousands of human workers would be involved in image classification every year.

The first version of the dataset was published in 2009 along with its accompanying research paper. Over 3 million images were labeled in two and a half years. Today, the dataset comprises over 14 million images spread over tens of thousands of meaningful concepts, and it has contributed to the development of computer vision systems that can identify images reliably. The project's website is at **http://image-net.org**.

The methodology behind ImageNet was an enormous success, and other projects also used large datasets with accurate classifications for their purposes. The popular voice assistants Siri (Apple, 2011) and Google Assistant (Google, 2016) also used identical techniques to learn how to recognize their user's speech.

Note: The Amazon Mechanical Turk service's name is a direct reference to The Turk because it is also a mechanized interface to work done by humans. Like a human was playing chess inside The Turk, many humans perform the Amazon service's tasks.

Artificial intelligence is, however, more than learning how to classify data.

In 1968, Marvin Minsky was researching the semantic representation of knowledge and the representation and modification of plans, which is said to have influenced the creation of new programming paradigms.

Knowledge representation and reasoning is the field of artificial intelligence dedicated to finding ways to represent information about the world so that a machine can use it to make decisions or complete tasks.

Today's research, like intelligent agents and autonomous vehicles, shows that many reasoning methods need to be combined to achieve artificial intelligence. It takes more than just learning how to register and classify information; it is necessary to know what to do with that information in each situation.

Thinking about intelligent systems

Working in artificial intelligence also means having an opinion on whether a machine can be considered intelligent at all. In other words, this research field also encompasses studying and understanding the nature of thought and reasoning.

Alan Turing, who is considered the father of theoretical computer science and artificial intelligence, wrote an influential paper titled *Computing Machinery and Intelligence* (1950). In the paper, Turing described a test that could be administered to a machine to prove whether the machine was capable of demonstrating intelligence.

Turing suggested a few versions of the test. One of those, called the "imitation game", has three participants: a human questioner, a human questionee, and a computer. The questioner does not know which one of the participants is human and can only communicate with them in written form. The computer's job is to convince the questioner that it is human (or that it is the human in the game). The version that is usually called the "Turing test" is slightly different; in this case, the interviewer needs to determine which of the participants is the machine.

The intended outcome is the same in any version. The computer must be intelligent if the interviewer cannot determine which participant is artificial. This affirmation was based on the argument that communication is the primary way to ascertain intelligence.

This argument is not without controversy. For example, in his 1980 paper *Minds, Brains and Programs*, John Searle defended that it is impossible to use the Turing

test to prove that a machine can think. He argued that a machine could manipulate symbols—written speech—without understanding their meaning. So, it could not be thinking in the same sense humans do.

John Searle offered a thought experiment to prove his point: the Chinese room. A way to describe this thought experiment is as follows. Picture a person inside a room with no knowledge of Mandarin but in possession of an instructions book containing many Mandarin sentences. The room has one slot in the wall through which written messages can be exchanged. People outside can send messages in Mandarin, and the person on the inside can reply by copying the appropriate responses from the book. From the outside, it would appear that the person in the room knows the Mandarin language.

Searle argues that the person in the room does not differ from a computer. The person is merely following instructions and does not understand any of the messages or their replies. Therefore, it cannot be concluded that a computer could understand them either.

Other thinkers have made their arguments, and this line of thought is not far from the philosophy of mind. One may also argue that the hard problem of consciousness is not too distanced from these discussions. Either way, Searle is not arguing against the possibility of a super-intelligent machine being built, but only that such a machine cannot think or fully understand what it is experiencing or communicating.

Note: The hard problem of consciousness is explaining why we say that some personal experiences feel like something. These subjective and conscious experiences are also called qualia. The mere existence of this problem is disputed by many philosophers and accepted by many others.

By now, it is clear that the field of artificial intelligence also includes disciplines other than engineering. The previous examples and discussions have already shown influences from game theory, mathematics, statistics, logic, philosophy, and even psychology. Communication theory, dealing with the encoding and transmission of information, and linguistics for natural language processing are included as well. Additionally, biology plays an important role—one of the significant sources of enthusiasm in research and fiction is replicating the way the human brain works and learns.

Defining machine learning

From the examples, tests, and thought experiments we discussed so far, it is evident that a machine must know what to do with the information it receives to be considered intelligent.

This knowledge may be represented as hardcoded rules, like the early expert systems we mentioned earlier. This representation is possible when the space state of the problem or application is known or adequately constrained and can be used to solve several problems related to planning or scheduling, for example.

Note: Planning and scheduling are also fields of work in the scope of the broader concept of artificial intelligence, just like machine learning.

Difficulties appear when the space state is primarily unknown or has a vast number of variables or possibilities. The game of chess is a good example. We mentioned earlier that the enormous number of possible moves made it a good example of how classical algorithms provided sub-optimal results.

This kind of unsatisfactory results usually stemmed from the enormous amount of time necessary to arrive at a solution. Naturally, researchers turned their attention to solving this issue.

Simon (1972) stated that "in the absence of an effective method guaranteeing the solution to a problem in a reasonable time, heuristics may guide a decision-maker to a very satisfactory, if not necessarily optimal, solution." Heuristics are techniques to optimize algorithms by reducing the computation time necessary to find a solution, often at the expense of absolute accuracy.

The usage of heuristics means that, in many scenarios, an approximate result is acceptable. However, it is still necessary to describe all rules or movements exhaustively before the algorithms can tackle their problems, which can be problematic or even impossible for humans. What if the computer could itself learn how to solve a problem?

It is believed that Arthur Samuel coined the term machine learning in 1959 while researching the application of the technique to a checkers-playing computer program. His paper *Some studies in machine learning using the game of Checkers* shows his verification of the fact that the program could learn how to play checkers better than its programmer.

Many machine learning techniques were developed once it was established that sufficiently approximated solutions to problems were acceptable and that it was possible to teach computer programs how to find them.

The fundamental premise of machine learning is that a computer program can learn by example, just like humans. Such programs are based on machine learning models.

A machine learning model is a program that takes some data as its input and outputs a prediction. It uses one or more algorithms to calculate its predictions, and these algorithms have parameters. The number, type and semantics of each parameter depends on the algorithm, but they all have one thing in common—their values can

be estimated from data. An example of a model parameter would be the slope used in the linear regression algorithm described in the next section.

For example, a model could predict a dog's breed based on its height and coat color. In this case, the expected output could be the confidence levels that the model assigned to each breed it knows, for example, 90% poodle and 10% German shepherd. Teaching the model requires feeding it data, sometimes large amounts. This process is called training. The model then adjusts its parameters iteratively based on the input data. These adjustments are necessary to optimize the algorithm's criteria and, ultimately, the model's predictions.

There are a few broad categories of machine learning algorithms:

- **Supervised learning**: This type of algorithm requires the expected output to be known with regard to the input data. The algorithms then tailor themselves to produce that output when new unseen data is received. The recognition of handwritten digits is an example of supervised learning; the output is known and includes the digits from 0 to 9.

- **Unsupervised learning**: As the name implies, the lack of supervision means that the program does not know the expected output. It must train itself to discover, for example, unknown patterns in data. An example would be detecting anomalies in data, such as an excessively high energy consumption that could indicate a faulty device.

- **Reinforcement learning**: This algorithm type resembles the way some animals are trained. It interacts with an environment to fulfill a goal, receiving positive or negative feedback. It learns the best way to navigate its problem space by trying to maximize the positive feedback (rewards). An example would be playing a game—good moves are rewarded, and wrong moves are penalized.

All algorithms known today do not fit in one category, and some applications require the usage of techniques from more than one category. That said, these are a good starting point.

One characteristic common to all of them is that they are not always meant to give a perfect answer. These answers are usually accompanied by a degree of confidence, indicating how close to a perfect result they are expected to be.

We now examine some well-known machine learning algorithms from an intuitive standpoint. Understanding the mathematics behind the algorithms is not required in many situations, but it helps to have an intuitive grasp of their mechanisms.

The data used to train some algorithms is already labeled, that is, each data tuple is associated with what we want to predict. So, these are supervised learning algorithms. Suppose we wanted to predict a dog's breed based on its height and

coat color. The different races are our labels, and the height and coat color are the input features in this case.

Linear regression algorithm

This algorithm was borrowed from statistics and is the most straightforward algorithm used in machine learning.

In a nutshell, it aims to find the line that best represents the linear relationship between two or more numeric variables, hence the term linear.

It can be better understood with the help of an example. Take a look at the following graph.

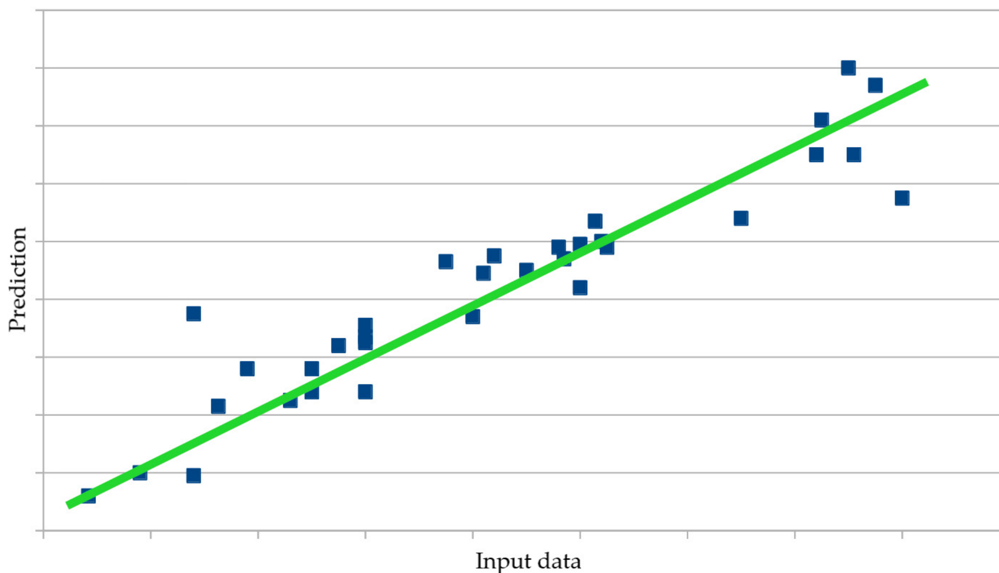

Figure 4.2*: An example of linear regression applied to some data*

The thick green line represents an approximation of the relation between the independent variable on the X-axis and the dependent variable on the Y-axis. The algorithm aims to find the line that predicts the value of Y when only the value of X is known, hence the term "regression"—modeling a target value based on one or more independent values.

A simple observation of the graph tells us that the line touches some examples with remarkable precision but is distant from others. This distance represents the error, or the lack of accuracy, of any predictions calculated using the line's parameters and dependent variables. A line can never touch all data points, so the objective is to find the line that yields the lowest error possible.

Many procedures were developed to discover the necessary parameters for this. The most well-known procedure is probably using the mean squared error to calculate the cost and a gradient descent technique to adjust the parameters.

The cost, also known as loss, can be defined as the difference between the actual and predicted values. The mathematical function used to calculate it depends on the algorithm and is called the cost function.

The gradient descent technique can be explained quite simply. Imagine a person descending a slope of a U-shaped valley in thick fog. They do not know when they reach the bottom; instead, they stop once they no longer notice a significant decrease in altitude with each step.

In this example, the altitude is our cost. The algorithm stops adjusting the parameters once the difference between consecutive error margins is no longer relevant. It may be desired to stop this adjustment earlier, for example, to avoid overfitting. The problem of overfitting is discussed in this chapter's *Opportunities to improve machine learning* section.

Clustering methods

Clustering, or clustering analysis, is the task of finding examples (data points) similar to others and grouping them by their similarities. Some groups—called clusters—are discovered after the task is complete, and each example is assigned to one of those groups.

The following figure is an example of a set of data submitted to a fictional clustering algorithm. This algorithm could have found the clusters as marked.

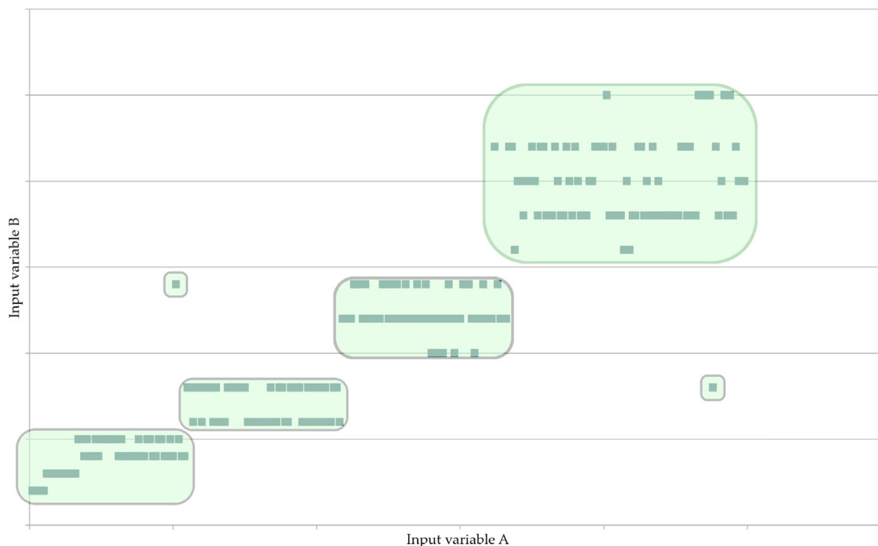

Figure 4.3: *Possible data submitted to a clustering algorithm*

Clusters are formed according to a measure of their similarity. This metric is obtained from the features that define the examples. The algorithm is unsupervised when there are no labels in the dataset, and it can be used, for example, to analyze market segmentation.

When the dataset contains labels, the supervised clustering algorithm becomes a classification algorithm. It then aims to determine which class the new data points (examples) belong to.

There are many clustering algorithms as well. One of the reasons is that it is not easy to arrive at a shared definition of a “cluster,” and another is that all algorithms are not suitable for all applications.

Neural networks

It is common knowledge that the brain contains numerous neurons, and each neuron is connected to other neurons. These connections are called synapses, which form a network and transmit stimuli between neurons and thus, across this network.

The first research work that effectively laid the foundation for modern theories of the brain was published by Alexander Bain (1873) and William James (1890). It is not surprising that, two centuries later, researchers in artificial intelligence became interested in applying such knowledge in their field. After all, if brains were composed of neurons and could learn, why couldn’t identical networks be built into a computer?

Once brought within the context of machine learning, neural networks also became known as **Artificial Neural Networks** (**ANN**) or **Simulated Neural Networks** (**SNN**). As the name implies, they mimic the way neurons communicate in the brain.

They are composed of artificial neurons organized in layers. There is one input layer, one output layer, and one or more intermediate or hidden layers.

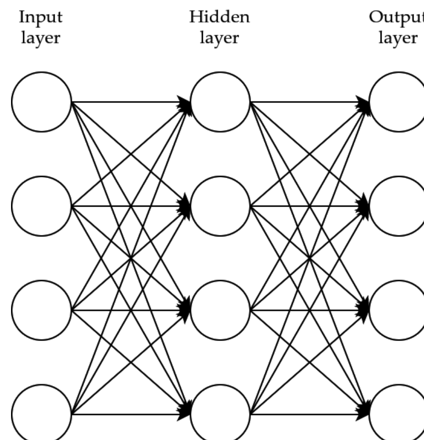

***Figure 4.4**: A neural network with one hidden layer*

Each neuron can be seen as a linear regression model with an output threshold (or bias) and weighted inputs. The weights are used to define the importance of each input in determining the output. The bias can be used to determine whether the neuron should activate. An activation function calculates the neuron's activation state based on one or more inputs and is affected by the bias value, which anticipates or delays the neuron's activation.

The output of one neuron is used to feed the input of the neurons in the following layer. The neurons in the output layer provide the result of the network's calculations. The arrows in *figure 4.4* represent this data flow between neurons.

Note: There are different types of neurons: perceptron, which output either zero or one, and sigmoid neurons, which output values between zero and one. Neural networks are typically built with the latter.

Neural networks can be trained with labeled or unlabeled data, making them suitable for supervised and unsupervised learning. The goal of their training remains the same—to minimize their output errors with relation to the training data. The classical training method uses the gradient descent technique to find the point of minimal error.

The network illustrated in *figure 4.4* is called a feed-forward network, and it is the most straightforward design. There are several other neural network architectures, like **Convolutional Neural Networks** (**CNN**) and **Recurrent Neural Networks** (**RNN**).

RNNs save a layer's output and feed it back to the input to calculate the following output. So, the typical type of input data for an RNN is a sequence of closely related data items. RNNs are strong in applications that require some memory. Language processing is a good example: sentences form sequences of closely related data items (words).

CNNs contain specialized layers and are primarily used in computer vision applications to classify images and perform object detection. They can also be used for unsupervised learning based on images. These specialized layers are called convolution layers and are used for feature detection, helping reduce the number of parameters.

These advanced neural network architectures are often implemented in the form of deep neural networks.

Deep neural networks

Figure 4.4 represents a neural network with one hidden layer. Neural networks with one or two hidden layers can also be called shallow neural networks. On the other hand, a neural network with many hidden layers is called a "deep" neural network.

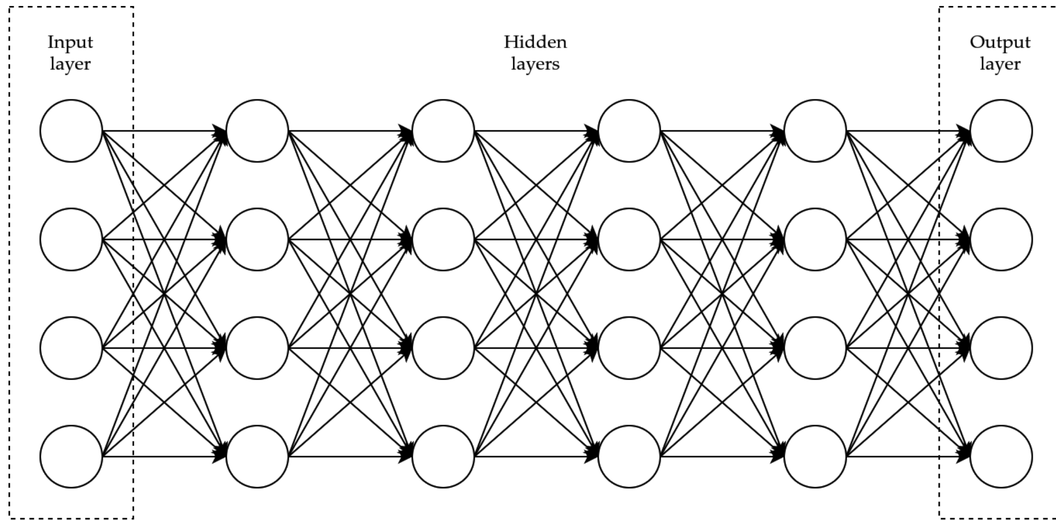

***Figure 4.5**: An example of a deep neural network*

All layers behave in the same way and are defined by the same mathematical equations as the shallow neural network. The only difference is that there are many of them.

Deep neural networks have been shown to provide much more accurate results for several applications as compared to shallow neural networks. The reason behind this is not clear yet, but any improvements firmly depend on the amount of training data available.

This lack of clarity illustrates one of the problems regarding neural networks that researchers are dealing with today—the difficulty of understanding how the network arrived at any particular result. In other words, the problem is a lack of explainability.

Opportunities to improve machine learning

Machine learning is not perfect and has both internal and external problems. We point out those that are, in our opinion, the main improvement opportunities currently under research.

Improving performance and accuracy

One of the reasons why machine learning only took off in the 21st century, especially in what concerns neural networks, is that the technology did not provide sufficient computational resources before that.

Training a machine learning algorithm with large amounts of data is computationally intensive work. Unfortunately, the **central processing units** (**CPUs**) or processors available at the time were designed for general-purpose computing and not for the number-crunching tasks required for these calculations.

So, training operations would take enormous amounts of time, limiting research efforts to what was practical.

Only with the arrival of GPUs and other specialized processors optimized for complex algebraic calculations did researchers have the resources to use more data or try better algorithms.

Better algorithms usually mean a new machine learning application or better accuracy with an existing application.

A machine learning algorithm's accuracy is measured by comparing the algorithm's output with the expected output for a testing dataset. In other words, data for working with machine learning is often divided into two or three distinct datasets: one for training, another for testing, and another one for validation. Just like the training dataset, the testing and validation sets contain data and the expected outcomes.

The validation dataset is used to estimate the model's behavior during the model construction phase, when the model's structure is still in its early stages and we don't want to spend a considerable amount of time training the model with the complete training dataset.

After the algorithm has been trained, it is fed the testing dataset. This set contains data that the algorithm has never seen, and its output is a reliable measure of its accuracy. The closer it is to the results expected from the testing set, the more accurate the algorithm is.

The algorithm or dataset needs to be adjusted if the results are not acceptable for the application's purpose.

It goes without saying that the search for an accurate model that generalizes well is a moving target, and it is as limited by technology as by the existing models. Research on new technology to support new models continues.

Note: There are two relevant concepts regarding a model's training accuracy: overfitting and underfitting. A model "overfits" when it is excessively close to the training data and thus, not generalizing well. Conversely, it "underfits" when it is not close enough to the data. It is easier to create an overfitting model than its opposite. In other words, if the model results' accuracy, when applied to the testing data, is significantly lower than the accuracy found during training, it means that it does not generalize well to "new" data and is probably overfitting.

Working toward explainability

Engineers can usually explain how or why a system works. In fact, most professionals do not feel comfortable if they do not understand the system themselves.

With simpler machine learning algorithms like statistical linear regression or clustering, we can explain how any given output was calculated because the model's parameters and its equations are known.

Hidden layers are introduced once a neural network or deep neural network is brought into the picture. Due to these hidden layers, it is no longer possible to derive one equation that explains the output: the model itself generates additional parameters internally as it is being trained. The hidden nodes represent these additional parameters. It is also very difficult, or even impossible, to figure out each feature's weight in determining the output.

These models do not expose their internals, which is why machine learning models can be known as **black box** models.

A model should be understandable so that engineers and data scientists can know, for example, if decisions are made ethically (see the following bias problem). Research is ongoing to find ways to explain how machine learning models arrive at a result. The resulting field of research is called **Explainable Artificial Intelligence** (**XAI**).

Avoiding bias

How can a machine learning algorithm be biased? After all, machines have no emotions or prejudices per se, and mathematics is not inherently biased, right?

In this chapter, we have seen that machine learning algorithms are trained by feeding them with data. This data is collected and curated by humans, and humans can indeed be biased, sometimes quite explicitly.

For example, the dataset may include features that correlate a person's gender, race, or beliefs with their purchasing power. Suppose a model is being trained to calculate a person's risk profile for credit purposes. Once those features are input into the model, its training considers them and produces results that discriminate against a population section.

It is also possible to produce a biased model by accident or oversight.

An example was the **Scottish Qualifications Authority** (**SQA**'s) attempt to use a machine learning system to adjust students' grades in 2020. These students had no exams due to the COVID-19 pandemic. Instead, their teachers recommended individual grades based on their experiences with the pupils during the school year. SQA was worried that it might lead to inflated grades, so they built a machine learning model to prevent such inflation. Unfortunately, this model considered

historical school performance and geographical data, so students saw their grades lowered because they lived, or had lived, in an area with a history of lower academic performance. In total, about 25% of the grades were lowered. The results were ultimately overturned, and the teacher's original recommendations were used.

Another example is ImageNet. As we mentioned earlier in this chapter, ImageNet is a dataset of about 14 million images labeled by tens of thousands of workers via the Amazon Turk. The dataset structure is sound, but those workers have transposed their biases into the labels they used to classify the images. Some workers also used derogatory words or racist slur in their classifications. Kate Crawford and Trevor Paglen raised this problem in *Excavating AI: The Politics of Training Sets for Machine Learning* (September 19, 2019 – **https://excavating.ai**). Ryan Steed and Aylin Caliskan have also addressed this matter in their paper *Image Representations Learned With Unsupervised Pre-Training Contain Human-like Biases* (2021). A project was started to improve the ImageNet dataset by crowdsourcing label review.

The problem of perpetuating bias via artificial intelligence is real. Some believe that each AI system will be a product of its time, just like people, so it is impossible to eliminate bias. Nevertheless, the problem is real, and efforts are underway to raise awareness of the issue and propose ways to detect and correct biased algorithms. Examples are the Fairlean project (**https://fairlearn.org/**) and the AINow research institute (**https://ainowinstitute.org**).

Thinking about security

All computer systems responsible for some relevant process or those that process valuable data are at risk of being cyberattack targets. Artificial intelligence systems are no exception to this rule.

A type of threat called an adversarial attack concerns machine learning algorithms. This attack works by making slight modifications in the algorithm's input data, causing the algorithm to misclassify or misbehave. These modifications are often invisible or ignored by humans performing the same task, but an ML algorithm is affected by them.

With the growing number of ML-enabled applications in systems that work and produce effects on the physical world, these attacks can yield serious consequences. Other types of attacks may be imperceptible even to human users and may be highly targeted: attackers can also manipulate the systems for their gain instead of harming others.

Autonomous or semi-autonomous vehicles operating on public roads are a common target for security researchers. Nir Morgulis et al. have found a technique that allowed them to print real traffic signs that deceive a machine learning model used in actual vehicles into misreading speed limits (*Fooling a Real Car with Adversarial*

Traffic Signs, 2019). Researchers at McAffee have identified similar vulnerabilities in systems used in Tesla vehicles of the time (*Model Hacking ADAS to Pave Safer Roads for Autonomous Vehicles*, 2020).

Probably because of the lack of explain-ability we discussed earlier, there is still no explanation for why adversarial attacks are so effective.

The current advice for any machine learning system exposed to real-world data, or data that could be manipulated, is the same as that for any classical system—to perform threat modeling and execute penetration testing. Such activities against ML are indeed much more complex, but they should be evaluated nevertheless.

Note: Penetration testing is a term used in software engineering and white-hat hacking. It represents the activities intended to gain unauthorized access to a system or impair it in the same way as a malicious attacker, but for research purposes. Defenses can then be built against such attacks using the insights gathered.

Conclusion

We have looked at the dream of artificial intelligence since antiquity until today. This work helped show that humanity has always dreamed about creating intelligent entities and has found ways to evolve its dream into reality. Science fiction continues to push the limits of imagination, and we can only look at our past experiences to speculate about how today's fiction shapes tomorrow's reality.

Practitioners in this field must know that machine learning is facing new challenges and threats, from societal and cultural bias to malicious attacks.

Basic machine learning algorithms have been explained from an intuitive perspective. This intuition is essential for understanding the work we do in the following chapters—training and using machine learning models to make predictions and classify images.

CHAPTER 5
Introduction to TensorFlow

Introduction

This chapter explains TensorFlow's fundamental concepts. It is one of the most popular frameworks that can be used to implement and train machine learning models.

At the same time, it shows how it can be installed in the three most commonly used operating systems. Python is the language used in TensorFlow's documentation and most examples, so we also cover Python installation.

Structure

We cover the following topics in this chapter:

- Installing TensorFlow
- Tensors
- Variables
- Graphs
- Simple model training
- Conclusion

Objectives

Upon completing this chapter, you should be able to read simple TensorFlow examples, experiment with TensorFlow functions in the Python shell, and understand the TensorFlow code used in the rest of this book.

Installing TensorFlow

TensorFlow is an open-source platform for machine learning developed by Google and initially released in 2015. It has gained popularity over the years. Its latest versions include features like support for JavaScript in browsers, the lightweight TensorFlow Lite library for mobile and edge devices that we use in this book, and a platform for deploying ML pipelines.

Other popular frameworks, like PyTorch or MxNet, are available, but we chose TensorFlow for this book primarily because of TensorFlow Lite and its imperative programming style.

Like any other framework, TensorFlow must be installed before it can be used. TensorFlow needs a Python environment to work, and its version must match TensorFlow's requirements.

At the time of writing, TensorFlow 2 has the following requirements. The most current ones can be found at **https://www.tensorflow.org/install/pip**.

- A 64-bit operating system, at least:
    - o Ubuntu 16.04
    - o Windows 7
    - o Mac OS 10.12.6
    - o Raspbian 9.0
- Python version between 3.6 and 3.8

  TensorFlow 1.15, the latest update in the 1.x version series, has slightly different requirements in what concerns the Python version:
- Python version 3.7 or below

We use TensorFlow 2 as much as possible in this book. Its API has been consolidated, and the execution model is now one step closer to the imperative model that Python, Java, or Kotlin programmers already know about.

However, some examples may require TensorFlow 1.15 for a couple of reasons. Either they require supporting libraries that are not yet updated, or they are too complex to train in a typical developer computer within a reasonable amount of

time. In the latter case, we try to use pre-trained models that may only be compatible with a specific version of TensorFlow.

The current recommendation is to use a version of Python compatible with both and try to keep the TensorFlow version as isolated from the system as possible. From the preceding list of requirements, we use Python 3.7.

Preparing Python

If you already have Python installed in your system, feel free to skip this section. Just ensure that the version is 3.7 or 3.6.

If you have never worked with the Python programming language, do not worry. It is easy to understand its basics from the perspective of a Java or Kotlin developer. We explain any usage of the Python language when it may not be clear for persons with a Java background.

One crucial difference between Python and Java (or Kotlin, for that matter) is that indentation matters in Python. There are no curly brackets to delimit code blocks and scopes; indentation is used instead.

Windows and Mac OS

Installer packages are available at **https://www.python.org/downloads/** if you need to install Python on a Windows or Mac OS machine. You only need to download the latest package of version 3.7 and run the installer.

Do not forget to add the Python executable to your PATH system variable if asked to do so by the installer.

Note: When you try to run Python from the command line in Windows 10, you may be prompted to disable the "python" and "python3" aliases from Settings > Manage App Execution Aliases. Go ahead and disable them: this only happens because we are not using the Python distribution available at the Microsoft App Store, and it is not a problem.

After installation is complete, you should have the Python executable available for the command prompt (terminal).

Alternative Mac OS installation (advanced)

If you already use the Homebrew package manager, you may use it to install Python 3.7 alongside the system's existing version. We also use the **`pyenv`** tool to manage different Python environments.

1. `brew install pyenv`
2. `pyenv install 3.7.10`

```
3. pyenv global 3.7.10
```

Add the following to your shell configuration file (**~/.zshrc** or **~/.bash_profile**, depending on the shell you use) to complete the pyenv setup:

```
1. if command -v pyenv 1>/dev/null 2>&1; then
2.  eval "$(pyenv init -)"
3. fi
```

Quit the terminal program and start it again or reload the configuration file. Once you run the **python --version** command, you should see it prints version 3.7.10.

Ubuntu Linux

The procedure becomes more complicated in an Ubuntu system because different Ubuntu versions already include different Python versions. For example, Ubuntu 18 comes with Python 2.7, whereas Ubuntu 20 comes with Python 3.8; neither is compatible with both TensorFlow versions.

Run the following commands to find out which version of Python is installed in an Ubuntu system:

```
1. python --version
2. python3 --version
```

We are using Ubuntu 20, upgraded from previous Ubuntu versions, so your results may vary. In particular, you may or may not have an executable named **python3** in your system, but this is not a problem.

If Python version 3.7 is not reported in your system, it needs to be installed.

The easiest way to install another Python version without causing any conflicts with the one used by your Ubuntu system is to install it from its source code.

First, we install the generic dependencies necessary for building Python:

```
1. sudo apt-get install build-essential checkinstall \
2.       libreadline-gplv2-dev libncursesw5-dev libssl-dev \
3.       libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev \
4.       libffi-dev zlib1g-dev lzma-dev
```

Then, we download, unpack, build, and install Python 3.7:

```
1. wget https://www.python.org/ftp/python/3.7.10/Python-3.7.10.tgz
2. tar zxf Python-3.7.10.tgz
3. cd Python-3.7.10
```

```
4. ./configure --enable-optimizations
5. sudo make altinstall
```

Once all commands complete successfully, you should have a **`pyhton3.7`** executable installed in **`/usr/local/bin`**. Test your setup by asking for Python's version:

```
1. /usr/local/bin/python3.7 --version
```

Creating virtual environments for TensorFlow

We recommend setting up a virtual Python environment for using each TensorFlow version.

Note: When using the following "python" command, be sure to use the correct command for your system! For example, in Ubuntu, you may use "python", "python3", or "/usr/local/bin/python3.7", depending on the results from the previous section.

It is necessary to install the virtualenv module first:

```
1. python -m pip install virtualenv
```

Then, change to a directory you want to use for working with TensorFlow and create one virtual environment for each major TensorFlow version.

It is common to have more than one version of Python installed in Ubuntu Linux systems. Do not forget to specify the full path to version 3.7 if it is not your default version or if you followed the preceding detailed method:

```
1. python -m virtualenv .tensorflow1 --python=/usr/local/bin/python3.7
2. python -m virtualenv .tensorflow2 --python=/usr/local/bin/python3.7
```

You may omit it otherwise. It is not required when Python is installed with pyenv (Mac OS), and is in the PATH (Windows):

```
1. python -m virtualenv .tensorflow1
2. python -m virtualenv .tensorflow2
```

Once the virtual environments are created, we can activate the virtual Python environment we want to use. It is mandatory to activate the correct environment for the correct TensorFlow version before using it.

First, change to the directory you chose while creating the virtual environments and activate the environment of your choice. The exact activation commands differ for every operating system.

Windows

For example, use the following if you want to activate the environment for TensorFlow 1:

```
1. C:\Users\vveloso\Work>.tensorflow1\Scripts\activate
2.
3. (.tensorflow1) C:\Users\vveloso\Work>
```

Note that the name of the currently active environment is placed in the prompt.

Ubuntu Linux and Mac OS

You should use the following if you wish to activate the environment for TensorFlow 1:

```
1. source .tensorflow1/bin/activate
```

Installing TensorFlow in the virtual environments

On Windows 7 or higher, you may need to install Microsoft's C++ redistributable libraries first. TensorFlow needs them to work. You can find a link to the correct download page at **https://www.tensorflow.org/install**.

TensorFlow 2

Change to your TensorFlow working directory and activate the virtual environment **`.tensorflow2`**.

We use the Python package installer (pip) to install TensorFlow. Let's update it first:

```
1. pip install --upgrade pip
```

Now, we install the latest version of TensorFlow:

```
1. pip install tensorflow
```

Note: At the time of writing, the latest version of TensorFlow was version 2.5.1. Some of the code examples may need to be adjusted to work on more recent versions of TensorFlow. If you need to specify the version, write "tensorflow==2.5.1" instead.

TensorFlow 1.15

Change to your TensorFlow working directory and activate the virtual environment **`.tensorflow1`**.

Let's update pip again as we have switched environments:

```
1. pip install --upgrade pip
```

Now, we install TensorFlow 1.5. Note that we must specify the version:

```
1. pip install tensorflow==1.15
```

Note: You may take advantage of faster TensorFlow executions if you have a CUDA®-compatible GPU card and run Windows or Ubuntu Linux.

Configuration is more complicated in this case and may only work for one TensorFlow version. It is not required for the work we do in this book, but you can visit https://www.tensorflow.org/install/gpu for further details.

Verifying the TensorFlow version installed

TensorFlow 1.15

Open a new terminal window, change to your TensorFlow working directory, and create a file with the following contents. We suggest naming it **`check-tf-1.py`**:

```
1. import tensorflow.compat.v1 as tf
2. hello = tf.constant("hello TensorFlow!")
3. sess = tf.Session()
4. print(sess.run(hello))
5. print(tf.__version__)
```

Activate the **`.tensorflow1`** environment and run the program.

You should see a message and the TensorFlow version at the end of the program's output. There is also other output related to TensorFlow's inner workings, and it depends on the operating system's configuration.

```
1. ❯ python check-tf-1.py
2.
3. b'hello TensorFlow!'
4. 1.15.0
```

TensorFlow 2

Create a file with the following contents. We suggest naming it **`check-tf-2.py`**:

```
1.  import tensorflow as tf
2. msg = tf.constant('hello TensorFlow!')
3. tf.print(msg)
```

```
4. print(tf.__version__)
```

Activate the **.tensorflow2** environment and run the program. Like with the previous program, you should see a message and the TensorFlow version at the end of the program's output. The exact version number may be different, but it must begin with 2.

```
1. ❯ python check-tf-2.py
2.
3. b'hello TensorFlow!'
4. 2.4.1
```

Note: All TensorFlow code examples in the following sections are meant to be used with TensorFlow 2.

The Python interpreter shell

Most code snippets in the following sections can be executed in one go as a program file, like the version verification, or they can be executed line by line in the Python interpreter shell. This shell is helpful for quick experiments.

You enter the shell by entering the **python** command in your terminal after activating your TensorFlow 2 virtual environment. *Figure 5.1* shows a Python shell started on the author's system by following these steps:

```
❯ source .tensorflow2/bin/activate
❯ python
Python 3.7.10 (default, Apr 10 2021, 12:07:21)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

***Figure 5.1**: Starting the Python shell*

Notice that the shell prompt is the **>>>** sign, so any Python code line you see in the following sections starting with **>>>** was typed in the interpreter shell.

Do not forget to import the TensorFlow package before calling any of its functions in the shell. Just input **import tensorflow as tf** before you start working.

Tensors

TensorFlow earned its name because most of its work, if not all, is based on data structures called tensors.

In linear algebra, tensors are generalizations of matrices for higher-order data and can represent data in ways that would be difficult otherwise (Xuan Bi et al., *Tensors in Statistics*, 2021). When one starts searching on the Internet or in libraries for study materials about tensors, most of the results focus on electromagnetism, relativity, and other physics-related domains.

Thankfully, understanding the tensors used in TensorFlow is much more straightforward. Simply speaking, a tensor can be seen as an array that can have multiple dimensions. Even with this formulation, we need to know some terminology from linear algebra.

Tensors can be used, for example, to represent the following types of data structures:

- Scalars, that is, single numbers
- Vectors
- Matrices

Figure 5.2 is a visual representation of arrays with varying number of dimensions representing the corresponding tensors.

The number of dimensions of a tensor is called its order. So, a zero-order tensor represents a scalar, a first-order tensor a vector, a second-order tensor a matrix, and so on.

TensorFlow uses shapes to describe a tensor's dimensions. A tensor's rank is the number of dimensions it has, and its shape is the length of each axis or dimension. It is often represented as an array whose length corresponds to the tensor's order, and each element is the size of the corresponding dimension.

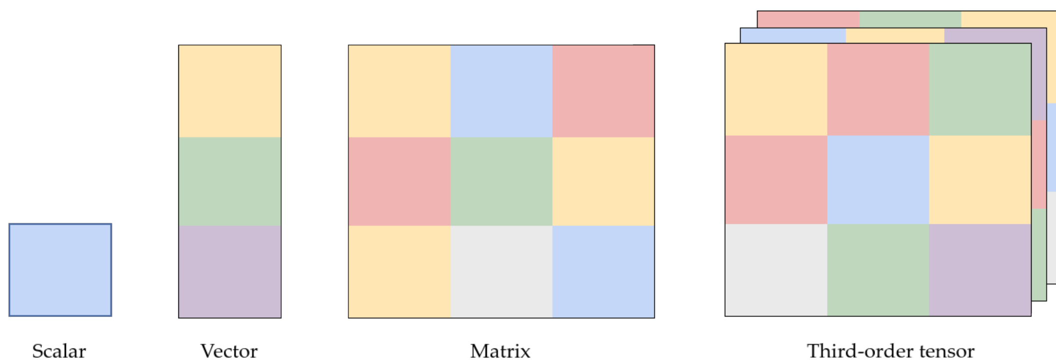

***Figure 5.2**: Tensors with different ranks*

As an example, the tensors illustrated in the preceding figure have the following shapes:

- The scalar has a shape []

- The vector has a shape [3] because it contains three elements
- The matrix has a shape [3, 3] because it is a 3x3 matrix
- The third-order tensor has a shape [3, 3, 3]

All these tensors were given dimensions of the same size to keep the illustration simple. However, it is typical and even expected to find real-world tensors with dimensions of different sizes.

That being said, how do we declare tensors in TensorFlow, and what kind of data can they contain?

As a typical array, all elements of a tensor need to be of the same data type. Tensors support storing signed and unsigned integers, floating-point values, complex floating-point numbers, Booleans, and strings.

Creating tensors with the constant() function

TensorFlow supports creating tensors in several ways. The simplest is the declaration of constant tensors. This is done using the **constant()** function. Consider the following snippet:

```
1.  import tensorflow as tf
2. c1 = tf.constant([1, 2, 3])
3. c2 = tf.constant([[1.4, 2.6, 4], [9.6, 10.7, 7.1]])
4. c3 = tf.constant(['a', 'vector'])
5. print(c1, c2, c3, sep='\n')
```

The preceding snippet should produce the following output:

```
1. tf.Tensor([1 2 3], shape=(3,), dtype=int32)
2. tf.Tensor(
3. [[ 1.4  2.6  4. ]
4.  [ 9.6 10.7  7.1]], shape=(2, 3), dtype=float32)
5. tf.Tensor([b'a' b'vector'], shape=(2,), dtype=string)
```

Note the shape and data type (TensorFlow shows the data type as **dtype**) of each tensor. We created tensors with Python integers, floating-point values, and strings. They show up as int32, float32, and string TensorFlow data types. TensorFlow also supports other data types. Most of them are listed in *Table 5.1*:

Type	TensorFlow data type
Integers (signed and unsigned)	int8, int16, int32, int64 uint8, uint16
Floating-point numbers	float32, float64
Strings	String
Boolean values	Bool
Complex numbers	complex64, complex128

***Table 5.1**: TensorFlow data types*

The **dtype** parameter can be used to specify the exact TensorFlow data type to apply during tensor creation:

```
1. c1 = tf.constant([1,2,3], dtype=tf.float64)
```

Creating tensors with generated data

In addition to the **constant()** tensor initialization function, we can initialize tensors with generated data.

They can be filled with zeros, ones, or any other value. The corresponding functions are, logically, **zeros()**, **ones()**, and **fill()**. The former two only need the desired tensor's shape as an argument, while the latter also needs the value that is to be placed in each tensor element.

```
1. c1 = tf.zeros([3])
2. c2 = tf.ones([3,2])
3. c3 = tf.fill([2, 3, 1], 42.0)
```

If we were to print these three tensors, the output would be similar to the following:

```
1. tf.Tensor([0. 0. 0.], shape=(3,), dtype=float32)
2.
3. tf.Tensor(
4. [[1. 1.]
5.  [1. 1.]
6.  [1. 1.]], shape=(3, 2), dtype=float32)
7.
8. tf.Tensor(
9. [[[42.]
```

```
10.   [42.]
11.   [42.]]
12.
13. [[42.]
14.   [42.]
15.   [42.]]], shape=(2, 3, 1), dtype=float32)
```

One-dimensional tensors containing sequences of values can be created with the **linspace()** and **range()** functions.

The **linspace()** function generates a fixed-size sequence of values within an interval, such that the first and last are the interval limits and the intermediate values are evenly spaced. It takes three parameters: the first and last values and the number of values to generate.

```
1. c1 = tf.linspace(3, 8, 5)
2. c2 = tf.linspace(3, 8, 6)
```

The tensors created by the preceding lines look as follows:

```
1. tf.Tensor([3. 4.25 5.5 6.75 8.], shape=(5,), dtype=float64)
2.
3. tf.Tensor([3. 4. 5. 6. 7. 8.], shape=(6,), dtype=float64)
```

The **range()** function also creates a tensor with a sequence of values, but it uses a step definition (the interval between values). It accepts the first and upper values and the interval between them as arguments. Unlike **linspace()**, **range()** does not include the upper value in its output.

Compare the following two usages of **range()** with **linspace()**. Both want values in the [1, 5) interval, but the first specifies a step of 1, whereas the second needs a step of 0.5:

```
1. c1 = tf.range(1, 5, 1)
2. c2 = tf.range(1, 5, 0.5)
```

The resulting tensors would be as shown below:

```
1. tf.Tensor([1 2 3 4], shape=(4,), dtype=int32)
2.
3. tf.Tensor([1.  1.5 2.  2.5 3.  3.5 4.  4.5], shape=(8,), dtype=float32)
```

Some functions create tensors with randomly generated values, differing in their distribution. Two of those are **`random.normal()`** and **`random.uniform()`**. Both of these functions receive the shape of the desired tensor as their first parameter. The former chooses random values with a normal distribution and allows the user to define the distribution's mean and standard deviation, so the probability of a value being chosen falls in a curve similar to *figure 5.3*. The latter chooses them with a uniform distribution, so all values are equally probable. Instead of the mean and standard deviation, it allows the user to define the minimum and maximum value limits.

Possible usages of these functions are illustrated in the following code snippet:

```
1. c1 = tf.random.normal([100], mean=0.0, stddev=1.0)
2. c2 = tf.random.uniform([100], minval=0.0, maxval=3.0)
```

There are other functions for initializing tensors with random values as well. For details on these and other functions, take a look at the TensorFlow documentation at **https://www.tensorflow.org/api_docs/python/tf**.

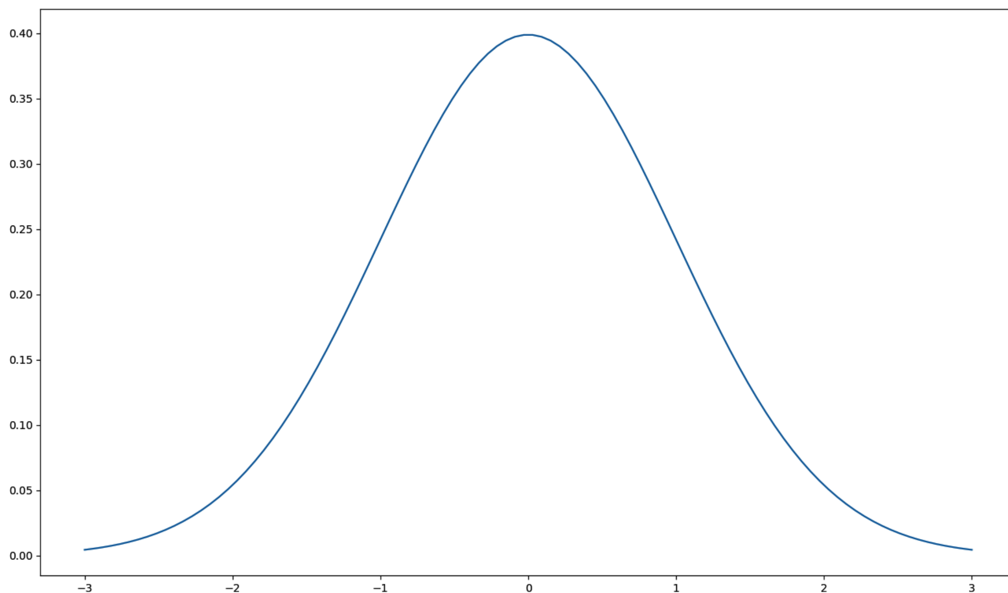

***Figure 5.3**: A normal distribution*

Operations with tensors

So far, we have not seen any practical differences between tensors and traditional multi-dimensional arrays. This section changes that.

TensorFlow defines many differentiable mathematical operations on tensors, which makes sense because machine learning algorithms are mathematical by nature, as we saw in *Chapter 4, An Overview of Artificial Intelligence and Machine Learning*.

TensorFlow's mathematical functions include exponential and logarithmic operations, rounding, comparisons, reductions, and many more. Listing them all is outside the scope of this book, but you can refer to the TensorFlow documentation at the previously mentioned address to know more about them.

The functions performing basic mathematical operations can be described as follows:

Function	Operation
`add(x, y)`	Adds two tensors element-wise
`subtract(x, y)`	Subtracts two tensors element-wise
`multiply(x, y)`	Multiplies two tensors element-wise
`divide(x, y)`	Divides the elements of two tensors using the Python division style: integer division results in a floating-point value
`scalar_mul(scalar, y)`	Multiplies the elements of a tensor by a scalar

Table 5.2: *Basic mathematical operations (x and y are tensors)*

The following listing contains a partial transcript of a Python shell session that demonstrates these functions' usage and results:

```
1.  >>> tf.add([1,2], [3, 4])
2. <tf.Tensor: shape=(2,), dtype=int32, numpy=array([4, 6],
   dtype=int32)>
3.  >>> tf.subtract([1,2], [3, 4])
4. <tf.Tensor: shape=(2,), dtype=int32, numpy=array([-2, -2],
   dtype=int32)>
5.  >>> tf.multiply([1,2], [3, 4])
6. <tf.Tensor: shape=(2,), dtype=int32, numpy=array([3, 8],
   dtype=int32)>
7.  >>> tf.divide([1,2], [3, 4])
8. <tf.Tensor: shape=(2,), dtype=float64, numpy=array([0.33333333,
   0.5 ])>
9.  >>> tf.scalar_mul(5, tf.constant([1,2]))
10.<tf.Tensor: shape=(2,), dtype=int32, numpy=array([5, 10],
   dtype=int32)>
```

Besides element-wise operations, TensorFlow supports linear algebra operations like matrix and vector operations:

Function	Operation
`linalg.matmul(a, b)`	Multiplies two matrices `a` and `b`
`linalg.matvec(a, v)`	Multiplies a matrix `a` by a vector `v`
`linalg.eye(num_rows, num_columns)`	Creates an identity matrix
`linalg.inv(a)`	Computes the inverse of an invertible square matrix `a`

Table 5.3: *Some matrix and vector operations*

The following listing taken from a Python shell session provides the respective demonstration:

```
1.  >>> a = tf.constant([[1,2,3],[4,5,6]])
2.  >>> b = tf.constant([[1,2],[3,4],[5,6]])
3.  >>> tf.linalg.matmul(a,b)
4. <tf.Tensor: shape=(2, 2), dtype=int32, numpy=
5. array([[22, 28],
6.        [49, 64]], dtype=int32)>
7.
8.  >>> v = tf.constant([1,2,3])
9.  >>> tf.linalg.matvec(a,v)
10.<tf.Tensor: shape=(2,),dtype=int32, numpy=array([14, 32],
   dtype=int32)>
11.
12. >>> tf.linalg.eye(3,3)
13.<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
14.array([[1., 0., 0.],
15.       [0., 1., 0.],
16.       [0., 0., 1.]], dtype=float32)>
```

The usual operators "+", "-", "*" and "/" are also defined for tensors.

When using tensors on arithmetic operations, they must have compatible shapes. For example, adding two tensors of different shapes, like a matrix and a vector, is impossible. In this case, the smaller tensor can be broadcasted into a compatible

shape. Vector broadcasting transforms a vector to another shape by replicating its elements as necessary. This replication is the reason for the operation's name.

The **broadcast_to()** function performs this action. It receives the original vector and the desired shape as parameters and outputs the broadcasted vector.

Usage of the **broadcast_to()** function is illustrated in the following listing:

```
1.  >>> tf.broadcast_to([1,2,3], [3,3])
2. <tf.Tensor: shape=(3, 3), dtype=int32, numpy=
3. array([[1, 2, 3],
4.        [1, 2, 3],
5.        [1, 2, 3]])>
```

Finally, we can also reshape and slice tensors and extract individual elements from them:

Function	Operation
slice(t, begin, size)	Extracts a slice from a tensor `t` starting at the `begin` index, with `size` shape
reshape(t, shape)	Creates a new tensor with the same elements as tensor `t` but with the specified shape
reverse(t, dimensions)	Reverses specific dimensions of tensor `t`
gather(t, indices, axis)	Extracts slices from an axis of tensor `t` according to indices
gather_nd(t, indices)	Extracts slices from tensor `t` according to the shapes defined as indices

***Table 5.4**: Some tensor transformation functions*

Tensor slicing can be demonstrated as follows. Remember that the **size** argument of the **slice** function is a shape, so it contains the size of each dimension in the desired slice.

```
1.  >>> a = tf.constant([[1,2,3],[4,5,6]])
2.  >>> v = tf.constant([1,2,3])
3.
4.  >>> tf.slice(v, [1], [2])
5. <tf.Tensor: shape=(2,), dtype=int32, numpy=array([2, 3],
   dtype=int32)>
6.
```

```
7.  >>> tf.slice(a, [0,1], [2,2])
8.  <tf.Tensor: shape=(2, 2), dtype=int32, numpy=
9.  array([[2, 3],
10.        [5, 6]], dtype=int32)>
11. >>> tf.slice(a, [1,1], [1,2])
12. <tf.Tensor: shape=(1, 2), dtype=int32, numpy=array([[5, 6]],
    dtype=int32)>
```

Reshaping is quite intuitive. Remember that the resulting shape must contain the same number of elements as the original tensor:

```
1.   >>> a = tf.constant([[1,2,3],[4,5,6]])
2.
3.   >>> tf.reshape(a, [6])
4.  <tf.Tensor: shape=(6,), dtype=int32, numpy=array([1, 2, 3, 4, 5,
    6], dtype=int32)>
5.
6.   >>> tf.reshape(a, [3,2])
7.  <tf.Tensor: shape=(3, 2), dtype=int32, numpy=
8.  array([[1, 2],
9.         [3, 4],
10.        [5, 6]], dtype=int32)>
```

Tensor reversing is not complicated either. Keep in mind that we need to indicate which dimensions of the tensor should be reversed:

```
1.   >>> a = tf.constant([[1,2,3],[4,5,6]])
2.
3.   >>> tf.reverse(a, [0])
4.  <tf.Tensor: shape=(2, 3), dtype=int32, numpy=
5.  array([[4, 5, 6],
6.         [1, 2, 3]], dtype=int32)>
7.
8.   >>> tf.reverse(a, [1])
9.  <tf.Tensor: shape=(2, 3), dtype=int32, numpy=
10. array([[3, 2, 1],
```

```
11.        [6, 5, 4]], dtype=int32)>
12.
13. >>> tf.reverse(a, [0,1])
14. <tf.Tensor: shape=(2, 3), dtype=int32, numpy=
15. array([[6, 5, 4],
16.        [3, 2, 1]], dtype=int32)>
```

Gathering specific elements from tensors may seem a little bit trickier.

Generally speaking, try to remember that the **gather()** function mostly extracts slices composed of entire tensor axises, whereas **gather_nd()** can extract slices composed of individual tensor elements. In other words, indices in **gather()** refer to an axis whereas they refer to individual elements in **gather_d()**.

```
1.  >>> a = tf.constant([[1,2,3],[4,5,6]])
2.  >>> v = tf.constant([1,2,3,4,5,6,7,8,9,10])
3.
4.  >>> tf.gather(a, indices=[0,2], axis=1)
5.  <tf.Tensor: shape=(2, 2), dtype=int32, numpy=
6.  array([[1, 3],
7.         [4, 6]], dtype=int32)>
8.
9.  >>> tf.gather(a, indices=[1], axis=0)
10. <tf.Tensor: shape=(1, 3), dtype=int32, numpy=array([[4, 5, 6]],
    dtype=int32)>
11.
12. >>> tf.gather_nd(v, [[1],[8],[4]])
13. <tf.Tensor: shape=(3,), dtype=int32, numpy=array([2, 9, 5],
    dtype=int32)>
14.
15. >>> tf.gather_nd(a, [[0,1],[1,1],[0,0]])
16. <tf.Tensor: shape=(3,), dtype=int32, numpy=array([2, 5, 1],
    dtype=int32)>
17.
18. >>> tf.gather_nd(a, [[[0,0],[1,1]],[[1,0],[1,2]]])
```

```
19. <tf.Tensor: shape=(2, 2), dtype=int32, numpy=
20. array([[1, 5],
21.        [4, 6]], dtype=int32)>
```

Tip: Refer to the documentation at https://www.tensorflow.org if you're wondering whether some operation that we have not covered here is available in TensorFlow. This can also help if you have doubts about their usage.

Variables

Although never explicitly mentioned, all tensors created in the previous section are immutable.

That said, the **`constant()`** function's usage to create them might have been a hint. It does not mean that constant values are used but that the resulting tensor is constant, and so, immutable.

TensorFlow variables are used to represent shared, persisted, and mutable state. They are implemented in the **`Variable`** class and behave like tensors because a tensor backs their data structure.

Provide an initial value to the TensorFlow **`Variable`** class to create a variable:

```
1.  >>> m = tf.constant([[1,2],[3,4]])
2.  >>> mvar = tf.Variable(m)
3.
4.  >>> vvar = tf.Variable([1,2,3])
5.
6.  >>> mvar
7.  <tf.Variable 'Variable:0' shape=(2, 2) dtype=int32, numpy=
    array([[1, 2],
8.  [3, 4]], dtype=int32)>
9.  >>> vvar
10. <tf.Variable 'Variable:0' shape=(3,) dtype=int32, numpy=array([1,
    2, 3], dtype=int32)>
```

Variables are assigned names automatically. Use the **`name`** argument when creating them if you want to customize their name:

```
1.  >>> tf.Variable(v, name='My Variable')
2. <tf.Variable 'My Variable:0' shape=(2, 2) dtype=int32, numpy=
```

```
3. array([[1, 2],
4.        [3, 4]], dtype=int32)>
```

Once created, the variable's contents can be changed by assigning them. Traditional Python assignments cannot be used as these are not programming language variables but TensorFlow variables. Instead, we must use the **assign()** function in the variable instance.

Suppose we wanted to multiply our matrix variable **mvar** from the previous example by two. The corresponding scalar multiplication function operates on the tensor backing the variable, but, as we know, it does not modify the tensor. A new tensor is returned instead. So, it is necessary to assign the operation's result back to the variable:

```
1.  >>> new_tensor = tf.scalar_mul(2, mvar)
2.  >>> mvar.assign(new_tensor)
3. <tf.Variable 'UnreadVariable' shape=(2, 2) dtype=int32, numpy=
   array([[2, 4],
          [6, 8]], dtype=int32)>
```

We can also add or subtract values from the variable's contents during its assignment:

```
1.  >>> mvar.assign_sub([[1,1],[1,1]])
2. <tf.Variable 'UnreadVariable' shape=(2, 2) dtype=int32, numpy=
   array([[1, 3],
         [5, 7]], dtype=int32)>
3.
4.  >>> mvar.assign_add([[2,2],[2,2]])
5. <tf.Variable 'UnreadVariable' shape=(2, 2) dtype=int32, numpy=
6. array([[3, 5],
7. [7, 9]], dtype=int32)>
```

Graphs

TensorFlow can work in two execution modes: eager and graph.

Note: The default execution mode in TensorFlow 2 is the eager mode.

Eager execution means that all TensorFlow functions execute their work as soon as they are invoked. This mode enabled all the experimentation in the Python shell

shown in the previous sections: the function was called, and the result was ready as soon as we entered the Python line of code.

On the other hand, graph execution means operations are first stored in a data structure and executed later. The entire process of building a TensorFlow graph can be compared to the compilation of Java code into bytecode: with TensorFlow, Python functions are compiled into graphs.

It is more complicated to write TensorFlow programs that take advantage of graphs, but this added complexity brings benefits in the form of frequent performance improvements. Graph operations can be automatically parallelized and optimized by the compiler.

In TensorFlow 1, users had to deal with a session concept that was used to execute graphs. However, TensorFlow 2 was redesigned and eliminated the need to handle sessions. So, the concept of graphs would be relegated to the background, allowing users to focus on the problem they want to solve.

Many operations with graphs are hidden underneath the higher layers of the TensorFlow and Keras API. However, it is helpful to have a basic understanding of their workings.

Graphs in TensorFlow 2 are composed of operations and tensor objects. Operations define units of computation, while tensors contain the data used and flowing between operations. Their desired sequence of execution defines the graph's structure.

A normal Python function can be executed within a TensorFlow graph by converting it using **`tf.function()`**. This function returns an instance of a data object known as a **`Function`** that compiles the Python function into a graph and executes it as needed.

Consider the following toy function for example:

```
1.  def some_function(x):
2.    r = tf.reverse(x, [0])
3.    return tf.scalar_mul(2, r)
```

It can be compiled to a graph simply by calling:

```
1. some_graph_function = tf.function(some_function)
```

From this moment on, **`some_graph_function()`** should produce the same result as **`some_function()`**. The only difference is that the former runs as a graph internally, whereas the latter runs imperatively.

The following diagram shows a possible visualization of the graph generated by this compilation.

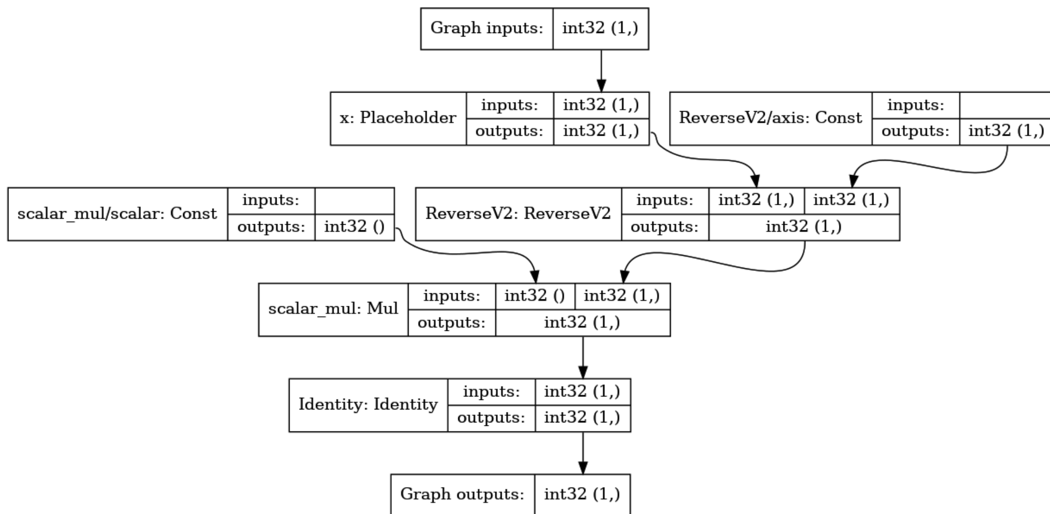

***Figure 5.4**: The graph of some_function()*

Note that this graph doesn't have to deciphered to use TensorFlow, nor is knowledge about their construction necessary. *Figure 5.4* is only meant to illustrate the elements on a TensorFlow graph.

It is necessary to keep some best practices in mind when writing functions to be compiled as a TensorFlow graph:

- Create TensorFlow variables outside of the function, passing them as arguments to be modified by the function
- Avoid taking arguments that are not TensorFlow objects
- Do not depend on outer Python variables: pass all dependencies as arguments to the function
- This is an optimization mechanism, so try to include as many computations inside the function as possible

These small steps help ensure that the function is compiled into a graph that preserves its imperative behavior.

Simple model training

Now that we have a high-level idea of how TensorFlow works, let's look at the training of a simple model.

The data for this example is a subset of the *Monitoring of CO2 emissions from passenger cars – Regulation (EU) 2019/631* dataset from the European Union Open Data project. The subset was obtained by reducing the number of features to four:

- The vehicle's mass in kilograms labeled as m (kg)
- The engine's displacement in cubic centimeters labeled as ec (cm3)
- The vehicle's CO2 emissions in grams per kilometer labeled as Enedc (g/km)
- The vehicle's fuel type: petrol or diesel

Rows from the reduced dataset were further processed to remove duplicated entries and entries with zero values in any column.

This reduced dataset is used to predict a vehicle's CO2 emissions. We present one model based only on the vehicle's mass and another based on its mass, displacement, and fuel type.

Our Python code begins by importing all the necessary libraries. Do not forget to activate the TensorFlow 2 virtual environment before starting work:

```
1. import pandas as pd
2. import tensorflow as tf
3. import matplotlib.pyplot as plt
4. import numpy as np
5. import seaborn as sns
6. from tensorflow import keras
7. from tensorflow.keras import layers
8. from tensorflow.keras.layers.experimental import preprocessing
```

Some libraries have not yet been used, so installing them in your system may be necessary. Use pip to install them after activating the virtual environment:

```
1. pip install pandas matplotlib numpy seaborn
```

- Pandas is a data analysis and manipulation tool documented at **https://pandas.pydata.org/**
- Matplotlib is a data visualization library from **https://matplotlib.org/**
- Numpy is a scientific computing package from **https://numpy.org/**
- Seaborn is a statistical data visualization package from **https://seaborn.pydata.org/**

All graphics in this section were generated with matplotlib or seaborn.

Loading and preparing the dataset

We then proceed to load the dataset from a CSV file:

```
1. co2 = pd.read_csv("CO2_subset.csv")
2. co2 = pd.get_dummies(co2, columns=['Ft'], prefix='', prefix_sep='')
```

The dataset contains a fuel type column that is not numeric; it defines a category, so it is necessary to translate it into a numeric column. Such translation can be accomplished using a transformation known as one-hot encoding. It works by turning the different categories into columns. Each row then gains a one (1) in the column corresponding to its original category, leaving the remaining columns set to zero (0).

The second line in the preceding code accomplishes this task by converting the dataset from a format like the one illustrated in the following table:

m (kg)	Enedc (g/km)	Ft	ec (cm3)
1253.0	95.0	diesel	1461.0
1448.0	132.0	petrol	1197.0
1395.0	130.0	petrol	1197.0
1165.0	85.0	diesel	1461.0

Table 5.5*: Original data as loaded from the CSV file*

It converts the dataset into:

m (kg)	Enedc (g/km)	ec (cm3)	diesel	petrol
1253.0	95.0	1461.0	1	0
1448.0	132.0	1197.0	0	1
1395.0	130.0	1197.0	0	1
1165.0	85.0	1461.0	1	0

Table 5.6*: After replacing non-numeric data with numeric columns*

From left to right, the columns represent the vehicle's mass in kilograms, its specific CO2 emissions in grams per kilometer, its engine capacity in cubic centimeters, and its fuel type.

We then proceed to divide the dataset into one training dataset and one test dataset, both containing the features and labels. Remember that features are the independent

variables and labels are the dependent variables. In other words, the features are the values fed into the model, and the labels are the model's output.

The purpose of the test dataset is to verify the accuracy of the model's predictions:

```
1. train_dataset = co2.sample(frac=0.8, random_state=0)
2. test_dataset = co2.drop(train_dataset.index)
3. train_features = train_dataset.copy()
4. test_features = test_dataset.copy()
5. train_labels = train_features.pop('Enedc (g/km)')
6. test_labels = test_features.pop('Enedc (g/km)')
7. test_results = {}
```

Lines 1 and 2 from the preceding excerpt reserve approximately 80% of the dataset for training and the remaining 20% for testing. The testing set is used to verify the model's accuracy after it is trained. We do not use a validation set for simplicity.

We then set the **`Enedc (g/km)`** column aside from both because it is our label, that is, it's what we want to predict.

The following excerpt creates a plot of the training dataset and displays it on the screen:

```
1. sns.pairplot(train_dataset[['m (kg)', 'Enedc (g/km)', 'ec (cm3)']],
   diag_kind='kde')
2. plt.show()
```

The plot should be identical to the one shown in *figure 5.5*. In this plot, each column is related to all the others in a matrix sharing the y-axes across a single row and the x-axes across a single column.

It is apparent from this plot that there is a relationship between the different columns in the dataset, by looking at the shapes of emissions-related cells.

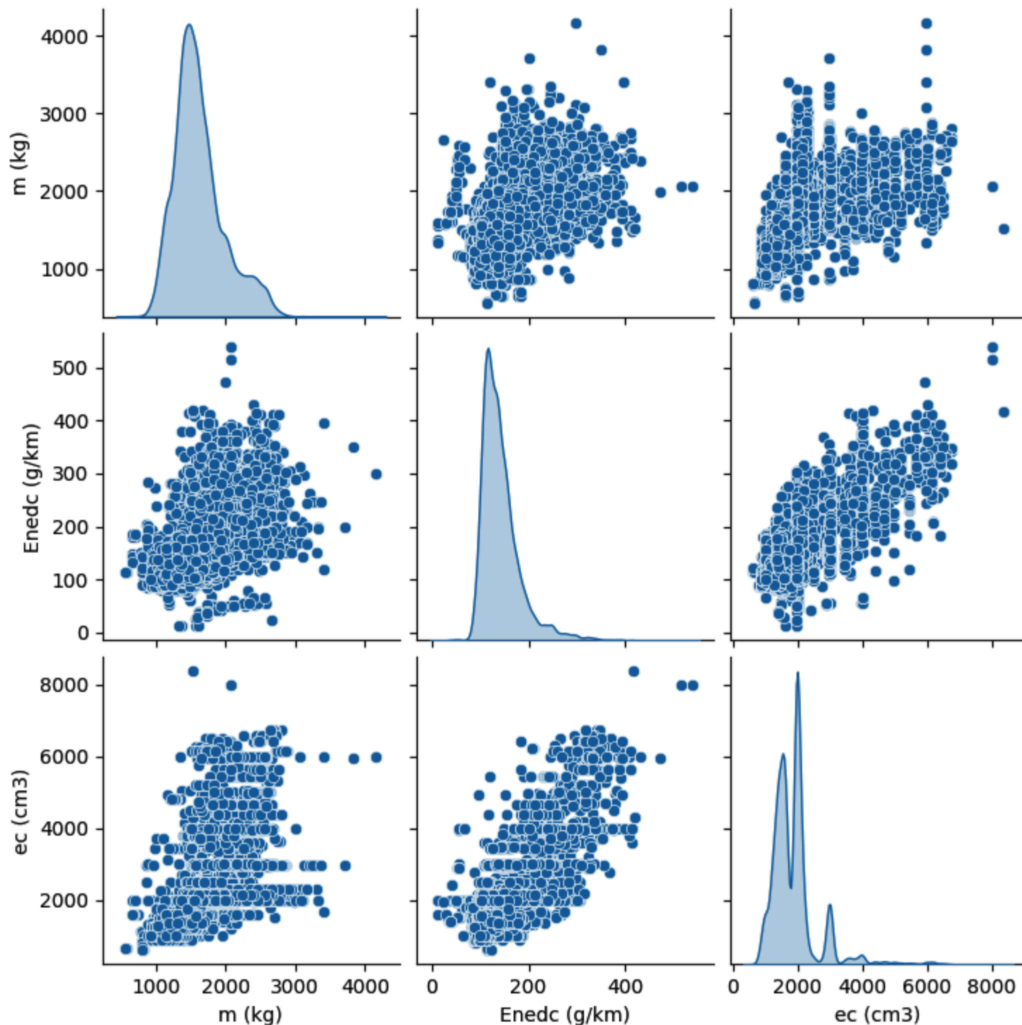

Figure 5.5: *Visualizing the CO2 emissions training dataset*

Training a linear regression model with one feature

Let's start with the simplest model possible: one feature and one label. We attempt to predict the amount of CO_2 emissions from the vehicle's displacement:

```
1.  def build_model(norm):
2.   model = tf.keras.Sequential([
3.       norm,
4.       layers.Dense(units=1)
5.   ])
6.   model.compile(
7.       optimizer=tf.optimizers.Adam(learning_rate=0.1),
8.       loss='mean_absolute_error')
9.   return model
10.
11. def fit_model(model, features):
12.  history = model.fit(
13.      features, train_labels,
14.      epochs=100,
15.      # configure logging
16.      verbose=1,
17.      # Calculate validation results on 20% of the training data
18.      validation_split = 0.2)
19.  return history
20.
21. # normalize across only one feature
22. displacement = np.array(train_features['ec (cm3)'])
23.  displacement_normalizer = preprocessing.Normalization(input_
    shape=[1,])
24. displacement_normalizer.adapt(displacement)
25.
26. displacement_model = build_model(displacement_normalizer)
27. displacement_model.summary()
```

We begin by defining two functions that help us work with the linear regression models.

The **`build_model()`** function uses the Keras sequential API to create a model with one neural layer. This API is called sequential because it allows us to define a linear

stack of layers. In the preceding listing, this is accomplished by providing an array of layers in lines 2-5. Two layers are defined: one normalization layer taken from the **build_model()** parameter and one neural layer implemented by the **Dense** class.

The **Dense** class is instantiated in line 4 by setting the units parameter to 1. This configuration results in a layer with one output and no activation function.

If you remember the description of the linear regression model from the previous chapter, we need to use a loss function along with a gradient descent algorithm to search for the best loss value. There are a few implementations of loss functions and gradient descent algorithms in TensorFlow. We are using the Adam gradient descent with a mean absolute error loss function in this example.

So, the model built by the **build_model()** function is a linear regression model with one neural layer. Once the **build_model()** function returns, it has already configured the model for training by calling its **compile()** function.

The **fit_model()** function executes the actual training. We define a fixed number of iterations, or epochs, for the training. It always runs this number of iterations, regardless of the loss function value.

Lines 21 to 24 create a normalization step across the displacement column, which is our feature for this model. Normalization aims to apply the same scale across all features and weights. Features often fall within different scales and ranges, so they must be comparable to each other somehow. The **Normalization()** class ensures that all data fits into a distribution centered around zero with a standard deviation of 1.

Finally, line 26 builds the model, and line 27 displays a summary of the model. This summary shows that the model we just created contains two trainable parameters out of a total of five parameters in the dataset:

```
1.  history = fit_model(displacement_model, train_features['ec (cm3)'])
2.
3. test_results['displacement_model'] = displacement_model.evaluate(
4.     test_features['ec (cm3)'], test_labels, verbose=1)
```

Now that the model has been created, it is time to train it. The training operation takes some time, and the amount of time needed depends on the computing resources available, the dataset size, the model structure, and the algorithms in use. This dataset is relatively small and the model is simple, so training should only take around one minute.

Line 3 from the previous excerpt runs the testing phase. We check how well the model performs against the testing dataset and store it for comparison purposes.

Once the model has been trained, we can use it to make predictions. We can also visualize the loss values generated during training. Let's first define some utility functions to generate the graphics:

```
1.  def plot_loss(history):
2.    plt.plot(history.history['loss'], label='loss')
3.    plt.plot(history.history['val_loss'], label='val_loss')
4.    plt.ylim([14, 30])
5.    plt.xlabel('Epoch')
6.    plt.ylabel('Error [g/km]')
7.    plt.legend()
8.    plt.grid(True)
9.    plt.show()
10.
11. def plot_enedc(x, y):
12.   plt.scatter(train_features['ec (cm3)'], train_labels,
          label='Data')
13.   plt.plot(x, y, color='k', label='Predictions')
14.   plt.xlabel('ec (cm3)')
15.   plt.ylabel('g/km')
16.   plt.legend()
17.   plt.show()
18.
19. def plot_predictions(model):
20.   x = tf.linspace(0.0, 8500, 8501)
21.   y = model.predict(x)
22.   plot_enedc(x, y)
```

The following two lines should plot and display the loss values and a set of predictions:

```
1.  plot_loss(history)
2.  plot_predictions(displacement_model)
```

Our model should have generated loss values similar to the ones shown in the following figure. Remember that the loss is nothing other than the neural network's

prediction error. There are two lines: one for the training loss and another for the validation loss. The former represents the error of predictions using the training dataset, whereas the latter is calculated using the validation dataset.

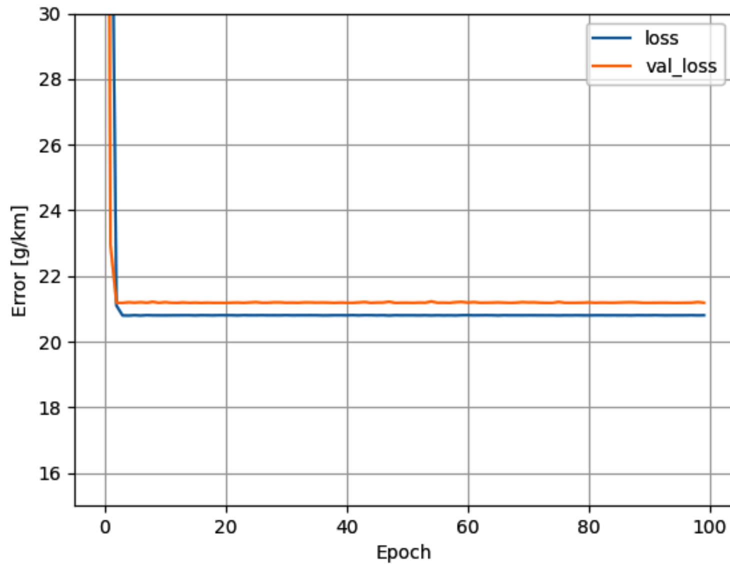

***Figure 5.6**: Single-variable linear regression loss values*

We get a plot similar to the following one if we now ask the model to make some predictions. It looks like the model is not too far off.

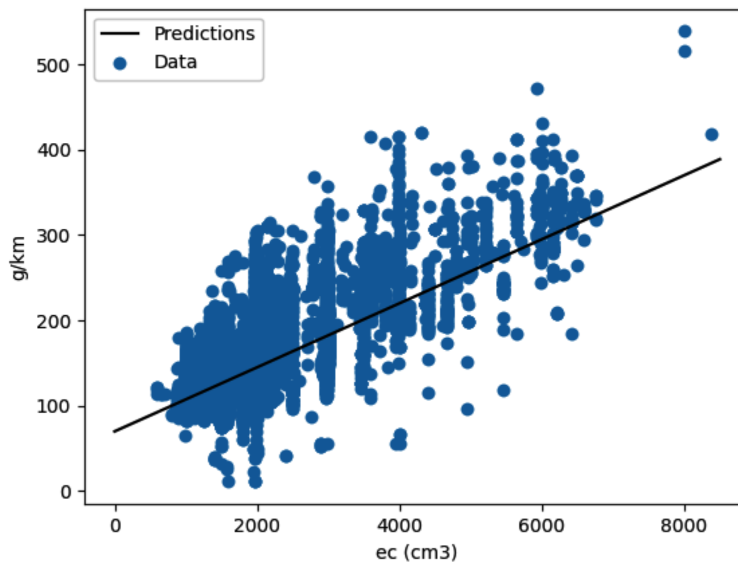

***Figure 5.7**: Single-variable linear regression model predictions*

Training a linear regression model with all features

We have used only one feature so far. What if we create a model with all features in the dataset? Would it show any improvement?

```
1. # Linear regression with multiple variables
2.
3. # Normalize across the entire set
4. normalizer = preprocessing.Normalization()
5. normalizer.adapt(np.array(train_features))
6.
7. # Build the model
8. full_displacement_model = build_model(normalizer)
9. full_displacement_model.summary()
10.
11. # Train and test the model
12. history = fit_model(full_displacement_model, train_features)
13. test_results['full_linear_model'] = full_displacement_model.
    evaluate(
14.     test_features, test_labels, verbose=1)
```

Let's train a model using the utility functions defined in the previous section, but this time on the entire dataset. Notice that the only difference is that the normalizer now works across all columns instead of just one; we no longer constrain the input shape.

It was unnecessary to modify the model created in the **`build_model()`** function as we do not restrict the shape of the network's inputs to a specific value. The dense neural layer used automatically configures itself according to the shape of the previous layer's outputs.

The model summary shown in line 9 of the preceding excerpt reports more trainable parameters than the single-feature model, as expected.

```
1. plot_loss(history)
```

Looking at the loss function graphic, it is apparent that the model with multiple variables performs better than the model with one variable.

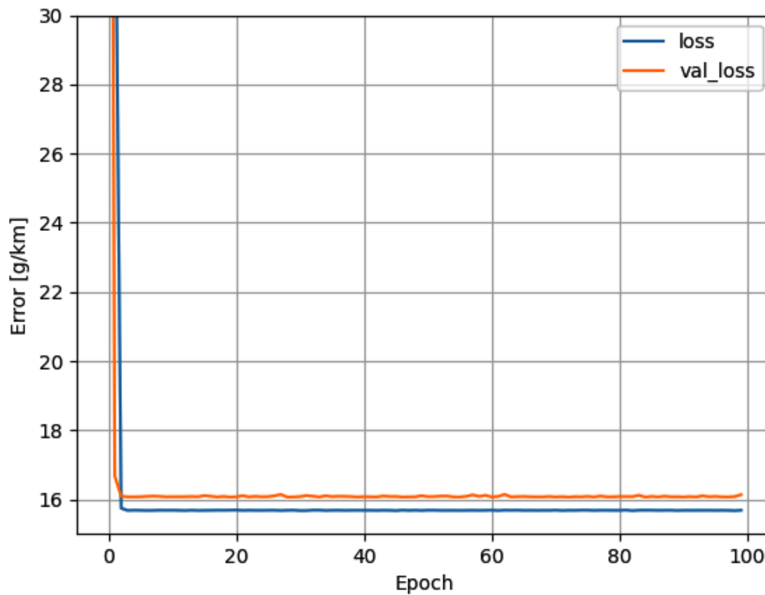

***Figure 5.8**: Multiple variables linear regression model loss*

Using a deep neural network for regression

One may wonder whether **Deep Neural Networks** (**DNNs**) perform better with this class of problems. It is pretty easy to adapt our code to use a DNN:

```
1.  def build_dnn_model(norm):
2.   model = keras.Sequential([
3.     norm,
4.     layers.Dense(64, activation='relu'),
5.     layers.Dense(64, activation='relu'),
6.     layers.Dense(1)
7.   ])
8.   model.compile(loss='mean_absolute_error',
9.     optimizer=tf.keras.optimizers.Adam(0.001))
10.  return model
```

The **build_dnn_model()** function in the previous excerpt creates a deep neural network. Note that there are two differences:

- We have added two more layers to the model in the Keras sequential API call.
- The new layers have non-linear activation functions. By default, the **Dense** layer is created with a linear activation function. We specified a non-linear activation function that allows the model to adjust itself better to data structures with some complexity. The ReLU activation function outputs only non-negative values; any negative values are constrained to zero. It was chosen because it generally allows models to perform well and be easy to train.

We can now build and train the model. Let's start with a single-variable model, like before:

```
1. dnn_displacement_model = build_dnn_model(displacement_normalizer)
2. dnn_displacement_model.summary()
3.
4.  history = fit_model(dnn_displacement_model, train_features['ec
   (cm3)'])
5. test_results['dnn_displacement_model']=dnn_displacement_model.
   evaluate(
6.     test_features['ec (cm3)'], test_labels, verbose=1)
```

There are many more parameters to train, but the loss function is about the same:

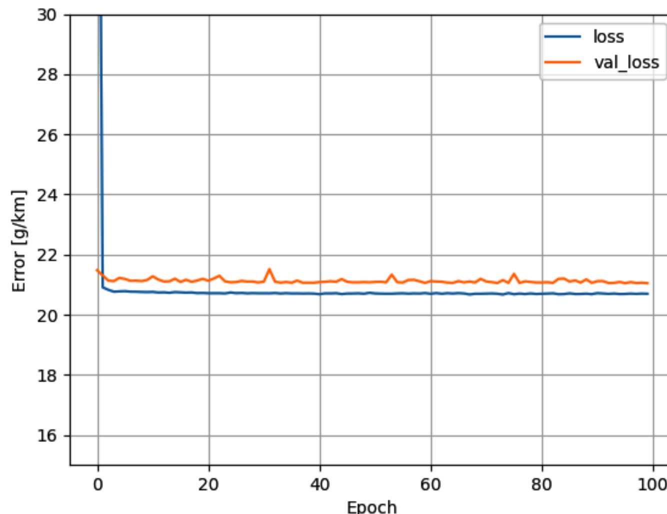

***Figure 5.9**: Single-variable DNN model loss values*

However, by looking at a plot of its predictions, we see that the DNN model can reflect the non-linearity of the data:

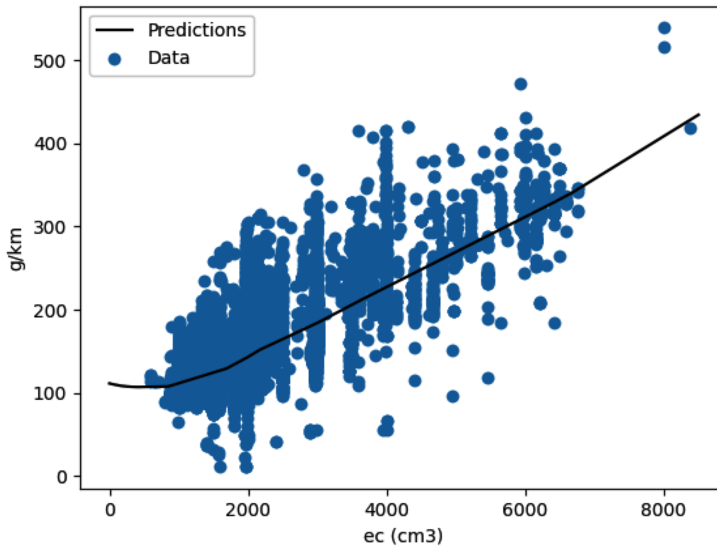

***Figure 5.10**: Single-variable DNN model predictions*

Let's train a DNN model based on all features:

```
1. full_dnn_model = build_dnn_model(normalizer)
2. full_dnn_model.summary()
3.
4.  history = fit_model(full_dnn_model, train_features)
5. test_results['full_dnn_model'] = full_dnn_model.evaluate(
6.     test_features, test_labels, verbose=1)
```

Again, the only difference is that we use the normalizer created with all columns in the features training dataset.

The model summary shows even more trainable parameters as expected, and the loss function has once again improved significantly, as shown in *figure 5.11*.

We have been collecting each model's evaluation results against the training dataset to compare their performances.

```
1.  pd.DataFrame(test_results, index=['Mean absolute error [g/km]']).T
```

The previous line displays the test results:

	Mean absolute error [g/km]
displacement_model	20.863403
full_linear_model	15.731938
dnn_displacement_model	20.852592
full_dnn_model	14.670211

***Table 5.7**: Performance of the different models based on the test data*

As we can see in *table 5.7*, the single-variable models have shown the same performance. In such simple cases, there is no gain from building a DNN. When all features are used, however, the DNN outperforms the linear model.

No unique tuning of either model's hyperparameters (layer parameters) was done, so different results may be obtained with different layer parameters.

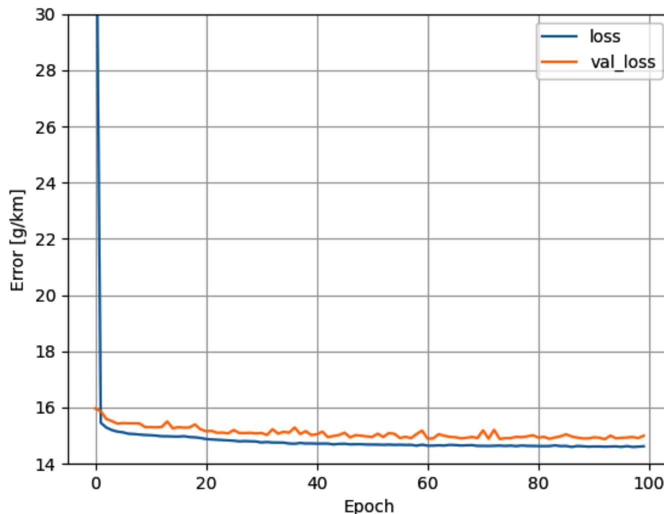

***Figure 5.11**: Multiple variables DNN model loss values*

We can now use the test training data to plot a graph of the model's predictions. This graphic can help visually examine the data pattern. The graphic's X-axis represents the true values, that is, the values known in the dataset, and the Y-axis represents the the values predicted by the model.

```
1.  test_predictions = full_dnn_model.predict(test_features).flatten()
2.
3. a = plt.axes(aspect='equal')
4. plt.scatter(test_labels, test_predictions)
```

```
5. plt.xlabel('True Values [g/km]')
6. plt.ylabel('Predictions [g/km]')
7. plt.xlim([0, 400])
8. plt.ylim([0, 400])
9. _ = plt.plot(lims, lims)
```

Looking at *Figure 5.12*, we notice that the model performs acceptably but shows a slight tendency toward lower emission values in general.

Tip: If you choose your dataset to build a regression model but the training stage outputs its loss values as "nan" (not a number), double-check to ensure that the data does not contain null (zero) or invalid values.

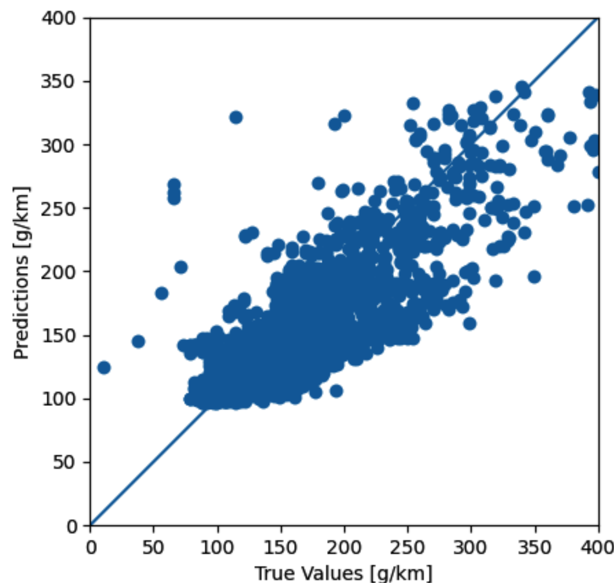

Figure 5.12: *Multiple variables DNN model predictions*

Conclusion

This chapter covered the basics of TensorFlow, along with its installation and usage with the Python programming language. We have now the foundations for reading TensorFlow code and understanding its underlying data structures, illustrated by a simple regression model.

The next chapter continues our work with TensorFlow. We use it to create a model capable of classifying images.

CHAPTER 6

Training a Model for Image Recognition with TensorFlow

Introduction

The previous chapter presented an approach to model training. Let's expand those concepts by training classification models for image recognition. We use two well-known datasets to create TensorFlow classification models for this purpose: one geared toward handwritten digits and another toward clothing.

Structure

We cover the following topics in this chapter:

- Recognizing handwritten digits
- Recognizing simple clothing items

Objective

By the end of this chapter, we have fully trained TensorFlow models for specific image recognition tasks. We also have understood the differences in behavior between the different types of models.

Recognizing handwritten digits

The **Modified National Institute of Standards and Technology** (**MNIST**) dataset is the most well-known dataset in the world of **Machine Learning** (**ML**). It is often considered the hello world dataset in what concerns ML projects. So, it is only fitting that our first image classification project should use it.

The MNIST dataset consists of images of all ten digits, handwritten. It is intended to train machine learning models to translate the handwritten digits to their numerical counterparts, in other words, to build an elementary **Optical Character Recognition** (**OCR**) system.

Preparing and loading the MNIST dataset

We begin by importing the necessary libraries and preparing some valuable functions to examine our dataset and progress as follows:

```
1.  def draw_image(img, label=''):
2.      plt.imshow(img)
3.      plt.xticks([])
4.      plt.yticks([])
5.      plt.grid(False)
6.      plt.xlabel(label)
7.
8.  def show_image(img):
9.      plt.figure()
10.     draw_image(img)
11.     plt.colorbar()
12.     plt.show()
13.
14. def draw_set_samples(images, labels):
15.     plt.figure(figsize=(5, 5))
16.     for i in range(9):
17.         plt.subplot(3, 3, i+1)
18.         draw_image(images[i], labels[i])
19.     plt.show()
20.
```

```
21. def plot(history, label, val_label):
22.     plt.plot(history.history[label], label=label)
23.     plt.plot(history.history[val_label], label=val_label)
24.     plt.xlabel('Epoch')
25.     plt.legend()
26.     plt.grid(True)
27.
28. def plot_history(history):
29.     plt.figure(figsize=(10, 4))
30.     plt.subplot(1, 2, 1)
31.     plot(history, 'accuracy', 'val_accuracy')
32.     plt.subplot(1, 2, 2)
33.     plot(history, 'loss', 'val_loss')
34.     plt.show()
```

The MNIST dataset is so well-known that it is referenced in the TensorFlow library as shown here:

```
1. mnist = tf.keras.datasets.mnist
2. (train_images, train_labels), (test_images, test_labels) = \
       mnist.load_data()
```

The preceding three lines are all it takes to transfer and load the dataset. We now have variables initialized with all training and testing data.

Let's take a peek at the dataset contents. The following figure represents the first image in the training dataset:

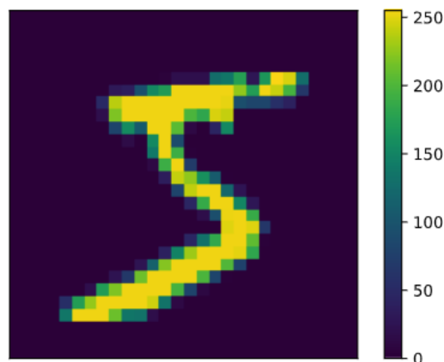

***Figure 6.1**: An image from the MNIST dataset*

It was obtained by calling the **show_image()** function on the first image in the dataset:

```
1. show_image(train_images[0])
```

If we also examine this image's shape, we see that it is a bidimensional array with dimensions (28, 28).

```
1. train_images[0].shape
```

So, it is a square image 28 pixels wide and 28 pixels high. Note that *figure 6.1* has its pixels colored according to their values. The scale of the bar to the right is [0, 255], which means that each pixel can have values in this interval. We are working with grayscale images because each image contains a single layer.

Most algorithms work better when their inputs are in the interval [0, 1]. So, we must scale the pixel's values to fall in this range. One way of doing this is as follows:

```
1. train_images = train_images / 255.0
2. test_images = test_images / 255.0
```

Now, all pixels are within the [0, 1] interval. You may use the **show_image()** method again to confirm that the scale has changed.

In addition to the **train_images** array, we have a **train_labels** array. This array contains the label for each image in the same position in its corresponding array. Labels are numbers from zero to nine, thus classifying each handwritten image as the equivalent number.

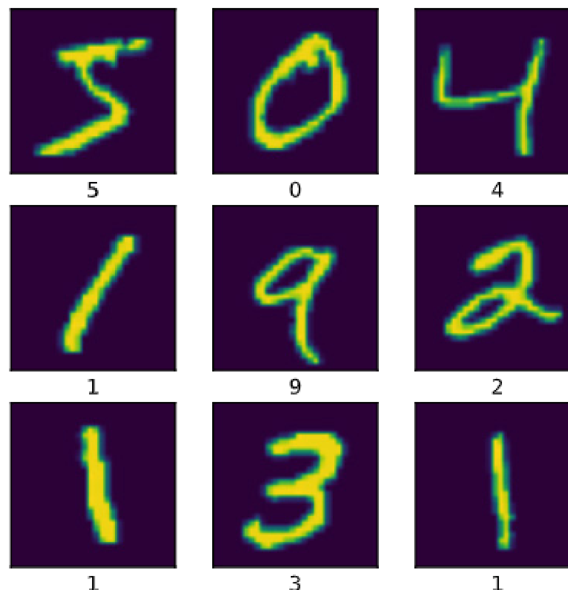

***Figure 6.2**: Some images and labels taken from the MNIST dataset*

Figure 6.2 is an example of this classification. It shows the first few pairs of images and labels from the training dataset.

Building the model

Now that the data is divided into training and testing sets and prepared, it is time to build the machine learning model.

The MNIST dataset comes originally from work by LeCun et al. (*Gradient-based learning applied to document recognition*, 1998) and is available in its original form at **http://yann.lecun.com/exdb/mnist/**. This web page also contains links to other research work published over the years examining the performance of different algorithms and methods applied to this dataset.

One can always see that, given the wide range of algorithms listed on that page, there is no single answer to machine learning problems, not even for those that appear to be as simple as the recognition of handwritten digits. Research and experimentation is necessary to find the model that best fits the problem at hand.

Taking inspiration from the works listed on the MNIST web page, let's first try a simple neural network with 300 hidden units and cross-entropy loss calculation, as shown in the following code snippet:

```
1. model = tf.keras.Sequential([
2.     tf.keras.layers.Flatten(input_shape=(28, 28)),
3.     tf.keras.layers.Dense(300, activation='relu'),
4.     tf.keras.layers.Dense(10)
5. ])
6. model.compile(optimizer='adam',
7.               loss=tf.keras.losses.SparseCategoricalCrossentropy(
8.                   from_logits=True),
9.               metrics=['accuracy'])
10.
11. history = model.fit(train_images, train_labels,
12.                     epochs=10, validation_split=0.2)
13.
14. test_loss, test_acc = model.evaluate(test_images,  test_labels,
15.     verbose=2)
```

```
16.
17. plot_history(history)
```

The model itself is built using the Keras sequential API as before. Lines 2-4 in the preceding code snippet define the model as follows:

- The input layer flattens the data, converting the two-dimensional tensors containing the images into a one-dimensional tensor for the following layers
- The hidden layer is a neural network layer with 300 units using the ReLU activation function
- The output layer is a neural network layer with 10 units, one for each label

Note: An activation function is fundamentally the function that calculates the output of each neuron based on its input(s). It is applied after the inputs are adjusted with weight and bias factors. Most functions allow the network to handle non-linear data.

The loss function used for the model's compilation in line 6 of the preceding code snippet calculates the cross-entropy loss when integers represent two or more classes. This function is appropriate for our problem because the labels are indeed integers, and we have 10 of them. Its **`from_logits`** parameter tells the function to calculate the loss on values in the (-Inf, +Inf) interval; by default, it expects the values to be part of a probability distribution.

Tip: In TensorFlow, the term logits is used to differentiate a neuron's output in the interval (-Inf, +Inf) from a probabilistic output in the interval [0, 1]. It is not related to the statistical function with the same name.

Line 11 of the previous code snippet trains the model across 10 iterations (epochs) using 20% of the training set as validation data. The resulting model is then verified against the test data in line 15, and the training statistics are plotted in line 18.

Figure 6.3 illustrates how these statistics are plotted, showing how the model has evolved during training:

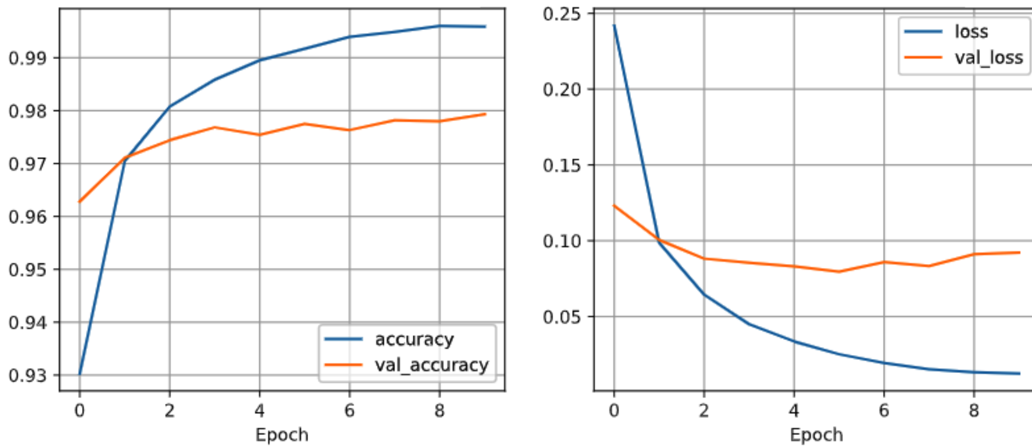

***Figure 6.3**: Training results of an NN model on the MNIST dataset*

Its accuracy kept improving, but its validation loss stagnated around the fifth iteration and appeared to worsen slightly.

When validation metrics diverge from training metrics, it usually means that the model is overfitting, that is, the model does not have a good performance with data it has never seen. An approach to prevent overfitting is discussed in the *Finding a better model* section later in this chapter.

Let's take a look at the testing results.

The last epoch resulted in a validation accuracy of 97.93%:

```
1. loss: 0.0098 - accuracy: 0.9970 - val_loss: 0.0923 - val_accuracy:
   0.9793
```

The testing reported an accuracy of 98%:

```
1.  313/313 - 0s - loss: 0.0792 - accuracy: 0.9800
```

We can see that the results from the testing phase are similar to the results from the last epoch. The testing phase even yielded a slightly better accuracy in comparison with the training validation accuracy.

A better accuracy while testing indicates that the model does not appear to be overfitting. With a 2% error rate, it is suitable for our purposes.

Researchers have built much more complicated models with error rates of 0.23%. However, these are outside of the scope of this book. You can read about them by following the links on MNIST's web page.

Saving the model

So far, we have worked with models that must be compiled and trained every time they are used. Repeating the training process is annoying with the small models and datasets we have used. When models become more complicated and the datasets become large, such repetition takes very long and needs so many resources that it becomes impracticable.

Thankfully, TensorFlow can save trained models to disk so that they can be reused later.

It could not be any simpler. The following single line of code creates a directory with the given name and stores the model weights, architecture, and training configuration in files inside it. This data format is called **`SavedModel`**.

```
1. model.save('digits')
```

The model can be loaded into memory later, even by a different Python program, simply by calling the **`load_model()`** function, as follows:

```
1. model = tf.keras.models.load_model('digits')
```

Testing the saved model with an image loaded from the disk

A machine learning model is only helpful if we can use real-world data to make predictions. Let's try to load a picture from the disk and ask our model what it represents.

We start with a new program. The first step is, naturally, to import the necessary libraries and load the previously saved model, as follows:

```
1.  import tensorflow as tf
2.  import numpy as np
3.
4. model = tf.keras.models.load_model('digits')
```

Then, we need to load the image we want to be recognized. The Keras API within TensorFlow exposes some functions for working with images. Its **`load_img()`** function can perform quite a few preprocessing steps with one single call, as shown in the following code snippet:

```
1. img = tf.keras.preprocessing.image.load_img('five.png',
2.     color_mode='grayscale',
```

```
3.      target_size=(28, 28),
4.      interpolation='bicubic')
```

An application capturing data in the real world needs to ensure that it is in the format expected by the machine learning model. The same applies to images.

Our model has the following requirements:

- Images must be grayscale, that is, each pixel must have only one intensity value
- Images must be exactly 28x28 pixels
- Pixels must have values in the interval [0,1]

The function call preceding executes the following tasks:

- Loads a picture in PNG format from a file named **`five.png`** in the current directory
- Transforms the picture into a grayscale picture instead of a color picture
- Resizes the picture so that it matches the expected image size

Note: For simplicity, we assume that the original image is already square and contains only the handwritten digit we want to recognize. A more practical application would need to fit a square bounding box around the handwritten digit and crop or resize it to that box.

The only image preprocessing step needed now is to scale the pixel values. We use the same technique as before:

```
1. subject = tf.keras.preprocessing.image.img_to_array(img) / 255.0
```

The image is ready, as shown in *figure 6.4*, but we cannot use it directly with the model:

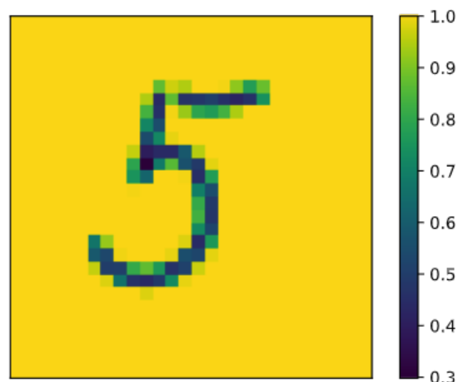

***Figure 6.4**: The test image after processing*

It is necessary to specify the batch size, both when training the model and when using it. The batch size is the number of samples processed in one single run and is determined by the tensor's first dimension. This information was already present in the MNIST dataset provided by TensorFlow when we trained the model.

However, this dimension is not included when we load an image, so we need to add one dimension to the image's tensor, as follows:

```
1. subject = tf.expand_dims(subject, 0)
```

Now, the tensor's first dimension is the batch size, and the following two dimensions contain the image itself.

We can finally ask the model to make a prediction, as follows:

```
1. predictions = model.predict(subject)
```

The predictions array contains the output of all 10 neurons on our model's output layer for each input image. Its contents are similar to the following:

```
1. [[-20.973839 -49.22007  -13.484433   4.35287  -80.24498   27.224888
    -23.330673  11.996965 -41.166916 -20.932955]]
```

Each array element refers to one class. Its value represents the model's degree of confidence that the input belongs to that class. We need to pick the index of the element with the highest degree of confidence.

Let's transform these values so that they can be interpreted as normalized probabilities (that is, percentages). The **softmax** function does this, as shown in the following snippet:

```
1. score = tf.nn.softmax(predictions[0])
```

The score tensor now contains percentages. The sum of all its elements is 1:

```
1. tf.Tensor(
2. [1.1683121e-21 6.3150229e-34 2.0900991e-18 1.1662971e-10 0.0000000e+00
    9.9999976e-01 1.1066205e-22 2.4355509e-07 1.9852653e-30 1.2170649e-
     21],
   shape=(10,), dtype=float32)
```

As we can see, the element with the highest value corresponds to the number five (index 5 in a zero-based indexing system), with a confidence level of 99.999976%:

```
1. print(
2.     "This image looks like a {} ({:.2f}% confidence)."
3.     .format(np.argmax(score), 100 * np.max(score))
4. )
```

The previous excerpt would output: **`This image looks like a 5 (100.00% confidence)`** because of rounding issues.

Note: Even when using the same input data, model architecture, number of epochs, and so on, it is not guaranteed that two different training runs will yield the same results. So, do not worry if your numbers are slightly different from ours. They will, however, be comparable. Add a call to the tf.random.set_seed() function, passing an integer value as its parameter, if you want to reproduce results consistently.

Recognizing simple clothing items

Inspired by the MNIST dataset, Han Xiao, Kashif Rasul, and Roland Vollgraf have created a dataset with an identical structure but containing clothing images instead (*Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*, 2017). They argue that the MNIST dataset is too easy for today's algorithms, so a new challenge is in order.

TensorFlow also includes a direct reference to the Fashion-MNIST dataset that makes it extremely simple to use.

Preparing and loading the Fashion-MNIST dataset

Since Fashion-MNIST has the same structure as MNIST, that is, 60.000 grayscale images of 28x28 pixels, we can reuse most of the code from the previous section. The reused code is not repeated in the following code excerpts.

Let's load the dataset and prepare it as we did with MNIST:

```
1. dataset = tf.keras.datasets.fashion_mnist
2. (train_images, train_labels), (test_images, test_labels) = \
3.     dataset.load_data()
4.  del dataset
5.
6. # Scale image values so each pixel value is in the [0, 1] interval.
7. train_images = train_images / 255.0
8. test_images = test_images / 255.0
```

Remember that we also need to scale the pixel values after loading and scaling the training and testing data. We are ready to use the dataset after this has been done.

The following figure shows a sample of the training data with the label corresponding to each picture:

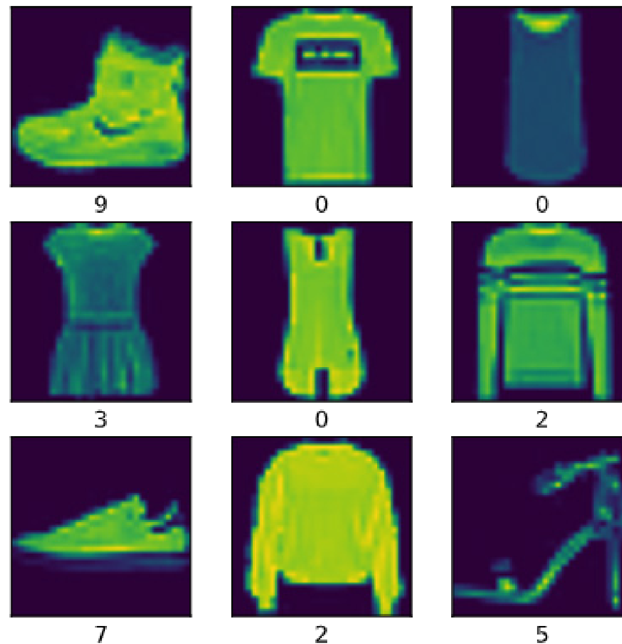

***Figure 6.5**: A sample from the Fashion-MNIST dataset*

Note that the labels are still numbers. They correspond to the 10 clothing item classes used to classify each item in the dataset, as shown in the following snippet:

```
1. classes = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
        'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

Looking at the array from the preceding excerpt, we can understand that the first row in *figure 6.5* shows an ankle boot, followed by a T-shirt and a top. T-shirts and tops are clustered together in the same class in this dataset.

Building the model

Do you remember that this new dataset was created to be a direct replacement of MNIST? We can reuse the model we created for the MNIST dataset and see how it fares with different images.

This time, however, let's do something a bit different. With our previous configuration of the MNIST model, we need to run the softmax function separately to transform the output layer's results into percentages. We can get percentages from the output

layer by specifying the softmax function as its activation function. This is shown as follows:

```
1. model = tf.keras.Sequential([
2.     tf.keras.layers.Flatten(input_shape=(28, 28)),
3.     tf.keras.layers.Dense(300, activation='relu'),
4.     tf.keras.layers.Dense(10, activation='softmax')
5. ])
6. model.compile(optimizer='adam',
7.               loss=tf.keras.losses.SparseCategoricalCrossentropy(),
8.                metrics=['accuracy'])
9.
10. history = model.fit(train_images, train_labels,
11.                     epochs=10,
12.                     validation_split=0.1)
13.
14. test_loss, test_acc = model.evaluate(test_images, test_labels,
    verbose=2)
```

The training phase ended with 89.30% validation accuracy:

```
1. Epoch 10/10
2. 1688/1688 [==============================] - 3s 2ms/step - loss:
   0.2245 - accuracy: 0.9169 - val_loss: 0.3092 - val_accuracy: 0.8930
```

The testing phase reported 88.69% accuracy:

```
1. 313/313 - 1s - loss: 0.3290 - accuracy: 0.8869
```

Since the accuracy reported by the testing phase is lower than the validation accuracy, we conclude that the model is overfitting by a small amount.

The following figure shows the training statistics for the training and testing phases:

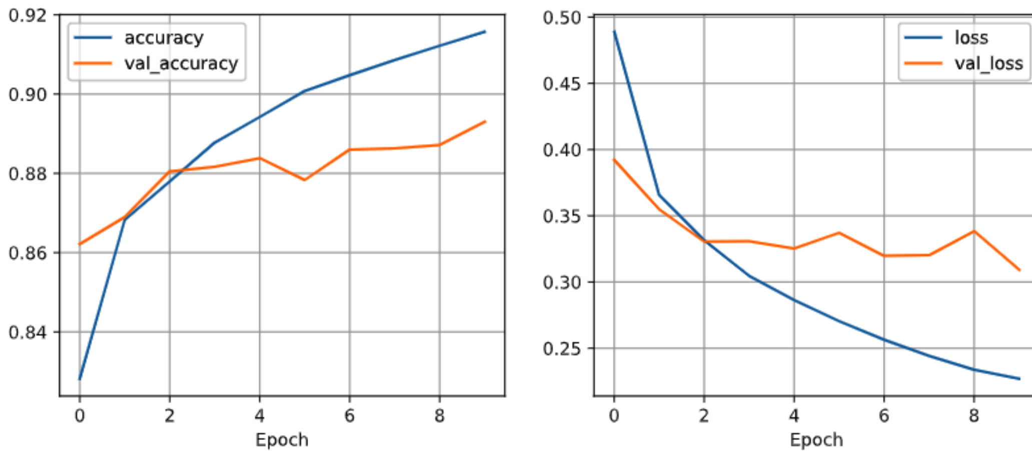

***Figure 6.6**: Training results of the MNIST network on Fashion-MNIST*

Figure 6.6 is similar to *figure 6.3* because they have the same architecture and parameters, and even their input data is similar.

The model can now be saved as follows:

```
1. model.save('fashion')
```

Finding a better model

We have concluded that the model was overfitting. Could there be a model configuration better suited to this problem?

As it turns out, there are many different models proposed for working with the Fashion-MNIST dataset.

We can try a model that Margaret Maynard-Reid proposed in her article *Fashion-MNIST with tf.Keras* (2018). It brings some extra layers, such as convolution, max pooling, and dropout. It is implemented as follows:

```
1. model = tf.keras.Sequential([
2.     tf.keras.layers.Conv2D(64, 2, padding='same',
3.         activation='relu',
4.         input_shape=(28,28,1)),
5.     tf.keras.layers.MaxPool2D(2),
6.     tf.keras.layers.Dropout(0.3),
7.     tf.keras.layers.Conv2D(32, 2, padding='same',
```

```
8.          activation='relu'),
9.      tf.keras.layers.MaxPool2D(2),
10.     tf.keras.layers.Dropout(0.3),
11.     tf.keras.layers.Flatten(),
12.     tf.keras.layers.Dense(256, activation='relu'),
13.     tf.keras.layers.Dropout(0.5),
14.     tf.keras.layers.Dense(10, activation='softmax')
15. ])
16. model.compile(optimizer='adam',
17.               loss=tf.keras.losses.SparseCategoricalCrossentropy(),
18.                metrics=['accuracy'])
```

Convolution is an operation that transforms each pixel using the average of its neighbors, often resulting in a sort of edge detection step. Since convolution steps are involved, this model is called a **Convolutional Neural Network** (**CNN**).

Max pooling is a downsampling operation and usually follows one or more convolution layers to reduce the number of features. Note that the number and placement of pooling layers control the image downsampling between layers.

The dropout layer sets input units randomly to zero during training at the rate specified as its first parameter. It then scales the remaining units by 1/1-rate, so their sum is unchanged. This operation prevents overfitting.

The new layers require the input layers to have a third dimension, called the channel dimension. So, we need to add an extra dimension to our training and testing image sets before training the new model, as follows:

```
1. new_train_images = tf.expand_dims(train_images, 3)
2. new_test_images = tf.expand_dims(test_images, 3)
```

Finally, the model can be trained like the previous one, as shown in the following code excerpt:

```
1. history = model.fit(new_train_images, train_labels,
2.                     epochs=10, validation_split=0.1)
3.
4. test_loss, test_acc = model.evaluate(new_test_images, test_labels,
   verbose=2)
```

This CNN model is more complicated than the first one we tried, so its training takes longer.

The last training step reported a validation accuracy of 90.63%:

```
1. Epoch 10/10
2. 1688/1688 [==============================] - 8s 5ms/step - loss:
   0.2661 - accuracy: 0.9026 - val_loss: 0.2457 - val_accuracy: 0.9063
```

Model evaluation reported a testing accuracy of 90.49%:

313/313 - 1s - loss: 0.2564 - accuracy: 0.9049

Its accuracy is better than the simple model, and it's overfitting by a smaller margin (0.14% versus 0.61%).

These improvements come from the different techniques implemented in the new model's layers:

- The convolution layer detects edges in the images, reducing the amount of information to the essential. Knowing the type of images in this dataset, reducing them to their edges should, logically, eliminate most noise.
- The max-pooling operation has two benefits. On the one hand, a downsampling operation reduces the image size, reducing the number of features the neural network needs to process while keeping the most important ones. On the other hand, this specific pooling operation helps reduce the model's sensitivity to slightly shifted, scaled, or tilted images.
- Finally, the dropout layer is explicitly used to turn off a subset of neurons during training, which changes with each iteration. So, on average, all neurons participate equally in the training process. The benefit of the dropout operation is that it helps prevent overfitting, even though the model may need more iterations (epochs) to complete its training.

Figure 6.7 also shows that the training evolution of our CNN is smooth; loss values evolve better during training. In comparison, loss values from *figure 6.6* almost stagnated after a couple of epochs.

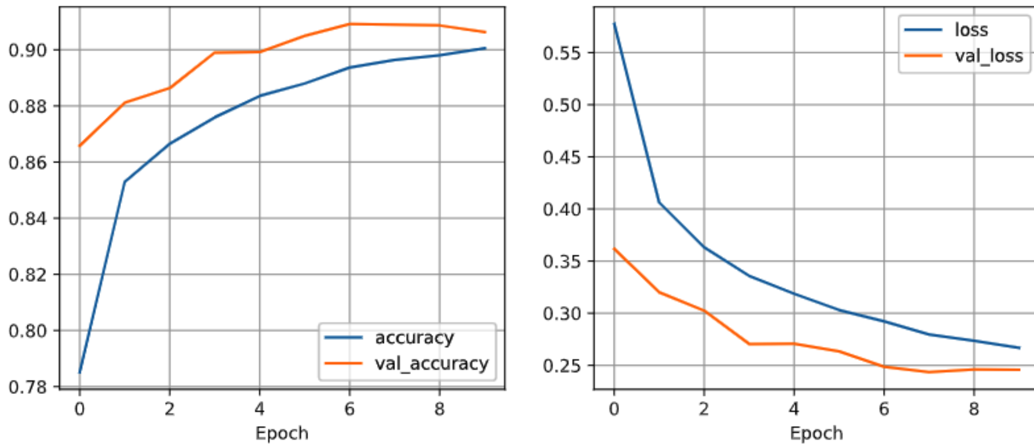

***Figure 6.7**: Initial results of a CNN trained on FashionMNIST*

Figure 6.8 shows the training evolution of the same CNN without the dropout layers to underline the importance of the dropout operation. Note how the validation loss and accuracy do not evolve favorably.

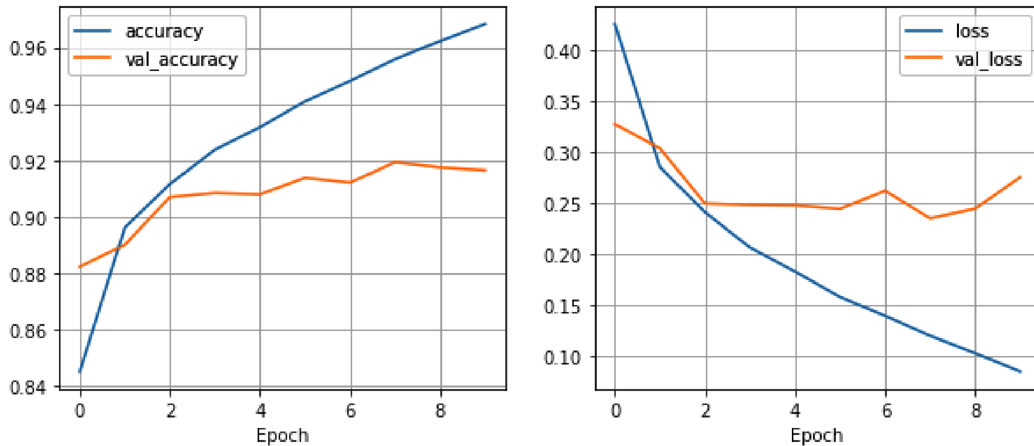

***Figure 6.8**: Effects of removing the dropout layers from the CNN*

We save this model for later use, as follows:

```
1. model.save('fashion-cnn')
```

Evaluating both models with a realistic image

We now have two machine learning models trained on the Fashion-MNIST dataset:

- A simple neural network
- A convolutional neural network

They have similar test results. The simple neural network achieved 88.69% accuracy on test data, and the CNN achieved 90.49% accuracy on the same test data.

At first glance, it would seem that they are quite equivalent. Would a 1.8% accuracy difference matter much for models with an error margin over 10%? However, we also noticed that the simple neural network was more overfitting than the CNN. Does this overfitting make a difference?

Let's apply these models to a realistic image that does not belong to the test set and has not been processed in the same way.

We use a similar version of the image loading and preprocessing code created for our MNIST model test. The difference is that this version does not need to apply an additional softmax conversion on the model's results and displays the probabilities for all classes in addition to the most probable. This way, we can better understand what the model is predicting:

```
1.  def classify_image_file(imageFile):
2.      # Load, scale the image down to 28x28 and make it grayscale
3.      img = tf.keras.preprocessing.image.load_img(imageFile,
4.          color_mode='grayscale',
5.          target_size=(28, 28),
6.          interpolation='bicubic')
7.      # Scalepixel values from the [0,255] to the [0,1] interval.
8.      img = tf.keras.preprocessing.image.img_to_array(img) / 255.0
9.      classify_image(img)
10.
11. def classify_image(subject):
12.     draw.show_image(subject)
13.     # Create a batch from this single image.
14.     subject = tf.expand_dims(subject, 0)
15.     # Get the first prediction from the sing-image batch.
16.     score = model.predict(subject)[0]
```

```
17.     print(
18.         "I'm convinced this is a {} ({:.2f}% confidence)."
19.         .format(classes[np.argmax(score)], 100 * np.max(score))
20.     )
21.     print("But it could also be:")
22.     for idx in range(0, score.shape[0]):
23.         print("  {:5.2f}% says it's a {}"
24.             .format(score[idx] * 100, classes[idx]))
```

We can now load the model and classify the test image, as follows:

```
1. model = tf.keras.models.load_model('fashion-cnn')
2.
3. classes = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
4.            'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
5.
6. classify_image_file("shirt.png")
```

The preceding code excerpt loads the CNN model. Experimenting with the simple model is only a matter of loading it and rerunning the classification.

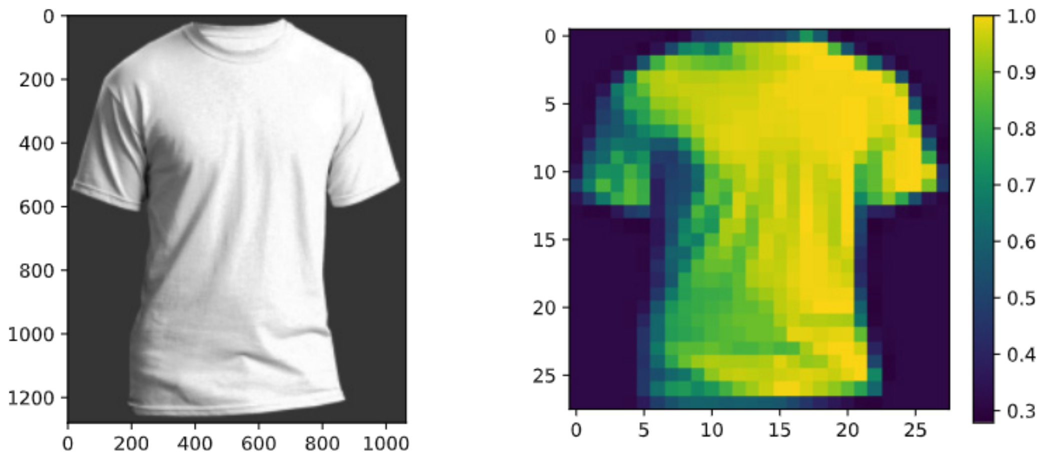

Figure 6.9*: The realistic image used to try the FashionMNIST models*

Figure 6.9 shows the source image used and how it looked before being submitted to the model. As with the recognition of handwritten digits, it was necessary to scale the image in size and pixel values.

It is relevant to mention that all images in the FashionMNIST training and test datasets have a dark background with pixel values close to or at zero. Note that our

realistic image also has a dark background. These models do not perform that well on images with a light background.

Let's see how both the models performed. Their full results are shown in *table 6.1*.

It is apparent that the CNN model is superior, at least when classifying images of t-shirts, like the one we used.

This improved performance is most likely the result of less overfitting, which means that the model works better with new data, and of the higher test accuracy the model has achieved.

	Neural network	**Convolutional neural network**
Prediction	**Bag (53.59% confidence)**	**T-shirt/top (33.33% confidence)**
Classes	40.68% says it's a T-shirt/top	33.33% says it's a T-shirt/top
	0.00% says it's a Trouser	0.38% says it's a Trouser
	0.69% says it's a Pullover	9.31% says it's a Pullover
	0.02% says it's a Dress	2.64% says it's a Dress
	0.01% says it's a Coat	0.76% says it's a Coat
	0.00% says it's a Sandal	1.58% says it's a Sandal
	5.00% says it's a Shirt	5.94% says it's a Shirt
	0.00% says it's a Sneaker	6.61% says it's a Sneaker
	53.59% says it's a Bag	26.35% says it's a Bag
	0.00% says it's an Ankle boot	13.09% says it's an Ankle boot

Table 6.1*: Results of both models on a realistic image*

Conclusion

We built two image classification models based on neural networks for two very different datasets in this chapter. Although humans would say that both are equally easy to classify, the higher complexity of the better performing model on the Fashion-MNIST dataset clearly shows that it's not the case with machine learning algorithms.

The latter dataset is more challenging to classify, requiring different kinds of layer operations like convolutions. This kind of operation is used in other neural networks, some of which are shown in this book's remaining chapters.

We also illustrated how overfitting could be a problem as it prevents a model from performing well on different data, and how the dropout operation is a technique to reduce overfitting.

In the next two chapters, we build an Android application capable of using these models to classify images of objects captured using the device's camera.

Chapter 7
Android Camera Image Capture with CameraX

Introduction

This chapter covers one of our image application building blocks—capturing live images using the device's camera. It showcases how the CameraX library, which is an Android library built to make it easier for developers to work with the device's camera in Android applications, can be used to streamline the entire process.

Structure

We cover the following topics in this chapter:

- Ways of working with cameras on Android
- Getting started with the CameraX API
- Requesting permissions to use the device's cameras
- Setting up the camera preview
- Taking a picture

Objectives

The advantages of using the CameraX library to capture images in real-time should be clear upon completing this chapter. We also create a new Kotlin-based Android application to use CameraX to capture images, building upon the work started in *Chapter 3, Building Our Base Application with Kotlin and SQLite*, and briefly introducing alternative application development techniques and libraries.

Ways of working with cameras on Android

Intents

The simplest way to implement image capture is to delegate the hard work to the camera application supplied with the device. This method is based on using intents to invoke the camera application and ask it to save the picture in the device's storage.

We have used intents in *Chapter 2, Event Handling and Intents on Android*, and as you may remember, switching between applications does not provide optimal user experience. Furthermore, this technique does not allow real-time image analysis.

The application flow for capturing an image using intents resembles the following:

1. Fire an intent to open the camera application. The intent must contain the path and file name for saving the captured picture in the JPEG format.
2. The user is redirected to the default camera application. They use its user interface to frame and take the picture.
3. The camera application saves the JPEG file and sets a result on the intent.
4. The user is redirected to the original application.
5. The original application reads and processes the JPEG file contents, eventually presenting its result to the user.
6. The whole process must be repeated if the image needs to be adjusted.

It is simple to implement, but the user experience is far from optimal, this is why hardly any applications use this method anymore.

Specialized camera APIs

There are three different APIs for working with cameras on Android applications.

Camera API

The Camera API is now deprecated, meaning it should not be used in new applications. It is mentioned to make the evolution toward different ways of working with the camera apparent.

When using the Camera API, the application needs to take care of the following tasks to show a preview to the user and capture an image:

1. Prepare the preview surface, which is commonly implemented using the **`SurfaceView`** class
2. Initialize the camera and set any settings, including the preview size
3. Connect the camera to the preview surface
4. Start the preview
5. Capture an image; we can also handle image frames individually
6. Restart the preview to capture another image, if desired
7. Stop the preview and release the camera

On top of these tasks, the application had to ensure that the camera is fully released once it is no longer in the foreground. Remember the application lifecycle described in *Chapter 1*, *Building an Application with Android Studio and Java*—we must implement the necessary callbacks to start, stop, acquire, and release the camera at appropriate times.

Another concern is the image size, aspect ratio, and orientation:

- Not all cameras support the same resolutions
- Not all cameras are placed in the same location on the device
- Not all cameras support the same aspect ratios

So, the application had to know how to select the best resolution and aspect ratio from the ones supported by the camera.

An additional difficulty is imposed by the user's ability to rotate the device, changing the application's layout. The preview had to be adjusted according to the device and camera orientations when this happens, and it often included resizing and rescaling the preview image.

Camera2 API

The Camera2 API was developed to support advanced camera use cases that were difficult or impossible to achieve using the old Camera API. The Camera2 API is available from the Android API level 21.

This objective means that the API became much more flexible and powerful but also more complicated.

In addition to all the preceding concerns listed for the old Camera API, the Camera2 API requires the application to do the following:

- Manage device-specific configurations
- Request, configure, and handle the different data streams explicitly; for example, one for the preview and another for the capture

CameraX API

Given the additional complexity of the Camera2 API, another API was also introduced for less complicated use cases with API level 21—the CameraX API.

CameraX is geared toward the concept of use cases, which means that the underlying resources are automatically configured with the most suitable settings for each use case:

- Previewing images on the display
- Running image analysis tasks
- Saving high-quality images

On top of that, this API is lifecycle-aware, alleviating the resource management concerns. It also tries to provide consistent behavior across devices, especially in terms of aspect ratio, rotation, orientation, and image sizes.

From this point forward, we use the CameraX API because the applications described in this book do not require any of the advanced features provided by the Camera2 API.

Getting started with the CameraX API

Let's see how we can use the CameraX API in a practical application. Our purpose is to build an Android application to capture a picture.

We start with a simple Android application using the Kotlin language and one activity. It is fundamentally the same application as in *Chapter 3, Building Our Base Application with Kotlin and SQLite*, without the database code.

> **Tip: The new CameraX API is available only from API level 21 onward, so do not forget to change the default API level in the project creation dialog box.**

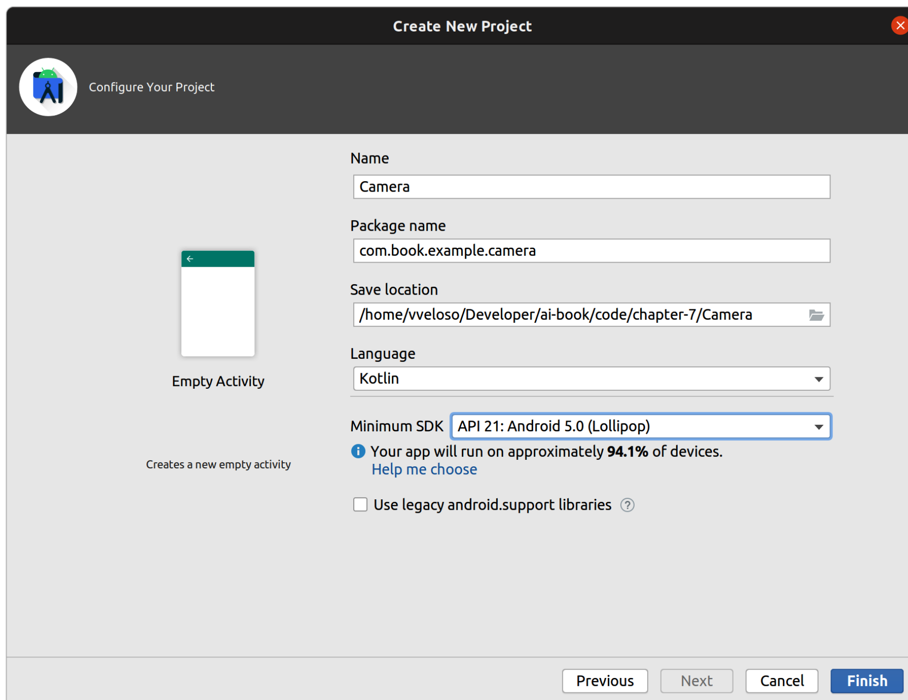

Figure 7.1: Creating the Camera application for Android 5.0 and higher

Once the application has been created, it is necessary to add the dependencies supporting the CameraX API to our application module's **build.gradle** file, as shown in the following code snippet:

```
1. dependencies {
2.
3.     // (other dependencies)
4.
5.     def camerax_version = '1.0.0-rc01'
6.     implementation "androidx.camera:camera-core:$camerax_version"
7.     implementation "androidx.camera:camera-camera2:$camerax_
         version"
8.     implementation "androidx.camera:camera-lifecycle:$camerax_
         version"
9.     implementation "androidx.camera:camera-view:1.0.0-alpha20"
10. }
```

For convenience, we use a Gradle plugin that simplifies our development work. Remember how it was necessary to use the **findViewById()** function in the previous chapters? This plugin generates code that eliminates that need by adding properties to the **Activity**, **Fragment** or **View** that provides direct access to their children views. It also adds properties to represent other getter and setter pairs in view classes.

The following line should be placed at the top of the same **build.gradle** file to enable this plugin:

```
1. apply plugin: 'kotlin-android-extensions'
```

Don't forget to synchronize the Gradle project in Android Studio after modifying the build files.

Note: The details already covered in chapters 1 through 3 are not repeated from this point onward. You can refer to those chapters and to the complete source code accompanying this book for such implementation details.

Requesting permissions to use the device's cameras

As we already know, Android applications need to ask for the user's consent to perform specific tasks and declare their need to access specific hardware.

The first step to use the device's cameras is to declare a dependency on such hardware and its related permissions in the application manifest. The following entries must be added to the **AndroidManifest.xml** file before the **application** tag:

```
1. <uses-feature android:name="android.hardware.camera.any" />
2. <uses-permission android:name="android.permission.CAMERA" />
```

Recent Android versions no longer grant these permissions when the application is installed. Instead, the application needs to request them from the user when they are needed.

The activity or fragment must check whether the appropriate permission has been granted when it needs to use the camera.

Permission checks are done using the **ContextCompat.checkSelfPermission()** function. If this function does not return the **PackageManager.PERMISSION_GRANTED** value, the application must request the desired permission explicitly.

```
1.  private fun cameraPermissionsGranted() =
2.      ContextCompat.checkSelfPermission(
3.          requireActivity().baseContext, Manifest.permission.CAMERA
```

```
4.     ) == PackageManager.PERMISSION_GRANTED
```

The preceding function is meant to be used from within a fragment. The **requireActivity()** function call in line 3 is not needed when using this function from an activity.

We check the permission at startup because our application starts the camera preview immediately:

- An activity calls this function on its **onCreate()** function after setting its content view
- A fragment does it on its **onAttach()** function because it needs a reference to the base context, and this reference is available through its parent activity

When the permission has not been granted, the easiest way to request it is to use the predefined Android permission dialog box. Just like with intents, this permission request API needs a request code provided by the calling application. You can define a constant with the integer value of your preference.

Call the following from a fragment:

```
1. requestPermissions(arrayOf(Manifest.permission.CAMERA),
2.     PERMISSION_REQUEST_CODE)
```

An activity does not have a built-in function for this purpose, so the call is slightly different:

```
1. ActivityCompat.requestPermissions(this,
2.     arrayOf(Manifest.permission.CAMERA),
3.     PERMISSION_REQUEST_CODE)
```

Once the call is made to the **requestPermissions()** function, the system presents the appropriate dialog box to the user requesting the specified permissions.

The result of this operation is communicated to the application through the activity or fragment callback **onRequestPermissionsResult()**. So, it is necessary to override this function in the calling activity or fragment.

This function provides the individual permissions requested and their corresponding grant results as parameters, so we can examine the parameters directly or use **checkSelfPermission()** again.

```
1.  override fun onRequestPermissionsResult(
2.     requestCode: Int,
3.     permissions: Array<out String>,
4.     grantResults: IntArray
```

```
5. ) {
6.     if (requestCode == PERMISSION_REQUEST_CODE) {
7.         if (cameraPermissionsGranted()) {
8.             // start the camera
9.         } else {
10.            Toast.makeText(context,
11.                R.string.permissions_not_granted,
12.                Toast.LENGTH_SHORT).show()
13.        }
14.    }
15.}
```

As shown in the preceding code excerpt, our application reuses the **cameraPermissionsGranted()** function to verify that the permission was granted and acts accordingly. There is nothing the application can do if the permission was not granted. It merely informs the user that it cannot proceed.

Tip: Users often enjoy a better experience if the Android Navigation component is used in the application. The accompanying code uses this component along with fragments. Any differences between the use of activities or fragments continue to be highlighted when necessary.

Setting up the camera preview

The first step to implementing a live camera preview is to have a surface in the application to draw the image frames.

Thankfully, the CameraX libraries already provide a ready-to-use view for this purpose. They even support scaling, rotation, and frame cropping to help with the difficulties we enumerated in the previous sections.

We begin setting up the camera preview by adding **PreviewView** to our layout. The following code snippet shows a possible configuration of **PreviewView** when placed inside a **ConstraintLayout**:

```
1. <androidx.camera.view.PreviewView
2.     android:id="@+id/previewView"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     app:layout_constraintBottom_toBottomOf="parent"
```

```
6.      app:layout_constraintEnd_toEndOf="parent"
7.      app:layout_constraintStart_toStartOf="parent"
8.      app:layout_constraintTop_toTopOf="parent" />
```

Depending on its version, AndroidStudio may not support adding this view directly from the editor. In this case, it is necessary to edit the layout XML code manually.

The second step is to request an instance of the **CameraProvider** class and set it up to start CameraX's preview use case once it is ready.

We split these tasks into two functions for readability and modularity. The first function, **configureCamera()**, requests the camera provider and invokes the second function, **configurePreview()**, once the camera provider is ready.

```
1.  private fun configureCamera() {
2.      ProcessCameraProvider.getInstance(requireContext()).also {
3.          cameraProviderFuture -> cameraProviderFuture.addListener({
4.              configureCameraUseCase(cameraProviderFuture.get())
5.          }, ContextCompat.getMainExecutor(context))
6.      }
7.  }
```

Note that the **ProcessCameraProvider.getInstance()** function does not return the camera provider instance immediately. Instead, it returns a **future** instance that represents an asynchronous process running in the background. One of the techniques to receive the result of this background process is to use a listener function, also called a callback. This callback is then invoked once the result is ready. By that time, the original application code is doing something else, and the background process has terminated. So, it is necessary to specify the executor where this listener is going to run. We chose the main executor because it is not associated with the user interface.

The **configureCameraUseCase()** function takes care of setting up CameraX's preview use case and is shown in the following code excerpt:

```
1.  private fun configureCameraUseCase(
2.          cameraProvider: ProcessCameraProvider) {
3.      val preview = Preview.Builder().build()
4.      preview.setSurfaceProvider(previewView.surfaceProvider)
5.
6.      val cameraSelector = CameraSelector.Builder()
```

```
7.              .requireLensFacing(CameraSelector.LENS_FACING_BACK)
8.              .build()
9.
10.     cameraProvider.bindToLifecycle(this, cameraSelector,
11.             preview)
12. }
```

Lines 3 and 4 from the preceding code snippet create a new instance of the **Preview** class and associate it with the **PreviewView** instance in our view.

A camera is requested in lines 6 and 7. We have chosen to request a camera facing the back of the device, but we can also request the front camera. At the time of writing, there is no fine-grained control of the precise camera that is used. CameraX selects the most appropriate camera according to its algorithms. After all, it was designed for simplicity.

Another critical task is accomplished in line 10—binding the camera provider to the use case and the current view's life cycle. This binding means that CameraX takes care of acquiring and releasing resources and hardware automatically when the activity or fragment comes to the foreground, goes to the background, or changes its layout. We do not need to write code to handle all the life cycle callbacks ourselves.

Finally, the **configureCameraUseCase()** function needs to be called from the permission handling blocks we built earlier (either in the activity creation or fragment view creation callback). We show the fragment usage as follows:

```
1.  override fun onViewCreated(view: View, savedInstanceState:
    Bundle?)  {
2.      super.onViewCreated(view, savedInstanceState)
3.      if (cameraPermissionsGranted()) {
4.          configureCamera()
5.      } else {
6.          requestPermissions(
7.              CAMERA_PERMISSIONS_REQUESTED,
8.              PERMISSION_REQUEST_CODE)
9.      }
10. }
```

Another call needs to be placed in the permissions result callback:

```
1.  override fun onRequestPermissionsResult(
```

```
2.      requestCode: Int,
3.      permissions: Array<out String>,
4.      grantResults: IntArray
5. ) {
6.      if (requestCode == PERMISSION_REQUEST_CODE) {
7.          if (cameraPermissionsGranted()) {
8.              configureCameraUseCase()
9.          } else {
10.             Toast.makeText(context, R.string.permissions_not_
                  granted,
11.                 Toast.LENGTH_SHORT).show()
12.         }
13.     }
14. }
```

The preview part of the application is ready to use once all the permission and camera management code is in place.

The following picture shows what the Android permissions request dialog box may look like. Its actual appearance varies according to the Android operating system version and device manufacturer. This specific picture was taken on an emulator.

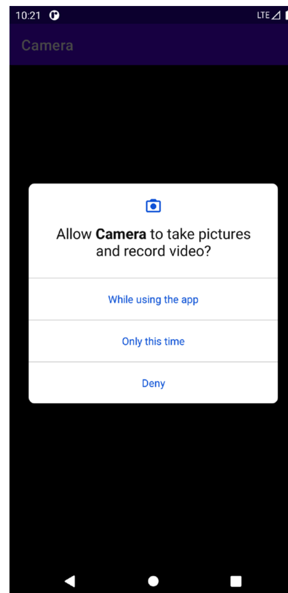

Figure 7.2: Camera permissions request dialog box

The live preview starts once the user gives their permission. The following screenshot shows the live preview on the author's device:

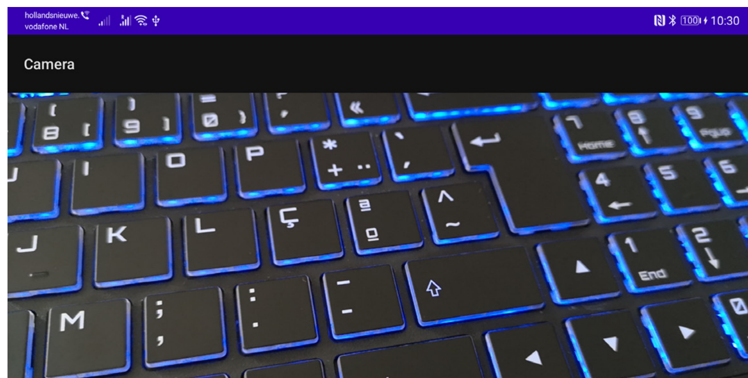

***Figure 7.3**: Camera preview working in landscape orientation*

The CameraX framework automatically makes all the necessary adjustments when the device is rotated and the application layout changes.

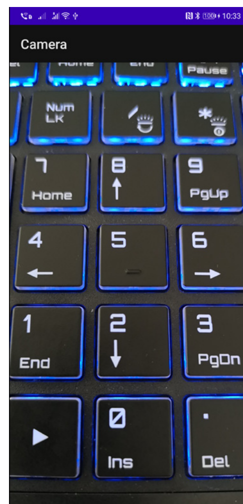

***Figure 7.4**: Camera preview working in portrait orientation*

Note how the pictures from both *figure 7.3* and *figure 7.4* have the appropriate aspect ratio to fit the preview area correctly.

Tip: One may grant or deny permissions to the application during development and experimentation. We can reset an application's permissions definitions through the corresponding section in the device's Settings application. The exact naming may vary, but it is usually "Permission manager" or "Application permissions."

Taking a picture

Now that the application has its live camera preview feature implemented, it is time to do something with it. Let's take a still picture and save it to the device's media location.

Configuring CameraX for image capture

The first step is to change our **`configureCameraUseCase()`** function to enable the CameraX image capture use case:

```
1.  private lateinit var imageCapture: ImageCapture
2.
3.  private fun configureCameraUseCase(
4.          cameraProvider: ProcessCameraProvider) {
5.      val preview = Preview.Builder().build()
6.      preview.setSurfaceProvider(previewView.surfaceProvider)
7.
8.      imageCapture = ImageCapture.Builder()
9.              .setTargetRotation(requireActivity().
10.                 windowManager.defaultDisplay.rotation)
11.             .build()
12.
13.     val cameraSelector = CameraSelector.Builder()
14.             .requireLensFacing(CameraSelector.LENS_FACING_BACK)
15.             .build()
16.
17.     cameraProvider.unbindAll()
18.
19.     cameraProvider.bindToLifecycle(this, cameraSelector,
20.             preview, imageCapture)
21. }
```

Note that we have configured a new use case in lines 8 through 11. These lines are the only CameraX configuration necessary to enable image capture.

Lines 9 and 10 instruct the **ImageCapture** class to store a specific rotation setting in the image metadata, and we set it to the current screen rotation. This rotation must be applied to the saved image to match the preview. It may need to be reset if the screen orientation changes, which is not being done in this example for simplicity.

Note: The getDefaultDisplay() function has been deprecated starting from API level 30. We still use it here because our application targets a lower API level, but you should remember this when writing applications that target recent Android versions.

We have added a cleanup operation in line 17. The application switches to another fragment to display the image once it is saved. It is necessary to unbind all use cases and bind new ones when the user returns to the preview; otherwise, it may not work correctly.

Saving the captured image as a JPEG file

The easiest way to capture an image is to save it as a JPEG image. To do so, we need the appropriate permissions if the user has configured their device to store images in external storage.

We first add a line to the application's manifest:

```
1. <uses-permission
2.     android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Then, we add this permission to the list of permissions being requested from the user. Instead of using the permission constant as earlier, an array is declared with all necessary permissions for the app for simplicity:

```
1.  private companion object {
2.      private val CAMERA_PERMISSIONS_REQUESTED = arrayOf(
3.              Manifest.permission.CAMERA,
4.              Manifest.permission.WRITE_EXTERNAL_STORAGE)
5.      private val PERMISSION_REQUEST_CODE = 100
6. }
```

Naturally, the **cameraPermissionsGranted()** function is also updated, so it uses the array:

```
1.  private fun cameraPermissionsGranted() =
2.      CAMERA_PERMISSIONS_REQUESTED.all {
3.          ContextCompat.checkSelfPermission(
```

```
4.              requireActivity().baseContext, it) ==
5.                  PackageManager.PERMISSION_GRANTED
6.       }
```

Ideally, each picture should be given a unique file name:

```
1.  private fun getPictureFile(mediaDir: File): File =
2.     File(mediaDir,
3.         SimpleDateFormat("yyyy-MM-dd-HH-mm-ss-SSS", Locale.US)
4.             .format(System.currentTimeMillis()) + ".jpg")
```

It is now necessary to determine the correct location to store the file:

```
1.  private fun getOutputDirectory(): File =
2.     Environment.getExternalStoragePublicDirectory(
3.         Environment.DIRECTORY_PICTURES).let {
4.           File(it, resources.getString(R.string.app_name)).apply {
5.                 mkdir()
6.             }
7.         }
```

Note: The getExternalStoragePublicDirectory() function has been deprecated and no longer provides a writable location starting from API level 29. We still use it here because our application targets a lower API level, but you should remember this when writing applications that target recent Android versions.

Adding a trigger button

It is impossible to take a picture if the user is unable to give that command, so we add a **`Take a picture`** button to our application. The following code excerpt illustrates how such a button might be placed in a **`ConstraintLayout`**:

```
1. <Button
2.     android:id="@+id/btnTakePicture"
3.     android:layout_width="wrap_content"
4.     android:layout_height="wrap_content"
5.     android:layout_marginBottom="16dp"
6.     android:text="@string/takePicture"
7.     app:layout_constraintBottom_toBottomOf="parent"
```

```
8.      app:layout_constraintEnd_toEndOf="parent"
9.      app:layout_constraintStart_toStartOf="parent" />
```

Finally, the image capture is triggered once the user clicks on the button. We need to be careful not to cause an application crash in case the camera is not fully initialized when the button is activated.

The button's listener is bound in the same way as in the previous chapters. We added the following code excerpt to the **onViewCreated()** function shown earlier:

```
1. btnTakePicture.setOnClickListener {
2.     takePicture()
3. }
```

The majority of the work for capturing an image is done in the **takePicture()** function, as shown here:

```
1.  private fun takePicture() {
2.   imageCapture?.let {
3.     val pictureFile = getPictureFile(getOutputDirectory())
4.     val outputOptions = ImageCapture.OutputFileOptions
5.         .Builder(pictureFile)
6.         .build()
7.     imageCapture.takePicture(
8.       outputOptions,
9.       ContextCompat.getMainExecutor(context),
10.      object : ImageCapture.OnImageSavedCallback {
11.        override fun onImageSaved(
12.              outputFileResults: ImageCapture.OutputFileResults) {
13.            val action = CameraPreviewFragmentDirections
14.                .actionCameraPreviewFragmentToPictureFragment(
15.                    pictureFile.absolutePath)
16.            requireView().findNavController().navigate(action)
17.        }
18.        override fun onError(exception: ImageCaptureException) {
19.            Log.e(TAG, "Failed to save the picture.", exception)
20.            Toast.makeText(context, R.string.picture_not_saved,
```

```
21.                  Toast.LENGTH_SHORT).show()
22.          }
23.        }
24.     )
25.   }
26. }
```

Line 2 ensures that the function does not attempt to use an uninitialized **ImageCapture** use case. The final image location and file name are calculated in line 3, which are then given to the **ImageCapture** object's **takePicture()** function in line 8.

Lines 10 through 23 create an instance of the callback object that receives the capture operation's result. The **onError()** function is called if an error occurs. When the operation completes successfully, the **onImageSaved()** function is called instead.

Lines 13 to 16 use a combination of the **Navigation** component and the **Safe Args** library to navigate to a fragment that displays the saved image. You do not need to use the **Navigation** component; we can implement this transition using activities and intents, as shown in previous chapters.

Displaying the captured image

A new fragment or activity can be used to display the image that was just captured. An **ImageView** can be added to its **ConstraintLayout** for this purpose:

```
1. <ImageView
2.     android:id="@+id/imageView"
3.     android:layout_width="wrap_content"
4.     android:layout_height="wrap_content"
5.     android:layout_marginBottom="16dp"
6.     app:layout_constraintBottom_toBottomOf="parent"
7.     app:layout_constraintEnd_toEndOf="parent"
8.     app:layout_constraintStart_toStartOf="parent"
9.     app:layout_constraintTop_toBottomOf="parent" />
```

The sample application also has a small text label to make it clear that the image being shown has been loaded from a file:

```
1. <TextView
2.     android:id="@+id/pictureCaption"
```

```
3.      android:layout_width="wrap_content"
4.      android:layout_height="wrap_content"
5.      android:layout_marginTop="16dp"
6.      android:text="@string/captured_picture"
7.      android:textSize="18sp"
8.      app:layout_constraintEnd_toEndOf="parent"
9.      app:layout_constraintStart_toStartOf="parent"
10.     app:layout_constraintTop_toTopOf="parent" />
```

Displaying the captured image with the correct rotation becomes easy with the help of the Glide library.

Add the following lines to the dependencies section in the module's Gradle build file:

```
1.  def glide_version = "4.12.0"
2.  implementation "com.github.bumptech.glide:glide:$glide_version"
3.   annotationProcessor  "com.github.bumptech.glide:compiler:$glide_
     version"
```

The full implementation of the fragment used in the demonstration application becomes very simple and is shown in the following code snippet.

If you choose to implement this view using an activity activated by an intent, do not forget to pass the file path as a parameter. It can be loaded using a call identical to the one in line 13.

The Safe Args library has generated the classes in line 3, so passing arguments to this fragment also became significantly simpler.

```
1.  class PictureFragment : Fragment() {
2.
3.   val args : PictureFragmentArgs by navArgs()
4.
5.   override fun onCreateView(
6.       inflater: LayoutInflater, container: ViewGroup?,
7.       savedInstanceState: Bundle?
8.   ): View? =
9.       inflater.inflate(R.layout.fragment_picture, container, false)
```

```
10.
11.     override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
12.         super.onViewCreated(view, savedInstanceState)
13.         Glide.with(this).load(args.filePath).into(this.imageView)
14.     }
15.
16. }
```

Conclusion

This chapter has shown how easy it is to use the high-level CameraX library to abstract several implementation details related to capturing images using the device's cameras.

Some auxiliary libraries were introduced that simplify some development tasks:

- The Safe Args library was used to pass parameters to fragments in a type-safe manner, combined with the Navigation component
- The Kotlin Android Extensions plugin generates code to transform child view lookups and getter and setter pairs into Kotlin properties
- The Glide library handles different image loading tasks efficiently

In the next chapter, the Camera application is transformed into a real-time image analysis application, integrating the machine learning models from the previous chapter in an Android application.

CHAPTER 8

Using the Image Recognition Model in an Android Application

Introduction

This chapter shows how we can use the machine learning models created in *Chapter 6, Training a Model for Image Recognition with TensorFlow*, in an Android application using TensorFlow Lite. An application is also created to analyze the images captured by the device's camera, building upon the techniques mentioned in the last chapter.

Structure

We cover the following topics in this chapter:

- Fundamentals of TensorFlow Lite
- Converting TensorFlow models into TensorFlow Lite
- Training an existing TensorFlow Lite model
- Setting up image analysis in the Android application
- Using TensorFlow Lite in the application
- Running TensorFlow Lite on dedicated hardware

Objectives

After going through this chapter, you are familiarized with the fundamental concepts of TensorFlow Lite, the procedure to convert regular TensorFlow models into their Lite version, and their integration in a working Android application.

Fundamentals of TensorFlow Lite

As its name implies, TensorFlow Lite is a framework that enables the usage of machine learning models in lower-powered devices such as:

- Mobile devices
- Embedded devices
- IoT devices

The extra care that was taken while creating TensorFlow Lite resulted in libraries optimized for size and power consumption. These optimizations mean that both the library and the model sizes are small (especially compared to the desktop versions) and that they are better suited for battery-operated devices.

Another important point is the difference in the format used in the saved models. TensorFlow Lite models are stored as a single file, whereas TensorFlow models are stored as a directory containing several files. This format makes TensorFlow Lite models easier to transfer to the devices.

The fundamental flow for using TensorFlow Lite models is the same as with TensorFlow, namely:

- Create the TensorFlow Lite model
- Use the TensorFlow Lite model to run inference in the device

The TensorFlow Lite models may be created in the following two ways:

- By converting a TensorFlow model into a TensorFlow Lite model
- By training an existing TensorFlow Lite model using custom data

It must be mentioned that, at the time of this writing, the set of features available for TensorFlow Lite models is not precisely the same as for TensorFlow. Some operators may not be supported, so some TensorFlow models may not be converted directly into TensorFlow Lite models. Documentation about the compatibility of operators between TensorFlow Lite and TensorFlow is available on the TensorFlow project web page (**https://www.tensorflow.org/**).

Converting TensorFlow models into TensorFlow Lite

The best (and recommended) way to convert a TensorFlow model into TensorFlow Lite format is to start from its saved representation. This statement means that the model was trained and saved, as we did in the previous chapter, before being converted.

A command line tool is available to convert models, but the current recommendation is that the conversion should be done through the Python API.

The **`TFLiteConverter`** class from the **`tf.lite`** package exposes a function conveniently named **`from_saved_model()`**. It knows how to load a saved model into a converter.

Once the converter is created with the saved model, a converted model can be created by calling the converter's **`convert()`** function.

The converted model is then ready to save to a file.

The following Python program shows the conversion procedure of the CNN model trained in the previous chapter against the Fashion MNIST dataset:

```
1.  import tensorflow as tf
2.
3. model = tf.lite.TFLiteConverter.from_saved_model(
4.     '../../chapter-6/fashion-cnn'
5. ).convert()
6.
7.  with open('converted-fashion.tflite', 'wb') as file:
8.     file.write(model)
```

We can optimize the converted models. These optimizations may be necessary or convenient in order to:

- Reduce the model size
- Reduce latency
- Convert the model to a data format supported by the target device

Generally speaking, the loss of accuracy caused by such optimization is minimal. However, model quantization can cause a significant drop in performance in some cases. As with all software development, testing and evaluating the model's performance before and after quantization is necessary.

We can choose from the following three optimizations:

- **Dynamic range quantization**: It quantizes the weights from the original floating-point values to 8-bit integers. It still uses floating-point kernels for computations.
- **Full integer quantization**: It quantizes the entire model but needs a reference data set to calibrate the variable tensors (input, activations, and output).
- **Float-16 quantization**: It quantizes the model weights to 16-bit floating-point values.

Each quantization type is appropriate for a specific use case and target device. For example, all are compatible with modern CPUs; float-16 quantization is suitable for GPUs; full integer quantization is compatible with microcontrollers that cannot handle floating-point values; and so on.

Optimizations are configured in the converter before calling the **`convert()`** function. The following code excerpt shows how to configure the converter to apply float-16 quantization:

```
1. converter = tf.lite.TFLiteConverter.from_saved_model('existing-model')
2. converter.optimizations = [ tf.lite.Optimize.DEFAULT ]
3. converter.target_spec.supported_types = [ tf.float16 ]
```

Training an existing TensorFlow Lite model

The TensorFlow Lite framework includes a library called Model Maker, which contains pre-built models that are suitable for some frequent tasks like:

- Image, text, and audio classification
- Object detection
- Recommendations

Users of these pre-built models only need to supply their custom dataset and train the chosen model with it. The API for using the different models is placed under the **`tflite_model_maker`** package.

As an example, let's train an image classification model with the Fashion MNIST dataset.

Lines 1 through 3 in the following Python program import the necessary libraries. Line 5 loads the dataset into memory, line 7 accomplishes model training, and line

9 performs the model's evaluation. Finally, line 11 saves the trained model in the **tflite** format.

Several image classification models are available in the library. We can choose one passing the **model_spec** parameter to the **create()** function. The library trains the EfficientNet Lite 0 model by default when this parameter is omitted.

```
1.  import tensorflow as tf
2.  from tflite_model_maker import image_classifier
3.  from tflite_model_maker.image_classifier import DataLoader
4.
5. (train_data, dummy, test_data) = DataLoader.from_tfds('fashion_
   mnist')
6.
7. model = image_classifier.create(train_data)
8.
9. loss, accuracy = model.evaluate(test_data)
10.
11. model.export(export_dir='./created-fashion-mnist')
```

The image classification model trained by default is a sequential model, and the training runs by five epochs, as follows:

```
1. Epoch 5/5
2. 1875/1875 [==============================] - 192s 103ms/step -
   loss: 0.7874 - accuracy: 0.8884
```

This default configuration reaches an accuracy of 88.84% during training with Fashion MNIST as shown:

```
1.  313/313 [==============================] - 32s 100ms/step - loss:
   0.7738 - accuracy: 0.8918
```

The corresponding evaluation step reported an accuracy of 89.18%.

We can switch to another supported model in the same category and modify the training parameters. The full list of supported models is available in the TensorFlow Lite Model Maker API documentation.

Setting up image analysis in the Android application

The new Android application built in this chapter uses the techniques explained in the previous chapter to implement camera access and a live camera preview.

For illustration purposes, the application uses a single activity to show the camera preview and image analysis results instead of multiple fragments this time.

The following fundamentals remain the same and are not repeated in this chapter:

- Add the required camera hardware and feature declarations to the application manifest.
- Implement requesting camera permissions in the activity. The starting point is the **`onCreate()`** function in the case of activities.
- Configure the CameraX library and start its preview use case.

The application's main activity uses a **`ConstraintLayout`** containing one preview view and one text view, as displayed in the following code excerpt. The latter is used to show the analysis result.

```
1.  <androidx.camera.view.PreviewView
2.      android:id="@+id/previewView"
3.      android:layout_width="match_parent"
4.      android:layout_height="0dp"
5.      app:layout_constraintBottom_toTopOf="@+id/
            txtClassificationResult"
6.      app:layout_constraintEnd_toEndOf="parent"
7.      app:layout_constraintStart_toStartOf="parent"
8.      app:layout_constraintTop_toTopOf="parent"/>
9.
10. <TextView
11.     android:id="@+id/txtClassificationResult"
12.     android:layout_width="wrap_content"
13.     android:layout_height="wrap_content"
14.     android:layout_marginBottom="16dp"
15.     android:text="@string/waiting_for_result"
16.     app:layout_constraintBottom_toBottomOf="parent"
```

```
17.       app:layout_constraintEnd_toEndOf="parent"
18.       app:layout_constraintStart_toStartOf="parent" />
```

A different use case is required for this new application—the image analysis use case. As its name implies, this use case enables running analysis tasks on each frame captured by the camera.

An **ExecutorService** is introduced in the application as the analysis tasks are expected to take some time. This executor is responsible for running tasks in a thread pool separate from the application's user interface and service threads. This way, the application's routine tasks are not affected by any image analysis algorithms.

The ExecutorService is used according to the following sequence of steps:

1. The first step is to declare the necessary property in the **Activity** class, as shown:

```
1.  private lateinit var cameraExecutor: ExecutorService
```

2. A single-threaded executor is then created in the **onCreate()** function, as follows:

```
1.  override fun onCreate(savedInstanceState: Bundle?) {
2.      super.onCreate(savedInstanceState)
3.      setContentView(R.layout.activity_main)
4.
5.      cameraExecutor = Executors.newSingleThreadExecutor()
6.
7.      // camera configuration goes here
8.  }
```

3. The executor needs to be disposed of when the activity is destroyed, so its cleanup is placed in the **onDestroy()** function, as shown:

```
1.  override fun onDestroy() {
2.      super.onDestroy()
3.      cameraExecutor.shutdown()
4.  }
```

The necessary infrastructure is now in place. The camera is configured, the preview is working, and an executor service is ready to service tasks.

The image analysis use case is added during the Camera configuration, just like the capture use case was added in the previous chapter.

Since the actual image analysis work needs to be implemented by the application, the next step is to create a class implementing the **`ImageAnalysis.Analyzer`** interface, as shown in the following code excerpt. This class contains a function that is called when a new image is ready to be analyzed.

```
1.  class ImageAnalyser : ImageAnalysis.Analyzer {
2.
3.      override fun analyze(image: ImageProxy) {
4.          image.close()
5.      }
6.
7. }
```

It is necessary to close the image reference before returning from the **`analyze()`** function so that the resources are freed and new images can be captured.

Finally, the newly created class can be linked to the new image analysis use case. If following the same structure as the image capture application, this configuration is done in the **`configureCameraUseCase()`** function, as follows:

```
1.  private fun configureCameraUseCase(
2.      cameraProvider: ProcessCameraProvider
3. ) {
4.
5.      // preview configuration
6.
7.      val imageAnalysis = ImageAnalysis.Builder()
8.          .setBackpressureStrategy(
9.              ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
10.         .build().apply {
11.             setAnalyzer(
12.                 cameraExecutor,
13.                 ImageAnalyser()
14.             )
15.         }
16.
17.     // additional camera selector and provider configuration
```

```
18.
19.     cameraProvider.bindToLifecycle(this, cameraSelector,
20.         preview, imageAnalysis)
21. }
```

There are two details worth mentioning about the preceding code excerpt.

As mentioned earlier, the image analysis is expected to take longer than the interval between two images captured by the camera. This expectation means that the analysis becomes late with regard to the image stream. One of two things can happen as a result of this delay: either all the image frames are buffered and wait for their turn be analyzed, or any incoming frames are dropped while the analyzer is busy.

The first detail is the choice of expected behavior, that is, the backpressure strategy setting in line 8 of the code excerpt. The **STRATEGY_KEEP_ONLY_LATEST** constant causes frames to be dropped while the analyzer is busy, so each new frame coming in for analysis is always the latest frame that the camera has captured. Buffering is activated by the **STRATEGY_BLOCK_PRODUCER** constant.

The second detail is the association of the newly created image analysis class with the analyzer use case in line 11 of the code excerpt. Here, we provide an instance of the class and the executor where the analysis runs.

Cropping captured images

Not all models work with the same aspect ratio, resolution, and color depth as the device's camera sensor. This means that the captured images need to be further processed before being submitted to the machine learning model.

Since the application user captures images with the help of a live preview, it makes sense that it should already show the appropriate image section and aspect ratio.

The specific aspect ratio depends on the model being used. All models trained in this and the previous chapter require a square image as an input with a specific size. Apart from ensuring the correct aspect ratio, the image must be resized before the model processes it. So, let's change our application's camera and preview configuration to reflect the aspect ratio requirement.

The first change is on the preview view's layout configuration by adding a dimension ratio constraint, as follows:

```
1. <androidx.camera.view.PreviewView
2.     android:id="@+id/previewView"
3.     android:layout_width="match_parent"
4.     android:layout_height="0dp"
```

```
5.      app:layout_constraintDimensionRatio="1:1"
6.      app:layout_constraintBottom_toTopOf="@+id/
          txtClassificationResult"
7.      app:layout_constraintEnd_toEndOf="parent"
8.      app:layout_constraintStart_toStartOf="parent"
9.      app:layout_constraintTop_toTopOf="parent"/>
```

Note the new **`app:layout_constraintDimensionRatio`** attribute with a value of **`1:1`**. This configuration results in a square preview view.

The second is to change CameraX's use case setup to include a viewport definition. Once set, CameraX calculates the largest possible crop rectangle based on the viewport settings. The preview view's viewport helps ensure that the crop rectangle matches the preview view ratio.

The **`configureCameraUseCase()`** function needs to be modified. A use case group needs to be explicitly created so that the viewport can be set as shown below:

```
1.  private fun configureCameraUseCase(
2.      cameraProvider: ProcessCameraProvider
3. ) {
4.      // preview, imageAnalisys and cameraSelector set up
5.
6.      val useCaseGroup = UseCaseGroup.Builder()
7.          .addUseCase(preview)
8.          .addUseCase(imageAnalysis)
9.      previewView.viewPort?.let {
10.         useCaseGroup.setViewPort(it)
11.     }
12.
13.     cameraProvider.unbindAll()
14.
15.     cameraProvider.bindToLifecycle(this, cameraSelector,
16.         useCaseGroup.build())
17.}
```

All images analyzed from this point onward have metadata that includes the crop rectangle to match the image the user sees in the preview view.

Converting captured YUV images to bitmaps

Image frames for analysis are delivered to the analyzer class in YUV format. Unfortunately, TensorFlow Lite does not support the YUV image format and only supports the RGB format.

> **Note: Images according to the YUV model are defined in terms of one luma component (Y) and two chrominance components (UV). The RGB model is completely different, defining images instead in terms of three channels: one for each primitive color (red, green, and blue).**

This incompatibility means that it is necessary to write code to convert YUV images to RGB. There are several ways to do this, one of which involves using the RenderScript framework.

Not only was the RenderScript framework designed to run computationally intensive tasks efficiently, but it comes with an intrinsic operation that handles this conversion.

The **`ImageAnalyser`** class can be extended with this conversion functionality. First, a parameter is added to the class's constructor because an Android Context reference is necessary to instantiate the RenderScript engine.

Then, both the RenderScript engine and the intrinsic operation are instantiated and saved as class properties to be reused later, as shown in the following code snippet:

```
1.  class ImageAnalyser(context: Context)
2.          : ImageAnalysis.Analyzer {
3.
4.      private val rs = RenderScript.create(context)
5.      private val script = ScriptIntrinsicYuvToRGB.create(
6.          rs, Element.U8_4(rs))
7.
8.      // implementation
9.  }
```

The actual conversion can be implemented as a Kotlin extension function, so its usage is simple.

The conversion function operates on an **`ImageProxy`** object, so it is not explicitly related to the image analysis task. Hence, it is a good approach to place it in a different Kotlin source code file. It is a top-level function, so, we can leverage Kotlin's capability of having top-level functions declared outside of a class, as shown here:

```
1.  fun ImageProxy.toBitmap(
2.          rs: RenderScript,
3.          script: ScriptIntrinsicYuvToRGB): Bitmap {
4.      val yuvBytes = toYuvByteArray()
5.
6.      val yuvType = Type.Builder(rs, Element.U8(rs))
7.          .setX(yuvBytes.size)
8.          .create()
9.      val input = Allocation.createTyped(
10.         rs, yuvType, Allocation.USAGE_SCRIPT
11.     )
12.
13.     val bitmap = Bitmap.createBitmap(
14.         width, height, Bitmap.Config.ARGB_8888
15.     )
16.     val output = Allocation.createFromBitmap(rs, bitmap)
17.
18.     input.copyFrom(yuvBytes)
19.     script.setInput(input)
20.     script.forEach(output)
21.
22.     output.copyTo(bitmap)
23.
24.     input.destroy()
25.     output.destroy()
26.
27.     val matrix = Matrix()
28.     matrix.postRotate(imageInfo.rotationDegrees.toFloat())
29.
30.     return Bitmap.createBitmap(
31.         bitmap, cropRect.left, cropRect.top,
```

```
32.         cropRect.width(), cropRect.height(),
33.         matrix, true
34.     )
35. }
```

The previous function allocates the necessary buffers, copies the YUV data from the underlying **ImageProxy** instance, and runs the intrinsic operation to perform the conversion. The image is copied to the target bitmap once the conversion is done.

Note that lines 27 to 30 of the previous code excerpt return a bitmap cropped and rotated according to the metadata present in the **ImageProxy**. This operation is necessary to ensure that the final bitmap has the same aspect ratio and orientation and contains the same image as the live preview.

For simplicity, the function allocates new buffers every time it is called. An optimization can be made by caching the buffers and reusing them between calls, and destroying them and allocating new buffers when the image dimensions change.

The function call made in line 4 of the preceding code excerpt converts the **ImageProxy** image data into one contiguous array. This is shown in the following code excerpt.

An interesting detail from this implementation is that the byte array returned is in NV21 format, containing the image layers in YVU order, as follows:

```
1.  private fun ImageProxy.toYuvByteArray(): ByteArray {
2.     require(format == ImageFormat.YUV_420_888)
3.         { "Invalid image format" }
4.
5.     val yBuffer = planes[0].buffer
6.     val uBuffer = planes[1].buffer
7.     val vBuffer = planes[2].buffer
8.
9.     val ySize = yBuffer.remaining()
10.    val uSize = uBuffer.remaining()
11.    val vSize = vBuffer.remaining()
12.
13.    val nv21 = ByteArray(ySize + uSize + vSize)
14.
15.    yBuffer.get(nv21, 0, ySize)
```

```
16.     vBuffer.get(nv21, ySize, vSize)
17.     uBuffer.get(nv21, ySize + vSize, uSize)
18.
19.     return nv21
20. }
```

In contrast, the **ImageProxy**'s layer array contains them in YUV order.

Using TensorFlow Lite in the application

Let's bring the machine learning models into our application. There are two APIs that can be used with TensorFlow Lite; they differ only in the model's ease of use for the developer:

- The TensorFlow Lite Interpreter API supports models without metadata. All models converted from TensorFlow lack metadata by default.
- The TensorFlow Lite Task Library and the Support Library work with models with metadata. Existing TensorFlow Lite models have metadata by default. We can also generate code for working with these models in Android applications.

Both model types have been created in the previous sections—*Converting TensorFlow models into TensorFlow Lite* and *Training an existing TensorFlow Lite model*—so that both APIs can be explored.

Creating modules for TensorFlow Lite APIs

Each API requires a different set of dependencies in the application, so it is cleaner to create one project module for each instead of bundling them all in the main application module.

It is easy to create a new Android library module for each API. Those who are familiar with Gradle projects can do this operation manually, but Android Studio can offer a helping hand. Just follow these steps:

1. Select the Project tree from the Project tool window
2. Right-click on the tree's root element (**ImageRecognition**)
3. Select **New** and then **Module**, as illustrated in *figure 8.1:*

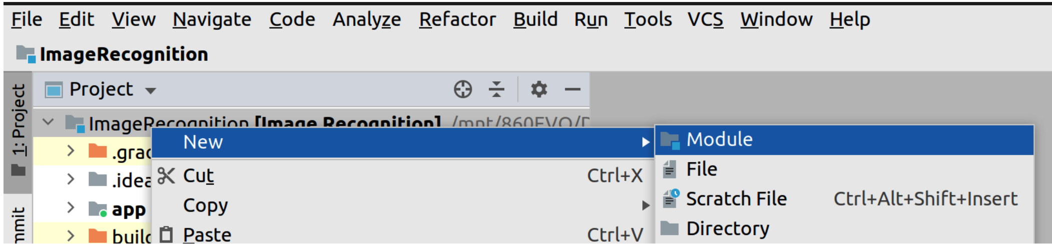

***Figure 8.1**: Adding a new module to the application*

4. Select the **`Android Library`** template in the module creation dialog box that appears. Choose a descriptive module name and a package name, as illustrated in figure 8.2.

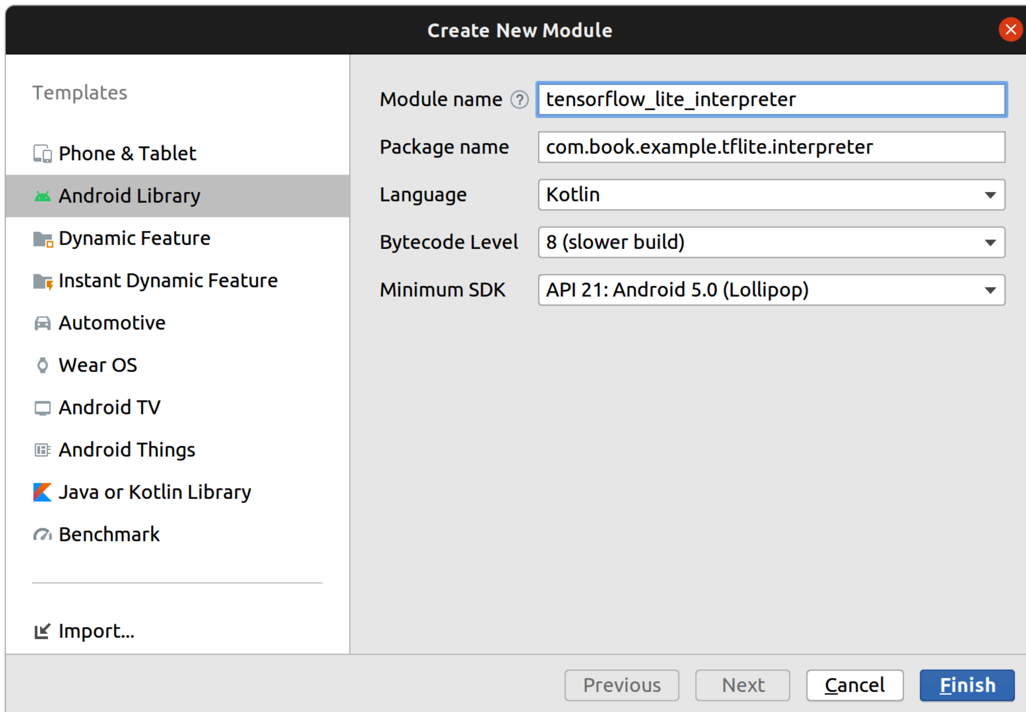

***Figure 8.2**: New Android library module settings*

We create two modules, one for each API. So, the suggestion is to name one as **`tensorflow_lite_interpreter`** and the other as **`tensorflow_lite_task`**.

5. Click on the **`Finish`** button to create the first module, and repeat the pricess to create the second module.

We need to add both modules' corresponding entries to the main application module's Gradle file once they are created. This is shown as follows:

```
1. dependencies {
2.
3.     // other dependencies
4.
5.     implementation project(':tensorflow_lite_interpreter')
6.     implementation project(':tensorflow_lite_task')
7.
8. }
```

It is now possible to use, from the main application, any code added to the new modules.

Due to how the TensorFlow Lite models included with the application are loaded, we must disable compression for the model file during packaging. Knowing that the model files have the **tflite** extension, compression is disabled by adding the following configuration to the Gradle project files in the application and modules:

```
1. android {
2.
3.     // other configurations
4.
5.     aaptOptions {
6.         noCompress "tflite"
7.     }
8.
9. }
```

Working with a converted model

First, the TensorFlow Lite model needs to be included in the project, as follows:

1. Create an **assets** directory inside the **src/main** directory of the **tensorflow_lite_interpreter** module. You may use Android Studio to create the directory using the following steps:

    a. Right-click on the **main** directory node in the Project tree

b. Select **New | Directory | assets**

2. Copy to this directory the **tflite** file corresponding to one of the converted models from *Converting TensorFlow models into TensorFlow Lite* or *Training an existing TensorFlow Lite model* as follows:

Figure 8.3: *Location of the model file inside a module*

3. Add a text file to the assets directory containing the model's labels, one label per line, as shown:

```
1. T-shirt/top
2. Trouser
3. Pullover
4. Dress
5. Coat
6. Sandal
7. Shirt
8. Sneaker
9. Bag
10. Ankle boot
```

4. Add the necessary dependencies on the TensorFlow Lite interpreter libraries to the module.

5. Add dependency entries to the **tensorflow-lite** and **tensorflow-lite-support** libraries in the **tensorflow_lite_interpreter** module Gradle file, as follows:

```
1. dependencies {
2.
3.     // other dependencies
4.
```

```
5.     implementation 'org.tensorflow:tensorflow-lite:2.5.0'
6.             implementation   'org.tensorflow:tensorflow-lite-
              support:0.2.0'
7.
8. }
```

A class can now be added with the necessary code for using the TensorFlow Lite model. The accompanying source code includes a **`ConvertedModelClassifier`** class in the **`com.book.example.tflite.interpreter`** package.

Working with a converted TensorFlow Lite image classification model is conceptually simple. The following steps are necessary to classify an image:

1. Load the classification labels
2. Load the TensorFlow Lite model
3. Create an instance of the **`Interpreter`** class based on the model
4. Set up input and output tensor processing pipelines, as necessary
5. Create the input buffer with the image that is to be classified
6. Create the output buffer that receives the classification results
7. Call the **`run()`** function to run the inference and obtain the results

The labels and model are loaded using methods from the **`FileUtil`** class, which is part of the TensorFlow Lite libraries, as shown in the following code excerpt:

```
1.  val labels = FileUtil.loadLabels(
2.          context, "labels-fashion.txt")
3.  val modelFile = FileUtil.loadMappedFile(
4.          context, "converted-fashion.tflite")
```

The interpreter is instantiated with the model as its constructor argument. Once the interpreter has loaded the model, its information about the input and output tensors can be used as follows:

```
1.  val interpreter = Interpreter(modelFile)
2.
3.  val inputTensor = interpreter.getInputTensor(0)
4.
5.  val outputTensor = interpreter.getOutputTensor(0)
6.  val outputBuffer = TensorBuffer.createFixedSize(
```

```
7.     outputTensor.shape(), outputTensor.dataType())
```

The image that is to be classified is loaded and transformed under the form of a **TensorImage** object, as shown here:

```
1.  private fun loadImage(bitmap: Bitmap)
2.          : TensorImage {
3.       val inputShape = inputTensor.shape() //{1, width, height, channels}
4.     val imageHeight = inputShape[1]
5.     val imageWidth = inputShape[2]
6.
7.     val tensorImage = TensorImage(inputTensor.dataType())
8.     tensorImage.load(bitmap)
9.     return ImageProcessor.Builder()
10.            .add(ResizeOp(imageWidth, imageHeight,
11.                ResizeOp.ResizeMethod.BILINEAR))
12.            .add(TransformToGrayscaleOp())
13.            .build()
14.            .process(tensorImage)
15.}
```

Note that the preceding excerpt uses an **ImageProcessor** object to transform the image into a format suitable for the model to use. Like with the Python version, the image needs to be resized and converted to grayscale.

An optimization can be made by caching the **ImageProcessor** instance until the orientation changes.

Inference can run once the bitmap is loaded in the correct format into a **TensorImage**, as shown in the following code excerpt:

```
1.  private fun classify(image: TensorImage): List<Pair<String, Float>> {
2.     interpreter.run(
3.         image.buffer,
4.         outputBuffer.buffer.rewind())
5.
```

```
6.   return TensorLabel(labels, outputProcessor.process(outputBuffer))
7.          .mapWithFloatValue
8.          .map { (key, value) -> Pair(key, value * 100.0f) }
9.          .sortedByDescending { (_, value) -> value }
10. }
```

The interpreter is called in lines 2 to 4 of the code excerpt to run the image through the model and gather its results in the reused output buffer.

The classification results are processed afterward. This function returns a list of all classes and their confidence levels as a probability, sorted in descending order. This list can be used to update the user interface with the most recent classifications.

Although not necessary with the Fashion-MNIST model trained in *Chapter 6, Training a Model for Image Recognition with TensorFlow*, an output processor has been used for demonstration purposes. The following excerpt declares it as an identity function, so the classifications remain unaltered:

```
1.  val outputProcessor: TensorProcessor = TensorProcessor.Builder()
2.          .add(NormalizeOp(0.0f, 1.0f))
3.          .build()
```

Any classification postprocessing, for example, when using quantized models, should be done at this point.

Finally, the image loading and the classification functions can be combined into one single operation, as depicted here:

```
1.  fun classify(bitmap: Bitmap):
2.          List<Pair<String, Float>> =
3.      classify(loadImage(bitmap))
```

This new **`classify()`** function takes a bitmap and returns the sorted classification labels with their corresponding confidence levels.

The **`ImageAnalyser`**'s callback can call the classifier's **`classify()`** function on each frame received from CameraX, as shown:

```
1.  private val classifier = ConvertedModelClassifier(context)
2.
3.  override fun analyze(imageProxy: ImageProxy) {
4.      try {
5.          val result = classifier.classify(
```

```
6.                imageProxy.toBitmap(rs, script)
7.            )
8.            // the result is then used somehow
9.        } finally {
10.           imageProxy.close()
11.       }
12. }
```

This excerpt shows one such invocation, assuming that the classifier code is organized in one class called **ConvertedModelClassifier**, whose instance is stored in a **classifier** property.

ImageAnalyser was implemented in the accompanying code to call a function once the result is ready. This function was implemented in the main activity so that the user interface is updated with the classification results.

The analysis task runs in a dedicated thread, so the user interface cannot be updated directly. Therefore, a **Handler** is used to run such an update in the application's main loop, as shown:

```
1.  class MainActivity : AppCompatActivity() {
2.      // (... other declarations ...)
3.
4.      private lateinit var imageAnalyser: ImageAnalyser
5.      private val mainThreadHandler: Handler =
6.          HandlerCompat.createAsync(Looper.getMainLooper())
7.
8.      override fun onCreate(savedInstanceState: Bundle?) {
9.          super.onCreate(savedInstanceState)
10.         // (... other code ...)
11.         imageAnalyser = ImageAnalyser(this) { result ->
12.             mainThreadHandler.post {
13.                 onPredictionResult(result)
14.             }
15.         }
16.         // (... other code ...)
17.     }
18.
```

```
19.         private fun onPredictionResult(result: List<Pair<String,
              Float>>) {
20.          txtClassificationResult.text = result
21.              .subList(0, 2)
22.              .map { "%s (%.0f%%)".format(it.first, it.second) }
23.              .joinToString(", ")
24.     }
25.
26.     // (... other functions ...)
27. }
```

The preceding excerpt shows how the analyzer is instantiated and configured so that the top two classification results are displayed to the user.

Naturally, the camera use case is configured with the **`ImageAnalyser`** instance instantiated in the **`onCreate()`** function instead of the local instance that was shown in an earlier section.

The application is ready to analyze the camera frames once everything has been put together.

Figure 8.4 is a screenshot of the application running on the author's device analyzing the model's test image, strategically loaded into a computer screen. We can see that the model successfully recognizes it as a T-shirt:

Figure 8.4: *Classifying the test image from a computer screen*

Real-world objects can work just as well, as *figure 8.5* shows. Lighting conditions, object placement and shape, and the scene's contrast will, of course, affect the classification results.

***Figure 8.5**: Classifying one of the author's new hiking boots*

Working with a trained existing TensorFlow Lite model

As stated earlier, by default, all pre-built models from the TensorFlow Lite library contain metadata describing the model's characteristics. The presence of metadata in the trained model means that the library and its related tools can automatically use this data, greatly simplifying the code needed to use the model in an application.

Thankfully, Android Studio has built-in support for TensorFlow Lite models with metadata.

Right-click on the **`tensorflow_lite_task`** module node in the project tree and select the **`New | Other | TensorFlow Lite Model`** options to add the model file to the module we have created for it. This is illustrated in *figure 8.6:*

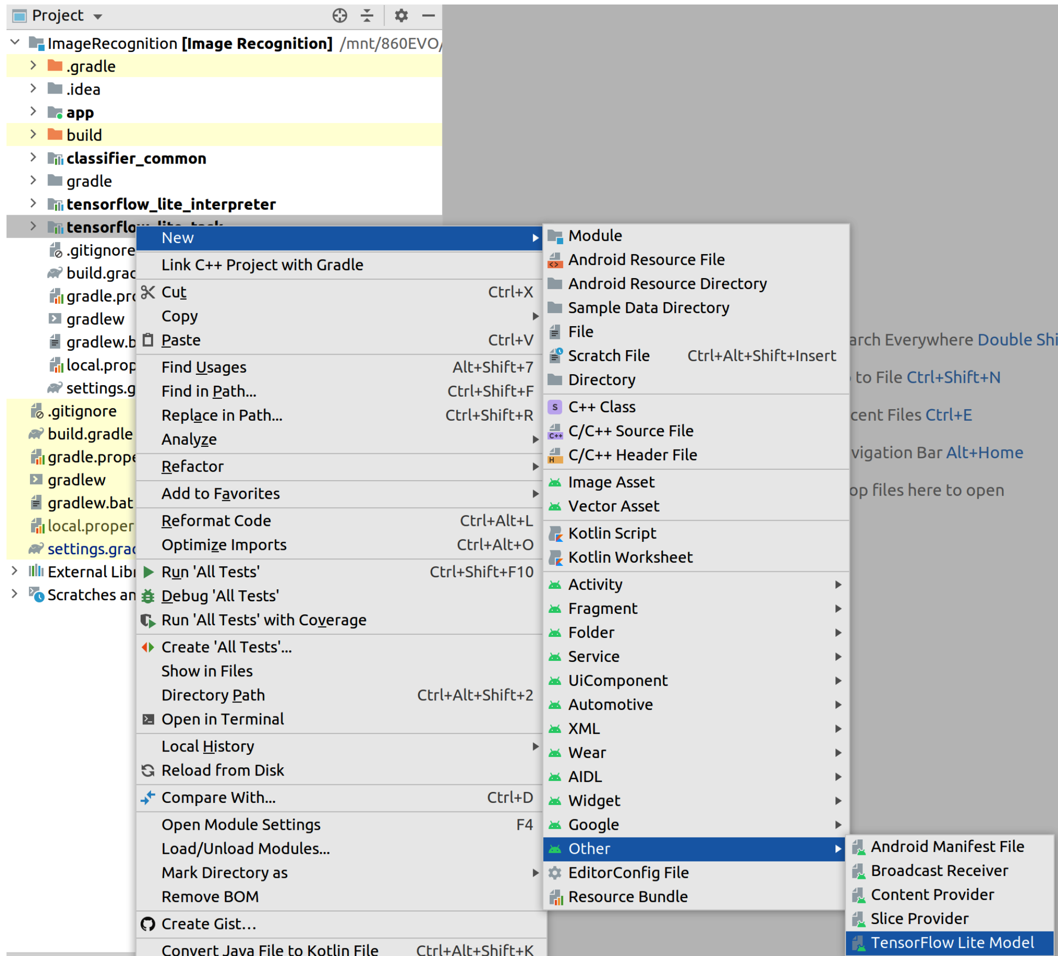

Figure 8.6: *Adding a TensorFlow Lite model to the project*

A dialog box appears, asking for the location of the TensorFlow Lite model file we wish to use. At the same time, it asks if we want the TensorFlow Lite dependencies to be added to the build configuration.

Choose the location of the pre-existing TensorFlow Lite model already trained with the Fashion MNIST dataset, saved in the *Training an existing TensorFlow Lite model* section. It is strongly advised to rename the **`model.tflite`** file to something more descriptive to avoid conflicts with the TensorFlow Lite API.

Figure 8.7 provides a suggestion of a descriptive name and recommended configuration options. Click on **`Finish`** to update the module's configuration.

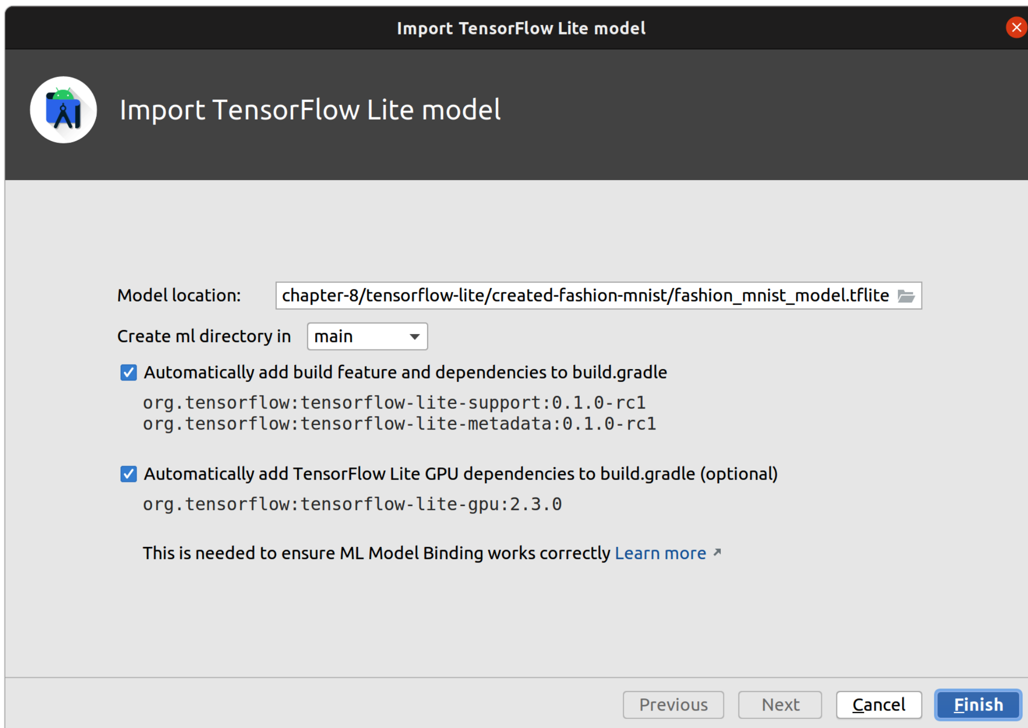

Figure 8.7: *New TensorFlow Lite model project configurations*

AndroidStudio then adds the model file to the **`ml`** project directory, includes the selected dependencies, and displays the information extracted from the model's metadata.

One additional step is necessary. As shown in *figure 8.7*, Android Studio may want to use specific versions of the TensorFlow Lite libraries. We have added newer TensorFlow Lite libraries to the **`tensorflow_lite_interpreter`** module's build configuration in the previous section. It is best to use identical versions in this module to avoid version conflicts or runtime problems.

Open the **`build.gradle`** file for the **`tensorflow_lite_task`** module and ensure that the dependencies section contains the following library versions:

```
1. dependencies {
2.
3.     // other dependencies
4.
5.     implementation "org.tensorflow:tensorflow-lite:2.5.0"
```

```
6.      implementation 'org.tensorflow:tensorflow-lite-support:0.2.0'
7.      implementation 'org.tensorflow:tensorflow-lite-metadata:0.2.0'
8.      implementation 'org.tensorflow:tensorflow-lite-gpu:2.5.0'
9.  }
```

The code necessary to run inference is relatively smaller than the code required to run inference on a converted model without metadata, as shown here:

```
1.  class TFLiteModelClassifier(context: Context) : Closeable {
2.
3.      private val model: FashionMnistModel =
4.          FashionMnistModel.newInstance(context)
5.
6.      override fun close() {
7.          model.close()
8.      }
9.
10.     fun classify(bitmap: Bitmap): List<Pair<String, Float>> =
11.         model.process(TensorImage.fromBitmap(bitmap))
12.             .probabilityAsCategoryList
13.             .sortedByDescending { category -> category.score }
14.             .map { category ->
15.                 Pair(category.label, category.score * 100.0f) }
16.
17. }
```

It could be even smaller, but a conversion is added in lines 14 and 15 in the code excerpt as we want to keep the same output format as before.

Note that the model class is generated automatically and is named after the **tflite** file name. The name suggested in *figure 8.7* was **fashion_mnist_model.tflite**, so the generated class was named **FashionMnistModel**.

Only a small change to the **ImageAnalyser** implementation is necessary to use this model, that is, create an instance of **TFLiteModelClassifier** instead of **ConvertedModelClassifier**, as follows:

```
1.  private val classifier = TFLiteModelClassifier(context)
```

The new model works as good as or even better than the converted model. *Figure 8.8* shows its results while running on the author's device:

***Figure 8.8**: The new model classifying one of the author's flip-flops*

Generating code for working with a model with metadata

The automatic code generation triggered when adding a model with metadata into the `ml` directory in an Android project is very useful. Still, it may sometimes be desirable to access its metadata or customize the input or output processors.

An additional TensorFlow package is available for generating an Android project with a model wrapper written in Java, Gradle build configurations, simple documentation, and of course, the model file itself.

This package is called TensorFlow Support and can be installed in the Python environment using the pip tool, as follows:

```
1. pip install tflite-support
```

The **`tflite_codegen`** command-line utility then becomes available. It needs the path to the model file, a Java package and class name for the generated code, and the destination directory.

The following excerpt shows an example of using this utility to generate code for our Fashion MNIST model:

```
1. tflite_codegen --model=./created-fashion-mnist/fashion_mnist_
   model.tflite \
2.   --package_name=com.book.example.tflite.task.wrapper \
3.   --model_class_name=FashionMnistModel \
4.   --destination=./created-fashion-mnist-wrapper
```

The resulting Java code can then be examined, used as-is, or customized as desired.

Running TensorFlow Lite on dedicated hardware

The techniques that were shown so far rely solely on the device's CPU to run inference tasks. Some devices have dedicated **Graphical Processing Units** (**GPUs**) and even specialized hardware, such as a **Digital Signal Processor** (**DSP**) or **Neural Processing Unit** (**NPU**). These features may result in reduced processing time during inference.

Graphical Processing Units

It is easy to enable TensorFlow Lite's support for GPUs using these steps:

1. Include the tensorflow-lite-gpu library in the project
2. Use the **`isDelegateSupportedOnThisDevice()`** function from the **`CompatibilityList`** class to determine whether the device is supported
3. Configure the interpreter to use the GPU delegate

Instantiate the **`Interpreter`** class with an **`Interpreter.Options()`** object carrying an instance of the GPU delegate when using it directly. This is shown as follows:

```
1.  val compatibility = CompatibilityList()
2.  val options = Interpreter.Options().apply {
3.     if (compatibility.isDelegateSupportedOnThisDevice) {
4.         addDelegate(GpuDelegate(
5.             compatibility.bestOptionsForThisDevice
6.         ))
7.     }
```

```
8. }
9. compatibility.close()
```

The procedure is similar when using a class generated automatically from a model with metadata, as shown in the following code excerpt:

```
1.  val compatibility = CompatibilityList()
2.  val options =
3.     Model.Options.Builder().apply {
4.         if (compatibility.isDelegateSupportedOnThisDevice) {
5.             setDevice(Model.Device.GPU)
6.         }
7.     }
8.     .build()
9. compatibility.close()
10. model = FashionMnistModel.newInstance(context, options)
```

A note of caution when using the GPU delegate is that the TensorFlow Lite interpreter must be instantiated in the same thread where the delegate is used to prevent runtime problems on some devices.

The code used to create an instance of the **`ImageAnalyser`** class was changed because the sample application uses a single-threaded executor service to run the image analyzer, which uses the interpreter to run inference.

This change causes these objects to be created within the scope of the said executor service. It is single-threaded, so the same thread is used to create the interpreter and run the analysis, as follows:

```
1. imageAnalyser = cameraExecutor.submit(Callable {
2.         ImageAnalyser(
3.             this,
4.             TFLiteModelClassifier(this)) { result ->
5.             mainThreadHandler.post {
6.                 onPredictionResult(result)
7.             }
8.         }
9.     })
10.    .get()
```

More than just GPUs

A new API is included with Android API level 28 (Pie) and above—the **Android Neural Networks API** (**NNAPI**). It provides acceleration for running models on devices with supported hardware, including GPU, DSP, or NPU. The TensorFlow Lite API can use the NNAPI, providing a simple interface for enabling such support.

When using an **`Interpreter`**, add the appropriate delegate to the **`Options`** object if running on a supported Android version, as shown:

```
1.  val options = Interpreter.Options().apply {
2.      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.P) {
3.          addDelegate(NnApiDelegate())
4.      }
5. }
```

If using a class generated automatically from a model with metadata, enable the corresponding flag in its **`Options`** object, as shown:

```
1.  val options =
2.      Model.Options.Builder().apply {
3.          if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.P) {
4.              setDevice(Model.Device.NNAPI)
5.          }
6.      }
7.      .build()
```

Faster performance is not guaranteed

We must keep in mind that the hardware acceleration delegates do not support all possible TensorFlow model operations or configurations.

Also, any additional application hardware requirements may not result in faster inferences. For example, the application may use the GPU for rendering, thus competing with the interpreter for GPU time.

So, the recommendation is to profile the application in as many devices as possible to determine if enabling these accelerations brings any speed improvements.

Conclusion

This chapter shows how to use TensorFlow Lite to run inference on an Android device based on image recognition models trained using TensorFlow in a more powerful computer.

We now know that there are pre-built models for various application classes created specifically with TensorFlow Lite in mind. These models can be trained with application-specific data and evaluated to see how suitable they are for whatever purpose a developer has in mind.

The next chapters build upon the concepts presented here, reusing the TensorFlow Lite framework and the CameraX image analysis use case as necessary.

Chapter 9
Detecting Faces with the Google ML Kit

This chapter covers the usage of the Google ML Kit library in an Android application to process images captured by the phone camera and detect faces that may be present in the image. Incorporating this library in our base image analysis application with CameraX is the first step toward building a face recognition application.

Structure

We cover the following topics in this chapter:

- Understanding the Google ML Kit
- Looking at face detection with the Google ML Kit
- Including Google ML Kit in our Android application
- Preparing the user interface
- Configuring the CameraX use cases
- Scanning images for faces in real-time

Objectives

By the end of the chapter, you have extended the base Android application using CameraX for live image analysis to include the detection of faces in captured images. You have a fully working face detection application then.

Additionally, you understand the different options available in the Google ML Kit library for detecting faces and the difference between face detection and face recognition.

Understanding the Google ML Kit

The field of computer vision has been active for decades, and some high-quality libraries have been developed to make computers "see" for a myriad of applications.

A reference in computer vision algorithms' implementation is the OpenCV (Open Source Computer Vision) library. This library has been growing over the past 20 years and includes machine learning algorithms. Its official website, **https://opencv.org**, stated that the library already included over 2,500 computer vision algorithms at the time of writing this book. Given its focus on performance and algorithm optimization, it is mainly intended for real-time computer vision applications.

OpenCV also supports Android, but the fact that it is a C++ native library, along with its size and feature set, makes its usage more complicated when compared with libraries explicitly developed for Android.

The Google ML Kit fills this usability gap for some of the most common computer vision tasks found in Android applications.

At the time of writing this book, the Google ML Kit contained implementations of machine learning models and algorithms tailored to the following tasks:

- Face detection
- Barcode scanning
- Image labeling
- Object detection and tracking
- Human body pose detection
- Text and digital ink recognition
- Selfie segmentation

This library is optimized for running on Android-based devices and does not need any cloud resources for its processing. Just like the image classification applications

developed in the previous chapters, the Google ML Kit library runs its models on the device.

This chapter focuses on the first algorithm—face detection. However, the basic techniques and concepts shown here also apply to the others, so it should not be too difficult to try them out as well.

Strictly speaking, the Google ML Kit library itself does not include the machine learning models required for each task. Instead, they must be added to the application using the library. The most straightforward way is to rely on Google Play Services to automatically download the model when the application is installed; we explore this option in the upcoming sections.

Note: This book does not cover using OpenCV in Android, but we recommend taking a look at its website to grasp the algorithms it provides. Its added complexity may pay off if it fits some other application.

Looking at face detection with the Google ML Kit

First and foremost, we must understand the significant difference between face detection and face recognition, or generally speaking, between object detection and object recognition.

The detection of an object—or face—in an image only means that the algorithm can perceive the presence of the said object and pinpoint its location in that image. So, it is unable to determine whether that object matches some other object seen earlier. In the case of face detection, it cannot tell that any detected face belongs to a specific person A or B, just that it is a face.

Often, face detection algorithms can detect and provide data about facial features, for example, the position or shape of the eyes, nose, and mouth. Since this data is extracted by face landmark detection models, such feature detection is not always combined with the face detection model. The Google ML Kit can also provide facial features. It calls them “landmarks” and “contours”, respectively, and they are only provided if requested. So, the algorithm runs slower when more information is requested about each face.

Other algorithms can use this additional data, such as some face recognition methods and expression or emotion classification.

The Google ML Kit is also capable of providing eye and mouth classifications, which means reporting that the face’s eyes are open or closed or that the mouth appears to be forming a smile.

Another feature offered by the Google ML Kit is face tracking. Tracking means assigning an identifier to each face and reporting their new positions as they move around the frame. This is not face recognition because it relies on visual tracking algorithms.

Finally, the Google ML Kit can provide face orientation toward the camera. This orientation can be expressed in terms of three angles: roll, pitch, and yaw.

All the preceding features can be toggled when configuring the Google ML Kit library in the application's code.

Including Google ML Kit in our Android application

Like with CameraX, including the Google ML Kit in an Android application is a matter of declaring the appropriate dependencies in the project and adding any desired configuration settings.

The starting point is an application already configured to use the CameraX library because this is the library used to capture images for analysis.

Enabling view binding

Although this step is not required to work with the Google ML Kit, it replaces the **`kotlin-android-extensions`** Gradle plugin to generate properties for accessing views in activities and fragments, eliminating the need to call **`findViewById()`**.

Applications built in the previous chapters used that plugin. This is a new way of doing it and is used in this project so that you can become familiar with both ways.

Activate this feature by modifying the **`android`** section in the module's Gradle file, as follows:

```
1. android {
2.   // other configurations
3.   buildFeatures {
4.     viewBinding = true
5.   }
6. }
```

Configuring the project's dependencies

We must choose the correct dependency to add the Google ML Kit libraries to the application. The choice depends on how the application accesses the model.

As mentioned in a previous section, the face detection model can be bundled with the application's package or downloaded at the time of installation with the help of Google Play Services.

Choose the following dependency to use Google Play Services to download the model at the time of installation:

```
1. dependencies {
2.
3.   // (other dependencies)
4.
5.   // Google ML Kit
6.   // Download the model dynamically through Google Play Services.
7.   implementation 'com.google.android.gms:play-services-mlkit-face-detection:16.1.7'
8. }
```

Alternatively, choose the following dependency to include the model in the application's package. Do not include both.

```
1. dependencies {
2.
3.   // (other dependencies)
4.
5.   // Google ML Kit
6.   // Bundle the model with the application.
7.   implementation 'com.google.mlkit:face-detection:16.1.1'
8. }
```

Any additional dependencies, such as the CameraX library, still need to be included. However, the applications developed in the previous chapters can be used as starting points.

Adding metadata for the Google Play Services

We must add some metadata to the application using the Google Play Services to download the model.

Add a **meta-data** element to the **application** element in the Android Manifest XML file, as demonstrated in the following code excerpt:

```
1. <application>
2.
3.   <meta-data
4.     android:name="com.google.mlkit.vision.DEPENDENCIES"
5.     android:value="face" />
6.
7.   <!-- activities here -->
8.
9. </application>
```

Even with a recently created application, the **application** element already contains attributes and other elements that should not be removed.

Preparing the user interface

This application aims to demonstrate how to work with the Google ML Kit for face detection and verify that the face detected in the source image can be extracted.

So, the sample application uses an interface that highlights the detected face and allows the user to find a good angle using a real-time camera preview.

The user interface's layout is shown in *figure 9.1*. Its XML definition is available in the accompanying code.

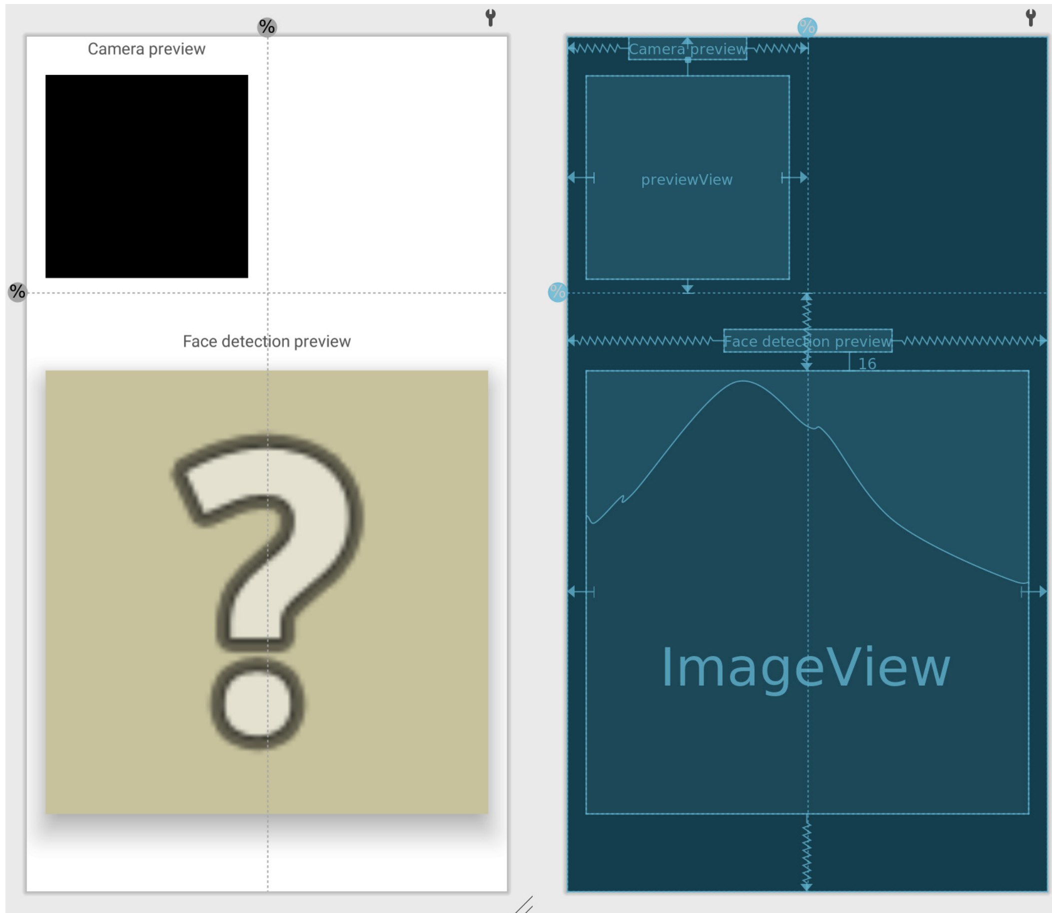

***Figure 9.1**: The sample face detection application's user interface*

The camera's preview is placed in the top-left corner, which allows us to use most of the screen to display the first face detected in the camera stream. An **`ImageView`** is used for this purpose.

As the application uses the view binding feature, its activity creation is slightly different from those built before. The binding code is generated automatically, but it needs to be connected to the activity's layout. This connection is made using the binding class to inflate the layout, which produces an instance of the binding class that can be used later to access the layout views.

The following code excerpt demonstrates the binding class's configuration:

```
1.  class MainActivity : AppCompatActivity() {
2.
3.   private lateinit var binding: ActivityMainBinding
4.
5.   override fun onCreate(savedInstanceState: Bundle?) {
6.     super.onCreate(savedInstanceState)
7.     binding = ActivityMainBinding.inflate(layoutInflater)
8.     setContentView(binding.root)
9.   }
10.
11. }
```

Now, it is time to set up the CameraX use cases.

Configuring the CameraX use cases

Almost all of the CameraX use case configuration is the same as mentioned in the previous chapter, except for the target resolution and the image analysis class.

Creating the image analysis class

A new image analysis class is created to support the Google ML Kit face detection process. There is one requirement—the application must display the detected face.

This requirement is fulfilled by providing the image analysis class with a function to be called when the face detection results are available, as shown in the following code excerpt:

```
1.  class RealtimeFaceDetector(
2.   context: Context,
3.   private val receiver: (Bitmap?) -> Unit
4. ): ImageAnalysis.Analyzer {
5.
6.   override fun analyze(proxy: ImageProxy) {
7.     proxy.close()
8.   }
```

```
9.
10. }
```

Line 3 in the preceding code fragment declares such a callback function. Note that the function's argument is a nullable bitmap. This choice is justified by the desire to indicate that no face has been detected in the camera pictures, which is achieved by passing a null bitmap to the callback function.

The callback function is implemented in the main activity, as illustrated in the following code excerpt. When a valid bitmap is received, it is shown in the **ImageView** dedicated to displaying the face detected. Otherwise, the standard Android help icon is used as a placeholder to indicate that no face was detected.

```
1.  private fun onFaceDetected(face: Bitmap?) {
2.   if (null != face)
3.     binding.faceView.setImageBitmap(face)
4.   else
5.     binding.faceView.setImageResource(
6.       android.R.drawable.ic_menu_help)
7.  }
```

Any other placeholder can also be used; this Android resource was chosen for simplicity.

```
1.  class MainActivity : AppCompatActivity() {
2.   // (other properties)
3.
4.   private lateinit var cameraExecutor: ExecutorService
5.   private lateinit var imageAnalyser: RealtimeFaceDetector
6.
7.   private val mainThreadHandler: Handler =
8.     HandlerCompat.createAsync(Looper.getMainLooper())
9.
10.  override fun onCreate(savedInstanceState: Bundle?) {
11.    // (activity creation)
12.
13.    cameraExecutor = Executors.newSingleThreadExecutor()
14.    imageAnalyser = RealtimeFaceDetector(this) { face ->
```

```
15.         mainThreadHandler.post {
16.           onFaceDetected(face)
17.         }
18.       }
19.
20.       binding.previewView.post {
21.         if (cameraPermissionsGranted()) {
22.           configureCamera()
23.         } else {
24.           requestPermissions(
25.             CAMERA_PERMISSIONS_REQUESTED,
26.             PERMISSION_REQUEST_CODE)
27.         }
28.       }
29.     }
30.
31.     override fun onDestroy() {
32.       super.onDestroy()
33.       cameraExecutor.shutdown()
34.     }
35.
36.     // (face detection callback)
37. }
```

The preceding code excerpt shows the same activity creation and destruction logic as in the previous chapter, adjusted to use view binding and the new image analysis class.

The logic for handling camera permissions remains the same and is not shown here again, but the use case configuration is slightly different.

The following code excerpt shows the CameraX use case configuration used in this Google ML Kit face detection application:

```
1.  private fun configureCameraUseCase(
2.    cameraProvider: ProcessCameraProvider
```

```
3. ) {
4.  val preview = Preview.Builder()
5.    .build().apply {
6.      setSurfaceProvider(binding.previewView.surfaceProvider)
7.    }
8.
9.  val imageAnalysis = ImageAnalysis.Builder()
10.        .setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_
   LATEST)
11.   .setTargetResolution(Size(480, 360))
12.   .build().also {
13.     it.setAnalyzer(
14.       cameraExecutor,
15.       imageAnalyser
16.     )
17.   }
18.
19. val cameraSelector = CameraSelector.Builder()
20.     .requireLensFacing(CameraSelector.LENS_FACING_FRONT)
21.   .build()
22.
23. val useCaseGroup = UseCaseGroup.Builder()
24.   .addUseCase(preview)
25.   .addUseCase(imageAnalysis)
26. binding.previewView.viewPort?.let {
27.   useCaseGroup.setViewPort(it)
28. }
29.
30. cameraProvider.unbindAll()
31.
32. cameraProvider.bindToLifecycle(this, cameraSelector,
33.   useCaseGroup.build())
34.}
```

It is identical to the one used in the previous chapter, with two differences:

- Line 11 specifies a target resolution. This target resolution is chosen according to the Google ML Kit's minimum resolution requirements, which are discussed in the next section. It does not mean that CameraX provides this exact resolution but that it attempts to choose a resolution that is a good fit.
- Line 20 requests a front-facing camera. This request aims to make it easier to test with the user's face, but the back camera can also be used.

Scanning images for faces in real-time

The Google ML Kit library can be brought into the application now that CameraX is set up as the necessary image capture infrastructure.

Configuring the minimum image resolution

One important detail about the Google ML Kit face detection library is that it has a requirement about the size of the image to be analyzed. It must not be smaller than 480×360 pixels. On the other hand, bigger resolution images may mean better accuracy, but they also mean longer processing times.

So, the recommendation is to run the detection on images with resolutions close to this minimum size when processing time needs to be kept as short as possible.

This reason is why our face detection application tells CameraX that its target resolution is 480×360. Of course, the library then attempts to choose the best possible resolution available.

The next step is to configure and create the face detector object. It's called to analyze each image captured by the camera, so its usage is restricted to our application's image analyzer class—the **RealtimeFaceDetector** class.

Creating the face detector object

The face detector object is easy to create. Just use the **getClient()** factory function from the **FaceDetector** class:

```
1.  private val detector: FaceDetector = FaceDetection.getClient(op-
    tions)
```

As usual, the devil lies in the details. Note that the **getClient()** function takes an argument aptly named **options**.

The Google ML Kit face detector provides several features that can be enabled and configured at the time of creation. These features have been mentioned at the

beginning of this chapter and are detailed as follows. *Table 9.1* contains a summary of the settings used to configure them on the Google ML Kit face detector.

Performance mode

The face detector can work in a mode that favors speed over detection accuracy or the other way around.

Landmark detection

The landmark detection feature allows the application to detect facial features and receive their corresponding positions as part of the face detection process.

At the time of writing, the following facial features are supported by the Google ML Kit face detector:

- Eyes
- Ears
- Mouth
- Nose base
- Cheeks

Most of these features are subdivided as left, right, top, or bottom; for example, the left eye, the bottom mouth section, the right cheek, and so on.

Naturally, only the facial features visible in the image can be detected. So, the angle at which the camera faces the person's head determines the number of landmarks detected and available in the face detector's results.

Contour detection

The contour detection feature is closely related to landmark detection because it provides the application with a set of points describing the contour of each landmark.

The face detector can provide the following contours, at the time of writing:

- Face
- Eyebrows
- Eyes
- Cheeks
- Lips
- Nose bridge and bottom

Like the landmark detection feature, only the visible landmark contours may be detected.

Classification

The Google ML Kit face detector currently supports two classifications:

- Eyes open
- Smiling

These classifications are provided as a percentage value representing the level of confidence in their presence. In other words, it represents how confident the model is that the face detected is smiling or has its eyes opened.

Minimum face size

We can define the minimum face size acceptable, expressed as a percentage of the face's width in relation to the image width. The default value is 0.1, which means that faces must be as wide as approximately 10% of the image, for example, at least 72 pixels wide in a 720 pixels wide image. These values are to be regarded as approximations because the detector can return results for slightly smaller faces than the requested size.

Changing the minimum face size may impact the detection speed. Requesting bigger faces means that smaller ones are ignored, so detection runs faster. On the other hand, lowering the threshold may mean that detection runs slower in comparison.

Face tracking

This feature assigns an identifier to each face detected so that the application can keep track of changes in the face's position in successive images.

Note that this is not real face recognition. Instead, the Google ML Kit uses motion detection algorithms and position comparison to provide this tracking feature.

Feature	Setting	Description
Performance mode	`PERFORMANCE_MODE_FAST`	Favor speed over accuracy
	`PERFORMANCE_MODE_ACCURATE`	Favor accuracy over speed
Landmark detection	`LANDMARK_MODE_NONE`	Landmark detection is disabled
	`LANDMARK_MODE_ALL`	Landmark detection is enabled
Contour detection	`CONTOUR_MODE_NONE`	Contour detection is disabled
	`CONTOUR_MODE_ALL`	Contour detection is enabled
Classification mode	`CLASSIFICATION_MODE_NONE`	Classification is disabled
	`CLASSIFICATION_MODE_ALL`	Classification is enabled

Feature	Setting	Description
Minimum face size	A floating-point value between 0.0 and 1.0	Represents a percentage of the image's width
Face tracking	A call to the `enableTracking()` function	Enable face tracking

***Table 9.1**: The different face detection features*

The configuration of all these features is accomplished by using an instance of the **`FaceDetectorOptions`** class. It is recommended to configure as many as possible even if the default settings are repeated, because then there can be no questions about the expected results.

The following code excerpt shows how the face detector options are defined in this chapter's application:

```
1.  private val options: FaceDetectorOptions=FaceDetectorOptions.
    Builder()
2.   .setPerformanceMode(FaceDetectorOptions.PERFORMANCE_MODE_FAST)
3.   .setLandmarkMode(FaceDetectorOptions.LANDMARK_MODE_NONE)
4.   .setContourMode(FaceDetectorOptions.CONTOUR_MODE_NONE)
5.   .setClassificationMode(FaceDetectorOptions.CLASSIFICATION_MODE_
       NONE)
6.   .setMinFaceSize(0.1f)
7.   .build()
```

All of these settings correspond to the Google ML Kit's default values.

Analyzing an image

Once the face detector object is created with the desired options, image analysis is triggered by calling its **`process()`** function.

The **`process()`** function accepts one argument of **`InputImage`** type that can be created using one of the several factory methods this type provides.

The application receives images captured by CameraX as an instance of **`ImageProxy`**. In turn, this proxy object provides an instance of **`Image`** containing the captured image in YUV format. Thankfully, Google ML Kit can process YUV images directly and efficiently.

So, it is recommended to use the **`fromMediaImage()`** factory method from **`InputImage`** to create the necessary parameter for the **`process()`** function.

The first approach to writing the contents of the image analyzer's **analyze()** function could be the following code excerpt:

```
1.   override fun analyze(proxy: ImageProxy) {
2.    val originalImage = proxy.image
3.    if (originalImage == null) {
4.      proxy.close()
5.      return
6.    }
7.    detector.process(InputImage.fromMediaImage(
8.        originalImage, proxy.imageInfo.rotationDegrees))
9.      .addOnCompleteListener {
10.       proxy.close()
11.     }
12. }
```

Note that the **process()** function also needs to know the input image's rotation regarding the final orientation desired. This information is necessary to use the Google ML Kit face detection model correctly and translate the output data into the expected orientation. It is retrieved from the CameraX image proxy.

Remember that CameraX's image proxy must be released before the **analyze()** function can be called again with a new image. Plus, image processing runs as an asynchronous task, so control is removed from the **analyze()** function before the processing results are ready.

Therefore, it is essential to add a listener method to free the image proxy after the image processing completes. This code is shown in lines 9 to 11 of the preceding excerpt.

Lines 2 to 6 are added to increase the application's robustness: **ImageProxy** declares that the result of the **getImage()** function (or its Kotlin property version) may be a null reference. We prevent null pointer exceptions by validating the reference before using it.

Using the face detection results

The image processor produces a list with information about any faces found in the image once the image processing is complete.

As explained earlier, image processing happens asynchronously, so it is likely not ready when the **analyze()** function returns. This problem can be solved by adding

another callback that is called when the analysis completes successfully. This callback then goes through the list of faces detected and uses their information as needed.

The following code excerpt shows how the new callback is added to the asynchronous task returned by the **`process()`** function.

Note how the first operation handles the possibility that the list of faces is empty in line 4. An empty list means that no faces were detected in the original image. In this case, a null face bitmap is given to the **`receiver()`** function.

```
1.  detector.process(InputImage.fromMediaImage(
2.      originalImage, proxy.imageInfo.rotationDegrees))
3.    .addOnSuccessListener { faces ->
4.      val faceBitmap = faces.firstOrNull()?.let { face ->
5.        val faceBounds = cancelRotation(face.boundingBox, proxy)
6.        if (faceBounds.setIntersect(proxy.cropRect, faceBounds)) {
7.          proxy.toBitmap(rs, script, faceBounds)
8.        } else {
9.          null
10.       }
11.     }
12.     receiver(faceBitmap)
13.   }
14.   .addOnCompleteListener {
15.     proxy.close()
16.   }
```

The second operation seems strange—a function called **`cancelRotation()`** is used to transform the bounding box reported by the face detector. Remember that the results are in the desired orientation's coordinate space, which can differ from the original image orientation. So, this function is necessary for two reasons.

The image conversion to bitmap in line 7 uses a function identical to the one in the previous chapter, except that the desired bounds are provided as an argument. The creation of the final bitmap uses an Android function that crops the bitmap before rotating it. This sequence means the bounding box's coordinates must be in the same coordinate space as the original image.

We must ensure that the bounding box falls within the image's dimensions, preventing problems caused by rounding errors. This verification is done in line 6 and also needs to be in the original image's coordinate space.

An alternative is to convert the original image to a bitmap and then work on the converted bitmap to extract a second bitmap containing only the face detected. This is, however, less efficient than transforming the coordinates.

The **`cancelRotation()`** function is shown in the following code excerpt:

```
1.  private fun cancelRotation(bounds: Rect, proxy: ImageProxy): Rect
   {
2.   val translatedRect = RectF(bounds)
3.   val matrix = Matrix()
4.   matrix.setRotate(
5.     proxy.imageInfo.rotationDegrees.toFloat(),
6.     proxy.cropRect.centerX().toFloat(),
7.     proxy.cropRect.centerY().toFloat())
8.   matrix.mapRect(translatedRect)
9.     return  Rect(translatedRect.left.toInt(),  translatedRect.top.
        toInt(),
10.    translatedRect.right.toInt(), translatedRect.bottom.toInt())
11. }
```

Its main job is to rotate the input rectangle around the center of the image proxy's cropping rectangle in the opposite direction of the image proxy's rotation angle.

Image processing may fail for some reason. The application should also be aware of any errors that occur during processing, and this can be accomplished by adding the third and final callback—the failure callback function.

The failure callback's job is to inform the receiver that no face could be extracted from the image.

The final image analysis function is then shown as follows:

```
1.  override fun analyze(proxy: ImageProxy) {
2.  val originalImage = proxy.image
3.  if (originalImage == null) {
4.    proxy.close()
5.    return
6.  }
7.  detector.process(InputImage.fromMediaImage(
8.      originalImage, proxy.imageInfo.rotationDegrees))
```

```
9.    .addOnSuccessListener { faces ->
10.     val faceBitmap = faces.firstOrNull()?.let { face ->
11.       val faceBounds = cancelRotation(face.boundingBox, proxy)
12.       if (faceBounds.setIntersect(proxy.cropRect, faceBounds)) {
13.         proxy.toBitmap(rs, script, faceBounds)
14.       } else {
15.         null
16.       }
17.     }
18.     receiver(faceBitmap)
19.   }
20.   .addOnFailureListener {
21.     receiver(null)
22.   }
23.   .addOnCompleteListener {
24.     proxy.close()
25.   }
26.}
```

Figure 9.2 illustrates the face detection application running on the author's device. Upon pointing the camera at a newspaper photograph, the Google ML Kit face detector detects a person's face. The previous image analysis function then extracts the detected face from the original image and sends it to the receiver. The latter displays the received face in the user interface's image view.

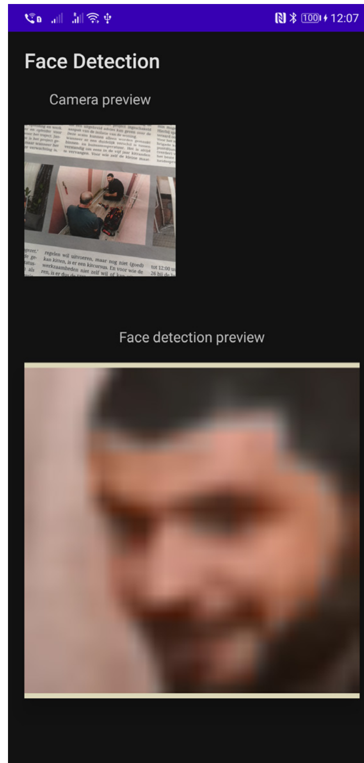

***Figure 9.2**: Detecting a face from a newspaper clipping*

Note that the face is blurry because it is pretty small in the original image. This configuration was chosen on purpose to protect the person's privacy.

Conclusion

The Google ML Kit libraries contain a face detection model that is easy to use combined with the CameraX framework and produces accurate results. Its additional features can be used to fine-tune its behavior and provide hints about the image that may be useful for some application purposes.

For example, a portrait application might want to know if the user is smiling to trigger image capture. On the other hand, an application that needs the user's image for face recognition may refuse to accept images where they are not facing the camera directly or have their eyes closed.

The next chapter takes the code base shown here and goes one step further, adding face recognition capabilities.

CHAPTER 10

Verifying Faces in Android with TensorFlow Lite

Introduction

This chapter takes you through the process of using a TensorFlow Lite model in an Android application to recognize the most prominent face in images previously captured by the CameraX library and analyzed by the Google ML Kit library.

Structure

We cover the following topics in this chapter:

- Understanding face recognition
- Understanding normalization
- Looking at the FaceNet model
- Working with the MobileFaceNets model
- Using the Euclidean distance to identify a face
- Incorporating the MobileFaceNets model in an Android application

Objectives

After completing this chapter, you have a simple Android application with complete face verification. Face recognition is added to the base Android application using CameraX and the Google ML Kit for live face detection in images. You have understood the fundamental mode of operation of most face recognition algorithms, and the method chosen for working with the results of the face recognition model is clear.

Understanding face recognition

As mentioned in the previous chapter, there is a significant but subtle difference between face detection and face recognition algorithms.

A face detection algorithm only aims to understand if a (human) face is present in a picture and provide its coordinates, very much like an object detection algorithm specialized to work with faces.

A face detection algorithm does not need to detect faces in a way that allows it to identify a face later on, possibly in a different setting, as being the same face it detected earlier. That is the job of the face recognition algorithm.

Humans possess a potent face detection and recognition mechanism that works effortlessly and produces accurate results, even when looking at a small part of a face. Humans can typically recognize a face by looking only at the triangle formed by the eyes and nose. However, only the eyes and eyebrows are often enough for accurate identification.

Besides face detection, one can say that there are two categories of tasks: face verification and face identification.

Face verification is the process of comparing one face to another or verifying whether a newly processed face matches pre-stored information about a specific face. It is a one-to-one matching process.

On the other hand, face identification (also called recognition) intends to find the identity of a newly processed face in a set of pre-stored information about several faces, for example, searching for a person's face in a database. It is a one-to-many matching process.

Note: As confusing as it may be, some sources use the term face identification to refer to face verification.

Researchers understood long ago that computer programs could not compare images of faces—or anything else for that matter—based on their pixel data only.

This sort of comparison does not produce reliable results because slight changes in lighting or position produce images with very different pixel data.

Logically, more targeted and detailed information is necessary, so the focus of research moved toward the representation of facial topology instead.

Intuitively, facial recognition algorithms work mostly by extracting information about points of interest in images of faces. These points of interest are called features.

The following figure illustrates how an imaginary algorithm might generate information about the facial features it finds:

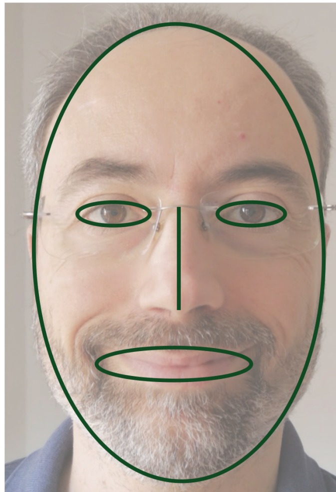

***Figure 10.1**: Possible features extracted by an algorithm*

The green lines in *figure 10.1* illustrate a few possible sets of points representing the position and boundaries of the most common facial features used by these algorithms. These include:

- Eye size and position
- Face or head format
- Nose format and position
- Mouth format and position

Note: An interesting detail is that such features may be extracted explicitly, like the Google ML Kit facial detection algorithm allows, or implicitly within the algorithm.

Early algorithms used techniques like image partitioning, histograms, and statistical analysis to extract helpful information in the form of features while simultaneously discarding irrelevant or redundant information.

Face recognition algorithms were carefully built, and the feature set was manually fine-tuned until the evolution of machine learning (ML) made **Convolutional Neural Networks** (**CNNs**) accessible.

CNNs are good feature extractors because, as it is understood today, they effectively extract low-level image features (for example, edges and circles) in their first layers, leaving the deeper layers with the task of partitioning the feature space and learning about image features. Today, the forefront of research relies on CNNs to perform feature extraction for face recognition and verification applications.

Features extracted by a CNN are not necessarily structured, that is, they are not necessarily presented as a vector with fixed positions for eyes, nose, or any other feature learned by the network because their internal structure depends on the CNN's configuration.

Nowadays, facial landmarks are often used explicitly to perform face image alignment before feeding the face image to the recognition model.

The approach used in this chapter uses the MobileFaceNets CNN, and it is independent of the structure of the CNN's output vector. The MobileFaceNets CNN was explicitly developed for face verification on mobile devices. Other state-of-the-art models, such as the FaceNet model, are computationally heavy or oversized to be used effectively on mobile devices.

Understanding normalization

This chapter uses terms like normalization, distance, Euclidian distance, and Euclidian normalization. It is essential to build some understanding of their meanings before we refer to them.

Normalization is a process whereby variable values are reduced to a standardized form so that they become comparable, that is, they can be compared to each other. One way of understanding the need for such an operation is unit conversion, for example, to compare temperatures measured in degrees Fahrenheit to temperatures measured in degrees Celsius, a conversion is required from one to the other.

There are many other forms of normalization, depending on the application. For example, the intensity of each pixel in the input images was scaled from [0, 255] to [0, 1] in *Chapter 6*, *Training a Model for Image Recognition with TensorFlow*. This scaling is also a form of normalization.

Normalizing a ratio scale

Some ML models produce outputs that describe an object instead of classifying it. For example, the models trained in *Chapter 6*, *Training a Model for Image Recognition*

with TensorFlow, were classification models that assigned a class to an object with a certain degree of confidence. On the other hand, the models used for object identification often describe them. For example, face identification models produce outputs that describe the face's features, so we must ensure that these outputs are somehow comparable among themselves.

If we visualize a fixed number of points placed over an object that describe the object's structure and then imagine that the object moves around the space, the object remains the same while the relative position of the points in relation to the observer changes. So, we can say that their scale changes, which is especially visible if the object moves nearer or farther.

In a ratio scale, measurements are unique up to a congruence or proportionality transformation. For example, if we take the points describing the object mentioned in the previous paragraph, they are closer together when the object is far away and further apart when the object is close. However, they describe the same object. It is necessary to reduce these sets of points to the same scale to compare them, in other words, to apply some ratio.

Going back to TensorFlow, we can draw a comparison between these imaginary points and the output of a ML model. Both represent a description of an object, and both assume different representations as the object moves in space.

We also know that TensorFlow ML models output a tensor and that a tensor is a generalization of a vector. So, we can say that these models' output is a vector.

The normalization of a ratio scale is accomplished by applying a transformation that creates a normalized version of the vector with length 1. Saying that a vector's length is 1 is equivalent to saying that its Euclidean norm equals 1. The Euclidian norm is a function that calculates the length of a vector from the origin in the Euclidian space.

Note: There is nothing special about the Euclidian space. It is the classical geometry space that we all studied since our introduction to geometry in school.

The outcome of the normalization process becomes apparent once we look closely at *figure 10.2.* Its top half represents one set of vectors, and the bottom half represents the same set of vectors after Euclidean (or L2) normalization.

When we compare both sets, it is clear that all vectors maintain their orientation in the normalized version. However, their lengths have been normalized and are now equal to 1. Note that all the points lie on a circle after normalization.

Once all the vectors produced by the model have the same length, they can be compared.

Note: In mathematics, a norm is a function from a vector space to the set of non-negative real numbers. In other words, it produces a single value bigger than or equal to zero when given a vector.

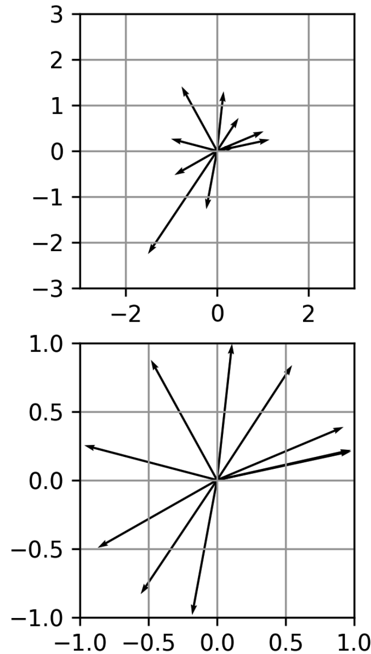

Figure 10.2: *A random set of vectors before (top) and after normalization (below)*

If two different sets of vectors point at the same location after normalization, we can say that they represent or measure the same object.

Using the Euclidean norm for normalization

In formal terms, the Euclidean norm of a vector is defined by the following formula:

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

This formula is also known as the square root of the sum of all squares.

A vector can be normalized by dividing each of its elements by its Euclidian norm, as shown here:

$$\hat{x} = \frac{x}{\|x\|_2}$$

Note: Do not panic at the sight of mathematical formulas. These formulas are shown only for reference and help make sense of the Kotlin code used in the application.

One possible Kotlin implementation of the Euclidean normalization, also known as the L2 normalization, is shown in the following code excerpt:

```
1.  fun euclideanNormalisation(vector: FloatArray): FloatArray {
2.      var norm = 0.0f
3.      for (idx in vector.indices) {
4.          norm += vector[idx].pow(2)
5.      }
6.      norm = sqrt(norm)
7.      return FloatArray(vector.size) { vector[it] / norm }
8. }
```

In Python, one can normalize a vector using the **`norm()`** function provided by the NumPy library. This function can be used as follows:

```
1.  import numpy as np
2. #...
3. vector = vector / np.linalg.norm(vector)
```

Now that we understand normalization in general and Euclidian normalization in particular, let's move on to face recognition models.

Looking at the FaceNet model

In 2015, Florian Schroff et al. published their paper *FaceNet: A Unified Embedding for Face Recognition and Clustering*.

The FaceNet model became a reference in face recognition, verification, and clustering due to its enhanced accuracy as compared to the top-performing models at the time.

It is also particularly interesting because its output is a 128-point measurement in the Euclidean space of the face image. This output vector is relatively compact, considering that the predominant models at the time worked with huge representations with thousands of dimensions.

The model's input is an image of a face whose exact specifications depend on the model implementation. The original paper's NN1 architecture, for example, used a 220x220 3-channel image as input for its first convolutional layer.

Apart from its efficiency, the fact that the face representation is placed in the Euclidean space means that standard techniques can be used to compare different representations. Calculating the Euclidean distance between representations is the simplest, and it is described in the *Using the Euclidian distance to identify a face* section later in the chapter.

The FaceNet model uses a deep convolutional network, followed by an L2 normalization layer that produces the final embeddings vector in the Euclidean space. Training of the FaceNet model, however, involved an innovation that the authors called triplet loss. *Figure 10.3* provides an overview of FaceNet's model structure:

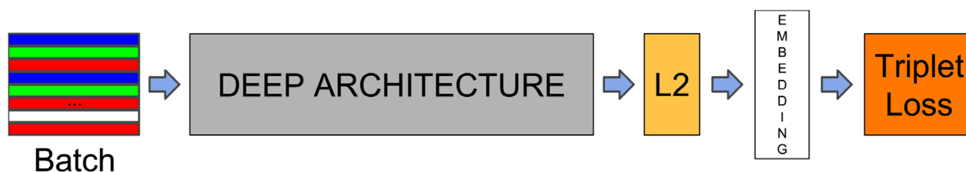

Figure 10.3*: The FaceNet model structure*

Triplet loss is a method that aims to minimize the distance between all faces of the same identity and maximize the distance between two faces of different identities within a margin. It involves an anchor face and positive and negative samples. The distance between the anchor and the positive sample should decrease with training, whereas the distance between the anchor and the negative sample should increase. Ideally, faces of the same identity produce embeddings vectors that are clustered together in Euclidean space.

Figure 10.4 illustrates the goal of the Triplet Loss calculation:

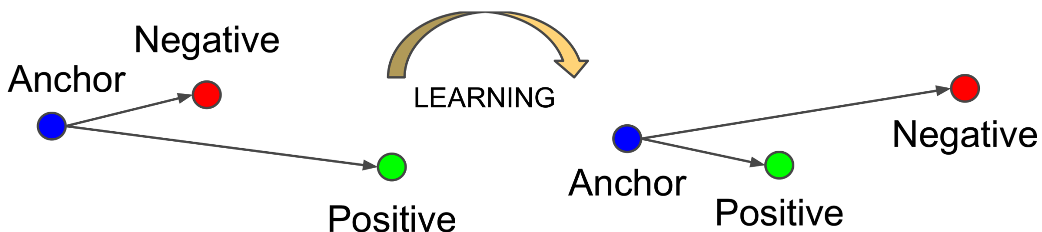

Figure 10.4*: The Triplet Loss method used in FaceNet*

Tip: Readers interested in the research behind the FaceNet model are encouraged to look up the paper mentioned earlier. It is available for free in scientific archives, for example, at https://arxiv.org/abs/1503.03832. Their authors kindly provided *figures 10.3* and *10.4*.

So, this model is a good choice for experimenting with face verification or identification in an application.

However, the team researching FaceNet used training sets with 100-200 million face images from about 8 million identities and trained their models in a cluster for 1,000-2,000 hours. This kind of training is outside the capacity of most domestic systems, so it's best to look for an existing implementation with a pre-trained model whenever possible.

Note: When using existing implementations, datasets, and pre-trained models for any purpose other than personal use, training, or academic research, always confirm that the terms of their license allow their use for the application intended.

This book focuses on TensorFlow and TensorFlow Lite, so we searched for FaceNet implementations in TensorFlow that could be easily converted to TensorFlow Lite.

One implementation of FaceNet in TensorFlow was created by David Sandberg (**https://github.com/davidsandberg/facenet**). Pre-trained models are available, but unfortunately, they are not easy to convert to TensorFlow Lite for the following reasons:

- The models do not use the Keras format
- The models support further training, which means they need to be converted into an inference-only model and frozen
- The models were created with TensorFlow version 1, which means that the code to perform the conversion needs to be written using an older API

There is another implementation of FaceNet available, created by Hiroki Taniai (**https://github.com/nyoki-mtl/keras-facenet**). This implementation was also created with TensorFlow version 1, but the pre-trained model is provided in the Keras format, which eases its conversion to the TensorFlow Lite format.

Converting Hiroki Taniai's implementation of FaceNet to TensorFlow Lite

We must begin by downloading the pre-trained model from the project's GitHub repository location.

The conversion of its Keras model to TensorFlow Lite is straightforward. It only requires the following lines of Python code:

```
1.  import tensorflow as tf
2.
3. converter  =  tf.compat.v1.lite.TFLiteConverter.from_keras_model_
```

```
   file(
4.     'facenet_keras.h5')
5. tflite_model = converter.convert()
6.
7.  with open('facenet.tflite', 'wb') as file:
8.     file.write(tflite_model)
```

The code in the preceding excerpt produces one TensorFlow Lite file that is approximately 91MB big. This file is too big for a mobile application.

Fortunately, there is a method to reduce the model's file size. This method is known as post-training quantization and consists of converting the model's parameters or activations from the original 32-bit floating-point format into formats with lower precision, like 16-bit floating-point or integers. This conversion happens in such a way that there are no significant losses in model accuracy.

The simplest form of post-training quantization for TensorFlow Lite models is called **dynamic range quantization**. This method converts the model's weights to 8-bit integers, but computations are done using floating-point values. Dynamic range quantization results in a model about four times smaller than the original, and it is compatible with CPUs.

Other forms of quantization bring smaller gains in model size or require reference data, that is, a subset of the training or validation data, to calculate quantized values. We chose dynamic range quantization due to its simplicity.

To save the model with dynamic range quantization, one needs to enable the corresponding option in the converter. So, the conversion code now becomes similar to the following code snippet:

```
1.  import tensorflow as tf
2.
3. converter  =  tf.compat.v1.lite.TFLiteConverter.from_keras_model_
   file('facenet_keras.h5')
4.
5. converter.optimizations = [tf.lite.Optimize.DEFAULT]
6. tflite_model = converter.convert()
7.
8.  with open('facenet.tflite', 'wb') as file:
9.     file.write(tflite_model)
```

Note how the converter receives optimization options in line 5 of the preceding code snippet.

Now that we have a TensorFlow Lite model file, it is time to verify its correct operation. We use some random data to do this. This operation aims to verify that the output vector's format and its normalization match what is expected.

The following code snippet loads the model, generates some random data, and runs an inference step based on it. Then, it outputs the shape and norm of the embeddings vector the model has produced.

```
1.  import numpy as np
2.  import tensorflow as tf
3.
4.  interpreter = tf.lite.Interpreter(
5.     model_path="facenet.tflite")
6.  interpreter.allocate_tensors()
7.
8.  input_details = interpreter.get_input_details()
9.  output_details = interpreter.get_output_details()
10.
11. # Generate a random input "image" and preprocess it
12. # before setting it as the model's input.
13. input_shape = input_details[0]['shape']
14. img = np.array(np.random.random_sample(input_shape),
15.     dtype=np.float32)
16. img = (np.float32(img) - 127.5) / 128
17. interpreter.set_tensor(input_details[0]['index'], img)
18.
19. # Obtain the embeddings from said random image.
20. interpreter.invoke()
21. embeddings = interpreter.get_tensor(output_details[0]['index'])
22.
23. print("Embeddings shape:", embeddings.shape,
24.     "norm", np.linalg.norm(embeddings))
```

Running this script on the newly created TensorFlow Lite mode should produce an output similar to the following:

```
1. Embeddings shape: (1, 128) norm 3.8391087
```

The shape of the embeddings vector is the expected one—a 128-dimensional vector. However, its Euclidean norm is not 1, which means it is not normalized.

An additional L2 normalization step, as described in the *Using the Euclidean norm for normalization* section, is required if one wants to use it in an application. The following code excerpt shows how this additional step can be implemented in Kotlin to be used directly with a TensorFlow Lite post-processor:

```
1.  class L2NormalizationOp: TensorOperator {
2.
3.    override fun apply(input: TensorBuffer): TensorBuffer {
4.       val shape: IntArray = input.shape
5.       SupportPreconditions.checkArgument(
6.          shape.size != 0 && shape[0] == 1,
7.          "Only single batches are supported."
8.       )
9.       val values: FloatArray = input.floatArray
10.
11.      var factor = 0.0f
12.      for (idx in values.indices) {
13.         factor += values[idx] * values[idx]
14.      }
15.      factor = sqrt(factor)
16.
17.     val normalized = FloatArray(values.size) { values[it] / factor
    }
18.
19.      return if (input.isDynamic)
20.            TensorBufferFloat.createDynamic(DataType.FLOAT32)
21.         else
22.            TensorBuffer.createFixedSize(shape, DataType.FLOAT32)
```

```
23.         .apply { loadArray(normalized) }
24.    }
25.
26. }
```

The quantized model file is still approximately 23MB. Although acceptable nowadays, it is still somewhat large for a mobile application.

Working with the MobileFaceNets model

Having realized the need for face verification models that could run efficiently on mobile devices, Sheng Chen et al. dedicated themselves to this matter and eventually published their paper *MobileFaceNets: Efficient CNNs for Accurate Real-Time Face Verification on Mobile Devices*.

In this paper, they describe an efficient CNN model optimized for fast and accurate face verification in mobile devices. It uses approximately one million parameters, whereas FaceNet, for example, uses approximately 7.5 million parameters in its NN2 Inception model. Its name notwithstanding, MobileFaceNets has a different model and was trained differently than FaceNet.

Not only did they design this new CNN model to be efficient on mobile devices, but they also used a global depth-wise convolution layer after the last layer of the face feature embedding CNN instead of the commonly used global average pooling layer. They demonstrated that this change brought significant advantages.

Tip: The MobileFaceNet paper detailing the theory and the research results is also freely available in scientific archives, for example, at https://arxiv.org/abs/1804.07573.

An implementation of MobileFaceNet for TensorFlow, with a pre-trained model, has been created by the **`sirius-ai`** GitHub user and is available at **https://github.com/sirius-ai/MobileFaceNet_TF**.

The pre-trained model is also available through this repository, but it is not easy to convert it to TensorFlow Lite.

Unfortunately, no suitable alternative open-source implementations could be found. The documentation in the project's GitHub issue tracker is most helpful and is available at **https://github.com/sirius-ai/MobileFaceNet_TF/issues/46**. The code is extensive and is not reproduced here because its licensing is not clear, but the accompanying repository for this book contains the converted model as a derived work of the original model.

The converted model's file size is approximately 5MB, which is a remarkable improvement from our converted FaceNet model.

The TensorFlow Lite version of this MobileFaceNet implementation was evaluated just like the converted FaceNet model. The evaluation results are similar to the following:

```
1. Embeddings shape: (1, 192) norm 1.0000001
```

We see that the output vector is slightly different from FaceNet's but still relatively compact and normalized in the Euclidean space. So, no additional normalization operations are required.

Just like with FaceNet, face embeddings can be compared using their Euclidean distances.

Using the Euclidean distance to identify a face

Considering that the FaceNet and MobileFaceNets models output embeddings vectors, one can state that if a model produces the same vector when presented with different face images, those images belong to the same person (identity).

However, it is improbable that two identical embeddings vectors are produced in any two faces with the same identity. For example, slight variations in lighting and position cause different outputs.

So, the question that must be asked is not "*Are these vectors equal?*" but rather "*Are these vectors sufficiently close?*". The embeddings vectors are normalized, so there must be a way to compare them.

In the section *Using the Euclidean norm for normalization*, we described how to calculate a vector's length or, in other words, a vector's distance from the origin. What about a vector's distance from another vector?

If we consider that an n-dimensional vector represents one point in an n-dimensional Euclidean space, we can calculate the distance between them. The distance is a real number that can be used to determine if two embeddings vectors are close enough to be considered the same identity.

The following formula is used to calculate the Euclidean distance between two points $p = (p_1, p_2, \ldots, p_n)$ and $q = (q_1, q_2, \ldots, q_n)$ in R^n:

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}$$

The L2-normalized Euclidean distance gives the same ordering of the cosine distance (the cosine similarity's complement) while avoiding expensive trigonometric calculations.

Finally, one needs to determine the threshold after which the distance becomes too large for the embeddings to belong to the same identity.

Such a threshold is defined depending on the metrics at use; for example, it may be chosen to minimize the rate of false positives and false negatives. Another metric can be the recognition rate, which is the rate of the number of correctly identified test images to the total number of test images. There are several other metrics, like the verification rate or the equal error rate, but we do not go over metrics in this book.

The FaceNet researchers have determined that a threshold of 1.24 was the optimal value for their dataset, so this value can be used as a starting point for comparing FaceNet embeddings vectors.

The MobileFaceNets model produces different vectors, but we still use 1.20 as a working threshold in this chapter's application. This threshold was chosen based on empirical tests with the application, so it is only appropriate for demonstration purposes.

Incorporating the MobileFaceNets model in an Android application

The application developed in *Chapter 9*, *Detecting Faces with the Google ML Kit*, can be used as a starting point. This new version of the application aims to incorporate a face detection model and evaluate its performance.

Adjusting the user interface

Its user interface needs to be modified slightly, and it seems appropriate to display the distance to some hardcoded embeddings vector.

The necessary code changes are related only to adjusting the position of the elements on the screen and incorporating a new text view used to display the calculated distance. This new view is defined as shown in the following code excerpt:

```
1. <TextView
2.     android:id="@+id/txtDistance"
3.     android:layout_width="wrap_content"
4.     android:layout_height="wrap_content"
5.     android:text="@string/distance_from_target"
```

```
6.     app:layout_constraintBottom_toTopOf="@+id/horizontalGuideline"
7.      app:layout_constraintEnd_toEndOf="parent"
8.      app:layout_constraintStart_toStartOf="@+id/verticalGuideline"
9.      app:layout_constraintTop_toTopOf="parent" />
```

The adjustments to the positions of the other elements are present in the accompanying code and are not shown here for brevity.

Figure 10.5 illustrates the new user interface design:

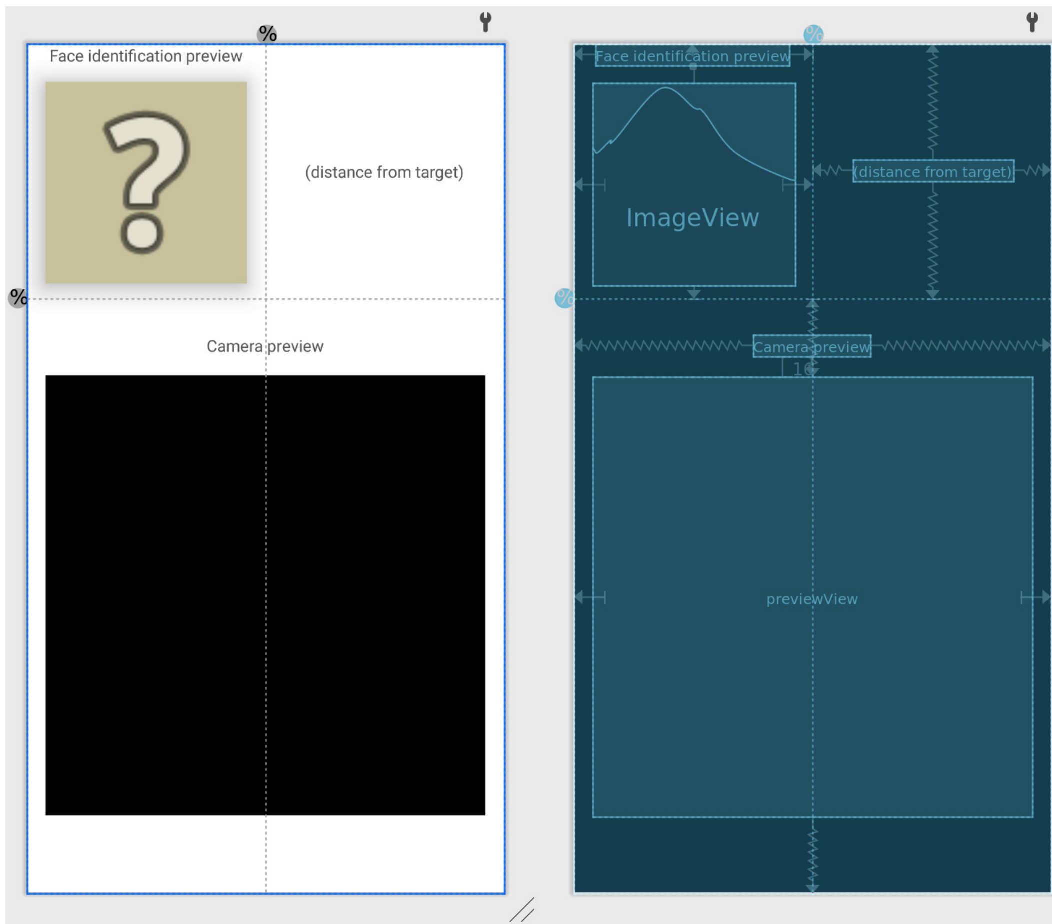

***Figure 10.5**: Application layout for displaying the distances calculated*

The image processing logic also needs to be adjusted to incorporate the face detection model.

Extracting embeddings from face images

Once the face image is obtained, it can be processed by the MobileFaceNet model.

The **`mobilefacenet.tflite`** file is placed inside the application module's main assets folder, as described in *Chapter 8, Using the Image Recognition Model in an Android Application*.

This TensorFlow Lite model does not contain metadata, so we chose to use the **`Interpreter`** class directly. A new class is created to manage the TensorFlow Lite model. The following code excerpt shows its initialization:

```
1.  class FaceNetEmbeddingsExtractor(context: Context): AutoCloseable
    {
2.
3.      private val interpreter: Interpreter
4.      private val modelFile: ByteBuffer
5.
6.      private val inputTensor: Tensor
7.      private val imageProcessor: ImageProcessor
8.      private val outputBuffer: TensorBuffer
9.
10.     init {
11.         modelFile = FileUtil.loadMappedFile(
12.             context, "mobilefacenet.tflite")
13.         interpreter = Interpreter(modelFile)
14.
15.         inputTensor = interpreter.getInputTensor(0)
16.         val inputShape = inputTensor.shape()
17.         val imageWidth = inputShape[1]
18.         val imageHeight = inputShape[2]
19.         imageProcessor = ImageProcessor.Builder()
20.             .add(
21.                 ResizeOp(imageWidth, imageHeight,
22.                     ResizeOp.ResizeMethod.BILINEAR)
23.             )
```

```
24.             .add(
25.                 NormalizeOp(127.5f , 128f)
26.             )
27.             .build()
28.
29.         outputBuffer =
30.             interpreter.getOutputTensor(0).let {
31.                 TensorBuffer.createFixedSize(
32.                     it.shape(), it.dataType())
33.             }
34.     }
35.
36.     override fun close() {
37.         interpreter.close()
38.     }
39.
40.     // (embeddings vector extraction)
41.
42.}
```

The initialization procedure is similar to the one used in *Chapter 8, Using the Image Recognition Model in an Android Application*.

Running inference to extract the embeddings vector is also simple. The following code excerpt shows its implementation in the **`FaceNetEmbeddingsExtractor`** class:

```
1.  class FaceNetEmbeddingsExtractor(context: Context): AutoCloseable
   {
2.
3.  // (initialisation)
4.
5.  fun embeddings(bitmap: Bitmap): FloatArray {
6.   val ticks = System.currentTimeMillis()
7.   interpreter.run(
8.       loadImage(bitmap),
```

```
9.        outputBuffer.buffer.rewind()
10.    )
11.   Log.i(TAG, "Extraction time: ${System.currentTimeMillis()-ticks}
    ms")
12.   return outputBuffer.floatArray
13.  }
14.
15. private fun loadImage(bitmap: Bitmap)
16.        : ByteBuffer {
17.   val tensorImage = TensorImage(inputTensor.dataType())
18.   tensorImage.load(bitmap)
19.   return imageProcessor
20.        .process(tensorImage)
21.        .buffer
22.  }
23.
24. companion object {
25.   const val TAG = "FaceNet"
26.  }
27.}
```

Nothing in this usage of a TensorFlow Lite model is new. A log entry has been added in line 11 of the preceding excerpt, so we can have an effortless indication of inference speed on the specific device running the application. Inference on a single face on the author's device takes approximately 120ms.

Writing a new image analyzer for face verification

The new image analyzer for face verification builds upon the same pattern used before:

- Uses the Google ML Kit to detect faces in the incoming images
- Allows the release of resources used by the detector and the model in a thread-safe manner
- Sends the results of the model to a processing function

The following code excerpt shows the initialization and cleanup parts of the face verification analyzer:

```
1.  class RealtimeFaceEmbeddingsExtractor(
2.    context: Context,
3.    private val receiver: (FloatArray, Bitmap) -> Unit)
4.     : ImageAnalysis.Analyzer, AutoCloseable {
5.
6.     private val extractor: FaceNetEmbeddingsExtractor =
7.         FaceNetEmbeddingsExtractor(context)
8.     private val detector: FaceDetector =
9.         FaceDetectorBuilder.build()
10.    private val conversion: ImageProxyYuvConversion =
11.        ImageProxyYuvConversion(context)
12.
13.    private val closureLock: Lock = ReentrantLock()
14.    private var closed: Boolean = false
15.
16.    override fun analyze(proxy: ImageProxy) {
17.        try {
18.            closureLock.lock()
19.            if (!closed) {
20.                process(proxy)
21.            }
22.        } finally {
23.            closureLock.unlock()
24.        }
25.    }
26.
27.    override fun close() {
28.        try {
29.            closureLock.lock()
30.            closed = true
```

```
31.             extractor.close()
32.             detector.close()
33.         } finally {
34.             closureLock.unlock()
35.         }
36.     }
37.
38.     // (image processing)
39.
40. }
```

The new **RealtimeFaceEmbeddingsExtractor** sends the results to a processing function, which is declared in line 3 of the preceding excerpt.

Lines 6 to 11 declare the necessary supporting objects, including **FaceNetEmbeddings Extractor**. Utility classes create the other objects to keep the analyzer implementation simple. These utility classes only aggregate the same behavior seen before and are supplied as part of the accompanying code.

The image processing part is shown in the following code excerpt:

```
1.  class RealtimeFaceEmbeddingsExtractor(
2.      context: Context,
3.      private val receiver: (FloatArray, Bitmap) -> Unit)
4.    : ImageAnalysis.Analyzer, AutoCloseable {
5.
6.    // (initialisation)
7.
8.    private fun process(proxy: ImageProxy) {
9.      val originalImage = proxy.image
10.     if (originalImage == null) {
11.       proxy.close()
12.       return
13.     }
14.     detector.process(InputImage.fromMediaImage(
15.         originalImage, proxy.imageInfo.rotationDegrees))
```

```
16.       .addOnSuccessListener { faces ->
17.         if (faces.isEmpty()) {
18.           receiver(EMPTY_EMBEDDINGS, EMPTY_BITMAP)
19.         } else {
20.           faces.first().let { face ->
21.           val faceBounds = cancelRotation(face.boundingBox, proxy)
22.           if (faceBounds.setIntersect(proxy.cropRect, faceBounds))
    {
23.             val faceBitmap = conversion.toBitmap(proxy, faceBounds)
24.               val embeddings = extractor.embeddings(faceBitmap)
25.               receiver(embeddings, faceBitmap)
26.             } else {
27.               receiver(EMPTY_EMBEDDINGS, EMPTY_BITMAP)
28.             }
29.           }
30.         }
31.       }
32.       .addOnFailureListener {
33.         receiver(EMPTY_EMBEDDINGS, EMPTY_BITMAP)
34.       }
35.       .addOnCompleteListener {
36.         proxy.close()
37.       }
38.   }
39.
40.  private fun cancelRotation(bounds: Rect, proxy: ImageProxy): Rect
    {
41.     // (existing implementation)
42.   }
43.
44.   companion object {
45.     val EMPTY_EMBEDDINGS: FloatArray = floatArrayOf()
```

```
46.     val EMPTY_BITMAP: Bitmap = createBitmap(1, 1, ARGB_8888)
47.   }
48. }
```

The same pattern is used in the preceding processing:

- Use success and failure callbacks to process the face detection result
- Ensure that the face image is in the correct orientation before sending it to the model
- Ensure that the face bounding rectangle is within the original image
- Run inference and send the results to the processing function
- Send empty results to the processing function if there is an error or no faces are detected

Note that the **`cancelRotation()`** function's implementation is not shown here for brevity; it is the same as in *Chapter 9, Detecting Faces with the Google ML Kit*.

Displaying the distance between two face embeddings vectors

The implementation of the **`MainActivity`** class remains almost identical to the one used in *Chapter 9, Detecting Faces with the Google ML Kit*, except that:

- The function receiving the result now has a different signature
- It is now necessary to show the distance to the reference face embeddings

So, the new function that receives the results is implemented as shown in the following code excerpt:

```
1.  private fun onFaceDetected(embeddings: FloatArray, faceBitmap:
    Bitmap) {
2.   if (embeddings.isEmpty()) {
3.      binding.faceView.setImageResource(android.R.drawable.ic_menu_
         help)
4.     binding.txtDistance.setText(R.string.distance_from_target)
5.   } else {
6.     val distance = euclidianDistance(
7.       testing,
8.       embeddings
```

```
9.      )
10.
11.     Log.i("Verification", "Distance: $distance")
12.
13.     binding.txtDistance.text = distance.toString()
14.     binding.faceView.setImageBitmap(faceBitmap)
15.   }
16. }
```

The function mainly aims to ensure that the different user interface elements show accurate information. The actual Euclidean distance calculation, depicted in lines 6 to 8 of the preceding excerpt, is implemented in a utility function.

This utility function implements the formula for the Euclidean distance in Kotlin as per the following code excerpt:

```
1.  fun euclidianDistance(a: FloatArray, b: FloatArray): Float {
2.      var sum = 0.0f
3.      for (i in a.indices) {
4.          sum += (a[i] - b[i]).pow(2)
5.      }
6.      return sqrt(sum)
7. }
```

The **onFaceDetected()** function uses the **euclidianDistance()** function to calculate the distance between the newly generated face embeddings and a face embeddings vector previously calculated and hardcoded in the application.

Obtaining face embeddings for testing

We must verify the correct operation of the application before moving forward, but we need a way to generate a face embeddings vector to use as a comparison.

The easiest way is to write an instrumented Android test that uses most of the utility classes and functions described previously to clip a face from some photograph and run it through inference.

The following code snippet shows a possible implementation of such a test:

```
1.  @Test
2.  fun embeddings() {
3.   val instrumentation = InstrumentationRegistry.getInstrumentation()
```

```
4.      val appContext = instrumentation.targetContext
5.
6.      val bitmap = extractFace(instrumentation, "identity_1_a.jpg")
7.
8.      val embeddings = FaceNetEmbeddingsExtractor(appContext)
9.                          .embeddings(bitmap)
10.
11.     assertThat(embeddings, `is`(notNullValue()))
12.     assertThat(embeddings.size, `is`(greaterThan(0)))
13.
14.     Log.i(TAG, embeddings.joinToString(",") { "${it}f" })
15. }
```

This test validates that the model produces a non-empty vector, and it also logs the embeddings vector's contents. This log entry is produced in a format that makes it easy to copy the **MainActivity**'s code for reference purposes.

Figure 10.6 shows this application running on the author's device. The application showed a distance of 0.81 to a hardcoded embeddings vector obtained through the instrumented test method.

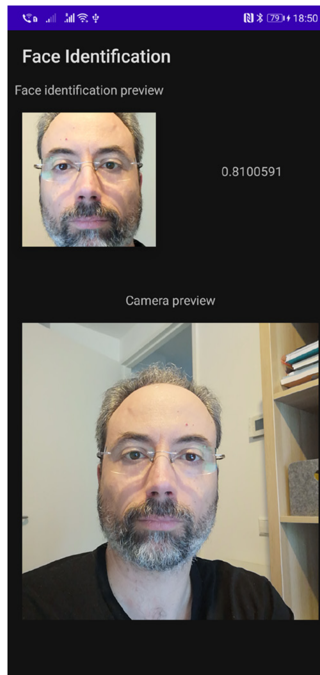

***Figure 10.6**: Identifying a face based on a previously generated embeddings vector*

This distance is below 1.20, so we can state with a high degree of confidence that the face image captured belongs to the same identity as the face image used in the instrumented test.

Conclusion

Face recognition and verification are different from face detection because they allow the association of an identity to different images representing the same face. In contrast, face detection merely knows where a face is present in an image.

When ML models output vectors describing their targets, we must ensure that they are normalized to be compared. The L2 or Euclidean normalization is frequently used. When vectors are L2-normalized, they can be compared by calculating their Euclidean distance, as long as the model was trained to cluster similar vectors together.

Both FaceNet and MobileFaceNets models can be used in an Android application for face recognition. The latter has the advantage of being smaller than the former, thus becoming more appropriate for mobile applications.

It is easy to alter a simple face detection application to incorporate face verification once all techniques described in this book are applied.

The next chapter further extends the application to include face registration, bringing it closer to a complete application by making use of face biometrics.

CHAPTER 11
Registering Faces in the Application

Introduction

This chapter shows how we can build an Android application for face recognition capable of extending its set of recognizable faces at runtime. It integrates the concept of adding a face persistently to the application so that it is recognizable between application runs.

Structure

We cover the following topics in this chapter:

- Building the identity store
- Implementing a view model to interface with the identity store
- Processing multiple faces from camera images
- Designing the user interface
- Adding new faces to the application
- Recognizing faces

Objectives

By the end of this chapter, you have built a fully functional Android application that runs face recognition on one or more new faces. The technique used to persist face embeddings locally is also understood.

Building the identity store

This chapter uses the term identity to refer to a tuple containing a name and a face embeddings vector. This interpretation is rather strict because identity is an abstract concept with many ramifications. Still, in this context, it solely represents a name that can be associated with any face whose embeddings are sufficiently close to the originals.

Whenever an application needs to store tuple-based data in a persistent manner that survives application runs, the first component that should be developed is an application database. We reuse the database management concepts described in *Chapter 3*, *Building Our Application with Kotlin and SQLite*, to build such a database using the Room library.

The tuple previously described contains a name and a face embeddings vector. Only one set of embeddings is sufficient, as long as the processed image has sufficient quality. However, we want to allow the user to add different embeddings with the same name for testing purposes.

So, the database entity for this demonstration application was defined as shown in the following code excerpt:

```
1.  @Entity
2.  data class Identity(
3.      @PrimaryKey(autoGenerate = true) veal id: Int,
4.      val name: String,
5.      val embeddings: FloatArray
6. ) {
7.      override fun equals(other: Any?): Boolean {
8.          // implementation
9.      }
10.
11.     override fun hashCode(): Int {
12.         // implementation
```

```
13.     }
14. }
```

Note that the database entity data class has had its **`equals()`** and **`hashCode()`** methods redefined, although their full implementation is trivial and not shown for brevity. This was done so that **`equals()`** and **`hashCode()`** could use the **`contentEquals()`** and **`contentHashCode()`** methods, respectively, from the embeddings array because the default implementations do not take array contents into consideration.

The `Room` library, however, does not support storing **`FloatArray`** objects directly. These objects need to be converted into a format that the `Room` library supports. We have chosen to store the array as a JSON-encoded string and built the converter class as shown in the following code excerpt:

```
1.  class FloatArrayConverters {
2.      private val floatStrategy = FloatArraySerializer()
3.
4.      @TypeConverter
5.      fun fromFloatArray(value : FloatArray): String =
6.          Json.encodeToString(floatStrategy, value)
7.
8.      @TypeConverter
9.      fun toFloatArray(value: String): FloatArray =
10.         Json.decodeFromString(floatStrategy, value)
11. }
```

The Kotlin serialization library provides the **`FloatArraySerializer`** and **`Json`** classes.

Now that the entity and the necessary data converter are defined, we can move forward and declare the **Data Access Object** (**DAO**). Our application has simple needs, as illustrated in the following code snippet:

```
1.  @Dao
2.  interface IdentityDao {
3.
4.      @Insert
5.      fun insert(identity: Identity)
6.
7.      @Query("SELECT * FROM Identity")
```

```
8.      fun findAll(): List<Identity>
9.
10. }
```

It only needs to insert new identities and list them to find the correct one during the recognition process.

The database class is also straightforward to declare. Its structure is shown in the following code snippet, except for the companion object that creates the database instance:

```
1.  @Database(entities = [ Identity::class ], version = 1)
2.  @TypeConverters(FloatArrayConverters::class)
3.  abstract class ApplicationDatabase: RoomDatabase() {
4.
5.      abstract fun identityDao(): IdentityDao
6.
7.      // companion object to create the database instance
8.
9.  }
```

This companion object is created exactly like in *Chapter 3, Building Our Application with Kotlin and SQLite*, so its code is omitted for brevity.

Projects created by Android Studio do not include the `Room` library and the Kotlin serialization library, so we need to change the application module's **`build.gradle`** file to include them, as follows:

```
1.  plugins {
2.      // (other plugins)
3.     id 'org.jetbrains.kotlin.plugin.serialization' version '1.5.20'
4.  }
5.  // (...)
6.  dependencies {
7.      // (other dependencies)
8.
9.      // Kotlin serialization
10.      implementation 'org.jetbrains.kotlinx:kotlinx-serialization-
            json:1.2.1'
```

```
11.
12.     // Room library
13.     implementation 'androidx.room:room-runtime:2.3.0'
14.     implementation 'androidx.legacy:legacy-support-v4:1.0.0'
15.     kapt 'androidx.room:room-compiler:2.3.0'
16.     implementation 'androidx.room:room-ktx:2.3.0'
17.     androidTestImplementation 'androidx.room:room-testing:2.3.0'
18. }
```

The preceding code excerpt shows the dependencies and plugins necessary to use the Kotlin serialization and `Room` libraries fully.

The demonstration application now has an implementation of a database it can use to store the identities. However, this database must be accessible from the application's views. The currently recommended way to do this is using an Android View Model.

Implementing a view model to interface with the identity store

The advantage of using the Android View Model pattern instead of handling the database directly within the activity or the fragment is that the view model has a separate lifecycle. For example, the view model is not destroyed if an attached activity is recreated because of a configuration change.

A view model also forces a separation of concerns, keeping data separate from the user interface logic. Furthermore, different fragments connected to the same activity can reuse the same view model transparently. This subject was also demonstrated in *Chapter 3, Building Our Application with Kotlin and SQLite*.

Due to the separation of concerns, our view model becomes responsible for the following tasks:

- Manage the database lifecycle
- Add identities to the database
- Obtain an identity based on a face embeddings vector

A few details need attention while implementing these tasks, so the construction of the view model is described in stages. The initial stage is the declaration of a class, as shown:

```
1.  class IdentityViewModel(application: Application)
2.     : AndroidViewModel(application)  {
```

```
3.
4. }
```

Thanks to the view model's characteristics, managing the database lifecycle is simple. The following code excerpt shows how the database instance is created and closed once the view model is destroyed:

```
1.  private val database = ApplicationDatabase.getDatabase(
2.      getApplication<Application>().applicationContext)
3.
4.  override fun onCleared() {
5.      super.onCleared()
6.      database.close()
7. }
```

Adding an identity record to the database is a little bit more involved because the database operation cannot be executed in the main application thread. So, we must ensure that it always runs in some other thread.

We use the view model's support for Kotlin coroutines to accomplish this separation. The view model ensures that any active coroutines are canceled when it is destroyed. The Kotlin coroutines allow us to define their execution context without further concerns.

The following code excerpt shows how this integration can be implemented:

```
1.  private val coroutineContext =
2.      viewModelScope.coroutineContext + Dispatchers.IO
3.
4.  suspend fun addIdentity(name: String, embeddings: FloatArray) {
5.      checkArgument(name.isNotBlank())
6.      checkArgument(embeddings.isNotEmpty())
7.      withContext(coroutineContext) {
8.          database.identityDao()
9.              .insert(Identity(0, name, embeddings))
10.     }
11. }
```

Lines 1 and 2 in the preceding excerpt declare a coroutines context that combines the context managed by the view model with an I/O dispatcher. The former ensures cancellation on destruction, and the latter ensures that the coroutine does not run in the main thread. The **`addIdentity()`** function is then declared as a coroutine whose database operation runs within this context.

The final responsibility of this view model is to match a face embeddings vector matching an existing identity. Considering that this application is unlikely to store many identities, a simple full table lookup algorithm calculates the Euclidean distances and selects the smaller distance below an identification threshold.

The following code excerpt shows such an implementation:

```
1.  suspend fun recogniseOrNull(embeddings: FloatArray)
2. : IdentityDistance? =
3.     withContext(coroutineContext) {
4.         calculateDistancesFrom(embeddings)
5.             .filter { it.distance < IDENTIFICATION_THRESHOLD }
6.             .minByOrNull { it.distance }
7.     }
8.
9.  private fun calculateDistancesFrom(embeddings: FloatArray)
10.: List<IdentityDistance> =
11.    database.identityDao().findAll()
12.        .map {
13.            IdentityDistance(
14.                it.id, it.name,
15.                euclidianDistance(it.embeddings, embeddings)
16.            )
17.        }
18.
19. companion object {
20.    private const val IDENTIFICATION_THRESHOLD = 1.20f
21.}
```

The **calculateDistancesFrom()** function returns a list of the Euclidean distances between the face embeddings vector provided as an argument and each of the face embeddings vectors stored in the database.

Then, the **recogniseOrNull()** function selects the ones that may be a positive match and picks the one with the lowest Euclidean distance value.

A threshold of 1.20 was selected based on the discussion from the previous chapter. This value can be adjusted based on the desired sensitivity. A lower value produces fewer false positives, whereas a higher value produces more false negatives.

Processing multiple faces from camera images

All examples presented so far only process one face from the Google ML Kit face detector's results. However, this face detector is capable of detecting the presence of multiple faces in the same image. So, why not attempt to recognize them all?

The **RealtimeFaceEmbeddingsExtractor** class from previous examples was refactored to process several faces and transfer all face embeddings vectors to its caller.

A new data structure was defined to carry information about each face to the user code, as shown in the following code excerpt:

```
1.  data class FaceEmbeddings(
2.      val face: Bitmap,
3.      val embeddings: FloatArray
4.  )
```

The class's constructor has changed accordingly. The user code now needs to provide a callback function that can receive a list of **FaceEmbeddings** objects. This change is shown in the following code snippet:

```
1.  class RealtimeFaceEmbeddingsExtractor(
2.          context: Context,
3.          private val receiver: (List<FaceEmbeddings>) -> Unit)
4.      : ImageAnalysis.Analyzer, AutoCloseable {
5.
6.  }
```

The already familiar housekeeping and composition patterns can be found in the **RealtimeFaceEmbeddingsExtractor** class.

The YUV image conversion functions, the Google ML Kit face detector class, and the **MobileFaceNet** embeddings extractor class are included by composition, as shown in the following code excerpt. Their implementation is identical to those presented in the previous chapters and is not shown here for brevity.

```
1.  private val extractor: FaceNetEmbeddingsExtractor =
2.      FaceNetEmbeddingsExtractor(context)
3.  private val detector: FaceDetector =
4.      FaceDetectorBuilder.build()
5.  private val conversion: ImageProxyYuvConversion =
6.      ImageProxyYuvConversion(context)
```

A synchronized closure function is used, along with a property holding the corresponding state, to ensure that any allocated resources are freed when the class is no longer necessary.

The synchronization is used to introduce a memory barrier, ensuring that any new value assigned to the property is visible in any thread and preventing the processing of any new images while resources are being freed.

This housekeeping pattern is shown in the following code snippet. Note how the implementation of the **analyze()** and the **close()** functions is interlinked through a **ReentrantLock**; the reentrant lock does not allow **process()** to be called while **close()** is freeing resources, and vice-versa.

```
1.  private val closureLock: Lock = ReentrantLock()
2. @Volatile private var closed: Boolean = false
3.
4.  override fun analyze(proxy: ImageProxy) {
5.      try {
6.          closureLock.lock()
7.          if (!closed) {
8.              process(proxy)
9.          }
10.     } finally {
11.         closureLock.unlock()
12.     }
```

```
13.}
14.
15. override fun close() {
16.     try {
17.         closureLock.lock()
18.         closed = true
19.         extractor.close()
20.         detector.close()
21.     } finally {
22.         closureLock.unlock()
23.     }
24.}
```

The **`process()`** function shown in the following code excerpt uses the face embeddings extractor to process all faces detected by the face detector. The callback function is called with the results or with an empty list if there are none.

```
1.  private fun process(proxy: ImageProxy) {
2.      val originalImage = proxy.image
3.      if (originalImage == null) {
4.          proxy.close()
5.          return
6.      }
7.      detector.process(InputImage.fromMediaImage(
8.              originalImage, proxy.imageInfo.rotationDegrees))
9.          .addOnSuccessListener {
10.             if (it.isNotEmpty()) {
11.                 facesDetected(proxy, it)
12.             } else {
13.                 receiver(emptyList())
14.             }
15.         }
16.         .addOnFailureListener {
17.             receiver(emptyList())
```

```
18.         }
19.         .addOnCompleteListener {
20.             proxy.close()
21.         }
22. }
```

Such implementation is still similar to the one presented earlier. The main difference is that the processing of the incoming image, including cropping the detected face images, was moved to a separate function. This new function is shown in the following code excerpt:

```
1.  private fun facesDetected(proxy: ImageProxy, faces: List<Face>) {
2.      if (closed) {
3.          return
4.      }
5.      val imageBounds = Rect(0, 0, proxy.width, proxy.height)
6.      val imageBitmap = conversion.toBitmap(proxy, imageBounds)
7.      receiver(
8.          faces.filter {
9.            it.boundingBox.setIntersect(imageBounds, it.boundingBox)
10.         }.map {
11.             val faceBitmap = Bitmap.createBitmap(imageBitmap,
12.                 it.boundingBox.left, it.boundingBox.top,
13.                 it.boundingBox.width(), it.boundingBox.height())
14.             FaceEmbeddings(
15.                 faceBitmap,
16.                 extractor.embeddings(faceBitmap)
17.             )
18.         }
19.     )
20. }
```

The first thing that this function does is to check if the underlying resources are still valid. Such verification is necessary because this function is called on the successful outcome of the face detector, which runs asynchronously.

Instead of converting each face independently, it is more efficient to first convert the entire image from the camera into a **`Bitmap`** and crop each detected face. The implementation becomes simpler this way, and this method also ensures that the camera image's underlying memory buffer is only consumed once.

Line 9 in the preceding code excerpt ensures that the detected face bounding box is valid. Lines 11 to 13 create a separate bitmap with the face's image, and lines 14 to 17 create the data transfer object containing the bitmap and the embeddings vector. This procedure is repeated for each face detected, and the resulting list is passed to the callback.

Using the face embeddings vectors is now the responsibility of the user code implementing the callback. The callback is implemented in a fragment that delegates recognition to its view model. Such implementation is described in the *Recognizing faces* section later in this chapter.

Designing the user interface

This demonstration application has two responsibilities: recognizing faces and adding faces to be recognized later. Therefore, it is logical to divide the user interface into at least two distinct sections, one for each task.

Such division is accomplished by implementing one fragment for face recognition and another to register new faces in the application.

Navigation between these fragments is accomplished through the Android Navigation component and its matching Safe Args library, as described in *Chapter 7, Android Camera Image Capture with CameraX*.

Adding new faces to the application

Adding a new face to the application to be recognized later consists of storing the new face's embeddings vector along with a user-friendly name, as discussed in the previous section, *Building the identity store*.

So, there is only one business rule—store only complete identities, that is, identities with a name and a face embeddings vector.

The user interface needs to implement the following controls before storing a new identity:

- The name must not be empty
- A face embeddings vector must be present

From the user's point of view, they see a camera preview, one text field to type a name in, and a button to store the information. *Figure 11.1* illustrates the layout chosen:

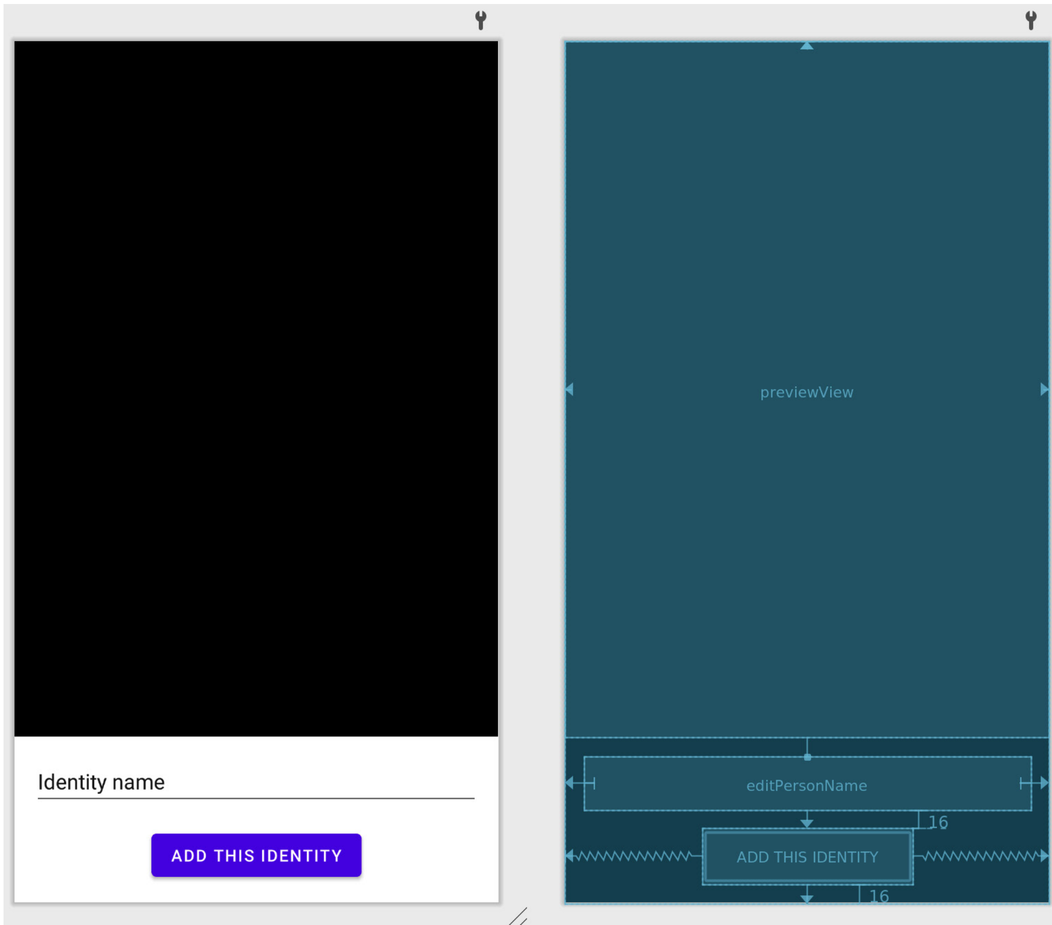

Figure 11.1*: User interface layout for adding a new face to the application*

The **AddNewFaceFragment**'s implementation begins by defining its runtime properties. The following code excerpt shows that it uses the same image analysis classes as the upcoming face recognition user interface:

```
1.  class AddNewFaceFragment : Fragment() {
2.
3.      private lateinit var binding: FragmentAddNewFaceBinding
4.
5.      private lateinit var cameraExecutor: ExecutorService
6.   private lateinit var imageAnalyser: RealtimeFaceEmbeddingsExtractor
7.
8. }
```

The purpose behind such reuse is to simplify the development of the application. However, more complicated user interfaces might need specialized code.

This class does not store the identity automatically once some face embeddings are present, so it needs some logic to store the name and the face embeddings until the user decides to store them. The **IdentityData** class shown in the following code excerpt serves this purpose:

```
1.  private data class IdentityData(
2.          var name: String = "",
3.          var embeddings: FloatArray = EMPTY_EMBEDDINGS) {
4.      val isComplete: Boolean
5.      get() = name.isNotBlank() && embeddings.isNotEmpty()
6. }
```

The **IdentityData** class also includes the **isComplete** property, which lets the fragment know when all data is present and a record can be created.

The following code excerpt shows how the **IdentityViewModel** and **IdentityData** classes are declared because they serve different purposes. The former contains the view model for the entire application, whereas the latter is a temporary model used only while the **AddNewFaceFragment** is active.

```
1.  private val viewModel: IdentityViewModel by activityViewModels()
2.  private val currentIdentityData = IdentityData()
```

The implementation of the functions handling the **AddNewFaceFragment**'s creation and destruction is already known from previous examples and is shown in the following code excerpt. They manage the camera executor's lifecycle and set up the automatically generated code for binding to the fragment's child views.

```
1.  override fun onCreate(savedInstanceState: Bundle?) {
2.      super.onCreate(savedInstanceState)
3.      cameraExecutor = Executors.newSingleThreadExecutor()
4. }
5.
6.  override fun onDestroy() {
7.      super.onDestroy()
8.      cameraExecutor.shutdown()
9. }
10.
```

```
11. override fun onCreateView(
12.     inflater: LayoutInflater, container: ViewGroup?,
13.     savedInstanceState: Bundle?
14. ): View {
15.     binding = FragmentAddNewFaceBinding.inflate(
16.         inflater, container, false)
17.     return binding.root
18. }
```

The heavy lifting required to initialize the camera and the image analyzer is done in the **`onStart()`** function.

This function needs to accomplish the following tasks:

- Build and configure the image analyzer, including the callback function invoked when new face embeddings vectors are ready
- Attach a listener to the **`Add this identity`** button, invoked when the user wants to store the last face embeddings received
- Configure the camera use cases with preview and image analysis

The following code excerpt illustrates the implementation of the **`onStart()`** function. It uses a utility method to configure the camera use cases with an identical implementation to previous usages. It is not shown here for brevity. The full implementation, including all utility methods and classes, is available in the accompanying code bundle.

```
1.  override fun onStart() {
2.      super.onStart()
3.
4.    imageAnalyser = RealtimeFaceEmbeddingsExtractor(requireContext(),
5.          this::onFacesDetected)
6.
7.      binding.btnAddThisIdentity.isEnabled = false
8.      binding.btnAddThisIdentity.setOnClickListener {
9.          onAddIdentityClicked()
10.     }
11.
12.     binding.editPersonName.text.clear()
```

```
13. binding.editPersonName.
        addTextChangedListener(PersonNameWatcher())
14.
15.     binding.previewView.post {
16.         configureCamera(this,
17.             cameraExecutor, binding.previewView,
18.             imageAnalyser, CameraSelector.LENS_FACING_FRONT)
19.     }
20. }
```

Note that a different listener is added to the view where the user writes the identity's name. This listener aims to ensure that a name is present before the user can store the new identity.

Such a listener is implemented in a class extending the Android's **FaceWatcher** interface and is shown in the following code excerpt:

```
1.  private inner class PersonNameWatcher: TextWatcher {
2.      override fun beforeTextChanged(
3.          s: CharSequence?, start: Int, count: Int, after: Int) {
4.      }
5.
6.      override fun onTextChanged(
7.          s: CharSequence?, start: Int, before: Int, count: Int) {
8.      }
9.
10.     override fun afterTextChanged(s: Editable?) {
11.         currentIdentityData.name = s.toString()
12.         binding.btnAddThisIdentity.isEnabled =
13.             currentIdentityData.isComplete
14.     }
15. }
```

Our application is only interested in the view's state after the text has changed, so the only function with a body in the **PersonNameWatcher** class is **afterTextChanged()**.

Its responsibility is to store the text in the fragment's private model class and manage the **Add this identity** button's state based on the availability of a complete data

set. In other words, the button is enabled only when both the name and the face embeddings are present.

Any resources used by the image analyzer need to be released when the fragment is no longer in use, so a simple **`onStop()`** function is also implemented, as shown in the following code excerpt:

```
1.  override fun onStop() {
2.      super.onStop()
3.      imageAnalyser.close()
4.  }
```

The **`onFacesDetected()`** function is called with a list of all face embeddings found in the camera's image every time a face is successfully detected and processed by the image analyzer.

This function is responsible for storing the detected embeddings in the fragment's private model and updating the **`Add this identity`** button's state. This way, the button's state is always correct, regardless of whether the name arrives first or the face embeddings. Its implementation is shown in the following code excerpt:

```
1.  private fun onFacesDetected(faces: List<FaceEmbeddings>) {
2.      lifecycleScope.launch(Dispatchers.Main) {
3.          currentIdentityData.embeddings = if (faces.isEmpty()) {
4.                  EMPTY_EMBEDDINGS
5.              } else {
6.                  faces.first().embeddings
7.              }
8.          binding.btnAddThisIdentity.isEnabled =
9.              currentIdentityData.isComplete
10.     }
11. }
```

Note that line 2 of the preceding code snippet runs the function's code with the help of the **`launch()`** function of the **`lifecycleScope`**'s object. It is taking advantage of the support for Kotlin coroutines to ensure that any changes made to the button's state and the private model run in the main thread.

It is not allowed to make any changes to an Android application's user interface state outside of the main thread. We take advantage of this limitation to update the private model in the main thread, thus avoiding any issues related to multi-threading.

Once the private model is complete, that is, a name and a face embeddings vector are set, the user can click the **Add this identity** button to store them.

The **onAddIdentityClicked()** function set as a listener in the fragment's **onStart()** function is called when they click on the button. Its implementation is shown in the following code excerpt:

```
1.  private fun onAddIdentityClicked() {
2.      lifecycleScope.launch(Dispatchers.Main) {
3.          viewModel.addIdentity(
4.              currentIdentityData.name,
5.              currentIdentityData.embeddings
6.          )
7.
8.          Toast.makeText(requireContext(),
9.              resources.getString(R.string.identity_added,
10.                 currentIdentityData.name),
11.             Toast.LENGTH_SHORT).show()
12.
13.         requireView().findNavController().navigate(
14.             AddNewFaceFragmentDirections
15.               .actionAddNewFaceFragmentToFaceRecognitionFragment()
16.         )
17.     }
18. }
```

Once again, the **launch()** function from the **lifecycleScope** object is used to start a coroutine scope, ensuring that all operations run in the main thread. This time it is not because the **onAddIdentityClicked()** function may be called in another thread, but because the view model's **addIdentity()** function is a coroutine, and coroutines can only be called from within a coroutine scope.

The remaining implementation of the **onAddIdentityClicked()** function is trivial:

- The view model is called to store the new identity in the database
- The user is quickly informed that the identity was stored
- The **FaceRecognitionFragment** is activated

Figure 11.2 shows the **`AddNewFaceFragment`** in action, ready to add the author's very relaxed identity to this Android face recognition demonstration application.

Now that the application contains some recognizable identities, it is up to the **`FaceRecognitionFragment`** to attempt matching new face embeddings to the stored identities.

***Figure 11.2**: Ready to add the author's identity to the application*

Recognizing faces

In this face recognition demonstration application, the **`FaceRecognitionFragment`** is the application's entry point because it displays all the latest face recognition results. In addition to displaying the face recognition results, it must allow the user to add new faces to the application by presenting a button with this purpose.

Figure 11.3 shows the user interface layout chosen for this demonstration:

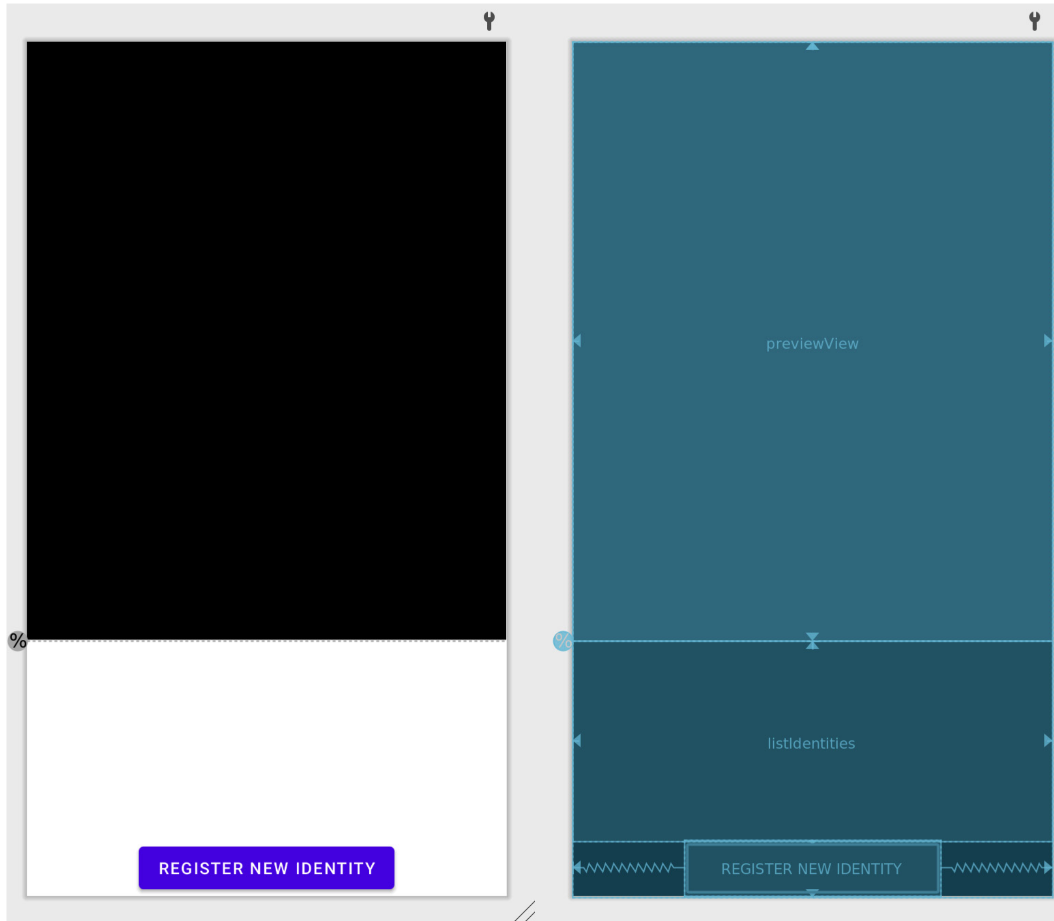

Figure 11.3*: The face recognition fragment's user interface layout*

The camera preview view on top is followed by a list of all faces detected and maybe recognized, finishing with a button that navigates to **`AddNewFaceFragment`**.

Listing the face detection results

A **`ListView`** supports the list of all faces detected with their recognition results. This **`ListView`**'s layout is defined with a custom **`GridLayout`**, so the face thumbnail is drawn on the left side and the recognition results on the right side. *Figure 11.4* shows this layout:

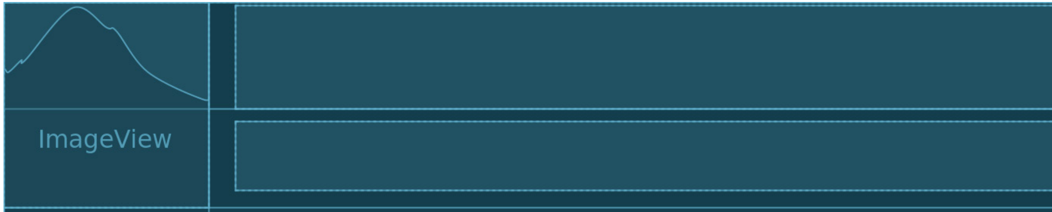

***Figure 11.4**: The face recognition results' list layout*

Since the **ListView** uses a customized layout, it also requires a customized adapter. An adapter for a **ListView** is responsible for inflating each row's layout and populating it with data. As such, it needs a data model to work.

The adapter's data class **Identity** is the glue joining the face bitmap from the face detector with the name and Euclidean distance from the view model. The following code excerpt shows its definition:

```
1.  data class Identity(
2.      val face: Bitmap,
3.      private val identityDistance: IdentityDistance?
4. ) {
5.      val name = identityDistance?.name ?: ""
6.      val distance = identityDistance?.distance?.toString() ?: ""
7. }
```

The adapter's implementation is trivial. It manages a list of **Identity** objects and copies their data to each row's views, as shown in the following code excerpt:

```
1.  class IdentitiesListAdapter(context: Context) :
2.    BaseAdapter() {
3.
4.    var identities: List<Identity> = emptyList()
5.    set(value) {
6.       field = value
7.       notifyDataSetChanged()
8.    }
9.
10.   private val inflater = LayoutInflater.from(context);
11.
```

```
12.    override fun getCount(): Int = identities.size
13.    override fun getItem(position: Int): Any = identities[position]
14.    override fun getItemId(position: Int): Long = position.toLong()
15.
16.    override fun getView(position: Int, convertView: View?,
17.                         parent: ViewGroup): View {
18.        val binding = if (convertView != null) {
19.            IdentityRowItemBinding.bind(convertView)
20.        } else {
21.            IdentityRowItemBinding.inflate(inflater, parent, false)
22.        }
23.        binding.imgIdentityFace.setImageBitmap(identities[position].
           face)
24.        binding.txtIdentityName.text = identities[position].name
25.        binding.txtIdentityDistance.text = identities[position].
               distance
26.        return binding.root
27.    }
28. }
```

Putting everything together

Now that all supporting pieces are in place, it is time to put the puzzle together.

The **FaceRecognitionFragment**'s implementation follows our typical resource management pattern on creation and destruction, as shown in the following code excerpt:

```
1.  class FaceRecognitionFragment : Fragment() {
2.
3.  private lateinit var binding: FragmentFaceRecognitionBinding
4.
5.  private lateinit var cameraExecutor: ExecutorService
6.  private lateinit var imageAnalyser: RealtimeFaceEmbeddingsExtractor
7.  private lateinit var identitiesListAdapter: IdentitiesListAdapter
8.
```

```
9.     private val viewModel: IdentityViewModel by activityViewModels()
10.
11.     override fun onCreate(savedInstanceState: Bundle?) {
12.         super.onCreate(savedInstanceState)
13.         cameraExecutor = Executors.newSingleThreadExecutor()
14.     }
15.
16.     override fun onDestroy() {
17.         super.onDestroy()
18.         cameraExecutor.shutdown()
19.     }
20.
21.     override fun onCreateView(
22.         inflater: LayoutInflater, container: ViewGroup?,
23.         savedInstanceState: Bundle?
24.     ): View {
25.     identitiesListAdapter = IdentitiesListAdapter(requireContext())
26.         binding = FragmentFaceRecognitionBinding.inflate(
27.             inflater, container, false)
28.         binding.listIdentities.adapter = identitiesListAdapter
29.         return binding.root
30.     }
31.
32. }
```

Line 9 of the preceding code excerpt initializes the shared view model, and lines 25 and 28 create the ListView's adapter and attach it to the ListView instance. Camera images are analyzed in a single-threaded executor. It is created and shut down in lines 13 and 18 in the **`onCreate()`** and **`onDestroy()`** functions.

The **`onStart()`** function takes care of initializing the image analyzer and camera use cases. It also prompts the user for the necessary camera permissions. The **`onStop()`** function shuts the image analyzer down. These functions' implementation is shown in the following code snippet:

```
1.  override fun onStart() {
2.      super.onStart()
3.
4.    imageAnalyser = RealtimeFaceEmbeddingsExtractor(requireContext(),
5.          this::onFacesDetected)
6.
7.      binding.previewView.post {
8.          if (cameraPermissionsGranted()) {
9.              setUpCamera()
10.         } else {
11.             requestPermissions(
12.                 CAMERA_PERMISSIONS_REQUESTED,
13.                 PERMISSION_REQUEST_CODE
14.             )
15.         }
16.     }
17.
18. }
19.
20. override fun onStop() {
21.     super.onStop()
22.     imageAnalyser.close()
23. }
```

The button that takes the user to the screen for adding new identities to the application is not bound until after the camera use cases are configured. This delay prevents the user from navigating away if the camera is not available or if the permissions were not granted. The function responsible for this sequence is shown in the following code excerpt:

```
1.  private fun setUpCamera() {
2.      configureCamera(this, cameraExecutor,
3.              binding.previewView, imageAnalyser,
4.          CameraSelector.LENS_FACING_BACK) {
```

```
5.          binding.btnAddIdentity.setOnClickListener(
6.              this::onAddIdentityButtonClicked)
7.      }
8. }
```

The **`Navigation`** component and the Safe Args library are used to switch to the **`AddNewFaceFragment`**, as shown in the following code snippet:

```
1.  private fun onAddIdentityButtonClicked(view: View) {
2.      requireView().findNavController().navigate(
3.          FaceRecognitionFragmentDirections
4.              .actionFaceRecognitionFragmentToAddNewFaceFragment()
5.      )
6. }
```

The code for requesting the permissions necessary to use the camera and the corresponding entries in the application's manifest file are not shown here for brevity. They are identical to the ones covered earlier and are included in the accompanying code.

All application components are implemented separately:

- The refactored camera image analyzer detects more than one face and extracts their face embeddings vectors using the **`MobileFaceNet`** model
- The view model interfaces with the identity store and can search it for the best match for a given face embeddings vector
- A list on the user interface displays each face and its identity data

So, building the face recognition callback logic is only a matter of calling them in the correct sequence. The **`FaceRecognitionFragment`**'s **`onFacesDetected()`** function does this and is shown in the following code excerpt:

```
1.  private fun onFacesDetected(faces: List<FaceEmbeddings>) {
2.      lifecycleScope.launch(Dispatchers.Main) {
3.          val info = faces.map {
4.              Identity(
5.                  it.face,
6.                  viewModel.recogniseOrNull(it.embeddings)
7.              )
8.          }
```

```
9.          identitiesListAdapter.identities = info
10.     }
11. }
```

The view model's interface uses coroutines to ensure that any database operations run outside the main thread, so a new coroutine scope needs to be created. It also needs to ensure that interactions with the list view run on the main thread. These concerns are handled in line 2 of the preceding code snippet.

Lines 3 to 8 in the same code snippet take the list of **`FaceEmbeddings`** objects received from **`RealtimeFaceEmbeddingsExtractor`** and convert them into **`Identity`** objects for **`IdentityListAdapter`**. The database search necessary is done in line 6 for each face embeddings vector. Line 9 updates the list adapter, which causes the search results to be displayed in the list view.

Running the Android face recognition application

Figure 11.5 shows the application recognizing the author's face, added as shown in *figure 11.2* from a lower quality photograph.

You may try, with different persons, settings, and ages, to find a recognition threshold that works well for your specific application. The generic threshold of 1.20 works well for a first approach.

Figure 11.5*: Recognizing the author's face from an older photo*

The face recognition demonstration application cannot recognize identities that were never added.

Figure 11.6 shows that although it did detect the presence of two faces in a poster for a workshop the author has given in cooperation with a colleague, only one of the two faces was registered in the application's identity store.

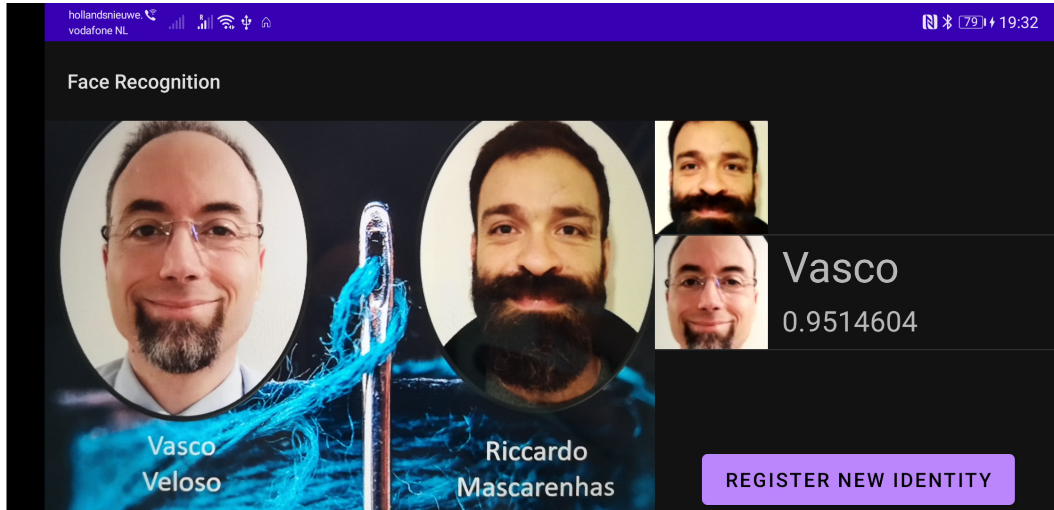

***Figure 11.6**: The application cannot recognize identities it does not know*

Conclusion

The Android application presented in this chapter puts together all the techniques, frameworks, and libraries worked on in all previous chapters. It can be used, for example, as the starting point for an application featuring user login via facial recognition.

The next chapter, *Image Processing with Generative Adversarial Networks*, shows another possible use of machine learning models, this time applied to image transformation.

CHAPTER 12
Image Processing with Generative Adversarial Networks

Introduction

This chapter demonstrates the possible usages of **Generative Adversarial Network** (**GAN**). We show how GANs can be used to generate realistic images automatically. GANs can also be used to process authentic images; these are presented as a GAN that applies a specific transformation effect to photographs.

Structure

We cover the following topics in this chapter:

- Understanding Generative Adversarial Networks
- Training a simple GAN
- Applying a special effect to photographs
- Taking anime-styled pictures in Android

Objectives

By the end of this chapter, you have understood the intuitive background behind GANs and developed an Android application that uses a GAN to apply an anime-like effect to images captured by the device's camera.

Understanding Generative Adversarial Networks

A **Generative Adversarial Network** (**GAN**) is a form of machine learning based on neural networks. As the name implies, a generative model aims to generate something, whereas the models we saw in the previous chapters are discriminative models. The former models generate new data instances and estimate such output's likelihood; the latter can discriminate between different data instances by applying a label and estimating the likelihood of correctly applying the label.

Note: GANs are only one type of generative model. There are also other types of generative models, just like there are different types of discriminative models.

Like other neural network models, a GAN is trained with a dataset that represents some distribution. The idea is that it learns to generate estimations of that distribution. Generally speaking, the initial input of a GAN is a set of random data.

In other words, a GAN may be trained with a set of pictures of dogs to generate new images of dogs. Alternatively, it may be trained with the MNIST dataset to generate images of handwritten digits, as we describe in the following section, *Training a simple GAN*.

GANs can be helpful in several applications, for example:

- **Restoring missing data**: An excellent example of this application is to upscale low-resolution images using GANs to generate the missing pixels in the resulting high-resolution image employing pixel value interpolation.
- **To provide inputs to other models or applications, such as the training of agents in a simulated environment**: A software agent that controls a robot might be fed fictional scenarios in a simulation by a GAN, enabling all kinds of tests without the risk of damaging equipment or hurting persons.
- **To transform or translate images**: Creating topological maps from satellite images or applying effects to photographs are examples of such an application.

Generative Adversarial Networks were first proposed by Ian Goodfellow et al. in their 2014 paper, *Generative Adversarial Networks*. They described GANs as being the result of a competition between two deep learning models: a generator and a discriminator.

The discriminator's role is to determine whether a data sample is real or fake; in other words, it needs to determine the probability that the sample comes from the training data set or a generator model. The generator learns to create plausible samples, aiming to produce samples good enough to fool the discriminator into

reporting that the generated samples are real. These networks are competing against each other, hence the network's adversarial nature. *Figure 12.1* shows an overview of this competition:

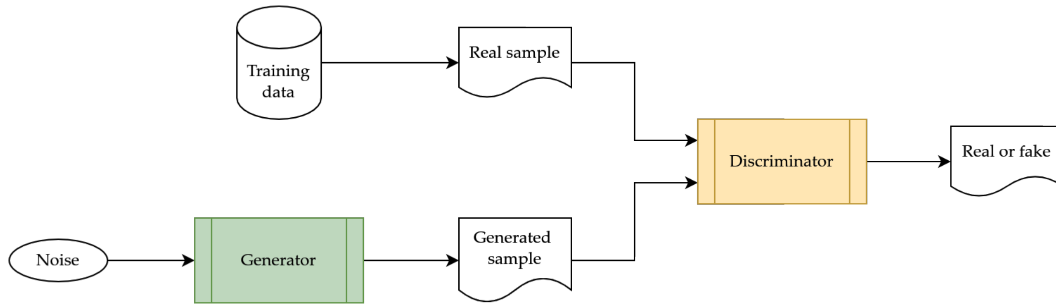

***Figure 12.1**: Overall GAN structure*

Given this competing nature, the behavior of a GAN differs from a deep learning network. Simply put, training a deep learning network is often a matter of minimizing its loss function's outcome. However, the ideal end state is an equilibrium as the generator and the discriminator in a GAN compete against each other in a zero-sum game. This equilibrium is known as the Nash equilibrium, named after the mathematician John Nash.

The Nash equilibrium is a game state where there is nothing to be gained by each player if they change their strategy, assuming that the other player also does not change theirs. In this state, the generator creates realistic images, and the discriminator is forced to guess (it guesses real half of the time, and fake the other half).

The problem is that it is difficult to detect this point of equilibrium, and there is no guarantee that it is ever reached. The most well-known problem preventing the equilibrium is mode collapse, which means the generator's outputs lose diversity.

Mode collapse can be described quite simply. Imagine a GAN being trained to generate handwritten digits. If the generator generates many good 2s, the discriminator may see so many 2s that it becomes less apt at discriminating other digits. The generator then continues to improve its 2s and eventually, generates mainly 2s. When the discriminator finally distinguishes the fake 2s from the real ones, the generator moves on to another digit, repeating the process. When this happens, the GAN is very good at faking a subset of the training data distribution at any given moment but very poor with the rest.

Since a GAN comprises two deep learning neural networks competing against each other, it follows that these networks need to be trained. They are trained together during each epoch, but they are updated separately between batches.

The discriminator needs to be updated first so that it can make helpful comparisons. It is trained using a method identical to the flow depicted in *figure 12.1*—samples classified as real are submitted along with samples classified as fake, making it better at distinguishing between them.

The generator is updated next. It produces a set of fake samples misrepresented as real ones because its goal is to deceive the discriminator. The discriminator's output represents its confidence that the input is real and is used to fine-tune the generator's parameters via backpropagation. *Figure 12.2* shows this setup:

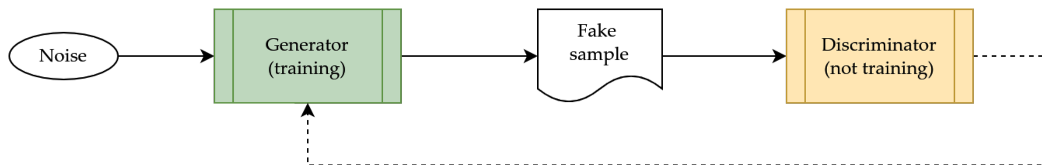

***Figure 12.2**: Training the generator*

Note that the discriminator model is not being updated while updating the generator, and vice-versa.

The generator can be used independently to produce the simulated samples once the GAN has reached a satisfactory state.

Training a simple GAN

We demonstrate the training process of a GAN through a model designed to generate handwritten digits based on the MNIST dataset. This model is part of the official TensorFlow documentation, and its training method has been modified to follow the convention used throughout this book.

The following code excerpt shows the beginning of the Python program, including dataset loading and preparation:

```
1.    import tensorflow as tf
2.    import matplotlib.pyplot as plt
3.    from tensorflow.keras import losses
4.    from tensorflow.keras import layers
5.    from tensorflow.keras import datasets
6.    import time
7.
8.   BUFFER_SIZE = 60000
9.   BATCH_SIZE = 256
```

```
10. GENERATOR_INPUT_SIZE = 100
11. NUMBER_OF_EXAMPLES = 16
12.
13. (train_images, train_labels), (_, _) = datasets.mnist.load_data()
14.
15. train_images = train_images.reshape(
16.   train_images.shape[0], 28, 28, 1).astype('float32')
17. train_images = (train_images - 127.5) / 127.5
18. train_dataset = tf.data.Dataset.from_tensor_slices(
19.   train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Lines 8 to 11 in the preceding code excerpt define some parameters used during training. The buffer size in line 8 represents the number of entries in the shuffled data set. The data set is divided into batches, with a fixed number of elements defined in line 9. As mentioned earlier, the generator takes some random data as input; the size of such data is defined in line 10. Finally, this code generates and periodically saves some example images for the user to visually evaluate the model's evolution; line 11 defines the number of example images generated each time.

Lines 13 to 16 in the preceding code excerpt load the dataset and add the channel dimension to the images. Line 17 normalizes the pixel values into the [-1, 1] interval, and lines 18 to 19 slice the dataset into random slices with 256 elements each.

It is time to build the models once the dataset is prepared. The following code excerpt shows the model used for the generator:

```
1.   def make_generator_model():
2.    model = tf.keras.Sequential([
3.        layers.Dense(7*7*256, use_bias=False,
4.            input_shape=(GENERATOR_INPUT_SIZE,)),
5.        layers.BatchNormalization(),
6.        layers.LeakyReLU(),
7.
8.        layers.Reshape((7, 7, 256)),
9.
10.       layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
11.         padding='same', use_bias=False),
```

```
12.         layers.BatchNormalization(),
13.         layers.LeakyReLU(),
14.
15.         layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
16.                padding='same', use_bias=False),
17.         layers.BatchNormalization(),
18.         layers.LeakyReLU(),
19.
20.         layers.Conv2DTranspose(1, (5, 5), strides=(2, 2),
21.             padding='same', use_bias=False, activation='tanh')
22.           ])
23.       return model
```

The first layer of the generator model takes a tensor with 100 elements as its input. The following layers reshape this input into a square matrix and keep changing its shape until it matches the shape of the images in the MNIST dataset.

Such shape changes are accomplished by the **Conv2DTranspose** layers, which implement an operation known as a transposed convolution. Simply speaking, these transposed convolution layers work backward, that is, from a tensor that could be the product of a convolution toward a tensor shaped like some original data.

The desired result is that the 100 initial elements of noise are transformed into an image of a handwritten digit similar to those in the MNIST dataset.

The following summary of the generator model helps visualize this transformation. In particular, note how the output shape of each layer progresses toward the final shape:

```
1.  _________________________________________________________________
2.  Layer (type)                 Output Shape              Param #
3.  =================================================================
4.  dense (Dense)                (None, 12544)             1254400
5.  _________________________________________________________________
6.  batch_normalization (BatchNo (None, 12544)             50176
7.  _________________________________________________________________
8.  leaky_re_lu (LeakyReLU)      (None, 12544)             0
9.  _________________________________________________________________
```

```
10. reshape (Reshape)            (None, 7, 7, 256)        0
11. _________________________________________________________________
12. conv2d_transpose (Conv2DTran (None, 7, 7, 128)        819200
13. _________________________________________________________________
14. batch_normalization_1 (Batch (None, 7, 7, 128)        512
15. _________________________________________________________________
16. leaky_re_lu_1 (LeakyReLU)    (None, 7, 7, 128)        0
17. _________________________________________________________________
18. conv2d_transpose_1 (Conv2DTr (None, 14, 14, 64)       204800
19. _________________________________________________________________
20. batch_normalization_2 (Batch (None, 14, 14, 64)       256
21. _________________________________________________________________
22. leaky_re_lu_2 (LeakyReLU)    (None, 14, 14, 64)       0
23. _________________________________________________________________
24. conv2d_transpose_2 (Conv2DTr (None, 28, 28, 1)        1600
25. =================================================================
26. Total params: 2,330,944
27. Trainable params: 2,305,472
28. Non-trainable params: 25,472
29. _________________________________________________________________
```

The discriminator model does the opposite—it takes a tensor with the same shape as the MNIST images and reduces it to one value. This output value represents the confidence that the input image is real. The following code excerpt shows the definition of the discriminator model:

```
1.  def make_discriminator_model():
2.      model = tf.keras.Sequential([
3.        layers.Conv2D(64, (5, 5), strides=(2, 2),
4.          padding='same', input_shape=[28, 28, 1]),
5.        layers.LeakyReLU(),
6.        layers.Dropout(0.3),
7.        layers.Conv2D(128, (5, 5), strides=(2, 2),
8.          padding='same'),
```

```
9.        layers.LeakyReLU(),
10.       layers.Dropout(0.3),
11.       layers.Flatten(),
12.       layers.Dense(1)
13.     ])
14.     model.compile(
15.         loss=losses.BinaryCrossentropy(from_logits=True),
16.         optimizer=tf.keras.optimizers.Adam(1e-4))
17.     return model
```

Note that the discriminator model is compiled as soon as it is defined, whereas the generator model is not. We do this because these models are trained under different configurations—the discriminator is updated individually, but the generator is updated using the discriminator's output. It is necessary to use two optimizers to alternate the generator and discriminator updates between batches.

The following summary of the discriminator model helps visualize the transformation of the 28x28 image into one prediction:

```
1.  _________________________________________________________________
2.  Layer (type)                 Output Shape              Param #
3.  =================================================================
4.  conv2d (Conv2D)              (None, 14, 14, 64)        1664
5.  _________________________________________________________________
6.  leaky_re_lu_3 (LeakyReLU)    (None, 14, 14, 64)        0
7.  _________________________________________________________________
8.  dropout (Dropout)            (None, 14, 14, 64)        0
9.  _________________________________________________________________
10. conv2d_1 (Conv2D)            (None, 7, 7, 128)         204928
11. _________________________________________________________________
12. leaky_re_lu_4 (LeakyReLU)    (None, 7, 7, 128)         0
13. _________________________________________________________________
14. dropout_1 (Dropout)          (None, 7, 7, 128)         0
15. _________________________________________________________________
16. flatten (Flatten)            (None, 6272)              0
```

```
17. _________________________________________________________________
18. dense_1 (Dense)              (None, 1)                 6273
19. =================================================================
20. Total params: 212,865
21. Trainable params: 212,865
22. Non-trainable params: 0
23. _________________________________________________________________
```

Finally, the GAN used to train the generator is defined. Looking back at *figure 12.2*, we see that the generator model is used to provide the discriminator's input. The following code excerpt shows the GAN's configuration:

```
1.  def make_gan(generator, discriminator):
2.   discriminator.trainable = False
3.   model = tf.keras.Sequential()
4.   model.add(generator)
5.   model.add(discriminator)
6.   model.compile(
7.         loss=losses.BinaryCrossentropy(from_logits=True),
8.         optimizer=tf.keras.optimizers.Adam(1e-4))
9.   return model
```

Remember that this GAN construction is meant to update only the generator, so the discriminator is placed in the GAN with training disabled.

All the models can now be created as per the following code excerpt:

```
1. generator = make_generator_model()
2. discriminator = make_discriminator_model()
3. gan = make_gan(generator, discriminator)
```

The training loop needs to be defined manually because of the GAN model's differentiated training requirements.

Each training step needs to perform the following tasks:

- Use the current state of the generator model to produce fake images
- Train the discriminator with the newly generated fake images and a set of images from the training set

- Train the generator using the updated discriminator state

The following code excerpt shows one way of implementing each training step:

```
1.  def train_step(real_images):
2.      batch_size = real_images.shape[0]
3.      noise = tf.random.normal(
4.        [batch_size, GENERATOR_INPUT_SIZE])
5.      generated_images = generator(noise, training=False)
6.      discriminator.trainable = True
7.      discriminator.train_on_batch(
8.          tf.concat([
9.            real_images,
10.           generated_images], axis=0),
11.         tf.concat([
12.           tf.ones((batch_size, 1)),
13.           tf.zeros((batch_size, 1))], axis=0)
14.     )
15.     discriminator.trainable = False
16.     gan.train_on_batch(
17.         noise,
18.         tf.ones((batch_size, 1))
19.     )
```

Note how the models are used with training enabled or disabled depending on their role in the GAN training cycle.

The complete training consists of repeating this training step until the network produces acceptable results (each repetition of the training step is called an epoch). The following code snippet shows the training loop:

```
1.  def train(dataset, epochs):
2.   for epoch in range(epochs):
3.     start = time.time()
4.     for image_batch in dataset:
5.       train_step(image_batch)
```

```
6.      print('Time for epoch {} is {} sec'.format(
7.        epoch + 1, time.time()-start))
```

Remember that the generator's input is a vector of random values, that is, noise. Every training session proceeds differently due to its random nature, so your results are different every time you train the model.

The following figures show samples of the generator's output after training sessions of different lengths. They are all different because of the random nature of the training, but they show the GAN progressing toward better quality output.

Figure 12.3 shows some outputs of the generator after 50 epochs. We can already recognize MNIST's handwritten digits' general shape and even a couple of numbers.

Figure 12.3*: The handwritten digit generator after 50 epochs*

Figure 12.4 illustrates one set of the generator's output after a training session that ran for 100 epochs. We can already recognize more digits, and their overall shape is better defined.

Figure 12.4*: The handwritten digit generator after 100 epochs*

Finally, *figure 12.5* depicts one last set of output after training the GAN for 500 epochs. We can recognize even more digits, and their shape continues to improve.

***Figure 12.5**: The handwritten digit generator after 500 epochs*

Fake images created by simple Generative Adversarial Networks such as this are seldom perfect, but some samples can be excellent. This demonstration GAN may or may not continue to improve after longer training sessions. Either way, it is a good example of the potential in GANs—it draws digits out of noise, after all!

Applying a special effect to photographs

Image transformation is another interesting class of GAN applications. Among the most popular are the special effects applied to photographs, which are common in social media and photography applications. Of course, we do not mean that all such effects are implemented with GANs, but surely a few involve machine learning.

Transforming images into anime-style pictures

Jie Chen, Gang Liu, and Xin Chen published a paper *AnimeGAN: A Novel Lightweight GAN for Photo Animation* in 2020. Their goal was to find a way to transform real-world photos into anime-style images.

Note: Anime is a Japanese term (アニメ) that refers to all animated works. In English, many people use it to refer to animation produced in Japan or employing a style similar to Japanese animation.

They examined the existing work in the fields of Neural Style Transfer and Image-to-Image translation. The former technique aims to create a new image by combining the content of one image with the style of another. The latter is typically used to map an image into a different domain, for example, grayscale to color or low-resolution to high-resolution.

Jie Chen et al. noticed that the existing networks either did not transfer the specific style of anime drawings with the desired quality or required many computational resources. So, they designed a generative adversarial network named AnimeGAN to tackle the specific problem of mapping images to the anime drawing style using minimal resources.

The AnimeGAN's generator is similar to a symmetrical encoder-decoder network, and the discriminator is a convolutional network employing spectral normalization.

An encoder-decoder network contains two components: the first is an encoder that takes a sequence with a variable length as an input and outputs some internal state with a fixed shape, and the second is a decoder that takes this state and maps it into a different variable-length sequence. So, a network of this type aims to map one input to an output where their corresponding lengths are different. Machine translation is a possible application of these models.

Besides the reduction in computational cost, Jie Chen et al.'s innovations were focused on the loss functions. Separate loss functions were employed for calculating grayscale, content, and color loss values carrying different weights, carefully chosen to produce a balanced result.

Note: The full version of the original AnimeGAN paper is available in the project's GitHub repository at https://github.com/TachibanaYoshino/AnimeGAN.

An improved version of the network, called AnimeGANv2, has been developed. We use this version for our demonstrations.

The AnimeGANv2 project repository in GitHub (**https://github.com/TachibanaYoshino/AnimeGANv2**) contains its TensorFlow implementation and instructions to use the datasets to train the network. The programs provided are only compatible with TensorFlow version 1 and tested with TensorFlow 1.8 and 1.15.

Fortunately, the authors included one pre-trained saved model in the repository; it is located in the **`pb_model_Hayao-64`** directory.

If you wish to reproduce their results and don't want to compromise your TensorFlow 2 environment, you can use a TensorFlow 1 virtual environment as described in *Chapter 5, Introduction to TensorFlow*.

Converting the GAN to the TensorFlow Lite format

The model's conversion to TensorFlow Lite is straightforward after downloading the entire **`pb_model_Hayao-64`** directory contents. The following code excerpt shows how it can be done:

```
1.  import tensorflow as tf
2.
3. converter = tf.compat.v1.lite.TFLiteConverter.from_saved_model(
4.     'pb_model_Hayao-64',
5.     signature_key = 'custom_signature')
6.
7. tflite_model = converter.convert()
8.
9.  with open('AnimeGANv2_Hayao-64.tflite', 'wb') as f:
10.  f.write(tflite_model)
```

One difference compared with the conversion programs used previously is that we must specify the name of the model's signature key. A model's signature is used to identify its inputs and outputs and has a default name. However, we can change the signature's name when saving the model; the TensorFlow Lite converter needs to know the new name when that happens.

Trying the GAN in the computer

The converted GAN model can be tested on the computer before building the Android application. The following code excerpt shows how this test can be conducted:

```
1.   import tensorflow as tf
2.
3.   image = tf.keras.preprocessing.image.load_img(
4.       'busy-street.jpg',
5.       target_size=(405, 540),
6.       interpolation='bilinear')
7.
8.   image_data = tf.keras.preprocessing.image.img_to_array(image)
9.   image_data = image_data / 127.5 - 1.0
10.  input = tf.expand_dims(image_data, axis=0)
11.
12.  interpreter = tf.lite.Interpreter(
13.      model_path='AnimeGANv2_Hayao-64.tflite')
14.
15.  interpreter.resize_tensor_input(0, input.shape, strict=True)
16.  interpreter.allocate_tensors()
```

```
17.
18. input_details = interpreter.get_input_details()
19. output_details = interpreter.get_output_details()
20.
21. interpreter.set_tensor(input_details[0]['index'], input)
22. interpreter.invoke()
23.
24. output_data = (interpreter.get_tensor(
25.     output_details[0]['index'])[0] + 1.) / 2 * 255
26. output_data = tf.cast(tf.clip_by_value(output_data, 0, 255),
27.     tf.uint8)
28.
29. tf.keras.preprocessing.image.save_img(
30.     'output-tflite.png', output_data)
```

Lines 3 to 6 in the preceding excerpt load the image from the disk. Note that it is being resized to be manageable so that the operation does not take long. For comparison, the AnimeGAN authors used 256x256 images in their experiments.

The image is then normalized, and lines 8 to 10 adjust its tensor. Lines 12 to 19 load the TensorFlow Lite model, retrieve its input and output information, and allocate the necessary memory. Finally, lines 21 and 22 invoke the model to run inference.

Image denormalization and pixel range adjustments are performed in lines 24 to 27 before the final image is saved to disk.

Figure 12.6 shows a photo of a street in Amsterdam used as input for a test of the TensorFlow Lite version of AnimeGANv2:

***Figure 12.6**: A photo of a street*

Figure 12.7 illustrates the image produced by the Generative Adversarial Network; it was transformed into a cartoonish drawing:

Figure 12.7*: An image of a street after being processed by AnimeGANv2*

Having tested the TensorFlow Lite model, we are now ready to incorporate it into an Android application.

Taking anime-styled pictures in Android

Building an Android application to take photos in an anime style is simple. We need to adapt the image capture application developed in *Chapter 7*, *Android Camera Image Capture with CameraX*, so that the captured image is transformed before being shown to the user.

Repurposing the image capture application

The original image capture application was developed with the following requirements in mind:

- Display a live camera preview
- Allow the user to capture a still image and save it in the device's storage
- Display the captured image to the user

Now that the TensorFlow Lite version of the AnimeGANv2 model is ready, there is only one additional requirement—to process the captured image before saving it.

Initially, the CameraX library was responsible for capturing and saving the image, but now we need to intervene in the process. The new application needs to capture and process the image according to the following steps:

1. Capture the image into memory instead of directly into storage.
2. Have AnimeGANv2 transform the image.
3. Save the transformed image in the device's photo store.

The captured image can only be shown to the user after these steps are complete.

Capturing an image into memory

To use CameraX to capture an image into memory instead of directly to a file in the device's storage, all that is necessary is to use another version of the **`ImageCapture`** class' **`takePicture()`** function.

One overload receives an instance of **`OnImageCapturedCallback`**. This object has its **`onCaptureSuccess()`** function called when the image is captured. With the default settings, the single function's argument is an instance of **`ImageProxy`** containing a JPEG-encoded picture.

The overall **`CameraPreviewFragment`**'s implementation of **`takePicture()`** becomes similar to the one shown in the following code excerpt:

```
1.     private fun takePicture() {
2.    imageCapture.takePicture(
3.        ContextCompat.getMainExecutor(context),
4.      object : ImageCapture.OnImageCapturedCallback() {
5.          override fun onCaptureSuccess(image: ImageProxy) {
6.          image.close()
7.          }
8.        override fun onError(exception: ImageCaptureException) {
9.            Log.e(TAG, "Failed to capture the picture.", exception)
10.             Toast.makeText(context, R.string.picture_not_captured,
11.                 Toast.LENGTH_SHORT).show()
12.       }
13.       }
14.   )
15.   }
```

The three previously mentioned steps need to take place within the scope of the **`onCaptureSuccess()`** function.

Transforming the captured image

The AnimeGANv2 model has one characteristic that makes it very flexible and restricts the optimization options available—its input dimensions are dynamic. Its input tensor's shape is [1, -1, -1, 3], meaning that the model accepts images of any size as its input.

Unfortunately, it also means that hardware acceleration is impossible in Android because the TensorFlow Lite's delegates for GPU and the NNAPI do not currently support models with dynamic inputs. The only optimization possible at the moment is to enable multithreading.

Note: The adventurous readers may try their hand at modifying the model to set a fixed input size. One way of doing this is to change the model creation and training code. Additionally, we can try a technique that's also used in the next chapter, *Describing Images with NLP*, to skip the first model nodes.

Such limitation also means that the model runs slowly on most devices, so we have decided to limit the maximum input image size to the same size used during training, that is, 256x256 pixels.

You are encouraged to experiment with different dimensions on your device and see the impact of small changes in the input dimensions on the processing time.

Having copied the AnimeGANv2 tflite model file to the Android application project's assets folder, we can build the class that applies the model to an image.

This class has the following responsibilities:

- Manage the TensorFlow Lite model
- Resize, rotate, and normalize the input image
- Run inference on the input image to obtain an anime-styled output image
- Deformalize the output into the standard image format

The base class structure shown in the following code excerpt takes care of the model management tasks:

```
1.  class AnimeGanModel(context: Context): AutoCloseable {
2.
3.      private val inputDataType: DataType
4.      private val interpreter: Interpreter
5.      private val modelFile: ByteBuffer
```

```
6.
7.      init {
8.          val options = Interpreter.Options().apply {
9.              setNumThreads(4)
10.         }
11.         modelFile = FileUtil.loadMappedFile(
12.             context, "AnimeGANv2_Hayao-64.tflite")
13.         interpreter = Interpreter(modelFile, options)
14.         interpreter.resizeInput(0,
15.             intArrayOf(1, IMAGE_WIDTH, IMAGE_HEIGHT, 3))
16.
17.         inputDataType = interpreter.getInputTensor(0).dataType()
18.     }
19.
20.     override fun close() {
21.         interpreter.close()
22.     }
23.
24. }
```

Note that multithreading is enabled in the interpreter via the options object created in lines 9 to 10 of the preceding excerpt. Additionally, we need to specify the expected input size before running inference because the model's input is dynamic. The model's input tensor is resized in lines 14 and 15.

Loading the image into the model's input tensor employs a familiar pattern, as shown in the following code snippet:

```
1.  private fun loadImage(bitmap: Bitmap, rotationDegrees: Int)
2.          : ByteBuffer {
3.      val tensorImage = TensorImage(inputDataType)
4.      tensorImage.load(bitmap)
5.      return buildProcessor(rotationDegrees)
6.          .process(tensorImage)
7.          .buffer
8.  }
```

Functions from the TensorFlow Lite library are used to build a tensor from an Android Bitmap, and then an image processor is applied before the tensor is converted into an array of bytes.

The image processor is responsible for resizing, rotating, and normalizing the input image and is created dynamically because of the rotation operation, as demonstrated in the following code excerpt:

```
1.    private fun buildProcessor(rotationDegrees: Int) =
2.      ImageProcessor.Builder()
3.          .add(
4.              ResizeOp(IMAGE_WIDTH, IMAGE_HEIGHT,
5.                  ResizeOp.ResizeMethod.BILINEAR)
6.          )
7.          .add(
8.              Rot90Op(4 - rotationDegrees / 90)
9.          )
10.         .add(
11.             NormalizeOp(127.5f , 127.5f)
12.         )
13.         .build()
```

Inference is performed by running the interpreter on the input tensor. Once again, the pattern is familiar and is depicted in the following code excerpt:

```
1.  fun process(bitmap: Bitmap, rotationDegrees: Int): Bitmap {
2.      val outputBuffer = allocateOutputBuffer()
3.      interpreter.run(
4.          loadImage(bitmap, rotationDegrees),
5.          outputBuffer.buffer.rewind()
6.      )
7.      return postprocess(outputBuffer.floatArray)
8.  }
9.
10. private fun allocateOutputBuffer(): TensorBuffer =
11.     interpreter.getOutputTensor(0).let {
12.         TensorBuffer.createFixedSize(
```

```
13.                  intArrayOf(1, IMAGE_WIDTH, IMAGE_HEIGHT, 3),
14.                  it.dataType())
15.     }
```

Just like the input vector's format, the output vector is a normalized array of floating-point values in the [-1, 1] range. Each tuple of three floating-point values represents the intensities of each pixel's (red, green, blue) – or RGB – color channels.

The final task is the denormalization of the output image and its conversion into an Android Bitmap. The numbers need to be transformed back to the [0, 255] interval and grouped in tuples of four byte-sized values. These tuples represent the intensities of each pixel's (**Alpha, Red, Green, Blue**) – or **ARGB** – transparency and color channels.

One possible way to implement such a transformation is shown in the following code snippet. This implementation may not be the most efficient, but it is easy to understand:

```
1.  private fun postprocess(data: FloatArray): Bitmap {
2.      val pixelCount = (data.size / 3)
3.      val pixels = IntArray(pixelCount)
4.      var floatPos = 0
5.      for (i in 0 until pixelCount) {
6.          pixels[i] = Color.rgb(
7.              ((data[floatPos++] + 1.0f) / 2.0f * 255.0f).toInt(),
8.              ((data[floatPos++] + 1.0f) / 2.0f * 255.0f).toInt(),
9.              ((data[floatPos++] + 1.0f) / 2.0f * 255.0f).toInt()
10.         )
11.     }
12.     return Bitmap.createBitmap(pixels, IMAGE_WIDTH, IMAGE_HEIGHT,
13.         Bitmap.Config.ARGB_8888)
14. }
```

Each element of the array declared in line 3 of the preceding code snippet contains one pixel's ARGB tuple. The original floating-point values are converted into a fully opaque ARGB pixel in lines 6 to 10. The final Android bitmap is created in lines 12 and 13 in the ARGB format.

The constants defining the image size are placed in the class' companion object, as shown in the following code excerpt:

```
1.  companion object {
2.      const val IMAGE_WIDTH = 256
3.      const val IMAGE_HEIGHT = 256
4. }
```

As mentioned earlier, the default configuration of the **ImageCapture** class results in a JPEG-encoded image being provided to our callback function.

The **AnimeGanModel** class we just showed can process only images in an Android bitmap format. We must convert that image into the Android bitmap format before invoking the model class.

A second class called **AnimeTransformation** is created to handle this conversion. We enforce the separation of concerns because it is not the model's responsibility to know how to handle whatever image format may be produced by its data source.

The **AnimeTransformation** class's implementation is shown in the following code excerpt:

```
1.  class AnimeTransformation(context: Context) : AutoCloseable {
2.
3.      private val model = AnimeGanModel(context)
4.
5.      override fun close() {
6.          model.close()
7.      }
8.
9.      suspend fun transform(proxy: ImageProxy): Bitmap =
10.         withContext(Dispatchers.IO) {
11.             val bitmap = bitmapFromJpeg(proxy)
12.             return@withContext model.process(
13.                 bitmap, proxy.imageInfo.rotationDegrees)
14.         }
15.
16.     private fun bitmapFromJpeg(proxy: ImageProxy): Bitmap {
17.         val jpegBuffer = proxy.planes[0].buffer
18.         val jpegSize = jpegBuffer.remaining()
19.         val data = ByteArray(jpegSize)
```

```
20.         jpegBuffer.get(data)
21.         return BitmapFactory.decodeByteArray(data, 0, data.size)
22.     }
23.
24. }
```

As you can see in lines 9 and 10 of the preceding excerpt, this class has an additional responsibility to run the image processing task as a coroutine. This way, the coroutine execution can be sent to a separate pool without interfering with the main application thread's operation.

Saving the modified image in the device's gallery

Once the image has been processed, the next logical step is to save it in the device's photo gallery.

Saving the image in the device's photo gallery is not difficult. All that is necessary is to prepare the required columns for the media store's database with the correct data and transfer the image data into the store.

One possible implementation of these steps is shown in the following code excerpt:

```
1.  spend fun saveImageToGallery(context: Context, bitmap: Bitmap)
2.      : Uri? = withContext(Dispatchers.IO) {
3.
4.      val name = "anime-" + SimpleDateFormat(
5.              "yyyy-MM-dd-HH-mm-ss-SSS", Locale.US)
6.          .format(System.currentTimeMillis());
7.
8.      val contentValues = ContentValues().apply {
9.          put(MediaStore.MediaColumns.DISPLAY_NAME, name)
10.         put(MediaStore.MediaColumns.MIME_TYPE, "image/jpeg")
11.         if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
12.             put(MediaStore.MediaColumns.RELATIVE_PATH,
13.                 Environment.DIRECTORY_PICTURES)
14.             put(MediaStore.MediaColumns.IS_PENDING, 1)
15.         }
```

```
16.   }
17.
18.    val resolver = context.contentResolver
19.    val imageUri = resolver
20.        .insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
21.                contentValues)
22.    imageUri?.let {
23.        resolver.openOutputStream(it)
24.    }?.use {
25.        bitmap.compress(Bitmap.CompressFormat.JPEG, 90, it)
26.        contentValues.clear()
27.        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
28.           contentValues.put(MediaStore.Video.Media.IS_PENDING, 0)
29.        }
30.        resolver.update(imageUri, contentValues, null, null)
31.    }
32.
33.    return@withContext imageUri
34.}
```

The media store database's content tuple is prepared in lines 8 to 16. The Android content resolver is then used to insert the new image's row in the media store in lines 18 to 21. The result of such insertion is a URI that uniquely identifies the new entry.

Based on that URI, the image data is written to the store in lines 22 to 31. The Android's content resolver design abstracts the final storage medium away. However, we know that an image file is being written in this specific case and appears in the device's gallery application.

Putting it all together

All the code necessary to apply an anime style to a JPEG image and save it to the device's media store is in place. We are now aware of the decision to limit this demonstration application to images 256x256 pixels in size and that image processing takes some time.

So, the preview view's aspect ratio was adjusted to achieve a what you see is what you get effect, so it is fixed at 1:1, that is, a square. An indefinite progress bar was

added so that the user can get visual feedback that the image was captured and is being processed. *Figure 12.8* illustrates these changes. The necessary changes in the XML layout files are included in the accompanying code package.

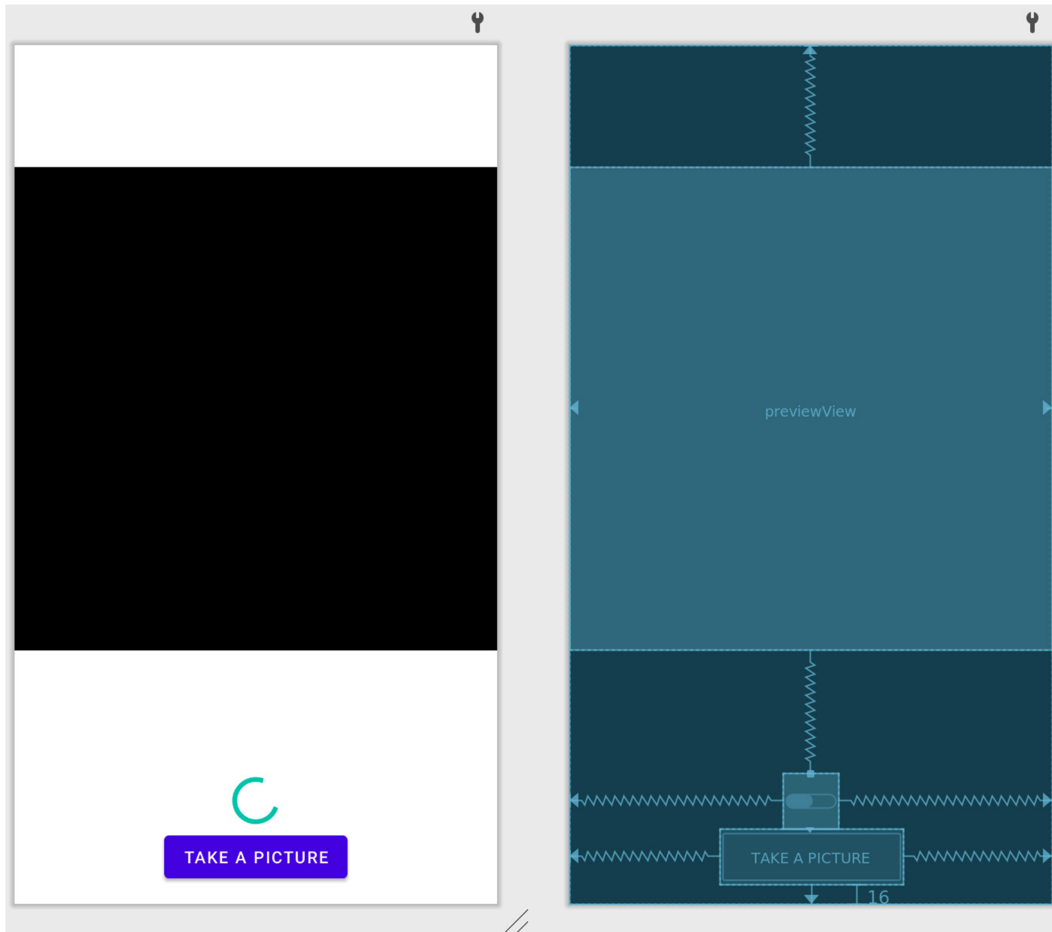

Figure 12.8: *The updated capture screen layout*

Introducing a fixed aspect ratio means that the CameraX use case configuration needs to be adjusted. The configuration of the **`ImageCapture`** use case and its binding to the camera provider, made in the **`CameraPreviewFragment`** class's **`configureCameraUseCase()`** function, is shown in the following code excerpt. The pattern should also be familiar.

```
1. imageCapture = ImageCapture.Builder()
2.     .setCaptureMode(CAPTURE_MODE_MAXIMIZE_QUALITY)
3.     .setTargetRotation(requireActivity().
4.         windowManager.defaultDisplay.rotation)
```

```
5.      .setTargetResolution(Size(
6.          AnimeGanModel.IMAGE_WIDTH,
7.          AnimeGanModel.IMAGE_HEIGHT))
8.      .build()
9.
10. // ... camera selector ...
11.
12.  val useCaseGroup = UseCaseGroup.Builder()
13.     .addUseCase(preview)
14.     .addUseCase(imageCapture)
15. previewView.viewPort?.let {
16.     useCaseGroup.setViewPort(it)
17. }
18.
19. cameraProvider.unbindAll()
20.
21. cameraProvider.bindToLifecycle(this, cameraSelector,
22.     useCaseGroup.build())
```

The image transformation classes can now be incorporated in the image capture callback. Its **onCaptureSuccess()** function implementation can be similar to the following code snippet:

```
1.  override fun onCaptureSuccess(image: ImageProxy) {
2.      lifecycleScope.launch(Dispatchers.Main) {
3.          progressBar.visibility = View.VISIBLE
4.          btnTakePicture.visibility = View.INVISIBLE
5.          saveImageToGallery(
6.              requireContext(),
7.              transformation.transform(image)
8.          )
9.          ?.also {
10.             val action = CameraPreviewFragmentDirections
11.                 .actionCameraPreviewFragmentToPictureFragment(
```

```
12.                     it.toString()
13.                 )
14.             requireView().findNavController().navigate(action)
15.         }
16.         progressBar.visibility = View.INVISIBLE
17.         btnTakePicture.visibility = View.VISIBLE
18.
19.         image.close()
20.     }
21. }
```

Line 2 in the preceding excerpt shows that the coroutine should always return to the main application thread because it manipulates user interface elements in lines 3 and 4 and 9 to 17. Lines 5 to 7 are responsible for processing and saving the image, and lines 10 to 14 switch to the fragment showing the processed image, sending it the saved image's URI.

The **`PictureFragment`** now needs to load the image from a URI instead of a file, so its **`onViewCreated()`** implementation needs a slight adjustment to tell the Guice library that the image source is a URI. Such adjustment is shown in line 4 of the following code excerpt:

```
1. override fun onViewCreated(view: View, savedInstanceState: Bundle?)
   {
2.     super.onViewCreated(view, savedInstanceState)
3.     Glide.with(this)
4.         .load(Uri.parse(args.filePath))
5.         .into(this.imageView)
6. }
```

Naturally, the application's Gradle build files also need to receive entries with the TensorFlow Lite library dependencies. These changes are similar to those made in different locations throughout this book and are available in the accompanying code.

Figure 12.9 shows the final preview from the author's device. Processing the image shown here took approximately 7 seconds, but this time varies depending on the device and its available resources.

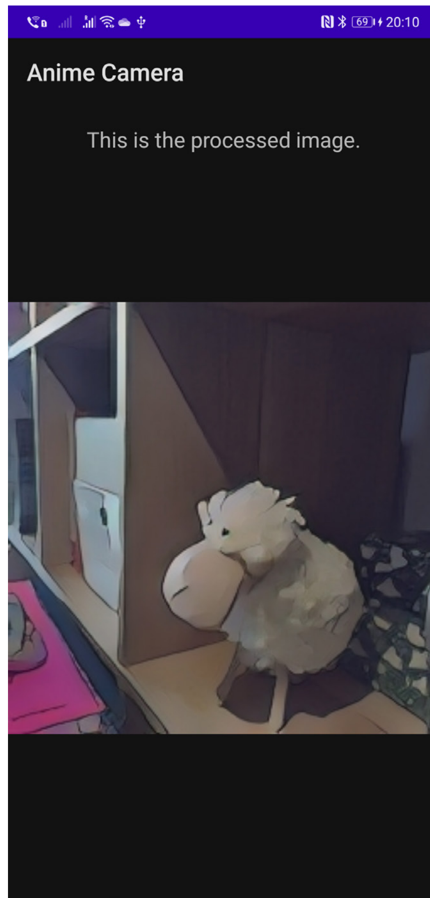

Figure 12.9*: The application just after processing a photo*

Conclusion

Generative Adversarial Networks are a concept with plenty of potential for achieving creative and transformational tasks more efficiently and, in some cases, with more quality than before. Its training can be very challenging, although some significant challenges have been identified. It is an active field of research whose goal is to make GANs as efficient and easy to train as deep learning networks.

The next chapter shows yet another application of machine learning—a neural network with a recurrent architecture that enables it to have some memory.

CHAPTER 13
Describing Images with NLP

Introduction

This chapter briefly touches the world of natural language processing, a task for which recurrent neural networks are particularly well suited. We showcase the Show and Tell model that attempts to describe the scenes depicted in images in plain English. We also build a sample application to use this model to provide descriptions of photographs.

Structure

We cover the following topics in this chapter:

- Understanding recurrent neural networks
- Evolving into long short-term memory networks
- Performing automatic image captioning
- Implementing automatic image captioning in Android

Objectives

By the end of this chapter, you have understood the intuition behind **Recurrent Neural Networks** (**RNN**) and **Long Short-Term Memory** (**LSTM**) networks. You also know how to use the LSTM-based Show and Tell model to automatically generate picture descriptions. An Android application is built to incorporate descriptions in pictures captured by the device's camera.

Understanding recurrent neural networks

All the neural networks described in this book so far are "forward-only" networks because each neuron's output is based only on its input. Nothing is left over after some piece of information moves through it.

Conversely, an RNN is formed by neurons or cells that possess one additional input and one additional output. These are used internally to propagate information derived from its previous computation to the following computation. The result is a neuron with a form of memory. *Figure 13.1* is a possible representation of an RNN neuron with its internal state feedback loop:

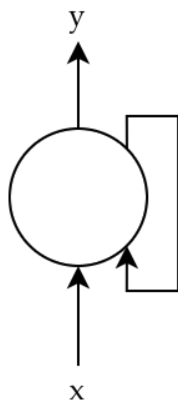

***Figure 13.1**: An RNN neuron with its state loop*

The inner workings of one RNN neuron can be more apparent by adopting a visualization technique called unrolling. The role of the network's internal state propagation mechanism as a form of memory becomes more explicit by unrolling the flow of information across a neuron over time. This is shown in the following figure:

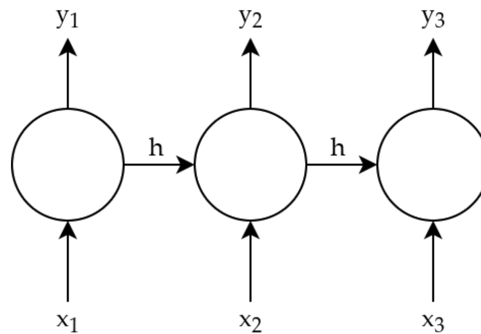

***Figure 13.2**: Unrolling the operation of an RNN neuron over time*

Figure 13.2 shows the unrolling of one RNN neuron's operation as it processes three inputs—x_1, x_2, and x_3—in sequence. Each subsequent step receives the internal (or hidden) state memorized by the previous step as input. That internal state is then mutated and propagated to the following step.

RNNs are particularly suited to processing an input of variable length for tasks that require maintaining context, known as sequence prediction problems. As the name implies, such problems involve predicting an output that depends on the input sequence. Translations, text generation, or speech recognition are some examples of sequence prediction problems.

Evolving into long short-term memory networks

A disadvantage of classical RNNs is that they have a short memory span. The internal state is transformed at each step, so it quickly loses any relation to the earliest inputs.

Consequently, sequence prediction problems requiring more extended information retention periods are not solved as effectively. For example, the language-related tasks mentioned previously become less efficient as the amount of information memorized declines.

As humans, it is easy for us to understand this limitation. Imagine that you are translating this sentence into some other language. How difficult would it be if you could remember the entire sentence as you translate it? What about if you could only remember the previous five words, or three, or even two?

Long Short-Term Memory (**LSTM**) networks were developed as an evolution in RNN research. Their fundamental characteristic is that each cell (neuron) has one

extra input and output. These are used to process and produce an additional set of network states. Since these states are often propagated with little transformation, the network's memory span increases (although each cell's state controls the actual degree of transformation).

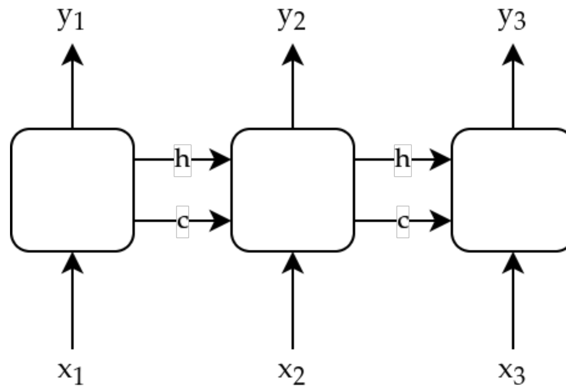

Figure 13.3*: Unrolling an LSTM cell*

Just like *figure 13.2*, *figure 13.3* shows one possible unrolling of an LSTM cell at work. Here, we can observe how the same internal state as before (*h*) is propagated alongside the new long-term memory state (*c*). Unlike the hidden state, the long-term memory state suffers little in the way of transformations. It helps determine the amount of information that should be retained from the previous hidden state, so it allows us to preserve inputs that were received farther away in time.

Tip: As with the other network models described, we stop at intuitive understanding and do not dive into RNNs or LSTM network models. Stanford University School of Engineering's 2017 CS231n lectures, particularly lecture number 10, is a good source of details. You can find these lectures at https://youtu.be/vT1JzLTH4G4.

Performing automatic image captioning

The problem of generating text that describes the contents of an image is a challenge for the field of artificial intelligence, building a path between computer vision and natural language processing.

It is a challenging task, significantly more complicated than the object classification or recognition tasks mentioned earlier in this book. However, it has many applications, the most evident of which is helping the visually impaired navigate the visually heavy world of digital media.

A challenge was raised in 2015, almost recognizing the difficulty of such an endeavor and encouraging further research in this field. It became known as the COCO 2015 Image Captioning Task, and its purpose was to hold a competition between image captioning algorithms.

Note: COCO or Common Objects in Context is a dataset developed for training object detection, segmentation, and captioning algorithms, and new challenges are proposed annually. The dataset and its underlying research paper are available at https://cocodataset.org.

One of the two algorithms that won the challenge is the Show and Tell algorithm, described in the 2016 paper *Show and Tell: Lessons learned from the 2015 MSCOCO Image Captioning Challenge* by Oriol Vinyals et al.

This algorithm was chosen as this chapter's basis for its quality and popularity and because it was implemented in TensorFlow. An open-source implementation called im2txt was made available by its original authors. The open-source community contributed with derived projects, providing fully or partially trained models so that implementers and researchers could study the model without spending a significant amount of computation resources and several days training the model.

One such derived project was built by Wei-Lin Ku (Hugh Ku) and is available at **https://github.com/HughKu/Im2txt**. We use it as the base for our work with TensorFlow Lite in this chapter.

Understanding the Show and Tell model

The Show and Tell model generates text from an input image.

We used a **Convolutional Neural Network** (**CNN**) to produce an embedding describing the image. This technique is similar to the one used for face recognition in *Chapter 10*, *Identifying Faces in Android with TensorFlow Lite*, where the embeddings produced by a CNN are used to identify human faces.

The embedding obtained by the CNN is then fed into an LSTM network for translation into English language text. This process is similar to a generation process, except that the source data is not random but an image.

Show and Tell is described as one model but can also be described as the composition of two models, as its high-level overview illustrated in *figure 13.4* suggests:

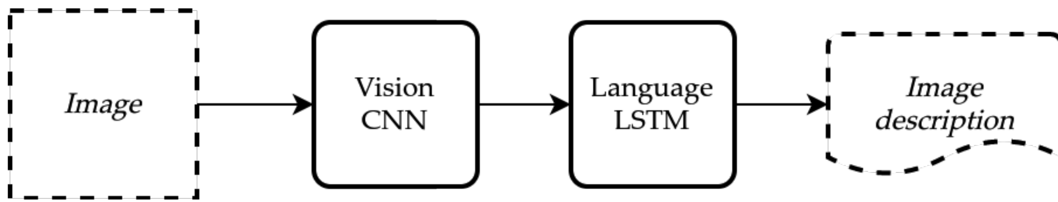

***Figure 13.4**: High-level overview of the Show and Tell model*

The caption-generating LSTM network takes the image embedding as its input and produces a probabilistic word classification as its output. Words are represented as an embedding model.

Note: A word embedding model aims to represent words as fixed-length vectors, with the advantage that the contextual relationship between them is directly related to their distance from one another. The TensorFlow documentation at https://www.tensorflow.org/text/guide/word_embeddings provides an excellent introduction to this concept.

The process for translating an image into its English description using the Show and Tell model has the following sequence:

1. Use the image processing CNN to extract an embedding describing the source image. This embedding is used to create the LSTM network's initial state.
2. Feed the state to the LSTM network. It produces a vector encoding the likelihood of each learned word occurring in the text at this point and a vector containing the new LSTM network state.
3. Pick the most probable words to build sentences. These are then fed to the network in the following iteration.
4. Repeat steps 2 and 3 using a suitable algorithm, like Beam Search, until one or more proper sentences are found. Refer to the *Implementing Beam Search* section later in the chapter for a description of Beam Search.
5. Choose the most probable sentence.

Given the probabilistic nature of this process, the results usually vary a bit between otherwise identical runs.

Some descriptions may be quite amusing when evaluated from a human perspective, but many are pretty accurate.

Converting the Show and Tell model to TensorFlow Lite

The accompanying code contains our version of Wei-Lin Ku's (Hugh) repository with all the necessary adjustments to obtain a TensorFlow Lite version of the Show and Tell model.

However, the TensorFlow checkpoint files containing the partially trained model are pretty big, so they still need to be downloaded before continuing. The URL can be found in the **`TFLITE.md`** or **`README.md`** file inside the **`im2txt`** directory.

You may also need to install additional libraries, like matplotlib or tensorboard, if you have not done so before. Use Python's pip tool if the programs report that any dependencies are missing.

Important: The Show and Tell model was initially developed with TensorFlow version 1. It is thus mandatory to use TensorFlow 1 to run inference and freeze the graph, as described in the following paragraphs.

We can try out the model once the checkpoint files are downloaded and extracted into the **`im2txt/model/Hugh/train`** directory. We use the **`im2txt/run_inference.py`** program to apply the model to one or more input images. We added sensible testing defaults to its command-line options, so it may be executed from its location, as follows:

```
1. python run_inference.py
```

The inference results are shown when the analysis of Hugh's test image is complete. They should be quite similar to the following output:

```
1. Captions for image test.jpg:
2.   0) a young boy wearing a hat and tie . (p=0.000195)
3.   1) a young boy wearing a blue shirt and tie . (p=0.000100)
4.   2) a young boy wearing a blue shirt and a tie . (p=0.000045)
```

This output proves that all files are placed in the correct locations and that the model is working correctly.

The next task that needs to be accomplished is freezing the graph. Graph freezing means the model's hyperparameters are saved as constants, and some additional information necessary for training is removed. The model becomes suitable for inference only.

We provided the **`freeze_model.py`** program to freeze the model. It also contains default values for its parameters so that it can be run from its location, as follows:

```
1. python freeze_model.py
```

Some output identical to the following lines is displayed once the model is frozen:

```
1. INFO:tensorflow:Froze 382 variables.
2. I0810   18:48:03.408173   140326452109696   graph_util_impl.py:334]
   Froze 382 variables.
3. INFO:tensorflow:Converted 382 variables to const ops.
4. I0810   18:48:03.551802   140326452109696   graph_util_impl.py:394]
   Converted 382 variables to const ops.
```

The frozen model is now saved in the **model/frozen_graph.pb** file and used as the source for the model's conversion to TensorFlow Lite.

Visualising the model with tensorboard

Use the **to_tensorboard.py** program to generate a compatible log file if you wish to visualize the Show and Tell model in the tensorboard application. This is shown as follows:

```
1. python to_tensorboard.py
```

The program creates a log file in the **model/Hugh/logs** directory. The tensorboard application can now be started with the following command:

```
1. tensorboard --logdir model/Hugh/logs
```

Once it is ready, the tensorboard program outputs one line with the URL to open its user interface. For example:

```
1. TensorBoard 1.15.0 at http://tuga:6006/ (Press CTRL+C to quit)
```

The tensorboard application's user interface is shown when the URL is opened on a browser, allowing inspection of the model's graph, as illustrated in *figure 13.5*:

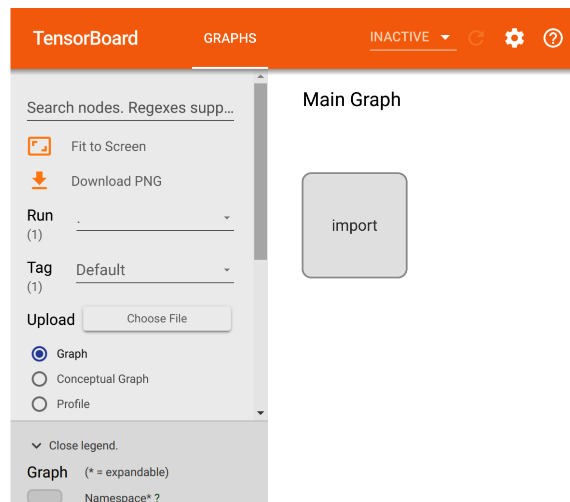

***Figure 13.5**: The tensorboard application's user interface*

The **import** node shown in *figure 13.5* is expandable, so double-clicking on it reveals further graph details.

Figure 13.6 shows the contents of the **import** node as displayed in tensorboard. We can recognize the main building blocks of the Show and Tell model:

- An InceptionV3 model used to extract embeddings from the input image
- The LSTM network used for translating the embeddings into English sentences

By examining the remaining namespace nodes, we can see that the **decode** node accepts an image in the JPEG format because it relies internally on the **DecodeJpeg** operation. Unfortunately, this operation is not available in TensorFlow Lite, which means the **image_feed** node cannot be used as the image input.

We feed the image directly to the InceptionV3 model in the interest of Python code reuse. This choice means that **ExpandDims_3** is our new image input tensor (we leave the usage of the **convert_image** node's input tensor as an exercise for the reader).

Apart from the image embeddings, the LSTM network has other inputs and outputs than the **input_feed** and **softmax** nodes shown in *figure 13.6*. Hidden inside the **lstm** node are its **state_feed** input tensor and its **initial_state** and **state** output tensors.

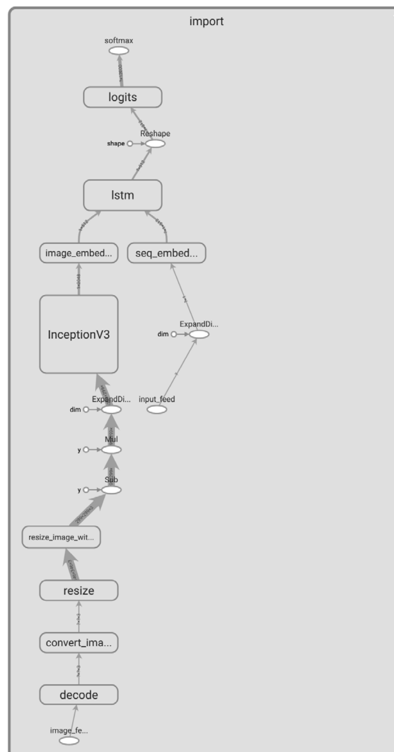

***Figure 13.6**: Overview of the Show and Tell model as shown on tensorboard*

The **initial_state** output tensor, as the name implies, contains the LSTM network's initial state after it is fed the image embeddings calculated by the InceptionV3 network. It is used only once, at the beginning of the inference cycle.

The LSTM network state is fed to the **state_feed** input tensor at each inference step. The **input_feed** tensor accepts the sequence of words predicted in every previous inference step.

The new state for the following step is provided through the **state** output tensor, and the inferred word probabilities are provided through the **softmax** output tensor.

Converting the frozen model to TensorFlow Lite

Now that the model has been frozen, we can use the provided Python program **tflite_convert.py** to produce the TensorFlow Lite model files.

Important: It is mandatory to use TensorFlow 2 to convert the Show and Tell model to TensorFlow Lite. The TensorFlow 1 converter does not read this frozen model correctly.

The Show and Tell model can be seen as two separate models, so we produced two TensorFlow Lite model files. One deals with extracting embeddings from the input image to produce the LSTM network's initial state, and the other runs the inference steps necessary to generate an image description.

The conversion program can be executed from its location, as follows:

```
1. python tflite_convert.py
```

It stores the two TensorFlow Lite model files—**imagenet.tflite** and **lstm.tflite**—inside the **model/** directory. Although they are already quantized, they are still pretty big: the former is about 49 MB, and the latter is about 14 MB.

The TensorFlow Lite ImageNet model is quantized down to 16-bit floating-point weights because it becomes unstable with integer quantization. This problem does not afflict the LSTM model, which benefits from the size reduction associated with integer quantization.

Testing the TensorFlow Lite models

A Python implementation of inference using the TensorFlow Lite models is also provided, so we can validate the correct operation of the converted models on the computer. This inference program is called **tflite_inference.py** and can be run from its location, as follows:

```
1. python tflite_inference.py
```

Once the inference is complete, it provides an output similar to the following:

```
1. Captions for image test.jpg:
```

```
2.    0) a young boy wearing a hat and tie . (p=0.000059)
3.    1) a man wearing a hat and a tie . (p=0.000050)
4.    2) a young boy wearing a hat and a tie . (p=0.000027)
```

The output is not the same as the original model, but it is close enough to verify that the TensorFlow Lite model operates correctly.

Remember that the TensorFlow Lite model files are quantized, that is, the original 32-bit floating-point weights' precision has been reduced to 16-bit floating-point or integer precision, depending on the file. This precision reduction means some differences are expected, but they are not large enough to compromise the overall model accuracy.

Now that we confirmed that the converted models are working as expected, we are ready to incorporate them into an Android application.

Implementing automatic image captioning in Android

Taking inspiration from one of the possible applications for a machine learning model such as this, our demonstration application creates automatic descriptions of images captured by the device's camera and saves them, embedded in the JPEG file as metadata.

Repurposing the image capture application

This new application does not have many differences compared to the image capture application or even the Anime application built in *Chapter 12*, *Image Processing with Generative Adversarial Networks*.

Therefore, we have chosen to reuse the base of the Anime application from *Chapter 12*, *Image Processing with Generative Adversarial Networks*. An identical application can be created from scratch, or the existing code can be copied and refactored as a new application. Should you opt for the latter, all folders and Kotlin packages need to be renamed, and the old model files must be removed alongside the corresponding processing code. We called this new application Photo Caption.

Incorporating the TensorFlow Lite model

The following files need to be copied into the Android application's assets directory: **`imagenet.tflite`**, **`lstm.tflite`**. Once they are in place, we can start writing the code to use them. It is implemented as the **`ShowAndTellModel`** class, which has the familiar signature shown in the following code excerpt:

```
1.  internal class ShowAndTellModel(context: Context): AutoCloseable {
2.
3. }
```

The models are loaded in the now usual way. Note that the TensorFlow Lite interpreters are configured to use the Android **Neural Network API** (**NNAPI**) when available. On the author's device, the options shown in the following code snippet represent 50% lower inference times as compared to the default values:

```
1.  private val imageNet: Interpreter
2.  private val ltsm: Interpreter
3.  private val imageNetModelFile: ByteBuffer
4.  private val ltsmModelFile: ByteBuffer
5.
6.  init {
7.      val options = Interpreter.Options().apply {
8.          setNumThreads(4)
9.          setUseNNAPI(true)
10.     }
11.     imageNetModelFile = FileUtil.loadMappedFile(
12.         context, "imagenet.tflite")
13.     imageNet = Interpreter(imageNetModelFile, options)
14.     ltsmModelFile = FileUtil.loadMappedFile(
15.         context, "ltsm.tflite")
16.     ltsm = Interpreter(ltsmModelFile, options)
17. }
18.
19. override fun close() {
20.     imageNet.close()
21.     ltsm.close()
22. }
```

Of course, not all devices support the NNAPI, and some may even be incompatible with these models. So, it is necessary to test it as extensively as possible before releasing an application to the general public that uses any acceleration.

The following code snippet shows how the initial LSTM network state is obtained from an input image:

```
1.  fun feedImage(bitmap: Bitmap): FloatArray {
2.      val outputBuffer = imageNet.allocateOutputBuffer(
3.          LTSM_INITIAL_STATE_OUTPUT)
4.      imageNet.runForMultipleInputsOutputs(
5.          arrayOf(loadImage(bitmap)),
6.          mapOf(imageNet.getOutputIndex(LTSM_INITIAL_STATE_OUTPUT)
7.                  to outputBuffer)
8.      )
9.      return outputBuffer.array()
10. }
```

The **`feedImage()`** function works just like the equivalent functions in other models we have used throughout this book. The main difference is that it calls the interpreter's **`runForMultipleInputsOutputs()`** function, so the input and output tensors are set explicitly in the call.

The **`loadImage()`** function is not shown for brevity. It is identical to the other image loading functions, resizing the input image to 299 x 299 pixels and normalizing it into the [-1, 1] interval.

The following section, *Implementing beam search*, shows that the process for generating the image's description is iterative and requires running inference multiple times with the LSTM network. The following code excerpt demonstrates how the LSTM network can be called to run one inference step:

```
1.  fun inferenceStep(inputFeed: LongArray, stateFeed: Array<FloatArray>)
2.          : InferenceResults {
3.      ltsm.resizeInput(ltsm.getInputIndex(LTSM_INPUT_FEED),
4.          intArrayOf(inputFeed.size))
5.      ltsm.resizeInput(ltsm.getInputIndex(LTSM_STATE_FEED),
6.          intArrayOf(stateFeed.size, stateFeed[0].size))
7.      val softmaxBuffer = ltsm.allocateOutputBuffer(inputFeed.size,
8.          LTSM_SOFTMAX_OUTPUT)
9.      val stateBuffer = ltsm.allocateOutputBuffer(inputFeed.size,
10.         LTSM_STATE_OUTPUT)
```

```
11.     ltsm.runForMultipleInputsOutputs(
12.         arrayOf(inputFeed, stateFeed),
13.         mapOf(
14.         ltsm.getOutputIndex(LTSM_SOFTMAX_OUTPUT) to softmaxBuffer,
15.             ltsm.getOutputIndex(LTSM_STATE_OUTPUT) to stateBuffer
16.         )
17.     )
18.     return InferenceResults(softmaxBuffer, stateBuffer)
19. }
```

Since each step processes sentences with different lengths and number of network state buffers, the first task that the **inferenceStep()** function needs to accomplish is resizing the interpreter's input buffers to accommodate its input parameter's data. The remainder of the function is a regular call to the TensorFlow Lite interpreter with multiple parameters.

The function returns an instance of the **InferenceResults** class containing the word probabilities array and the new network state. This class is shown in the following code snippet:

```
1.  data class InferenceResults(val softmax: Array<FloatArray>,
2.                                val state: Array<FloatArray>)
```

Implementing beam search

The LSTM network predicts the probability of each word of its vocabulary following the previously predicted word at each inference step. So, each step for an n-word vocabulary yields n predictions. Each word prediction represents a new possible sentence at each step.

For example, let's say that the first inference step reports that the most probable words are ("A", "One"). The second step might report that ("person", "man") could follow "A" and ("person", "woman") could follow "One." We now have a solution space of ("A person," "A man," "One person," "One woman"). The following step would predict more words, causing the solution space to grow exponentially. At the same time, we can visualize this progression as a graph, where each word is a node leading to several different words.

Beam Search is a search algorithm that explores a graph by traveling through the most promising node in a limited set. It orders all partial solutions according to a heuristic, keeping only the most promising ones. So, it is a good choice for searching the solution space of the LSTM network and is the one used by the Show and Tell

model authors. The implementation orders sentences based on the sum of each prediction probability's natural logarithm.

Processing the vocabulary file

The **word_counts.txt** file needs to be copied into the Android application's assets directory. It is located in the **im2txt/data/Hugh** directory from the **im2txt** repository and is used to translate the LSTM network's output to English because the LSTM network only works with word indices.

The following code excerpt contains the class responsible for loading the contents of the **word_counts.txt** file. Note that it associates each word from the **word_counts.txt** file with an index value corresponding to the word's line number in the file. Once this task is complete, it ensures that the special start, end, and unknown word markers are part of the vocabulary map. The start word marker is important to inform the LSTM network that a new sentence is being started. On the other hand, the end marker informs our code that the network has completed a sentence.

```
1.  class Vocabulary(context: Context) {
2.
3.      val idToWord: Map<Int, String>
4.      val startId : Int
5.      val endId : Int
6.
7.      init {
8.          val words = loadSingleColumnTextFile(context,
9.              "word_counts.txt", Charsets.UTF_8)
10.             .map { it.split(" ")[0] }
11.
12.         check(words.containsAll(listOf(START_WORD, END_WORD)))
13.             { "Incomplete vocabulary" }
14.
15.         val wordToId = HashMap<String, Int>(words.size)
16.         idToWord = HashMap(words.size)
17.
18.         var index = 0
19.         for (word in words) {
20.             wordToId[word] = index
```

```
21.             idToWord[index] = word
22.             index++
23.         }
24.
25.         if (!wordToId.containsKey(UNK_WORD)) {
26.             wordToId[UNK_WORD] = index
27.             idToWord[index] = UNK_WORD
28.         }
29.
30.         startId = wordToId[START_WORD]!!
31.         endId = wordToId[END_WORD]!!
32.     }
33.
34.     companion object {
35.         private const val START_WORD = "<S>"
36.         private const val END_WORD = "</S>"
37.         private const val UNK_WORD = "<UNK>"
38.     }
39. }
```

Implementing the supporting data structures

We need to keep track of each sentence generated, so a data structure needs to be defined to capture the current sentence state as it is built. The following code excerpt shows the **Caption** class definition:

```
1.  data class Caption(val sentence: List<Long>,
2.                    val state: FloatArray, val logprob: Double)
3.      : Comparable<Caption> {
4.
5.      override fun compareTo(other: Caption): Int =
6.          when {
7.              this.logprob == other.logprob -> 0
8.              this.logprob < other.logprob  -> -1
9.              else -> 1
```

```
10.         }
11.
12. }
```

The `Caption` class contains the words composing the sentence, the confidence associated with the sentence, and the LSTM network state. This LSTM network state needs to be fed back to the network to continue inference and predict the next word in the sentence. It can, of course, be the network's initial state as well.

It is necessary to select the best sentences, in other words, the sentences with the highest confidence, so the `Caption` class is also comparable according to confidence.

Another supporting data structure is a priority queue that contains a limited number of sentences. When the maximum number of sentences is reached, it drops the sentence with the lowest confidence each time a new sentence needs to be stored. This queue ensures that only the sentences the model is more confident about are processed. The `TopN` class implements this queue and is depicted in the following code snippet:

```
1.  internal class TopN<T: Comparable<T>>(private val n: Int) {
2.
3.     private val queue = PriorityQueue<T>(n)
4.
5.     fun push(element: T) {
6.         queue.offer(element)
7.         if (queue.size == n) {
8.             queue.poll()
9.         }
10.    }
11.
12.    fun extract(sorted: Boolean = false) =
13.        queue.toList()
14.            .let { if (sorted) it.sorted() else it }
15.            .also { queue.clear() }
16.
17.    fun isEmpty() =
18.        queue.size == 0
19. }
```

Running the beam search

The **Captioning** class implements the beam search algorithm. Only the actual search is shown and described here for brevity. The full implementation is available in the accompanying code.

The algorithm chooses the most probable words according to their confidence values. Sentences are classified by their confidence scores, which are calculated by summing the natural logarithms of each word's confidence values.

As a general overview, the search algorithm comprises the following steps:

1. Obtain the LSTM network's initial state by feeding it the image to be described.
2. Maintain a list of partial captions (sentences) and a list of complete captions. Complete captions are finalized sentences, whereas partial captions are parts of the solution space.
3. Run an inference step to obtain the predictions for the current list of partial captions.
4. Select the n most probable words for each partial caption, where n is the algorithm's beam size.
5. Append the predicted words to each caption, storing them in the correct list (partial or complete).
6. Keep only the k better sentences.
7. Repeat steps 3 to 6 until the maximum caption length is reached or until there are no more partial sentences.

The following code excerpt shows the implementation used in the Photo Caption Android application:

```
1.  val initialState = model.feedImage(image)
2.
3. partialCaptions.push(Caption(
4.     listOf(vocabulary.startId.toLong()), initialState, 0.0
5. ))
6.
7.  for (l in 0 until MAX_CAPTION_LENGTH) {
8.   val partialList = partialCaptions.extract()
9.   val inferred = inference(partialList)
```

```
10.   for ((partialIndex, partialCaption) in partialList.withIndex())
    {
11.     val wordProbabilities = findTopBeamValues(
12.       inferred.softmax[partialIndex])
13.     for (wordProbability in wordProbabilities) {
14.       if (wordProbability.value < 1e-12) continue // avoid ln(0)
15.         val caption = Caption(
16.          partialCaption.sentence + wordProbability.index.toLong(),
17.           inferred.state[partialIndex],
18.           partialCaption.logprob + ln(wordProbability.value.
              toDouble())
19.         )
20.         captions { wordProbability.index == vocabulary.endId }
21.           .push(caption)
22.       }
23.   }
24.   if (partialCaptions.isEmpty()) break
25.}
26.
27. return captionsToStrings(
28.   captions { !completeCaptions.isEmpty() }
29.)
```

Note that a sentence is considered complete if the code of its last word corresponds to the end marker. The LSTM network produces this marker like any other word; thus, it works like a period.

Saving metadata in JPEG image files

Our demonstration application saves the image description in the JPEG file. This requirement makes it necessary to populate the JPEG file's EXIF metadata structure while saving the file to the device's gallery.

As demonstrated in *Chapter 12, Image Processing with Generative Adversarial Networks*, saving the file to the device's gallery implies writing its contents to an **`OutputStream`**. Unfortunately, there is no direct support in the Android SDK for this operation. So,

we use the Apache Commons Imaging library to save the JPEG-encoded image data with EXIF metadata to the **ContentResolver**'s **OutputStream**.

The Apache Commons Imaging library can be added to our application simply by including the following entry in the application's module Gradle file:

```
1. implementation 'org.apache.commons:commons-imaging:1.0-alpha2'
```

The contents of the **ImageProxy** containing the captured image are already JPEG-encoded, so the image can be saved simply by creating the metadata entries and copying the **ImageProxy** buffer to the file. The following code excerpt shows how this is implemented in the Photo Caption application:

```
1.  private fun saveWithComment(image: ImageProxy, comment: String,
2.                              output: OutputStream) {
3.   val outputSet = TiffOutputSet()
4.        outputSet.orCreateExifDirectory.add(EXIF_TAG_USER_COMMENT,
           comment)
5.        outputSet.rootDirectory.add(TIFF_TAG_ORIENTATION,
           orientation(image))
6.   ExifRewriter().updateExifMetadataLossless(
7.     image.jpegData,
8.     output,
9.     outputSet
10.  )
11.}
12.
13. private fun orientation(image: ImageProxy) =
14.  when (image.imageInfo.rotationDegrees) {
15.      90 -> ORIENTATION_VALUE_ROTATE_90_CW
16.     180 -> ORIENTATION_VALUE_ROTATE_180
17.     270 -> ORIENTATION_VALUE_ROTATE_270_CW
18.    else -> ORIENTATION_VALUE_HORIZONTAL_NORMAL
19.  }.toShort()
20.
21. val ImageProxy.jpegData: ByteArray
22. get() {
```

```
23.   val jpegBuffer = planes[0].buffer
24.   val jpegSize = jpegBuffer.rewind().remaining()
25.   val data = ByteArray(jpegSize)
26.   jpegBuffer.get(data)
27.   return data
28. }
```

Note that the metadata created by the **saveWithComment()** function needs to include the image orientation because the **ImageProxy**'s contents are stored as per the device's sensor orientation. This orientation did not necessarily match the device's orientation when the image was captured, so we must store the rotation value that needs to be applied to the image to ensure that it appears upright when displayed. Applications and libraries reading the JPEG file honor this value and automatically apply the required transformation.

Note: Many Android gallery applications do not support displaying EXIF user comments. Your device's gallery application may not show the image description.

The Photo Caption application displays the captured image alongside its description after processing and saving it to the device's gallery. The **PictureFragment** class received the ability to read JPEG file metadata, as shown in the following code excerpt:

```
1.   override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
2.       super.onViewCreated(view, savedInstanceState)
3.       val imageUri = Uri.parse(args.filePath)
4.
5.       requireContext().contentResolver.openInputStream(imageUri)
6.           .use {
7.               val metadata =
8.                   getMetadata(it, "stream") as JpegImageMetadata
9.               pictureCaption.text = metadata
10.                  .findEXIFValueWithExactMatch(EXIF_TAG_USER_
                         COMMENT)
11.                  .stringValue
12.          }
13.
14.      Glide.with(this)
```

```
15.         .load(imageUri)
16.         .into(this.imageView)
17. }
```

The image represented by the **filePath** fragment argument is read twice. The Apache Commons Imaging library's **getMetadata()** function is first used to read its metadata and retrieve the saved description. Then, the Glide library is used to load it into the fragment's **ImageView** to be displayed.

Putting it all together

The **CameraPreviewFragment** class is first updated to keep a reference to the **Captioning** class, ensuring that the latter is released in its **onDestroy()** function.

Image processing and storage is performed in the **takePicture()** function's **onCaptureSuccess()** callback, like the Anime Camera application. This function's new version selects the most probable image description and saves it to the gallery.

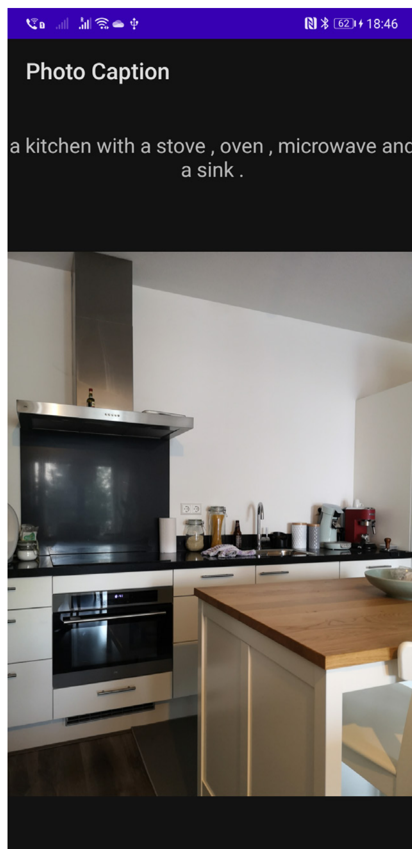

Figure 13.7: *The application describing the author's kitchen*

Once the image is saved to the gallery, the **PictureFragment** shows the image and its description to the user. The following code excerpt shows how these new steps were implemented:

```
1.  val bestCaption = captioning.caption(image).first()
2.
3. saveImageToGallery(requireContext(),
4.     image, bestCaption
5. )?.also { uri ->
6.     requireView().findNavController()
7.         .navigate(
8.             CameraPreviewFragmentDirections
9.                 .actionCameraPreviewFragmentToPictureFragment(
10.                    uri.toString()
11.                )
12.        )
13.}
```

The first sentence is chosen as the best because the **Captioning** class's **caption()** function already returns a sorted list.

Figure 13.7 shows the Photo Caption application describing a picture of the author's kitchen. Note how the Show and Tell model produced an accurate, albeit not totally correct, description—there is no microwave in the picture. The model was probably confused by the black panel between the induction plate and the extractor hood.

You shall find that presenting different pictures to the model may yield surprisingly accurate results. These results can be amusing or even baffling at times.

Conclusion

In this chapter, we saw that it is possible to build neural networks with some memory, allowing previous inference results to influence the following inference steps. Traditional recurrent neural networks possess only short-term memory, whereas long short-term memory networks add some longer-lasting memory, as the name suggests.

An Android application that produces a description of an input image in plain English was implemented. This application has shown how to work with multiple TensorFlow Lite models, particularly with models that require more than one input tensor or offer more than one output tensor.

This book has offered an introduction to the world of artificial intelligence and machine learning, described the most commonly used algorithms, and exemplified how Android applications can be built to take advantage of such technologies. Hopefully, you have the desire to delve deeper into your new machine learning challenge. Good luck!

Index

A

activity
  adding, to application 48-52
activity layout modification
  Android Studio layout editor, using 39
  component identifier, changing 45
  component, placing 42, 43
  ConstraintLayout 41, 42
  performing 38
  string resources, using in components 44
  XML version, of button 45, 46
adversarial attack 130
Android
  cameras, working with 190
  SQLite, implementing 87-92
Android activity lifecycle
  create stage 17
  destroy stage 19, 20
  killed stage 20
  launch stage 17
  pause stage 19
  restart stage 19
  resume stage 19
  running stage 19
  shut down stage 20
  start stage 17, 18
  stop stage 19
Android application
  creating 4, 5
  intents 53
  lifecycle 16, 17
  running 9

running, on Android emulator 9-12
running, on real device 12
SQLite database, using 87
Android application manifest 26-29
Android permissions 29-31
required features, declaring 31, 32
Android application resources 20, 21
identifying, in application code 22, 23
localization 23, 24
other types of resources 25, 26
qualifiers 21, 22
string resources 24, 25
Android camera
image capture 201
intents 190
permissions, requesting 194-196
specialized camera APIs 190
Android emulator
application, running on 9-12
Android Neural Networks API (NNAPI) 238
Android project structure 6-9
Android Studio
setting up 2, 3
simple Android application, creating 4, 5
URL 2
Android Virtual Devices (AVD) 10
AnimeGANv2 project 327
anime-styled pictures, in Android
captured image, transforming 332-337
capturing 330-342
image capture application, repurposing 330, 331
image capturing, into memory 331
modified image, saving in device gallery 337, 338
artificial intelligence
depictions, in literature 114, 115
future 114
history 114-119
working with 119, 120
Artificial Neural Networks (ANN) 125
automatic image captioning
performing 346, 347
automatic image captioning, in Android
beam search, implementing 356
beam search, running 360, 361
image capture application, repurposing 353
implementing 353
metadata, saving in JPEG image files 361-364
supporting data structures, implementing 358, 359
TensorFLow Lite model, incorporating 353-356
vocabulary file, processing 357

B

black box models 129

C

Camera2 API 191
Camera API 190
considerations, while using 191
camera preview
setting up 196-200
CameraX API 192
using 192-194

CameraX use cases
  configuring 248
  image analysis class, creating 248-252
central processing units (CPUs) 128
clustering 124, 125
code view 40
component identifier 45
ConstraintLayout 41
convolution 183
Convolutional Neural Network (CNN) 126, 183, 264, 347

D

deep neural networks 126, 127
  using, for regression 164-168
delete() function 91
design surface selector 40
design view 40
device
  application, running on 12-16
  Chrome OS, configuring 13
  configuring 12, 13
  connecting, to computer 14
  macOS, configuring 13
  Ubuntu Linux, configuring 13
  Windows, configuring 13, 14
dtype parameter 143
dynamic range quantization 212, 270

E

encoder-decoder network 327
Euclidean distance
  using, for face identification 274, 275
Euclidean norm
  using, for normalization 266, 267
Explainable Artificial Intelligence (XAI) 129

F

face detection algorithm 262
face detector object
  creating 252
FaceNet model 267-269
face recognition algorithms 263
face recognition demonstration application
  face detection results, listing 306, 307
  faces, recognizing 305, 306
  implementing 308- 312
  new faces, adding 298-305
  running 312, 313
  user interface, designing 298
Fashion-MNIST dataset
  loading 179, 180
  model, building 180-182
  model, finding 182-185
  preparing 179, 180
float-16 quantization 212
full integer quantization 212
fundamentals, Kotlin language
  classes and objects 84-87
  code organization 73
  data types 73, 74
  functions 77-79
  Lambda expressions 79-81
  nullability 82-84
  packages 72
  type inference 76, 77
  variables and properties 74-76
  visibility 73

G

gather() function 150

Generative Adversarial Network (GAN) 316-318

  converting, to TensorFLow Lite format 327, 328

  example 316

  simple GAN, training 318-326

  trying, in computer 328-330

Google driver package

  reference link 13

Google ML Kit 242, 243

  face detection 243, 244

  including, in Android application 244

  user interface, preparing 246-248

Google ML Kit face detector

  classifications 254

  contour detection 253

  face tracking 254, 255

  landmark detection 253

  minimum face size 254

  performance mode 253

Google ML Kit, in Android application

  metadata, adding for Google Play Services 246

  project's dependencies, configuring 245

  view binding, enabling 244

graphs, TensorFlow 152-154

  best practices 154

H

handwritten digits

  recognizing 170

Hiroki Taniai’s implementation, of FaceNet

  converting, to TensorFlow Lite 269-273

I

identity store

  building 288-291

  view model, implementing 291-294

image analysis

  captured images, cropping 217, 218

  captured YUV images, converting to bitmaps 219-222

  setting up, in Android application 214-217

image analysis class

  creating 248-250

image capture

  CameraX, configuring 201, 202

  captured image, displaying 205, 206

  saving, as JPEG file 202, 203

  trigger button, adding 203-205

image scan, for faces in real-time

  face detection results, using 256-260

  face detector object, creating 252

  image, analyzing 255, 256

  minimum image resolution, configuring 252

  performing 252

image transformation

  into anime-style pictures 326, 327

  special effect, applying to photographs 326

improvement opportunities, machine learning

  bias, avoiding 129, 130

explainability, working towards 129
performance and accuracy, improving 127, 128
system security 130, 131
intents, Android application 53
activity, showing with 54, 55
another application, starting with implicit intent 65-68
creating 54
explicit intent 53
implicit intent 53
intent filters 64
using 63
value, returning from target activity 56-63
Internet of Things (IoT) 16

K

Kotlin language 70-72
fundamentals 72

L

language selector 40
layout constraints
visualizing 41
layout design section
code view 40
design surface selector 40
design view 40
language selector 40
orientation selector 40
preview API version selector 40
preview device selector 40
split view 40
theme selector 40
view options 40
layout editor
sections 39
using 39
linear regression algorithm 123, 124
linspace() function 144
locale 24
Long Short-Term Memory (LSTM) networks 346
evolving into 345

M

machine learning
clustering methods 124, 125
defining 120-122
improvement opportunities 127
linear regression algorithm 123, 124
neural networks 125
machine learning algorithms
reinforcement learning 122
supervised learning 122
unsupervised learning 122
machine learning models
evaluating, with realistic image 186-188
Matplotlib 155
URL 155
MNIST dataset
loading 171, 172
model, building 173-175
model, saving 176
preparing 170, 171
saved model, testing with image loaded from disk 176-178

MobileFaceNets model
incorporating, in Android application 275
working with 273, 274
MobileFaceNets model, in Android application
distance between face embeddings, displaying 283, 284
embeddings, extracting from face images 277-279
face embeddings, obtaining for testing 284-286
new image analyzer, writing for face verification 279-283
user interface, adjusting 275, 276
Model-View-Controller (MVC) pattern
applying 32
dependencies 33
disadvantages 33
Model-View-Presenter (MVP) pattern 33
dependencies 34
disadvantages 34
Model-View-ViewModel (MVVM) pattern 34, 35
disadvantages 35
Modified National Institute of Standards and Technology (MNIST) 170
multiple faces
processing, from camera images 294-298

N

Nash equilibrium 317
neural networks 125
Artificial Neural Networks (ANN) 125
Convolutional Neural Networks (CNN) 126
deep neural networks 126, 127
Recurrent Neural Networks (RNN) 126
Simulated Neural Networks (SNN) 125
normalization 264
Euclidean norm, using for 266, 267
ratio scale, normalizing 264-266
Numpy 155
URL 155

O

Object-Relational Mapping (ORM) 93, 94
advantages 94
onCreate() function 89
onUpgrade() function 89
operations, with tensors 145
linear algebra operations 147
mathematical operations 146
Optical Character Recognition (OCR) system 170
orientation selector 40

P

Pandas 155
URL 155
permissions
requesting, to use camera 194-196
preview API version selector 40
preview device selector 40
Python
alternative Mac OS installation 135, 136

download link 135
installing, on Mac OS 135
installing, on Ubuntu Linux 136, 137
installing, on Windows 135
preparing 135
Python interpreter shell 140

Q

query() function 91

R

range() function 144
receive events
reacting to 47, 48
registering to 46, 47
Recurrent Neural Networks (RNN) 126, 344, 345
reinforcement learning 122
resource identifier (ID) 22
Room-based databases
creating 101, 102
Data Access Objects (DAO), creating 97-100
database migrations, using 106, 107
database views, using 103, 104
data entities, defining 96, 97
object references, converting to database types 104-106
object references, not supported 104
queries, running outside of main thread 108, 109
testing 109-112
working with 94-96
Room library 94

S

Seaborn 155
URL 155
Show and Tell model 347, 348
converting, to TensorFlow Lite 349, 350
frozen model, converting to TensorFlow Lite 352
TensorFlow Lite models, testing 352, 353
visualising, with tensorboard 350-352
simple clothing items
recognizing 179
simple model training, TensorFlow 154, 155
dataset, loading 156-158
dataset, preparing 156-158
deep neural network, using for regression 164-168
linear regression model, training with all feature 163, 164
linear regression model, training with one feature 158-162
Simulated Neural Networks (SNN) 125
split view 40
SQLite 87
consequences 93
implementing, in Android 87-92
SQLite-based databases
testing 92, 93
SQLite database
using, in Android application 87
SQLiteOpenHelper 88
supervised learning 122
synapses 125

T

TensorFlow 134
  download link 138
  eager execution 152
  graphs 152-154
  installing 134
  simple model training 154, 155
  variables 151, 152
TensorFlow 1.15
  installing 138, 139
TensorFlow 2
  installing 138
  requirements 134
TensorFlow documentation
  reference link 145
TensorFlow installation 134
  in virtual environments 138
  Python, preparing 135
  version, verifying 139
  virtual environments, creating 137
TensorFlow Lite 210
  Android Neural Networks API (NNAPI) 238
  automatic code generation, for adding model with metadata 235, 236
  converted model, working with 224-231
  existing model, training 212, 213
  fundamentals 210
  Graphical Processing Units 236, 237
  modules, creating for APIs 222-224
  running, on dedicated hardware 236
  TensorFlow models, converting into 211, 212
  trained existing model, working with 231-235
  URL 210
  using, in application 222
tensor reshaping 149
tensor reversing 149
tensors 140-142
  creating, with constant() function 142, 143
  creating, with generated data 143-145
  operations 145
  transformation functions 148, 149
tensor slicing 148
theme selector 40

U

unsupervised learning 122
update() function 90
user events 38

V

variables, TensorFlow 151, 152
view options 40
virtual Python environment
  creating, for TensorFlow 137
  on Mac OS 138
  on Ubuntu Linux 138
  on Windows 138