

Hands-on TinyML

Harness the power of Machine Learning on the edge devices



Rohan Banerjee



Hands-on TinyML

*Harness the power of Machine Learning
on the edge devices*

Rohan Banerjee



www.bpbonline.com

Copyright © 2023 BPB Online

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

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

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

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-446

www.bpbonline.com

Dedicated to

My Beloved Little Niece:

Arjama

About the Author

Rohan Banerjee is a practicing data scientist having more than 12 years of relevant industry experience. He completed his M.Tech from IIT Kharagpur in 2011. His areas of interest include advanced data science, machine learning, embedded machine learning, digital signal and image processing. Rohan is currently associated with Baker Hughes Company. Before that, he was with TCS Research, Tata Consultancy Services where he published more than 40 technical papers in international conferences, journals, and also contributed in enhancing their intellectual property portfolio. Rohan is an avid reader of contemporary literature, a traveler, and a quiz enthusiast.

About the Reviewers

- **Tushar Chugh** is a machine learning engineer at Google, focusing on search ranking. With a background in robotics from Carnegie Mellon University, he previously contributed to GM's self-driving car perception systems. Passionate about applied machine learning, deep learning, and computer vision, Tushar has experience developing innovative technologies at Qualcomm and Microsoft in ML and tech domains.
- **Yogesh M Iggalore** has over a decade of experience in product development and is actively involved in all the disciplines of product architecture, hardware design, firmware development, testing, and cloud integration. He is currently interested in product development in TinyML.

Acknowledgements

There are many people I want to express my gratitude to. First and foremost, I would like to sincerely thank my family members for their unwavering support and encouragement throughout my journey — I could have never completed this book without their support.

I am grateful to various online resources, blogs, and materials that enriched my learning in order to write the book. I would also like to acknowledge the valuable feedbacks of my colleagues and co-workers during many years working in the tech industry. I am particularly grateful to Mr. Avik Ghose from TCS Research for providing me the opportunity to learn and work on TinyML. I gratefully acknowledge the effort of Mr. Tushar Chugh and Mr. Yogesh M Iggalore for their technical scrutiny and suggestions for improving the quality of this book.

My sincere gratitude also goes to the team at BPB Publication for being supportive enough to provide me quite a long time to finish the book and also for all the valuable editorial reviews.

Finally, I would like to thank all the readers who have taken an interest in the book. Your encouragement has been invaluable.

Preface

TinyML is an emerging trend in machine learning, that aims at deploying complex machine learning and neural network models on low-powered tiny edge devices and microcontrollers. Modern deep learning algorithms are computationally expensive and result in large model size. They are often hosted on dedicated servers having enormous computing resources. As users, we generate the data at our end and send them via the internet to process remotely. Owing to the limitations in network bandwidth, roughly 10% of all our data can be sent over the internet. Processing of the data on the edge can revolutionize the current paradigm. Thanks to TinyML, large machine learning models can be shrunk in order to effectively deploy on smaller devices having few hundred kilobytes of RAM and few megabytes of flash memory. Such devices can operate 24x7 with a minimum power consumption. Moreover, being entirely offline, the applications not only consumes zero network bandwidth, but also preserves user privacy.

TinyML is going to be the next big thing in machine learning. Major tech giants are heavily investing in standardizing the hardware and software stack. In this book, we cover the basic concepts of TinyML through practical coding examples to enable the readers to learn the basic concepts of TinyML and develop their own applications. Rather than discussing every single mathematical concept behind the machine learning algorithms, the book primarily focuses on end-to-end application development through coding examples. The projects covered in this book are implemented in open-source software commonly used in industry and academics.

This book is divided into 10 chapters. The details are listed as follows.

Chapter 1: Introduction to TinyML and its Applications – covers the basic concept of EdgeML and TinyML, their potential applications, and challenges. It briefly covers the hardware and software platforms required to create TinyML. We also discuss the process flow of creating TinyML applications.

Chapter 2: Crash Course on Python and TensorFlow Basics – covers the basics of Python which is now the de facto programming language in machine learning for both research and creating production ready software. We start with the basic concepts of Python along with various libraries such as Numpy, Matplotlib. The later part of the chapter covers the key aspects of TensorFlow. TensorFlow is a free

and open-source software library for machine learning and neural networks. The chapter briefly covers some of the fundamental concepts of TensorFlow through coding examples.

Chapter 3: Gearing with Deep Learning – briefly talks about neural networks. We begin with the concept of a simple Artificial Neural Network (ANN), various activation functions, and backpropagation to learn the weights. Later, we talk about Convolutional Neural Network (CNN), a popular deep neural network architecture used in modern image processing and computer vision applications.

Chapter 4: Experiencing TensorFlow – guides us to develop our first neural network using TensorFlow and Keras. Keras is a set of deep learning APIs in Python, running on top of TensorFlow, providing high level of abstraction in developing large neural networks. We begin with implementing a simple ANN for classification of handwritten digit images. Later, we implement our first CNN architecture.

Chapter 5: Model Optimization Using TensorFlow – talks about how a large TensorFlow model can be effectively compressed in order to deploy on smaller edge devices using TensorFlow Lite. We create a base CNN model using TensorFlow and convert it into the lighter TFLite model. The chapter also covers TensorFlow Model Optimization Toolkit, a software library for optimizing large neural networks for easy deployment and execution. We learn about different model optimization techniques, such as quantization, weight pruning and weight clustering through coding examples using the APIs provided by TensorFlow Model Optimization Toolkit. Finally, we summarize the impact of various optimization techniques on the base CNN in terms of model size and accuracy.

Chapter 6: Deploying My First TinyML Application – guides us to create the first real TinyML application on Raspberry Pi, a commercially available low-powered edge device. We create a neural network for classification of offline images on Raspberry Pi. The chapter covers two important topics, MobileNet and transfer learning. MobileNet is an optimized neural network architecture specially designed for low-powered mobile edge devices. Transfer learning is another interesting concept in machine learning, where we can reuse a pre-trained model on a new

problem. Transfer learning is particularly useful when we do not have sufficient training data to create a model from scratch.

Chapter 7: Deep Dive into Application Deployment – guides us to implement a more practical TinyML application of real-time on-device person identification from live video stream recorded by a camera. The application is again deployed on Raspberry Pi using various open-source software.

Chapter 8: TensorFlow Lite for Microcontrollers – covers the basics of TensorFlow Lite for Microcontrollers, a highly optimized software tool for porting TensorFlow models on low-powered microcontrollers. We implement a simple neural network that modulates the voltage output of a linear potentiometer and successfully deploy it on Arduino Nano 33 BLE Sense, the recommended microcontroller board for creating TinyML applications.

Chapter 9: Keyword Spotting on Microcontrollers – guides us implementing an on-device speech recognition application. Keyword spotting is an important requirement in modern voice assistant services, such as Amazon’s Alexa or Apple’s Siri. In this chapter, we implement a simple keyword spotting application on Arduino. We first implement a basic keyword detection system using TensorFlow to understand the key concepts of audio processing. Later, we implement a real keyword spotting application using Edge Impulse, a free software platform for designing end-to-end TinyML application and deploy on an Arduino device with minimum code writing.

Chapter 10: Conclusion and Further Reading – summarizes our learnings in the book and covers some recent trends in TinyML.

Code Bundle and Coloured Images

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

<https://rebrand.ly/q35pmfm>

The code bundle for the book is also hosted on GitHub at **<https://github.com/bpbpublications/Hands-on-TinyML>**. 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**.

Join our book's Discord space

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

<https://discord.bpbonline.com>



Table of Contents

1. Introduction to TinyML and its Applications.....	1
Introduction.....	1
Structure.....	3
Objectives.....	3
Brief overview of Machine Learning.....	4
<i>Supervised Machine Learning</i>	5
<i>Unsupervised Machine Learning</i>	6
Machine Learning and Deep Learning.....	6
Edge computing and TinyML.....	7
Applications of TinyML.....	9
Hardware for deploying TinyML.....	10
Software for TinyML.....	13
Process flow of creating TinyML applications.....	13
Prerequisites—hardware and software.....	16
Conclusion.....	17
Key facts.....	17
2. Crash Course on Python and TensorFlow Basics.....	19
Introduction.....	19
Structure.....	21
Objectives.....	21
Colab Notebook.....	22
Python variables.....	24
<i>Python strings</i>	25
<i>Lists</i>	26
<i>Tuple</i>	27
<i>Dictionary</i>	28
Conditional and logical operations.....	28
Loops in Python.....	30
Functions in Python.....	30
Python libraries.....	32

<i>NumPy library</i>	32
<i>Random number generation</i>	37
<i>Matplotlib library</i>	39
<i>Pandas library</i>	41
Introduction to TensorFlow.....	42
<i>Tensors and datatypes</i>	44
<i>Differentiation in TensorFlow</i>	46
<i>Graphs and functions in TensorFlow</i>	47
<i>End-to-end Machine Learning algorithm using TensorFlow</i>	48
Conclusion.....	54
Key facts.....	55
Further reading.....	55
3. Gearing with Deep Learning	57
Introduction.....	57
Structure.....	58
Objectives.....	59
Theory of artificial neural networks.....	59
<i>Binary cross entropy loss function</i>	62
<i>Neural network activation functions</i>	63
<i>Sigmoid activation function</i>	64
<i>Tanh activation function</i>	65
<i>ReLU activation function</i>	65
<i>Softmax function</i>	66
<i>Learning the neural network weights—the backpropagation algorithm</i>	66
Introduction to Convolutional Neural Network.....	69
<i>Architecture of a CNN</i>	71
<i>Input layer</i>	71
<i>Convolutional layer</i>	71
<i>Pooling layer</i>	78
<i>Fully connected layer or dense layer</i>	79
<i>Output layer</i>	79
<i>Putting them all together</i>	79
Neural network hyperparameters.....	80
<i>Number of layers</i>	81

<i>Learning rate</i>	81
<i>Dropout</i>	81
<i>Regularization</i>	82
<i>Choice of optimization algorithm</i>	82
<i>Mini-batch size</i>	82
Conclusion	83
Key facts	83
Further reading	84
4. Experiencing TensorFlow	85
Introduction	85
Structure	86
Objectives	86
Keras and TensorFlow	87
Classification of handwritten digits using a feedforward neural network	90
<i>Data processing</i>	92
<i>Model implementation</i>	94
Implementation of a Convolutional Neural Network	97
Evaluation metrics in classification models	105
Conclusion	107
Key facts	108
5. Model Optimization Using TensorFlow	109
Introduction	109
Structure	110
Objectives	111
Experiencing TensorFlow Lite	111
TensorFlow Model Optimization Toolkit	120
<i>Quantization</i>	121
<i>Weight pruning</i>	128
<i>Weight clustering</i>	134
Collaborative optimization	138
Conclusion	143
Key facts	144

6. Deploying My First TinyML Application	145
Introduction.....	145
Structure.....	146
Objectives.....	147
The MobileNet architecture.....	147
<i>Depthwise separable convolution</i>	148
Image classification using MobileNet.....	148
<i>Brief introduction to transfer learning</i>	152
<i>Implementing MobileNet using transfer learning</i>	153
<i>Creating an optimized model for a smaller target device</i>	154
<i>Evaluation of the model on the test set</i>	157
Introduction to Raspberry Pi.....	158
Getting started with the Pi.....	160
<i>Installing the operating system</i>	161
<i>Setting up the Pi</i>	162
<i>Remotely accessing the Pi</i>	164
Deploying the model on Raspberry Pi to make inference.....	165
Conclusion.....	173
Key facts.....	173
7. Deep Dive into Application Deployment	175
Introduction.....	175
Structure.....	177
Objectives.....	177
System requirement.....	178
The face recognition pipeline.....	179
Setting up the Raspberry Pi for face recognition.....	180
<i>The Raspberry Pi camera module</i>	180
<i>Installing the necessary libraries</i>	184
Implementation of the project.....	185
<i>Data collection for training</i>	185
<i>Model training</i>	189
<i>Real-time face recognition</i>	192

Conclusion	196
Key facts.....	197
8. TensorFlow Lite for Microcontrollers	199
Introduction.....	199
Structure.....	201
Objectives.....	202
Arduino Nano 33 BLE Sense	202
<i>Setting up the Arduino Nano</i>	204
First TinyML project on the microcontroller—modulating the potentiometer 209	
<i>Required components</i>	210
<i>Connecting the circuit</i>	211
<i>Read potentiometer to control the brightness of the LED</i>	212
<i>Creating a TensorFlow model to modulate the potentiometer reading</i>	215
<i>Inference on Arduino Nano using TensorFlow Lite for Microcontrollers</i>	222
Conclusion	228
Key facts.....	229
9. Keyword Spotting on Microcontrollers.....	231
Introduction.....	231
Structure.....	233
Objectives.....	233
Working principles of a voice assistant	234
Implementation of a keyword spotting algorithm in Python	235
<i>Audio spectrogram</i>	241
<i>Designing a Convolutional Neural Network model for keyword spotting</i> ..	247
Introduction to Edge Impulse	251
Implementing keyword spotting in Edge Impulse.....	253
Model deployment	264
Conclusion	266
Key facts.....	267

10. Conclusion and Further Reading	269
Introduction.....	269
Structure	270
Objectives.....	270
Brief learning summary.....	271
TinyML best practices	273
AutoML and TinyML	275
Edge ML on smartphones	277
Future of TinyML.....	277
Further reading.....	278
Appendix	281
Index	283

CHAPTER 1

Introduction to TinyML and its Applications

Introduction

The year 2022 brought **Artificial Intelligence (AI)** to a new level of endless possibilities through the applications of a Generative Pre-training Transformer, ChatGPT. ChatGPT is an AI language model that uses advanced machine learning models to generate human-like text. Needless to say that AI and Machine Learning are hot topics in modern technology. We are living in a world where we are using them everywhere in our day-to-day activities, knowingly or unknowingly. Although the two terms, AI and Machine Learning, are often used synonymously, there are subtle differences between them.

Artificial Intelligence is the science of imbuing human-like intelligence in machines via computer programming to make them behave like humans and, therefore, solve real human problems. In short, through AI, a computer system tries to simulate human reasoning using maths and logic. AI can be applied to many different sectors and industries, including but not limited to healthcare industries for suggesting drug dosage, banking and finance sector for identifying suspicious activities, self-driving cars, and so on. Machine Learning is a subset of AI, where a machine is programmed to learn from past experience in order to predict the outcome of a future event without explicitly being programmed for that. The concept is analogous

to the way we all learn. We gather knowledge from various mediums, for example, reading books, guidance and advice from parents and teachers, and from our day-to-day experiences. Based on that knowledge, we can act in a new situation, like writing in an exam. Machine Learning has many sub-fields. Deep learning is a subset of machine learning that simulates the behavior of the human brain through a specially designed architecture called the **Artificial Neural Network (ANN)**. Deep learning can deal with large unstructured data with minimum human interaction, and hence, has gained lots of attention in recent times. In today's world, we are immensely dependent on Machine Learning and deep learning techniques in our daily activities. Our smartphones are loaded with numerous applications directly using machine learning. When you click a photo on your smartphone and upload it on social media, it automatically detects various objects in the photo, the place where it is captured, and even suggests you to tag your friends who are present in the photo. All these happen thanks to some Machine Learning applications such as object detection, geo-locating mapping, and face recognition. Similarly, while searching for news on the internet, we often opt for searching by voice. It falls under speech recognition, another popular application of Machine Learning. When you shop at an e-commerce site, you are often surprised at how the website accurately knows your preference and recommends you accordingly. This also happens because of some Machine Learning algorithms that learn from your past purchase history and recommend you accordingly.

Machine Learning, particularly deep learning algorithms, are computationally expensive and often require a powerful hardware accelerator like a **Graphics Processing Unit (GPU)** to operate. Such applications typically run on large computers and dedicated data centers. However, the data is generated by the users, on their personal devices like smartphones. In the traditional approach, user data is sent to a dedicated remote server machine via the internet for running the machine learning jobs. However, is this practically feasible? We generate gigabytes of data every day. Is it practically possible to send all these data to a remote system? That would consume enormous network bandwidth. What about the network delay? Recently, there has been a trend called Edge ML that aims at shrinking the machine learning models to run them on edge devices like our smartphones.

In this book, we are going to introduce TinyML, which takes Edge ML one step further and allows it to run machine learning algorithms even on the smallest microcontrollers. It is a subset of applied machine learning that fits large machine-learning and deep learning models to tiny embedded systems running on microcontrollers or other ultra-low power processors. Technically, embedded systems need to be powered by less than 1 milliwatt so that they can run for months, or even years, without needing

to replace batteries. TinyML is one of the hottest trends in the field of embedded computing. Research suggests that global shipments of TinyML devices will reach 2.5 billion by 2030. Several tech-giants are currently working on chips and frameworks that can be used to build more systematized devices in order to standardize the field. TinyML is expected to cause ground-breaking advancements in complex machine learning tasks to solve our day-to-day problems.

Structure

In this chapter, we will discuss the following topics:

- Brief overview of Machine Learning
 - Supervised Machine Learning
 - Unsupervised Machine Learning
- Machine Learning and Deep Learning
- Edge computing and TinyML
- Applications of TinyML
- Hardware for deploying TinyML
- Software for TinyML
- Process flow of creating TinyML applications
- Prerequisites—hardware and software

Objectives

TinyML is a subfield of modern Machine Learning that aims at compressing large Machine Learning models to deploy them on low-powered, low footprint, resource-constrained edge devices and microcontrollers. Though it sounds amazing, deploying a large Machine Learning model on a smaller edge device is not easy. A reduction in model size often comes with a degradation in performance. Hence, rather than compression, the main focus is on optimizing a model for a target device.

This book is intended to cover the fundamentals of TinyML through practical projects so that the readers can have an in-depth idea of how TinyML works with some hands-on experience. The primary objective of this book is to make you familiar with TinyML programming using open-source software packages so that you can create your own TinyML projects from scratch. Rather than detailing the underlying complex mathematics involved in machine learning and deep neural network algorithms, our key focus is to learn the programming aspects using practical

examples. However, interested readers are encouraged to learn the mathematical aspects from various available resources for a better understanding of how various machine learning algorithms were actually derived.

In this introductory chapter, we will briefly cover the fundamentals of TinyML. We will begin with the key aspects of machine learning and deep learning. Then, we will talk more about TinyML as a technology, its applications, and the hardware and software recommended to create real TinyML projects.

Brief overview of Machine Learning

Before starting with TinyML, we should have some fundamental concepts of machine learning. Machine Learning is a branch of artificial intelligence that focuses on developing computer algorithms based on data to imitate the human learning process. Now, the question arises: where do we need machine learning?

Suppose you have measured the temperature of the day as 25 degrees in the Celsius scale using a thermometer and wish to convert it to the Fahrenheit scale. There is a well-known formula for doing the conversion, which is given by: $\frac{C}{5} = \frac{F-32}{9}$. You

can simply put $C = 25$ in the equation and get F as 77. Here, you have both the data and the rule that relates to the data. However, the situation is quite different in real-life applications where you have data, but you often do not have a known mathematical formula to relate them. For example, suppose you want to predict the price of a house in a suburban locality in Delhi. What would you typically do? There is no known mathematical formula to solve the problem. Machine learning can help us to do so. Machine learning is all about data. If we have the right amount of data, it can help us to find suitable relations between them. In order to predict the price of a new house, you first need to gather certain information for a few other houses in the same locality to empirically estimate the price of a new house. For example, you could collect the area of those houses, the number of rooms, the distance of the properties from the main road, and so on. You also need to collect the current prices of those houses. Here, the price of the house can be considered as a dependent variable, which is determined by the independent variables such as the area of the house, number of rooms, distance from the main road, and so on. The dependent variables are also called the target values or labels in some cases, and the independent variables are called as features. With machine learning, we can build a model to find the relationship between the dependent and the independent variables. The resulting model can predict the price of another house if the features are provided as input.

Similarly, suppose a pharmaceutical company is planning to launch a new blood pressure-controlling medicine. Before that, they want to investigate the impact of that medicine on people. If the recorded stable blood pressures after the intake of certain dosages of the medicine are experimentally noted on a diverse group of people, a machine learning model can be created to predict what should be the ideal medicine dose for a patient having a certain range of blood pressure.

Machine learning approaches primarily fall into two categories, **supervised** and **unsupervised** machine learning.

Supervised Machine Learning

Supervised machine learning approaches take both features and corresponding targets or labels as input to create a model which can be used to predict the target value of a new unseen example data using the features. Supervised machine learning algorithms are commonly used for classification and regression. In classification, a machine learning model is designed to predict a discrete class label from the features, for example, predicting the presence of a cat or a dog in an image, or identifying numerical digits from handwritten expressions. In regression, a machine learning model predicts a continuous value, for example, predicting the price of an asset or predicting the salary of a person. *Figure 1.1* provides a basic block diagram showing various components of the supervised learning approach:

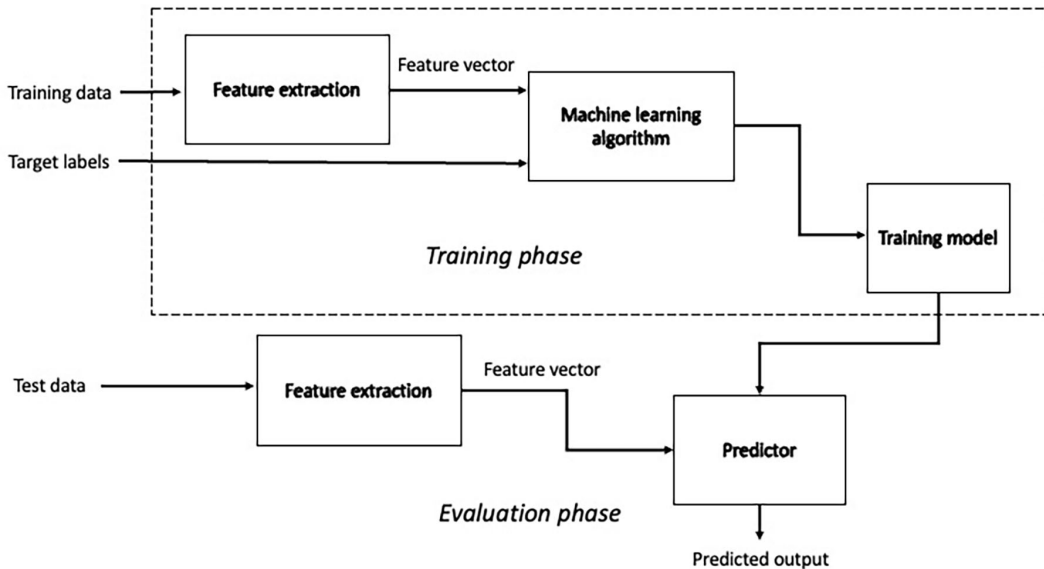


Figure 1.1: Block diagram of supervised learning approach

Supervised learning involves two phases, training and evaluation. During training, it takes labeled training data as input and tries to create a mathematical relationship between them by adjusting some parameters, which are called the model weights. Once the training is done, the model can be used for the prediction of unseen test data. Feature extraction is a very important step in machine learning. The relevant set of information extracted from the input that directly determines the target values is called as features. In the previous example of house pricing, the number of rooms or area of the house can be considered as features. Similarly, for a classification problem, if you are given labeled images of cats and dogs as input, color of the animal in the image, its facial structure, the presence or absence of whiskers, and so on could be the relevant feature to create the classifier.

Training of a supervised learning algorithm has the following three basic components:

- A **decision process** that makes a guess of the target values from the input features.
- An **error function** that finds how good the guess is with respect to the actual target values or labels.
- An **optimization process** that iteratively adjusts the decision process via modifying the model weights to reduce the error between the guessed and the actual target values.

Linear regression, logistic regression, support vector machine, and Artificial Neural Networks (ANN) are popular examples of supervised learning algorithms.

Unsupervised Machine Learning

Unsupervised machine learning algorithms deal with unlabeled data. That means you only have the features but not the labels. Such algorithms try to find the hidden patterns of the input data based on the features and group them together to form clusters. All data in a particular cluster share similar properties. Unsupervised learning is typically used in applications such as customer segmentation, similarity detection, product recommendation, data dimensionality reduction, and so on, where you really do not know the target labels. A few examples of unsupervised learning algorithms are principal component analysis and K-means clustering.

Machine Learning and Deep Learning

Deep learning is the newest yet most popular branch of machine learning that works particularly well on unstructured data. Deep learning algorithms can be considered as mathematical evolution of traditional Machine Learning algorithms. Refer to the

basic block diagram of supervised learning in *figure 1.1*. A machine learning algorithm cannot learn from raw unstructured data. It first needs to extract a set of relevant features, and the features are then used to train the model. Feature extraction is a manual process in traditional machine learning. Finding the optimum feature set is probably one of the most difficult tasks, which might require domain expertise in the field application. Deep learning algorithms can directly take raw data as input and can extract the relevant features automatically, therefore, bypassing the need for manual feature extraction. Deep learning techniques are particularly useful to process unstructured data (for example, text and images).

Deep learning approaches analyze data in a way similar to the human brain. They have a layered structure of **Artificial Neural Network (ANN)**, which is inspired by the biological nervous system. An ANN comprises of neurons or nodes in a layered structure where each layer is connected to another layer to analyze complex patterns and relationships in data. A typical neural network requires huge training data but minimum human intervention to function. The ability of deep learning algorithms to work with minimum human intervention makes them extremely popular in modern data science in diverse practical applications such as computer vision, speech recognition, natural language processing, and so on. In this book, we will heavily use deep neural networks in various projects, primarily the **Convolutional Neural Network (CNN)**.

Edge computing and TinyML

Machine learning, more specifically deep learning algorithms, are computationally expensive. In reality, it may take from several hours to several days to train a large neural network model using sophisticated hardware acceleration platforms such as **Graphics Processing Unit (GPU)** or **Tensor Processing Unit (TPU)**. Such hardware platforms are maintained by large enterprises at large distributed data centers. As individual users, we generate data at our end in our personal edge devices, like smartphones or tablets, in the form of text, audio, video, or image. However, the devices we possess are not always capable of running complex machine learning models for an application. Machine Learning operations are traditionally performed on the cloud. Users' data is typically sent to the backend data center that hosts the machine learning model via the internet for processing, and the result is transferred back to our device. For effective data management and processing, all our devices are connected to the internet to create an ecosystem called the **Internet of Things (IoT)**. The interconnection between traditional machine learning and IoT is no doubt effective as we get all our jobs done seamlessly. However, it has its own drawbacks. A few key challenges are listed as follows:

- **Data privacy and security:** In traditional machine learning, IoT devices send their data to a cloud network for processing. This is prone to cyber-attacks, and hence, has severe security and privacy issues.
- **Power consumption:** Machine learning models consume enormous power. A research team at the University of Massachusetts estimated in 2019 that deep learning sessions of a machine learning model could generate up to 626,155 pounds of CO₂ emission, which is roughly equal to the carbon emission of five cars over their lifetime.
- **Network bandwidth and latency:** It would require an infinite bandwidth to support hundreds and thousands of IoT devices, to continuously stream their data to the cloud for processing. Another key aspect is of network latency. Latency is termed as the time lag in sending and receiving the data between an IoT device and the server over the network. In slower networks, the latency is higher, and the user often needs to wait for a long period of time to get a response from the server. It is undesirable for user engagement in real-time applications.

Edge AI and Edge ML have emerged as the next frontier of development for IoT systems. In Edge AI, data is produced, handled, and processed locally. Instead of sending to the cloud, the analytics happens in the edge device, such as smartphones, single board computers, IoT devices, or edge servers. Real-time processing allows a faster response and reduced latency and bandwidth use. Applications of Edge AI can be seen in object detection, speech recognition, fingerprint detection, autonomous driving, and so on.

Tiny machine learning, commonly known as TinyML, takes Edge AI one step further in order to run machine learning algorithms even on the smallest microcontrollers with the least amount of power possible. TinyML is a rapidly growing field in machine learning. Instead of GPUs or microprocessors for computation, TinyML entirely relies on less capable processing units that consume very less power, typically in the range of a few milliwatts. Such processors are frequently Cortex-M based, having only a few hundred kilobytes of **Random Access Memory (RAM)**, a few megabytes of flash memory, and clock rates in the tens of megahertz. Therefore, TinyML applications ensure low power consumption, low latency in running a machine learning model. They also ensure to preserve user privacy as the data is entirely being processed on the edge device.

Applications of TinyML

TinyML applications are extremely energy efficient. A standard **Central Processing Unit (CPU)** consumes around 70–85 Watts, and a GPU consumes up to 500 Watts of power to operate. On the other hand, TinyML models operate on microcontrollers that consume only a few milliwatts or microwatts. Such devices are intended to run for several weeks or even months without recharging or changing of the batteries. This brings down the overall carbon footprint. Processing at the edge also ensures low latency and improved data privacy. These devices are relatively basic in terms of computation hardware, making them available at a cheaper price. TinyML is successfully applied in various practical applications across industries, explained as follows:

- **Predictive maintenance:** Large industrial machines are prone to making faults. Predicting a fault of a machine ahead of time is important in any industry to avoid a potential shutdown. Under normal health conditions, most machines exhibit some standard properties in terms of mechanical noise, vibration, torque, and so on. A deviation from the normal range can be an alarm for the potential fault of the machine in the near future. Continuous monitoring of the machine is possible by gathering relevant information on various properties by installing sensors on the body of the machine for analysis. Such analysis is typically done 24×7 using small microcontrollers with minimum power consumption, as the frequent replacement of the device battery is impractical.
- **Healthcare:** TinyML is bringing in affordable solutions in early disease screening and medical diagnostics, which can be used in developing nations to supplement limited healthcare facilities. Off-the-shelf electronic devices in the form of wristbands or smart watches are readily available that use TinyML algorithms to measure physiological parameters like heart rate and blood pressure. Such devices can also predict abnormal heart rhythms like atrial fibrillation, which can be an early sign of a heart attack.
- **Agriculture:** There are mobile phone applications for assisting farmers to detect diseases in plants just by taking a few pictures of the diseased plants to run on-device machine learning algorithms for analysis. As the applications do not need images to send to the cloud, they can help the farmers in remote areas where stable internet connectivity remains an issue.
- **Voice-assisted devices:** Voice-assisted devices such as Amazon Echo, Google Home, or iPhone's Siri have become very popular these days. Such devices listen to your voice command and can act accordingly, such as playing your favorite music, turning ON/OFF the room light, and so on. This is a

perfect example of where TinyML and traditional machine learning work together. The microphone of the voice-assisted device continuously analyses the background sound to detect a wake-up keyword such as “OK Google!”, “Alexa!”, or “Hey Siri!”. The keyword detection process has to be extremely light-weight, on-device, and low-powered. This is where TinyML is deployed. As soon as the keyword is detected, the device wakes up and records your following voice instruction like “*What’s the weather going to be like today?*” or “*Play my favorite music.*”, which is sent to the cloud for processing via more powerful natural language processing algorithms which are not possible to run at the edge.

- **Ocean life conservation:** TinyML applications are used for real-time monitoring of whales in North America to avoid whale strikes in busy shipping lanes.

Hardware for deploying TinyML

A complex deep learning model can have several thousand to millions of trainable parameters, resulting in a large model size in the range of several megabytes to gigabytes. In general, model size is not a big issue when machine learning applications are deployed on a remote server that virtually has infinite memory space for storage. However, the scenario is different in TinyML, having a few hundred kilobytes of RAM.

Deciding the correct hardware for deploying TinyML models is often challenging. You have to keep several factors in mind, for example, the device form factor for your application, how much memory storage your model requires, the maximum allowable power consumption by your application, what sensors you might require for the collection of data and their interfaces, whether you require on-device training, the approximate price of your application, and so on. Our smartphones and tablets are great examples of edge devices. The past few years have witnessed a rapid proliferation of smartphones. Modern smartphones are rich in computing resources and in-built sensors. You can even train medium size neural networks on them. Smartphones and tablets are a great choice to run Edge ML applications that involve a strong user interface, for example, on-device face recognition for person identification, high-definition videography, gaming, natural language processing, and so on.

A **Single Board Computer (SBC)** is another popular device for edge computing in IoT-based applications. An SBC is a small portable computing device built on a single printed circuit board with a microprocessor, memory, and **input/output (I/O)** devices. Although an SBC has much smaller memory and lesser powerful processor

than a personal computer, it comes at a much cheaper price and drives significantly low power to operate. An SBC can draw its required power to operate from a power bank or the USB port of a computer. SBCs can easily interface with external sensors like servo motors and ultrasound sensors. They are typically used in academic projects and industrial applications where edge devices of small form factors are required to be directly connected to external devices for data collection and analysis. *Figure 1.2* shows a picture of a Raspberry Pi device, a popular SBC used in various commercial applications and academic projects. The device is powerful enough to run optimized deep learning models.



Figure 1.2: Raspberry Pi 3, Model B+, a popular single board computer

When we think of deploying extremely low-profile TinyML applications to operate 24×7 , the primary target hardware are the microcontrollers. A microcontroller is a compact **Integrated Circuit (IC)** designed to perform a specific task in an embedded platform. They are much smaller in size than a smartphone or an SBC and have much lesser computing resources. However, they are extremely low-powered. A microcontroller primarily contains a CPU that connects all other components in a single system. The CPU fetches data, decodes, and executes the assigned task. The CPU clock speed typically ranges between 16 megahertz and 64 megahertz in a microcontroller. They have a small amount of computation memory along with a certain amount of **Read Only Memory (ROM)** or flash memory for the storage of data and programs. The typical RAM size is 64 to 256 kilobytes, and the flash memory can be of 2 megabytes. There are several I/O ports to communicate with external devices. Microcontrollers may also contain one or more in-built timers and counters, **Analog to Digital Converter (ADC)** and **Digital to Analog Converter (DAC)**, to read data from external sensors. Microcontrollers can be divided into various categories depending on the underlying architecture, memory, and instruction sets. They are used in applications such as machine health monitoring, space research, autonomous cars, and so on, which need to operate for a prolonged duration without frequently replacing the battery.

Microcontrollers are significantly different than a computer system. A computer system is designed to perform multiple different tasks concurrently, whereas a microcontroller is specifically designed for one particular application, such as turning ON/OFF an LED, rotating a servo motor, or controlling a robotic arm. Microcontrollers have a much-constrained hardware environment. The CPU clock speed of a powerful microcontroller can be up to 64 megahertz, with 256 kilobytes of RAM and only 1–2 megabytes of flash memory. On the other hand, modern computer systems come with several gigahertz of CPU clock speed, 8–16 gigabytes of RAM and terabytes of storage area. Microcontrollers do not have an operating system. They draw much smaller power compared to a computer, which is in the range of milliwatts or microwatts. Hence, they can operate for several weeks without recharging or replacing the battery, making them extremely popular for continuous operation in edge computing applications. *Figure 1.3* shows few of commercially available microcontroller units popularly used in TinyML applications:



Arduino Nano 33 BLE Sense



SparkFun Edge



Raspberry Pi Pico



ESP-32-S3

Figure 1.3: Popular microcontrollers for TinyML applications

Software for TinyML

Apart from hardware, there are certain software libraries and platforms we heavily rely on creating TinyML applications, from model optimization to on-device inferences. The optimization algorithms aim at compressing a base machine learning model in such a way that ensures a minimum degradation in performance. **TensorFlow** is Google's open-source machine learning framework for easy development of machine learning models. **TensorFlow Lite** is a mobile library for optimizing and deploying large TensorFlow models on mobile devices running Android or iOS or Linux based embedded devices like Raspberry Pi or even some microcontrollers. It is a cross-platform that converts a TensorFlow model into a special format called flatbuffer that can be optimized for speed or storage. For microcontrollers, there is another version of TensorFlow, **TensorFlow Lite for Microcontrollers**. It is written in C++ 11 and requires a 32-bit platform, which is mostly compatible with ARM Cortex-M Series processors used in microcontrollers. There also exist software platforms like Edge Impulse or Neuton for creating highly optimized end-to-end TinyML applications from scratch without writing a single line of code.

Process flow of creating TinyML applications

Depending upon the nature of the application, the target hardware for TinyML can be a smartphone, an SBC or even a smaller microcontroller unit. A complex machine learning or a deep learning model may not exhibit the desired performance on an edge device. For example, a deep neural network for image classification can have several thousands of parameters resulting in a large model of few megabytes. A standard microcontroller contains only few hundred kilobytes of RAM. Such a microcontroller cannot even load the machine learning model in its memory. While executing a machine learning application, the training model should not occupy the entire memory space. There has to be enough memory space available for the input data, storing the intermediate variables also for the output. Hence, a standard machine learning model needs to be compressed and optimized in order to effectively run on an edge device.

In machine learning, training of the model is the most computationally expensive and time-consuming job. In general, model training is not preferred at the edge. Instead, it is commonly done on a powerful computer or server having sufficient computational power. The edge devices are primarily intended to make inferences on test data. *Figure 1.4* depicts the block diagram of the different steps involved in developing a TinyML application.

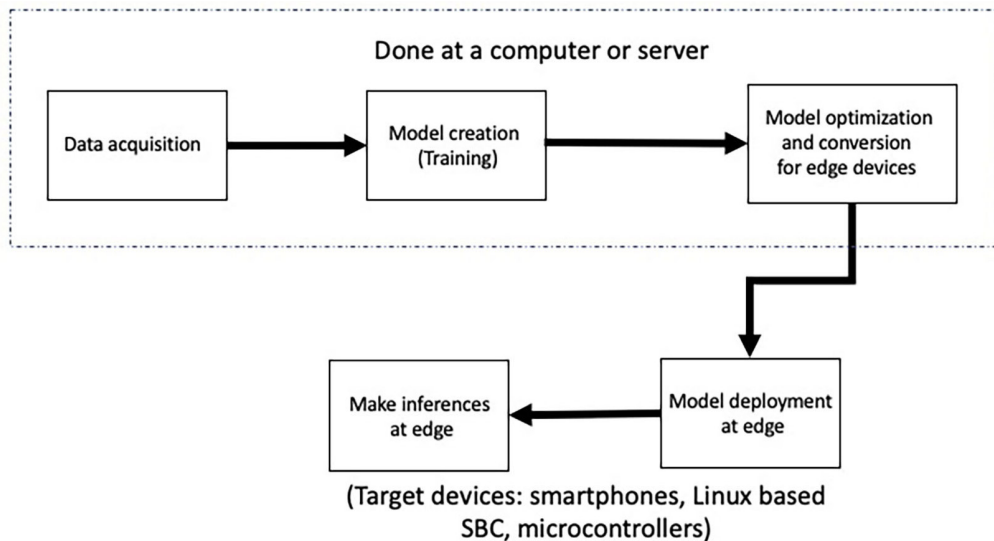


Figure 1.4: Block diagram indicating different steps of creating a TinyML application

- **Data acquisition:** Getting the right data is the key to training any machine learning model. There is no exception in TinyML. The accuracy of a machine learning algorithm heavily depends on the diversity of the dataset upon which the model has been trained. A machine learning model is not guaranteed to work on test data that is statistically different than the training data. Suppose you want to create a TinyML application that will be deployed on a Raspberry Pi for the classification of images captured by a USB camera. Suppose all your images in the training data were recorded using a high-quality DSLR camera. Despite reporting very good accuracy in internal validation on the training data, your model may not work in the target platform. The quality of images captured by the USB camera during evaluation is of much inferior quality compared to a DSLR camera used to collect the training data. In order to get the optimum performance, your training data should contain images recorded by all possible types of cameras that would be used in the deployment setup. Similarly, if you want to design a speech recognition system and you train your model on voice samples recorded from young male people, your model will most probably fail on female voice or even on the voice of elderly men.
- **Model creation:** Once a good amount of training data is acquired, we are good to go for training the model. Machine learning engineers often opt for a trial-and-error method for finding the optimum machine learning architecture. The entire dataset is split into three parts, training, validation, and test sets. The model is first created on the training set and is internally evaluated on

the validation set. The performance on the validation set is used as feedback to tune the model architecture. A model performing very well on the training set, but doing unsatisfactory on the validation set is an undesirable condition which called an overfit. Similarly, a model which does not perform on both training and validation set is called an underfit. A perfectly trained model should produce similar performance on both training and test sets. Once you are satisfied with the performance on the validating set you can finalize the model and evaluate on the test set and then release the model for deployment.

Throughout this book, we will use **TensorFlow** for developing our machine learning models in Python programming language. TensorFlow contains various libraries and Application Programming Interfaces (APIs) for the quick and easy creation of complex neural networks from scratch. One can even use a pre-trained neural network architecture and retrain it on his/her own data using transfer learning.

- **Model optimization and conversion for edge devices:** We cannot readily deploy a machine learning model to make inferences. The model needs to be optimized for the target devices. In model optimization, our key objective is to convert the original model to a smaller and faster model with a minimum impact on overall performance. Model optimization can be achieved in various ways, such as *quantization*, *weight pruning*, and *weight clustering*.

In quantization, the model activations and model weights are converted from 32-bit floating points to 16-bit floating points or even 8-bit integers, which results in a smaller model. The integer-based models execute much faster. In a large neural network, a significant number of parameters have very little impact on overall performance. We use weight-pruning to remove a few such insignificant parameters from the network. Although there is a negative impact on model performance due to pruning, it introduces model sparsity which helps to compress the model. Similarly, in weight-clustering, the number of unique weights in the model is reduced by grouping them into a given number of different clusters.

In this book, we will primarily use **TensorFlow Lite** to run our TinyML applications on edge devices like smartphones or Raspberry Pi. We will also use **TensorFlow Lite for Microcontrollers** to further optimize the models to deploy on the microcontroller units.

- **Model deployment at the edge:** Model training and optimization are mostly done on a resourceful machine having the necessary computation capacity. The optimized model is then deployed on the target edge device. There are **Software Development Kits (SDK)** specific to the target platform used for model deployment.

- **Make inference:** Once the model is successfully deployed, we are ready to go for on-device inference. Both TensorFlow Lite and TensorFlow Lite for Microcontrollers provide optimized APIs to make inferences on diverse mobile edge devices and microcontrollers.

Prerequisites—hardware and software

The readers do not necessarily require an in depth knowledge of machine learning to follow this book. However, we assume the readers have a strong mathematical base in terms of linear algebra and probability theory. We also assume that the readers have some fundamental programming knowledge in Python and C/C++. Although we will briefly cover some of the commonly used Python libraries in *Chapter 2, Crash Course on Python and TensorFlow Basics*, it might be difficult for beginners to totally understand the concept. Interested readers are, thus, encouraged to check the abundantly available resources on Python in printed or electronic mediums to get familiar with the programming concept at their own pace.

Throughout this book, we will use **Google Colab** for designing and optimization the machine learning models in Python programming language. Colab is a free Python editor where one can write and execute Python scripts on cloud. As a result, the readers do not require to install any environment at their end to run Python scripts but can use the power of GPU and TPU over the cloud for faster training of deep learning models. We will primarily use **TensorFlow** and **TensorFlow Lite** to implement various projects.

Two different edge devices will be used for implementation. The first two projects will be implemented on **Raspberry Pi 3, model B+**, and the remaining two on **Arduino Nano 33 BLE Sense**. Both are commercially available low-cost edge devices for TinyML applications. Raspberry Pi is a Linux based SBC popularly used in various academic projects for edge computing. It has enough computing resources to run optimized deep learning models. It can easily communicate with external sensors, I/O devices and camera for data collection and user interaction. One can even execute Python scripts on the device including TensorFlow Lite. On the other hand, Arduino Nano 33 BLE Sense is a microcontroller-based embedded hardware platform which is considered as the suggested development board for deploying TinyML applications. It comes with a host of inbuilt sensors, including a microphone, nine-axis accelerometer, gyroscope magnetometer, and temperature sensors so that it can be directly used in a number of practical projects without connecting to external sensors. Arduino has its own human readable language to program. The language is very similar to C/C++ with some special features and methods specific to the hardware. One having some fundamental knowledge in

C/C++ can very easily write codes for Arduino. Arduino IDE, a freely available software is used for writing the programs, code compiling and deployment. With TensorFlow Lite, one can convert a base TensorFlow model into an equivalent C++ library for Arduino as well as use the other optimized libraries to write on-device inference programs.

Conclusion

We have come to the end of the first chapter of this book. TinyML is an emerging field of advanced machine learning. In this chapter, we have briefly introduced the concept of TinyML along with its potential applications and various hardware platforms for implementation. TinyML aims at compressing and optimizing large machine learning and deep learning models to effectively run them on low-powered edge devices like mobile edge devices or microcontrollers. TinyML is a new paradigm in machine learning which ensures that the analysis is done on the device itself, where the data has been recorded. By doing this, not only consumes lesser network bandwidth but also preserves user privacy as the data recorded by the edge device is not sent to the cloud for processing. Although we are still in a nascent stage, TinyML has a lot of promise to enable AI research from a new perspective, which can revolutionize the industry. In the upcoming chapter, we will start learning the programming aspect to implement our own TinyML applications.

Key facts

- TinyML is a branch of machine learning that aims at designing extremely optimized machine learning algorithms for small-edge devices and microcontrollers.
- These applications are extremely low-powered, highly portable, and cost-effective.
- TinyML is commonly used in real-time applications that involve continuous monitoring. A few examples are the automatic prediction of machine failure, keyword spotting from human speech, traffic surveillance, and so on.

Join our book's Discord space

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

<https://discord.bpbonline.com>



CHAPTER 2

Crash Course on Python and TensorFlow Basics

Introduction

In the first chapter, we talked about the basics of TinyML, its utility, and potential applications in various industries as well as our everyday life. TinyML brings the power of machine learning to small edge devices or microcontrollers that can be easily deployed in various practical applications at home, office, and in industries such as retail, agriculture, or healthcare. Compared to traditional machine learning models, TinyML requires less computation and consumes less energy to operate. However, you will only be able to appreciate the true power of TinyML when you will learn how to create your own TinyML application, which can be used in your daily life.

From this chapter, we will gradually learn how to create a TinyML application from scratch using various open-source libraries. We will start our journey by creating a simple deep neural network model on the desktop for image classification and gradually learn different steps to design an optimized light-weight neural network that can run on small microcontroller devices. In modern data science, complex machine learning and deep learning models are commonly implemented in a scripting language called **Python**. In this chapter, we will briefly learn the basics of

the Python programming language. All the machine learning models we are going to implement in this book will be primarily created in Python. Hence, it is important to have strong basics in Python for implementing TinyML.

Python is a very popular computer programming language that was created by *Guido van Rossum* in the late 1980s. The first version of Python was released in 1991 under the GNU **General Public License (GPL)**. It is a free, high-level, interactive, and object-oriented programming language. Python is an interpreted programming language. Python scripts are processed at runtime by the Python interpreter. The interpreter translates and executes the program one statement at a time. Python language is heavily used in academics and is also used by all major tech-companies such as Google, Amazon, Meta, and so on. Unlike C/C++, you do not need to compile a Python program before executing. You can even directly interact with the interpreter from a Python prompt to execute your program. Python works on almost all popular platforms and operating systems, such as Windows, macOS X, Linux, or even Linux-based edge devices like Raspberry Pi and a few microcontrollers. Python syntax is simple enough, which makes it highly readable. It supports very high-level dynamic data types and automatic garbage collection. Furthermore, Python codes can be easily integrated with other programming languages such as C, C++, or JAVA. As a result, Python is popularly used for server-side Web scripting, software development, and mathematics.

Python is perhaps the most globally accepted programming language used in machine learning. Other popular high-level programming languages include R, JavaScript, MATLAB, and so on. However, Python is predominantly used in statistics, data-science, machine learning, and deep learning applications, from rapid prototyping to production-ready software development. Some of the key advantages of Python are as follows:

- It can handle big data and perform complex mathematics optimally.
- Python is simple and easy to learn. Hence, it is extremely popular among researchers and developers.
- Python offers a large number of libraries and frameworks for easy implementation of complex machine learning algorithms.
- Being platform independent, Python applications can run on multiple operating systems. Hence, it is convenient to use Python for both development and deployment.
- Finally, Python has a great user community to disuses your issues.

Throughout this book, we will use Python 3 to develop all our projects, which is the recent stable version of Python.

Structure

In this chapter, we will discuss the following topics:

- Colab Notebook
- Python variables
- Conditional and logical operations
- Loops in Python
- Functions in Python
- Python libraries
 - NumPy library
 - Random number generation
 - Matplotlib library
 - Pandas library
- Introduction to TensorFlow
 - Tensors and datatypes
 - Differentiation in TensorFlow
 - Graphs and functions in TensorFlow
 - End-to-end Machine Learning algorithm using TensorFlow

Objectives

The key objective of this chapter is to briefly cover the fundamental aspects of the Python programming language and a few of its libraries popularly used in data science and machine learning applications. Although Python is a vast programming language having lots of functionalities and external libraries to learn, we will briefly cover a few of them through simple code examples to get an idea of how to use them in machine learning. The knowledge learnt in this chapter will be heavily used in subsequent chapters in implementing the projects. The readers who already have some fundamental knowledge in Python can skip the first few sections of this chapter or can quickly go through the code examples to brush up their skills. Later in this chapter, we will briefly cover the basics of TensorFlow, a software library

popularly used in creating large and deployment-ready machine learning and deep neural network models.

Colab Notebook

As mentioned earlier, Python is an interpreted programming language. In order to execute Python, you need to install some software on your computer. The list of software includes the Python interpreter, various supporting libraries that you might require in your applications, and optionally an **Integrated Development Environment (IDE)**, such as Thonny, PyCharm, or Jupyter, for editing or quick debugging of the programs. A detailed instruction for installing Python distribution and various libraries on your computer can be found on the official website of Python¹.

In this book, we will use Colab Notebook, a cloud-based environment for Python programming to implement the projects. Collaboratory, commonly known as Google Colab or simply Colab, is a Google research project providing a free Web browser-based Jupyter Notebook environment for writing and executing interactive Python codes. It requires no software installation or setup at the user's end, as the code execution is entirely done in the cloud. Hence, you do not need a high-end computer to execute large and complex Python codes. The user can use Google Colab virtually from any device with an internet connectivity to connect to a Python runtime for code execution. Moreover, you can freely use the power of **Graphics Processing Units (GPU)** or **Tensor Processing Units (TPU)** hosted at Google's data centers to speed up the code execution. The notebooks created at Colab are stored in your Google Drive, which you can easily maintain and also share with others. Most of the necessary libraries popularly used in developing deep neural network architectures are preinstalled in Colab, which will give you a seamless programming experience.

The best user experience of Google Colab can be found by using the Google Chrome Web browser. Google Colab can be accessed simply by typing and hitting the URL <https://colab.research.google.com/>. The welcome page of Colab will look similar to what is shown in *figure 2.1*:

1 <https://www.python.org>

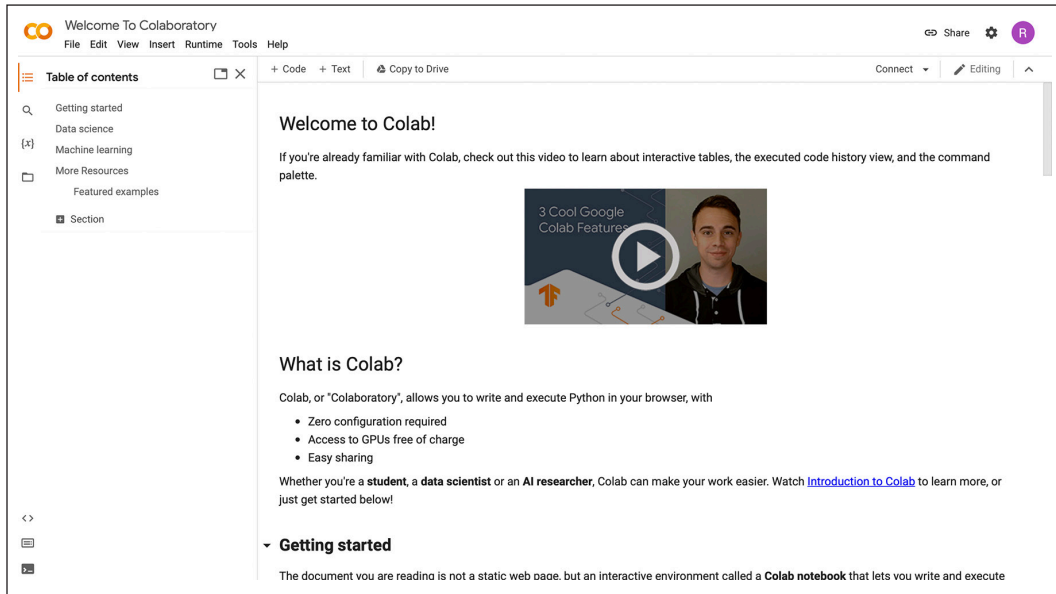


Figure 2.1: Google Colab welcome page

You are encouraged to watch the welcome video on the Web page to get a quick overview of Colab. You need to link your Gmail account with Colab so that your notebook files are automatically saved in your Google Drive. Let us create a new notebook from the **File** tab at the top left side of the browser. Select **File** → **New notebook**.

You will get a new Colab notebook to implement your project, as shown in *figure 2.2*. It contains a code cell where you write your program and buttons to perform various operations. You can rename your project by clicking on the filename. The project can also be saved on your local machine as an `.IPYNB` file.



Figure 2.2: Sample Colab notebook

Now, perform the following:

1. Once a new project is created, click on the **Connect** button on the right side of the notebook. It will allocate you the necessary resource and connect you to a hosted Python runtime to execute your code in the cloud.

2. You can also connect to a GPU or TPU. Go to **Runtime->Change runtime type**. However, GPU resource is only allocated for a given period of time and is expected to be used only when heavy computation is required.
3. You can write your code inside the code cell of the notebook (Refer *figure 2.2*). Click the play button at the left to execute your code inside the cell. The output will be displayed underneath the cell. You can add new code cells by clicking the **+Code** button and execute the cells individually.

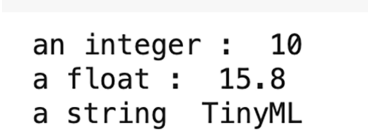
Now, let us learn some basics of Python programming through practical examples.

Python variables

Understanding the variables and data types is important in learning any programming language. Python has five standard data types, **numbers**, **string**, **list**, **tuple**, and **dictionary**. Unlike C/C++, Python variables do not require an explicit declaration to reserve memory space. The equal (=) sign is used to assign values to variables. Go to your Colab notebook and add a new code cell. Enter the following code and execute. It will declare and print different variable types in Python. The code output is shown in *figure 2.3*:

```
>>x = 10
y = 15.8
z = 'TinyML'

print('an integer : ', x)
print('a float : ', y)
print('a string ', z)
```



```
an integer : 10
a float : 15.8
a string TinyML
```

Figure 2.3: Python variable declaration

You can disable a code line from execution by adding the # character in front of the line. The function **print()** is used to display the output in the console. Python supports standard mathematical operations such as addition (+), subtraction (-), multiplication (*), and so on. See the following code. Add a new code cell and execute the following lines of code. The output is shown in *figure 2.4*:

```
>>a = 10
b = 20.2
c = 5
sum = a+b
subtract = b-a
mult = a*c

print('addition : ', sum)
print('subtraction : ', subtract)
print('multiplication : ', mult)

        addition :  30.2
        subtraction :  10.2
        multiplication :  50
```

Figure 2.4: Examples of mathematical operations in Python

Python strings

- Python *strings* are sets of characters represented within quotation (' ') or (" ") marks.
- The plus (+) operation is used for addition, and the asterisk (*) is for the repetition of the string.
- The colon (:) operator is used to get the characters within a string in between any two given indexes. In Python, the first index is represented by zero, and the final index is indicated by -1.

Refer to the following code for an example showing some basic string operations in Python. Run it on a new code cell. The code output is shown in *figure 2.5*:

```
>>str1 = 'Hello'
str2 = 'TinyML'
str3 = str1+str2    # addition of two strings
str4 = str1*2      # repeats the string
```

```
print(str3)
print(str4)
print(str3[0])    # prints the first character of a string
print(str3[3:6]) # prints fourth to sixth characters of a string
print(str3[-1:]) # prints the last element of a string

HelloTinyML
HelloHello
H
loT
L
```

Figure 2.5: Basic string operations in Python

Lists

- Python **lists** are compound data types comprising multiple items separated by commas and enclosed within brackets ([]).
- Lists are similar to arrays in C, but they can contain heterogeneous data types (that is, both numbers and strings).
- Lists are mutable, which means you can change the elements in a list.
- The elements in a list are indexed at 0 at the beginning, and the last element is annotated by `-1`. The colon (`:`) is used to get elements between two certain indexes within a list.
- The plus (`+`) operator concatenates two lists one after another (note, the `+` operation does not do an element-wise addition). The `append()` function is used to add a new element to a list.

See the following code for basic string operations. The code output is shown in *figure 2.6*:

```
>>list_1 = [1, 2, 3, 4, 5]    # a list of integers
list_2 = [10, 10.45, 'Tom'] # string of various data types
list_3 = list_1 + list_2    # add the two lists

print('list_3 : ',list_3)
print(list_3[2:5])        # print from third to fifth element of the list
```

```

print(list_3[-3:])      # print the last three elements of the list

list_3.append(50)      # add an element at the end of the list

print('list_3 ',list_3)

list_3 : [1, 2, 3, 4, 5, 10, 10.45, 'Tom']
        [3, 4, 5]
        [10, 10.45, 'Tom']
list_3 : [1, 2, 3, 4, 5, 10, 10.45, 'Tom', 50]

```

Figure 2.6: Example of Python list

Tuple

Python **tuples** are also sets of comma-separated values similar to lists but are enclosed in parentheses (()). Unlike lists, tuples are immutable, and hence, they cannot be updated. See the following code for understanding the concept of the tuple:

```

>>tuple_1 = (110, 20, 30, 40, 50)      # a tuple of strings

tuple_2 = (20, 25.4, 'Tom', 'Jerry') # a tuple a of various data types

tuple_3 = tuple_1 + tuple_2           # add the tuples to a get a new one

print(tuple_3)

print(tuple_3[2:6])                  # print from third to sixth element

list_4 = [10, 50, 100, 120]          # define a list

list_4[2] = 70.                      # modify the third element of the list

print(list_4)

tuple_1[2] = 25                       # will throw an error as tuple cannot be updated

```

Refer to the code output in *figure 2.7* for understanding the difference between a list and a tuple. The Syntaxes of various operations on a tuple are quite identical to the operations on a list. Please note while modification of an element in a list is possible, it is not supported in a tuple. Hence, we get an error in Line 12 of the preceding code when we try to modify the elements in a tuple. Since Python is an interpreted

programming language, it executes the program line by line and displays the output till Line 11 before throwing the error in Line 12.

```
(110, 20, 30, 40, 50, 20, 25.4, 'Tom', 'Jerry')
(30, 40, 50, 20)
[10, 50, 70.0, 120]
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-2-ea21548c0e90> in <module>()
     10 print(list_4)
     11
--> 12 tuple_1[2] = 25      # will throw an error as tuple can not be updated

TypeError: 'tuple' object does not support item assignment
```

Figure 2.7: Example of tuple

Dictionary

- Python dictionaries are similar to hash tables where data is stored in key:value pairs.
- Dictionary items are printed with curly brackets. An item can be referred by the key names.
- Dictionary items can be modified, added, or removed, but duplicate items are not allowed.

In the following code snippet, we create a dictionary and show various operations:

```
>>mydict ={'name': 'John', 'year': 1984, 'skillset' :['C', 'JAVA',
'Python' ]}

print(mydict['name'], mydict['skillset'])      #print dictionary items

mydict.update({'year' : 1990})                 # update one item

mydict.update({'company' : 'ABC International'}) # add one item

mydict.pop('skillset')                         # remove the item skillset
```

Conditional and logical operations

Python supports various conditional operations using the *if*, *elif*, and *else* keywords. Standard *and*, *or* logical operators are also supported. Unlike other programming languages, Python uses indentation to define the scope of the code and does not use

curly brackets ({}). Indentation is maintained by adding some white space before the code line. You need to be very careful in maintaining the indentation in Python programming. Failing to maintain the indentation will produce errors. Standard operations used to compare two variables in conditional operations and their syntax are as follows:

- a is equal to b : `a==b` (note, `a=b`, assigns the value of variable b to variable a)
- a is not equal to b : `a!=b`
- a is greater than b : `a>b`, a is greater than or equal to b : `a>=b`
- a is lesser than b : `a<b`, a is lesser than or equal to b : `a<=b`

The following example shows the usage of conditional operations to find the maximum between two numbers. See the usage of indentation in the code.

```
>>x = 30
y = 40

if x>y:
    print("x is bigger than y")
elif x==y:
    print("x and y are equal")
else:
    print("y is greater than x")
```

The following code example shows how both conditional and logical operations (*and*, *or*) can be used in Python to find the maximum of three different numbers:

```
>>x = 30
y = 50
z = 40

if x>y and x>z:
    print("x is the maximum")
elif y>x and y>z:
    print("y is the maximum")
```

```
else:  
    print("z is the maximum")
```

Loops in Python

In a programming language, the loop statements are used to execute a group of statements multiple times if a condition is met. Python has two popular types of loops, *while* loop and *for* loop. The codes within the loops are indented.

In a *while* loop, we first set a condition, and the set of instructions are executed in the loop as long as the condition is true. The following example code uses the while loop to print the even numbers between 1 and 10:

```
>>i = 1  
  
while i<=10:  
    if i%2 == 0:  
        print(i)  
    i = i+1
```

In *for* loops, a set of instructions is iteratively executed. Refer to the following code example that reads all elements in a given list, multiplies each element with 5, and saves in another list for printing using *for* a loop.

```
>>list_a = [4, 7, 9, 11, 15] # input list  
  
list_b = [] # define an empty list where the multiplied values will  
be stored  
  
for i in list_a:          # reads every element in the list on by one  
    j = i * 5  
    list_b.append(j)  
  
print(list_b)           # print the new list after exiting the for loop
```

Functions in Python

A function is a reusable code block that is used to perform a single task. It can be called and reused multiple times inside your program. Python functions behave similar to other programming languages. The key features are provided as follows:

- In Python, a function is defined using the **def** keyword, followed by the function name and the input arguments that are provided within parentheses and ends with a colon (:).
- The code block within a function is indented.
- A function ends with a return statement that gives the output values. A function can have single, multiple, or no return values.
- The variables defined within a function block have local scope within that function only, whereas variables defined outside that function have global scope.

The following code example shows how to define a function in Python. It performs addition, subtraction, and multiplication of two numbers by defining a function `my_func()`.

```
>># function definition
def my_func(num_1, num_2):
    sum = num_1 + num_2
    diff = num_1 - num_2
    mult = num_1 * num_2
    return sum, diff, mult
# function call
a, b, c = my_func(25, 15)
print("The sum = ", a)
print("The difference = ", b)
print("The multiplication = ", c)
```

The code output is shown in *figure 2.8*:

```
The sum = 40
The difference = 10
The mutiplication = 375
```

Figure 2.8: Defining functions in Python

Python libraries

As a programming language, a popular feature of Python is the availability of various libraries. A library can be defined as a collection of code modules that can be easily called from our program for specific operations. Python has become one of the most popular programming languages in data science and machine learning thanks to its readily available libraries, which help in the quick implementation of machine learning architectures, data visualization, and report generation without writing many lines of code.

The core Python distribution comes with a collection of standard libraries of more than 200 standard modules. The standard libraries contain built-in modules written in C and provide various capabilities such as interacting with the operating system (the **os** module), accessing files **input/output (I/O)**, performing various mathematical operations (the **math** module), parsing command line arguments (the **sys** module), and so. Apart from that, there are external libraries that need to be installed separately. In the following sections, we will briefly discuss a few libraries, which are popularly used in machine learning applications.

NumPy library

NumPy is a popular Python library for working with numerical arrays and is commonly used in linear algebra and data science. A NumPy array is identical but faster than a Python list and comes with many more functionalities. The NumPy array object is called **ndarray**. An ndarray is a multi-dimensional object of homogeneous data. The shape of the array is a tuple integers specifying the size of each dimension. NumPy arrays are highly optimized, and the array elements are stored in continuous memory locations, ensuring faster access and processing of data compared to lists.

In order to work with a library, one needs to import it into the workspace. In Python, a library is imported by the **import** keyword. While importing the NumPy package, we create an alias **np** with the keyword **as** to avoid the repetitive usage of the word **numpy** in the program.

NumPy and most of the Python libraries required in machine learning are already installed in the Colab environment. So, you just need to go to your Colab notebook and type the following command to import the NumPy library.

```
>>import numpy as np
```

In Python, the functions inside a library module are called by using a dot (.) operator along with the module name. A NumPy array can be created by passing the elements

separated by coma as input arguments to the function `array()` under the NumPy module. It will create an ndarray object. Similarly, A Python list or a tuple can also be converted to a NumPy array using the same function. The following code example shows how to create a NumPy array and how to convert a Python list and tuple to a NumPy array.

```
>>import numpy as np

arr_1 = np.array([1,2,3,4]) # define an array
print(arr_1)
print("type of arr_1 : ", type(arr_1))

my_list = [10, 20, 30 ,40] # define a list
arr_2 = np.array(my_list) # convert the list to an array
print(arr_2)

my_tuple = (50, 100, 200) # define a tuple
arr_2 = np.array(my_tuple) #convert the tuple to an array
print(arr_2)
```

See the code output in *figure 2.9*:

```
[1 2 3 4]
type of arr_1 : <class 'numpy.ndarray'>
[10 20 30 40]
[ 50 100 200]
```

Figure 2.9: Creating a NumPy array

The preceding arrays are 1-dimensional. NumPy supports 0, 1, 2, 3 or higher dimensional arrays. The array dimension can be returned by the `ndim` attribute. Refer to the following example to print the dimension of a NumPy array.

```
>>arr_2d = np.array ([[1,2,3,4], [5,6,7,8]]) # a 2-D array
arr_3d = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]]) # a 3-D
array
```

```
print(arr_2d.ndim)
print(arr_3d.ndim)
```

Indexes in a NumPy array start with 0 to access the first item, index 1 to access the second element, and so on. The last item is indexed as `-1`. In the previous example, the third element in the second row in the 2-D array, `arr_2d`, can be accessed by `arr_2d[1, 2]`, that is, `arr_2d[1, 2] = 7`. Similarly, in the 3-D array, `arr_3d[1, 0, 2] = 9`. The colon operation is used to select all the elements in a row or a column in an array. For example, `arr_2d[:, 2]` will print the entire row matrix at the second column. So, `arr_2d[:, 2] = [3, 7]`. Similarly, `arr_3d[1, 1, :] = [10, 11, 12]`.

We often need to generate data arrays in our program. The `np.arange()` function creates an array of evenly spaced number within a given range. Executing `np.arange(0, 10)` will return an array of 10 elements from 0 to 9. By default, the distance between two elements, that is, the step-size in the array is 1. However, one can provide the step-size between two elements as the third input argument in the function. Remember, the end-point of the interval is excluded. Similarly, the `linspace()` function returns evenly spaced samples within a given range. There is a subtle difference between the two functions. The function `arange()` allows you to define a step-size and infers the number of steps between the given ranges, whereas `linspace()` allows you to define how many values you want within the given interval.

The following example shows the usage of `arange()` and `linspace()`.

```
>>arr1 = np.arange(0,10)      #creates an array between 0 and 10
arr2 = np.arange(10,30,3)    #creates an array between 10 and 30 with
                             step-size 3
print("arr1 = ",arr1)
print("arr2 = ",arr2)
#using linspace
print(np.linspace(0,20,10))  #creates an array between 0 and 20 in 10
                             steps
```

See the code output in *figure 2.10*:

```
arr1 = [0 1 2 3 4 5 6 7 8 9]
arr2 = [10 13 16 19 22 25 28]
[ 0.          2.22222222  4.44444444  6.66666667  8.88888889 11.11111111
 13.33333333 15.55555556 17.77777778 20.          ]
```

Figure 2.10: Usage of `arange()` and `linspace()` in NumPy:

The functions `zeros()` and `ones()` under NumPy create an array of zeros and ones of a given dimension. They will create one-dimension and multi-dimension arrays where the inputs are given in the form of a single number or a tuple. Refer to the following code example, the output of which is shown in *figure 2.11*:

```
>>arr_zeros_1d = np.zeros(5)

arr_zero = np.zeros ((3,3))

arr_one = np.ones ((4,3))

print("arr_zeros_1d", arr_zeros_1d) # creates 1-D array of zeros
print("arr_zero = ", arr_zero)      # creates 3X3 array of zeros
print("arr_one = ", arr_one).       # creates 4X3 array of ones

arr_zeros_1d [0. 0. 0. 0. 0.]
arr_zero = [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
arr_one = [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

Figure 2.11: NumPy arrays of zeros and ones

NumPy also allows you to rearrange the dimension of an array without changing the data using the `reshape()` function. You can also convert a 1-D array into a multi-dimensional array. See the following example and the corresponding output in *figure 2.12*:

```
>>array = np.arange(0,12) # create a 1-D array

print("original array")

print(array)              #print the original array
```

```
print("2-D converted")

print(array.reshape((4,3))) # rearrange into a 2-D array of 4 rows and
3 columns

print("3-D converted")

print(array.reshape((2,3,2))) #rearrange into a 3-D array

original array
[ 0  1  2  3  4  5  6  7  8  9 10 11]
2-D converted
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
3-D converted
[[[ 0  1]
 [ 2  3]
 [ 4  5]]

 [[ 6  7]
 [ 8  9]
 [10 11]]]
```

Figure 2.12: Reshaping a NumPy array

NumPy has a wide list of inbuilt functions to perform various mathematical operations on arrays. Apart from simple mathematical operations such as addition, subtraction, one can perform more complex operation such as finding the numerical mean of the elements, element-wise multiplication between two arrays, and so on. NumPy also has inbuilt functions to perform matrix operations such as matrix multiplication, transpose, rank, finding Eigen values, and Eigen vectors. You can also perform operations on rows and columns. Specify `axis = 0` for column-wise and `axis = 1` for row-wise operations. The default is `axis = None`. A few popular mathematical operations are shown in the following code. Modify and execute the code at your end to play with various mathematical operations.

```
>>arr_1 = np.array([[1, 2 ,3], [4, 5, 6], [11, 12, 13]])

arr_2 = np.array([[7,8,9],[10,11,12], [14, 15, 16]])

print("sum of the arr_1 and ar_2")

print(np.add(arr_1, arr_2))
```



```
print("subtract between arr_2 and arr_1")
print(np.subtract(arr_2, arr_1))
print("element wise multiplication")
print(np.multiply(arr_1, arr_2))
print("matrix multiplication of arr_1 and arr_2")
print(np.matmul(arr_1, arr_2))
print("average of elements in arr_1=", np.mean(arr_1))
print("average of elements in arr_1 column-wise=", np.mean(arr_1,
axis=0))
print("average of elements in arr_1 row-wise=", np.mean(arr_1, axis=1))
```

Random number generation

In *Chapter 3, Gearing with Deep Learning*, we will learn about neural networks. When we train a neural network, we basically iteratively update the weights for different units (nodes) of the network to reduce a loss function. The initial values of the weights are set by some random numbers, which are updated to the optimum values. In Python, we can easily generate an array or a matrix of random numbers of any dimension. The **random** module under NumPy contains various functions for generating random numbers.

The **randint()** function generates a random integer number, an array, or a matrix of any given dimension within a given range as given in the input argument. Here, The random numbers are drawn from a discrete uniform distribution. The **rand()** function gives a random floating point number between 0 and 1 drawn from a uniform distribution. The function only takes the dimension of the array or matrix you want to generate as input and generates it with random samples. The **randn()** function is similar to the **rand()** function, but it draws random samples from a standard normal distribution of zero mean and unit variance to populate the matrix.

Please note as the numbers are generated randomly, each time you call these functions, a new set of random data will be generated. Hence, when you execute the code, the output printed in the console will be entirely different than what is shown in the code examples.

The following code example shows how random numbers can be generated using the **random** module under NumPy:

```

>>from numpy import random

random.seed(5)

print("A random number between 0 and 10 = ", random.randint(10))

print("A 3X3 rmatrix of random numbers between 10 and 20 = ")

print(random.randint(10,20, (3,3)))

print("A 3X3 rmatrix of random numbers between 0 and 1 = ")

print(random.rand(3,3))

print("A 3X3 rmatrix of random numbers from a unit normal
distribution = ")

print(random.randn(3,3))

# create a large matrix with random.randn(). Its mean and variance
will be close to 0 and 1

randn_matrix = random.randn(500,100)

print("mean and variance of a large randn matrix : ", (np.
mean(randn_matrix), np.var(randn_matrix)))

```

The code output in one run is shown in *figure 2.13*: Run it several times to see the different output in different run.

```

A random number between 0 and 10 = 3
A 3X3 rmatrix of random numbers between 10 and 20 =
[[16 16 10]
 [19 18 14]
 [17 10 10]]
A 3X3 rmatrix of random numbers between 0 and 1 =
[[0.2968005  0.18772123  0.08074127]
 [0.7384403  0.44130922  0.15830987]
 [0.87993703  0.27408646  0.41423502]]
A 3X3 rmatrix of random numbers from a unit normal distribution =
[[ 0.91187364 -1.44384155  1.82444017]
 [ 1.45762512 -0.91025815 -0.99001894]
 [-0.81754812 -1.16827947 -0.21077965]]
mean and variance of a large randn matrix : (0.0013921817469718315, 0.9992936178062981)

```

Figure 2.13: Generating random numbers in NumPy

As mentioned, when you call the functions under the *random* module, you will get completely different sets of random numbers every time you execute. However, you can get the same output by setting a fixed seed at the beginning. This is done by calling the `seed()` function at the beginning before calling the functions that generate random numbers. It takes an integer as input and sets the seed accordingly

to generate a fixed set of random numbers. Just add `random.seed(5)` at the beginning of the previous example, and you will get the same set of outputs. Replace 5 with a different integer to get a completely different set of data. Refer the following code

```
>>random.seed(5)

    print(random.rand(3,3))
```

It will produce the same output every time.

Matplotlib library

Matplotlib is a Python library for data visualization, plotting of data in graphical format, and creation of interactive animations. Most of the utilities of Matplotlib lie under the `pyplot` module, which is popularly used for static data plotting. The library can be imported by the following command:

```
>>import matplotlib.pyplot as plt
```

You can plot both Python lists or NumPy arrays. A simple example of plotting a sinusoidal wave stored in a NumPy array using the matplotlib library is shown in the following example. We define a sinusoidal function and plot it using matplotlib libraries. The `plot()` function is used to plot the waveform in the console. We can also assign labels to the axes and add a title to our plot. It has two main input arguments. The first argument is the array corresponding to the *x-axis*, and the second one is the array corresponding to the *y-axis*. Both arrays should have the same dimension. You can also set the axis names and titles for your plot. Refer to the following code:

```
>>import numpy as np

    import matplotlib.pyplot as plt

    x = np.linspace(0,50,100)

    y = np.sin(x)*np.pi/180

    plt.plot(x, y)

    plt.xlabel("time axis")

    plt.ylabel("amplitude")

    plt.title("a sinusoidal wave");
```

The output is shown in *figure 2.14*:

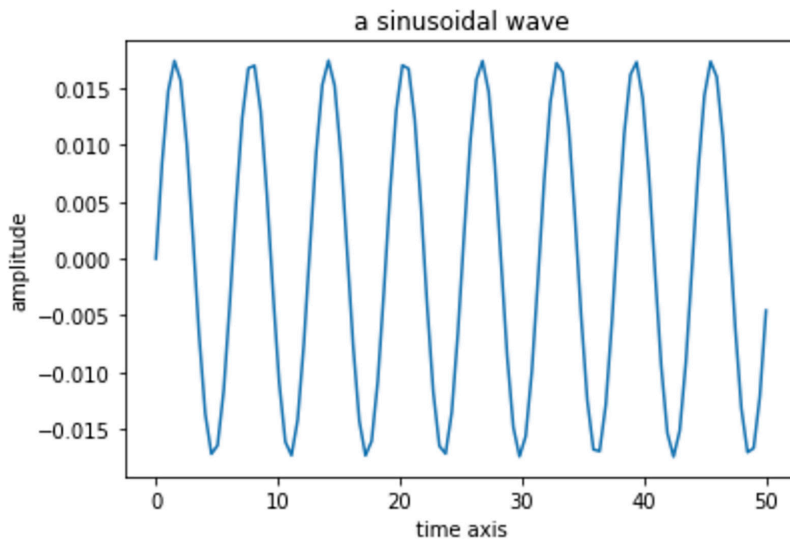


Figure 2.14: Plotting a sinusoidal waveform using Matplotlib

Note that if you are using IDEs other than Colab or executing your scripts from a command line, you have to add `plt.show()` at the end of the preceding code in order to visualize the plots.

Matplotlib allows the usage of various colors and font sizes in the plot. The generated plots can also be saved as images for creating reports or presentations. Using the `subplot()` function, you can draw multiple plots in a single image. In the following example, we define a sinusoidal wave, add some random noise to it, and plot the two waveforms separately using the `subplot()` function. In the following code, we have plotted the two waveforms vertically. They can also be plotted horizontally by changing it as `plt.subplot(1,2,1)` and `plt.subplot(2,2,1)`, respectively. You are encouraged to try that too. Similar to the previous example, you can also provide axis labels and add titles to the plots. Refer to the following code:

```
>>from numpy import random
    x = np.linspace(0,50,100)
    y = (np.sin(x)*np.pi/180)*2
    y_noisy = y+ np.random.normal(0,0.05,size=100)
    plt.subplot(2,1,1)
    plt.plot(x,y);
```

```
plt.subplot(2,1,2)
plt.plot(x,y_noisy);
```

The code output is shown in *figure 2.15*:

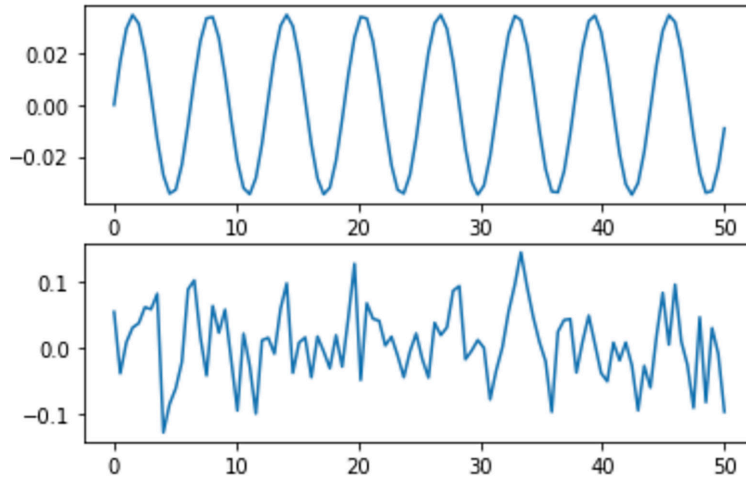


Figure 2.15: Example of subplot in Matplotlib

Apart from these, Matplotlib also supports statistical plots for data visualization, such as pie-charts, bar-plots, histograms, and so on.

Pandas library

Pandas is another popular library having various functions for reading, analyzing, and manipulating data. Large-scale data can be easily processed in Pandas. Execute the following command to import the Pandas library.

```
>>import pandas as pd
```

Pandas deals with three different data structures, *Series*, *DataFrame*, and *Panel*. A *Series* is a labeled 1D homogeneous array of immutable size. *DataFrames* and *Panels* are labeled 2D and 3D structures of heterogeneous data with mutable size. While the Pandas library has wide functionalities, we will briefly discuss here about Pandas **DataFrames**. *DataFrames* are popularly used to store multi-dimensional data obtained from various structured formats such as **.TXT** and **.CSV**.

In the following code example, we read the California housing dataset, “**california_housing_train.csv**,” which is already stored in the Colab workspace. This dataset contains housing prices in California along with various attributes that determine

the price. A CSV file can be read in Pandas by calling the `read_csv()` function, and the content will be loaded in a DataFrame. The `columns` attribute will give you different column names available in the CSV file. A DataFrame can also be converted into a NumPy array for further processing.

```
>>import pandas as pd

df = pd.read_csv("https://storage.googleapis.com/mledu-datasets/
california_housing_train.csv", sep=",")

print(type(df))

print(df.columns)
```

Refer to *figure 2.16* for code output:

```
<class 'pandas.core.frame.DataFrame'>
Index(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
       'total_bedrooms', 'population', 'households', 'median_income',
       'median_house_value'],
      dtype='object')
```

Figure 2.16: Reading csv file using Pandas

Data corresponding to a particular column can be fetched by calling the column name. In the preceding example, `df['population']` will give you all the entries corresponding to the population column. Using Pandas, we can perform various mathematical operations such mean, mode, and so on, and also perform filtering of data based on certain queries. The following code example prints the average and median population in the dataset. Then it plots those entries where the population is higher than 20,000:

```
>>print("maxium population = ", df['population'].max())

print("median population = ", df['population'].median())

filter = df['population'] > 20000

print(df[filter])
```

Introduction to TensorFlow

TensorFlow is an end-to-end, open-source platform popularly used for the quick implementation of machine learning algorithms. A rich ecosystem of tools, libraries, and community resources has made it extremely popular among machine learning

researchers and practitioners to develop and deploy various machine learning algorithms with greater efficiency and flexibility. TensorFlow has become popular in recent times for the quick development of complex deep neural network architectures for both experimentation and developing production-ready software.

TensorFlow was originally developed by Google within their Machine Intelligence Research Organization to conduct various machine learning and neural networks related research. The initial version was released in 2015 under the Apache License 2.0. TensorFlow 2.0, the newest stable version, was released in 2019. TensorFlow is highly flexible. It supports a wide variety of programming languages, including Python, C++, and Java. Moreover, it can run on CPU, GPU, and TPU for a faster processing of large machine learning applications. TensorFlow is available in Linux, macOS X, and Windows platforms. It also supports TensorFlow Lite, a highly optimized lighter version of the original TensorFlow that is available on mobile computing platforms such as Android, iOS-based smartphones, and Linux-based single board computers like Raspberry Pi. TensorFlow Lite models can be further optimized using a few standard APIs to run on microcontroller units. Hence, TensorFlow is heavily used in TinyML applications. Visit the official TensorFlow website for more details². To summarize, some of the key features of TensorFlow are as follows:

- It is open-source
- Efficiently works with multi-dimensional data
- Provides a higher level of abstraction, which reduces the code length for the developer
- Supports various platforms and architectures
- Highly scalable and provides greater flexibility for quick prototyping

TensorFlow, along with all its dependencies, is already installed in Colab. So, you just need to import the libraries to write codes without any package installation. TensorFlow can be imported by typing the following command in the code cell.

```
>>import tensorflow as tf
```

Once TensorFlow is imported, you can check the version by typing the command `tf.__version__`. At the time of writing this book, the TensorFlow version in Colab is 2.12.0, which may change with time. TensorFlow 2, or TF 2, is the newest version which is significantly different compared to the previous version TF 1.x. In this book, we will use TF 2 for all our programming. However, TF 2 provides a backward compatibility module to use TF 1.x. The eager execution mode of TF 2 makes it easier to create a machine learning architecture with lesser lines of code.

² <https://www.tensorflow.org>

Tensors and datatypes

Tensors are the backbone of TensorFlow. A **tensor** can be considered as an n -dimensional array similar to a NumPy `ndarray`. Tensors are defined by their dimensions like, scalar number (0-D), vector (1-D), matrix (2-D), and other higher dimensional arrays. TF 2 supports various data types, like:

- 8-bit, 16-bit and 32-bit, and 64-bit integers
- 32-bit and 64-bit floating point numbers
- 8-bit unsigned integers
- strings
- Boolean numbers

TF 2 has two primitive types of data, constants, and variables. A constant is an immutable value that is defined only once during declaration. A variable is mutable and can be modified and recomputed during programming.

The following code example shows the usage of constants in TensorFlow using TF 2. Note, unlike TF 1.x, TF 2 does not require to define a `tf.Session()` to execute an operation.

```
>>a_scalar = tf.constant(1, name="const1")

a_vector = tf.constant([10.0, 20.1, name="const2")
a_matrix = tf.constant([[20, 25.5],[5.8, 27], name="const3")

print(a_scalar)
print(a_vector)
print(a_matrix)
print("shape of a_matrix = ", a_matrix.get_shape())
print("rank of a_matrix = ", tf.rank(a_matrix))
```


Refer *Figure 2.17* for the code output:

```
tf.Tensor(10, shape=(), dtype=int32)
tf.Tensor([10.  20.1], shape=(2,), dtype=float32)
tf.Tensor(
[[20.  25.5]
 [ 5.8 27. ]], shape=(2, 2), dtype=float32)
shape of a_matrix = (2, 2)
rank of a_matrix = tf.Tensor(2, shape=(), dtype=int32)
```

Figure 2.17: Defining constants in TensorFlow

In the preceding code example, we have defined a scalar, a vector, and a 2-D matrix as constants. Constants are defined by `tf.constant()`. Optionally, we can also provide the name of a constant. Printing a constant displays its content, shape, and data type of the content. We can also get the tensor shape by calling the `get_shape()` function. The function `tf.rank()` gives the rank of a tensor.

Variables in TensorFlow are defined using `tf.Variable()`. The elements in a tensor variable can be modified. A tensor can also be converted into a `NumPy()` array. Refer to the following code example illustrating how to define variables in TensorFlow:

```
>>var_1 = tf.Variable([[[[1.,2.,3.],[4.,5.,6.]],[[7.,8.,9.],[10.,11.,
12.]])])

print(var_1.value())

var_1[0,1,2].assign(25)
print("new tensor after modifying an element = ")
print(var_1.value())

# convert in to numpy array
numpy_var = var_1.numpy()

print("data type after conversion to numpy array = ", type(numpy_var))
```

Similar to NumPy, TensorFlow provides plenty of functionalities for standard mathematical operations such as addition, subtraction, multiplication, averaging of data, and various matrix operations.

Differentiation in TensorFlow

While training a neural network, we often use an algorithm called backpropagation to update the network weights. Backpropagation involves a series of differentiation operations to calculate the gradient of the weights. TensorFlow provides the API `tf.GradientTape` for optimized differentiation. Once a mathematical expression is defined, `GradientTape.gradient(target, source)` computes the gradient of the `target` with respect to the `source`. The following code example shows the usage of `GradientTape()` on scalar tensors to perform a simple differentiation task on the function `y` at `x = 5`.

```
>>x = tf.Variable(5.0)

with tf.GradientTape() as tape:
    y = x**3+ 5*x+ 2
dy_dx = tape.gradient(y, x) # calculates the gradient 3*x**2 + 5
print(dy_dx.numpy())
```

Execute the code and check whether it calculates the differentiation accurately. The differentiation operation can be further extended on multi-dimensional tensors. In the following example, we define a variable termed as `loss` from multi-dimensional tensors `w`, `b`, and constant `x`. The function `tf.reduce_mean()` computes the mean of elements across dimensions of a tensor. Then, we compute the gradient of the variable `loss` with respect to `w` and `b`:

```
>>x = tf.constant([[1.,2.,3.],[4.,5.,6.]])
w = tf.Variable(tf.random.normal(3, 2)), name='w')
b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')

with tf.GradientTape(persistent=True) as tape:
    y = tf.matmul(x, w) + b
    loss = tf.reduce_mean(y**2)
    print(loss)

[d1_dw, d1_db] = tape.gradient(loss, [w, b])
print(d1_dw.shape)
```

Graphs and functions in TensorFlow

In TensorFlow 2, operations are executed eagerly by default, and eager executions are imperative where the operations are executed immediately as they are called in the program. Hence, it is easier to debug and provides better readability. However, in TensorFlow 1.x, the default was graph mode. Graph execution allows code portability outside Python. Here, computations are executed as a TensorFlow graph, which creates a portable solution. A graph (**tf.Graph**) comprises of a set of operations (**tf.Operation**), which are basically the nodes of the graph, and tensors which represent the units of data flowing between the operations. The graphs can be saved, run, and restored without the underlying Python code. As a result, TensorFlow graphs can be deployed in environments that do not have a Python interpreter, such as smartphones or microcontrollers.

We can switch from eager execution to graph execution in TensorFlow 2 using **tf.function**. A graph can be created in TensorFlow using **tf.function**, which takes a Python function as input and creates a *Function*, a Python callable that builds a TensorFlow graph from a standard Python function. A Function encapsulates multiple graphs and enables faster execution and deployability. Moreover, graph execution is much faster than eager execution.

The following code example shows how a simple function, **my_func()**, can be converted into a Python callable Function, **tf_myfunc()** using **@tf.function**. The function multiplies two tensors, and the result is added with another tensor to produce the output. When executed, it is converted into a graph for execution.

```
>>def my_func(x, y, b):
    return tf.matmul(x,y) + b
# convert it to a Function
@tf.function
def tf_myfunc(x):
    y = tf.constant([[2.0], [3.0]])
    b = tf.constant(4.0)
    return my_func(x, y, b)
# call the function tf_myfunc
tf_myfunc(tf.constant([[10.0, 15.0]])).numpy()
```

Refer to *figure 2.18* for the code output:

```
array([[69.]], dtype=float32)
```

Figure 2.18: Converting a Python function into a Python callable Function in TensorFlow

While `tf.Graph` converts the in-built TensorFlow operations into a graph, a dedicated library called **AutoGraph**, is used to convert the remaining Python logic for graph-generating code. You can view the graph-generating output of **AutoGraph** for the preceding example by executing the following command:

```
>> print(tf.autograph.to_code(my_func))
```

End-to-end Machine Learning algorithm using TensorFlow

So far in this chapter, we have discussed the basics of Python and TensorFlow with simple examples to get some insights. Now, we will apply our learning to create our first machine learning application, a simple linear regression model. Linear regression is a supervised machine learning approach that finds a linear relationship between a set of dependent (y) and independent (x) values so that if a new set of x values is given as input, the model can predict the corresponding y value with some accuracy. Linear regression assumes a linear relationship exists between x and y , which can be represented by the following:

$$y = w \cdot x + b$$

Here, the parameters w and b are termed as the slope and intercept, which are the two constants in the equation. These two parameters are learnt during training on a given set of data having x and corresponding y values. It tries to find the best possible values for w and b so that a straight line can be fitted on the data. Once done, we can predict an unknown y from a given x using the w and b values obtained in training. The independent variable x can be single or multi-variate. In machine learning, w and b are also called the learning parameters. To be more precise, w is called as weight, and b is called as bias. The following steps occur during training: Initially, some random values are assigned for w and b , and the corresponding y value is calculated for all the x values in the training data in the training data using the preceding equation. As expected, the calculated y values will be no way near the actual values of y . We define a loss function, J , which measures the **Mean Squared Error (MSE)** between the predicted and actual y values for all training examples, Hence:

$$J = \frac{1}{n} \sum_{i=1}^n (\text{pred}_i - y_i)^2$$

Here, n is the number of training data, pred_i , and y_i are the predicted value and actual value of i^{th} data in the training set. The objective of the training process is to iteratively update the weights to get the best values for w and b to minimize the value of J . Gradient Descent is an optimization approach popularly used to meet the objective. It is a first-order optimization algorithm to find the local minimum of a function. It comprises the following steps:

1. Make some initial assumptions of the weights (that is, w and b).
2. Calculate the first-order derivative of the loss function, J , with respect to the weights to compute the gradient or slope.
3. In order to get a local minimum, move away from the gradient of the loss function at the current point by *alpha* (α) times. Next, update the current weights. The term, α is called the learning rate.
4. Repeat Steps 2 and 3 until J is minimized.

Now, we will write a script using TensorFlow to build a single-variate linear regression model. Create a new notebook in Colab. Let us start by importing the necessary Python libraries in the first code cell:

```
>>import tensorflow as tf

import numpy as np

from numpy import random

import matplotlib.pyplot as plt
```

Next, we will create our dataset for our application. Our dataset comprises both the dependent variable, y , and the independent variable, x , that follow a linear relationship with a slope (w) = 2 and an intercept (b) = 3. To give it a more practical feel, the y values are added with some random noise (say σ). Hence, the actual relationship is as follows:

$$y = 2 \cdot x + 3 + \sigma$$

Let us assume the x values range between 0 and 5, and we create 500 data points in our dataset. The following code generates our data.

```
>>x = np.linspace(0.,5., 500)

y = 2 * x + 3 + np.random.randn(len(x))

plt.plot(x,y, '*')
```

```
plt.xlabel('x values')  
plt.ylabel('y values')
```

When executed, the code shows the distribution of the data points in a scatter plot. See *figure 2.19*:

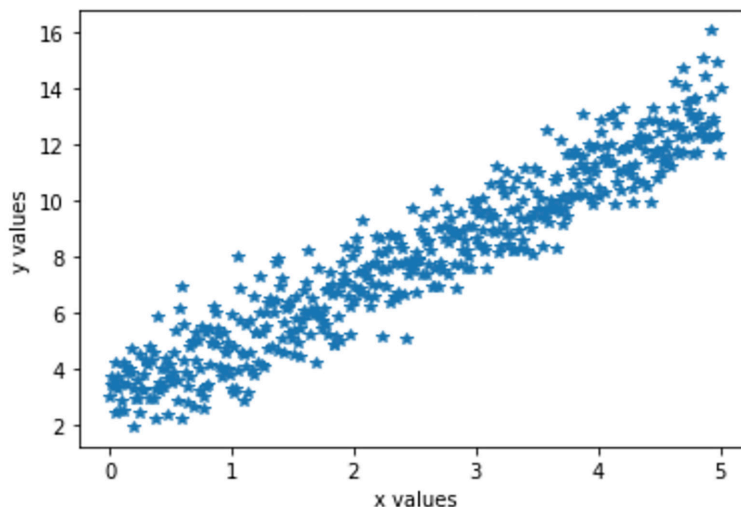


Figure 2.19: Scatter plot of the generated data points in our example

Figure 2.19 shows there is a clear visual linear relationship between x and y . We will create a linear regression model to find the best-fit line.

Now, we will split the dataset into training and test set. The training data will be used to learn the values for w and b . The test data will be used for performance evaluation. For that, we will use a Python library, *scikit-learn*. It is an open-source library featuring various machine learning algorithms.

The following script will randomly split our datasets into two parts, 75% of the data for training and the remaining 25% for testing. Assigning a value to the parameter **random-state** will ensure we get the same split every time, which is good for the repeatability of the results.

```
>>from sklearn.model_selectionimport train_test_split  
  
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size =  
0.25, random_state = 100)
```

In this problem, we have two trainable parameters w and b . We set some random values as their initial weights.

```
>>w = tf.Variable(np.random.randn())  
    b = tf.Variable(np.random.randn())
```

Now, we will define two helper functions. The first function calculates the loss function, which is the MSE between the predicted and actual values. In the second function, we calculate the gradient of the loss function with respect to w and b . We will use the `tf.GradientTape` API to perform the operation. Refer to the following code:

```
>>def calc_mse(x, y, w, b):  
    y_pred = w * x + b  
    mse = tf.reduce_mean(tf.square(y_pred - y))  
    return mse  
  
def calculate_gradient(x, y, w, b):  
    with tf.GradientTape() as tape:  
        loss_value = calc_mse(x, y, w, b)  
  
        w_grad, b_grad = tape.gradient(loss_value, [w, b])  
    return w_grad, b_grad
```

Now, we are all set to train our linear regressor. We will set a learning rate (α) of 0.001 and train for 500 epochs to reduce the loss. The parameters w and b are updated based on the calculated gradient in every epoch. We also store the corresponding MSE value in every epoch in a list. Refer to the following code.

```
>>num_epochs = 500  
  
    learning_rate = 0.001  
    loss = []  
    epoch_list = []  
  
    for epoch in range(num_epochs):  
        w_grad, b_grad = calculate_gradient(x_train, y_train, w, b)
```

```
dW, dB = w_grad * learning_rate, b_grad * learning_rate
w.assign_sub(dW)
b.assign_sub(dB)

loss.append(calc_mse(x, y, w, b))
epoch_list.append(epoch)

if epoch % 10 == 0:
    print(f"Epoch: {epoch}, loss {calc_mse(x, y, w, b):.3f}")
```

Now, execute the code. It will take few seconds to run on Colab using the CPU. It will also print the MSE value in every 10 epochs. We can use the following code to plot and see how the overall MSE loss gets reduced with increasing epochs.

```
>>plt.plot(epoch_list, loss)

plt.xlabel('number of epochs')

plt.ylabel('loss')
```

The plot in *figure 2.20* shows how the loss is reducing with epochs, which indicates that the predicted values are getting closer to the actual values:

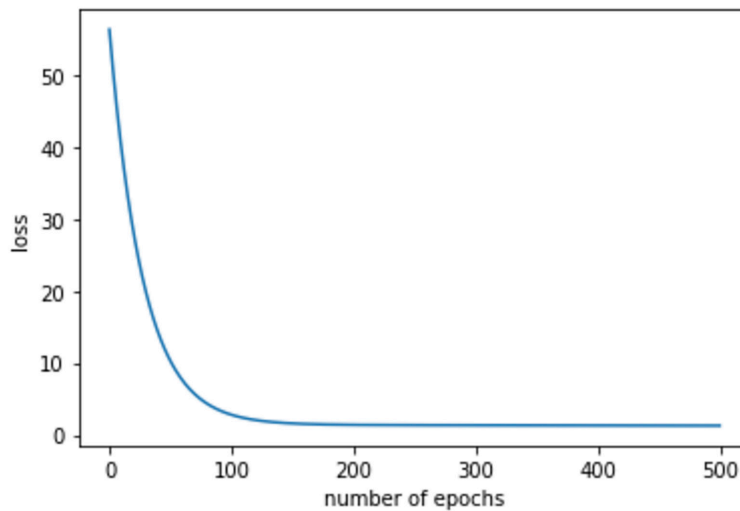


Figure 2.20: Plotting MSE loss versus epochs

We will take the value of w and b at the end of 500 epochs as their optimum value, and these are stored in a Python dictionary for prediction.

Finally, we will create a function to check how our linear regressor performs on the test data. The following `predictor()` function can be used for the prediction of y from new x values based on the training parameters.

```
>>params = {"weight":w, "bias":b}

def predictor(x):

    return params["weight"] * x + params["bias"]
```

Before prediction, we will check how good the linear regressor model is. The following code makes a scatter plot of the training data and plots the straight line based on the w and b values obtained via training the linear regressor:

```
>>plt.plot(x_train, y_train, '*')

x_train_sort = np.sort(x_train)

pred_final = predictor(x_train_sort)

plt.plot(x_train_sort, pred_final, 'r');

plt.xlabel("x values");

plt.ylabel("y values");
```

We can conclude that the resulting straight line is a good fit for the training data. See *figure 2.21*:

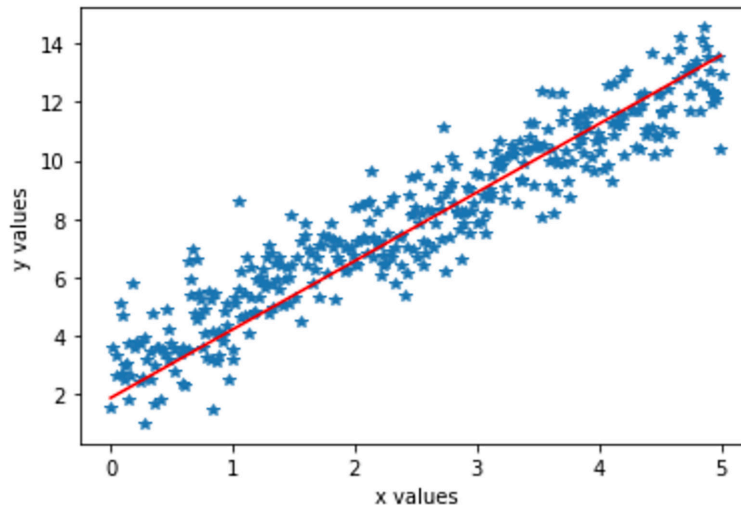


Figure 2.21: Fitting the straight line obtained by linear regression on the training data

The following code segment predicts the value of y on the entire test set using the function `predictor()`. It also calculates the MSE between the predicted and the actual values of y . Finally, it shows the predicted values on randomly selected five test data points along with the actual values for comparison.

```
>>y_test_pred = predictor(x_test)

Print("MSE on all test data", calc_mse(x_test, y_test,
params["weight"], params["bias"]).numpy())

# select 5 random test data and check the performance

indx = np.random.randint(0, 250, 5)

for i in range(len(indx)):

    print(f"predicted value : {y_test_pred[i]:3f}, actual value {y_
test[i]:.3f}")
```

Conclusion

Python is a very popular programming language in machine learning for researchers and also for developing production-ready software. A good understanding of Python is a prerequisite to fully understand the rest of the topics covered in the book. This chapter has been designed as a crash course to brush up on your Python programming knowledge before moving to the more complex part of programming in deep learning and TinyML applications. We assume that the readers have some fundamental knowledge of Python programming. Those who require an in-depth knowledge of Python and its various libraries are strongly encouraged to read the various resources available in print and also in digital mediums. The readers should try to execute the examples covered in this book at their end and are also strongly encouraged to modify, tweak, and extend them to gain more confidence in writing their own codes independently.

In this chapter, we have briefly covered the fundamental aspects of Python through examples. We have also discussed a few of the Python libraries, which are frequently required in building machine learning applications. Subsequently, we have briefly discussed about TensorFlow, an open-source library that is very popular in creating machine learning and advanced neural network applications. We have covered various datatypes of TensorFlow through examples. Finally, we have shown how TensorFlow can be used in creating an end-to-end machine learning application by creating a simple univariate linear regression model.

Throughout this book, we have used Google Colab notebook for writing and executing our Python scripts, which is entirely free to use, cloud-based, and does not necessarily require any software installation on your host machine. However, you need to be connected to the internet to get access to a remote Python runtime in order to execute your scripts. For experimental purposes, the readers are also encouraged to create the Python environment on their machine by installing the necessary libraries in order to run a few of the test examples. Python and TensorFlow can be easily installed on Windows, Linux, or macOS X computers. The readers can refer to the official website of Python and TensorFlow for detailed guidance for installation. The websites also contain detailed documentation and other various aspects of Python with examples, which the readers can find useful.

Key facts

- Python is a general-purpose, high-level, interactive, and interpreted program language popularly used in machine learning and data science.
- Python codes are portable and can be run on various operating systems, and can be easily integrated with other programming languages such as C++, JAVA, and so on.
- Google Colab is a browser-based Python IDE where you can easily run your programs on cloud.
- NumPy is a popular Python library for performing various mathematical operations on multi-dimensional arrays.
- Matplotlib allows to plot interactive graphs in Python.
- You can read and parse data from large CSV files and manipulate data using Pandas.
- TensorFlow is an end-to-end open-source platform for building and deploying machine learning and deep neural network applications.
- Tensors are multi-dimensional data array.
- Thanks to the graph execution mode, TensorFlow programs are portable across various platforms.

Further reading

1. Beazley, David, and Brian K. Jones. *Python cookbook: Recipes for mastering Python 3*. "O'Reilly Media, Inc.", 2013.
2. Pajankar, Ashwin. "Introduction to Python." In *Python Unit Test Automation*, pp. 1-17. Apress, Berkeley, CA, 2017.

3. Cutler, Josh, and Matt Dickenson. "Introduction to Machine Learning with Python." In *Computational Frameworks for Political and Social Research with Python*, pp. 129-142. Springer, Cham, 2020.
4. Joshi, Prateek. *Artificial intelligence with Python*. Packt Publishing Ltd, 2017.
5. Campesato, Oswald. *TensorFlow 2 Pocket Primer*. Stylus Publishing, LLC, 2019.
6. Singh, Pramod, and Avinash Manure. *Learn TensorFlow 2.0: Implement Machine Learning and Deep Learning Models with Python*. Apress, 2019.

Join our book's Discord space

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

<https://discord.bpbonline.com>



CHAPTER 3

Gearing with Deep Learning

Introduction

In Chapter 1, *Introduction to TinyML and its Applications*, we briefly talked about machine learning and deep learning. In this chapter, we will learn more about deep learning. Deep learning is a subset of machine learning that imitates the learning process of the human brain. Deep learning is possibly the hottest technology right now in computer science, which is heavily used by all major enterprises and start-ups in different applications. There is a subtle difference between deep learning and traditional machine learning. Traditional machine learning algorithms, such as linear regression, support vector machine, or random forest require structured data, termed as features, as inputs for making a prediction. That means raw unstructured data cannot be directly applied to a machine learning framework. Let us take a real-life example. Suppose you want to identify an animal in an input image. What you do? You look for certain unique properties, like the shape and size of the animal, its color, and some other distinguishing markers, such as the presence of whiskers or canine, and so on. These can be termed as the features of the animal.

We do a similar thing in traditional machine learning. The user needs to compute the features from the input data containing its salient properties as a basic pre-processing step. The features are mostly a set of numerical values. These are then applied to

the machine learning algorithm as input for performing a certain task, for example, classifying an object from an image or predicting an event. Machine learning algorithms can be supervised or unsupervised. As mentioned earlier, a supervised machine learning algorithm has two phases, training, and testing (evaluation). The machine learning model is created during training, and it is evaluated in testing. The performance of a machine learning application widely depends upon how discriminating the input features are. It can be expected that computing the relevant features manually from the input is often difficult, time-consuming, and may require application-specific domain knowledge.

Deep learning tries to eliminate the pre-processing step in machine learning by automatizing feature extraction. It can take unstructured data like rich media and entertainment data, geospatial data, and audio or text data as input and automatically extracts the relevant features during training. For example, you can directly train a deep learning algorithm with an ample number of labelled images of dogs and cats to create a classifier that can accurately classify cats and dogs from unseen images. The deep learning model will automatically determine the decisive features from the input data during training and use them for prediction. However, the extracted features might not be human interpretable.

Artificial Neural Network (ANN), commonly known as a neural network or a feedforward network, is the heart of deep learning models. An ANN comprises several connected layers, with each of them having multiple **nodes**, which are also called as **neurons**. There is an input layer, several intermediate hidden layers, and an output layer. Each node has an associated weight and a threshold. If the input to a node is above the threshold, the node is activated, and the information is passed to the next node in a forward direction, all the way toward the output. There are more complex deep neural network architectures like **Convolutional Neural Networks (CNN)** that have multiple layers to progressively extract higher-level features from the input. A CNN produces much superior performance than ANNs in image processing and computer vision applications.

Structure

In this chapter, we will discuss the following topics:

- Theory of Artificial Neural Network (ANN)
 - o Binary cross entropy loss function
 - o Neural network activation functions
 - o Learning the neural network weights—the backpropagation algorithm

- Introduction to Convolutional Neural Network (CNN)
 - Architecture of A CNN
 - Putting them all together
- Neural network hyperparameters

Objectives

In this chapter, we will briefly talk about the basic theory of neural networks along with some of the underlying mechanisms. We will start with the concept of a simple neural network, the activation functions, and the learning process of a neural network via backpropagation. Later, we will also discuss about **Convolutional Neural Network (CNN)**, a powerful deep learning algorithm popularly used in modern image processing and computer vision applications. The concept of neural networks involves lots of underlying mathematics. However, in this chapter, we will try to bypass the complex mathematical formulation as much as possible and only cover the basic fundamentals of neural networks for an easy understanding of the readers. The learning in this chapter will be required in the later part of the book to understand the TinyML projects, which will primarily use CNN as the machine learning model.

Theory of artificial neural networks

The concept of an artificial neural network, or ANN, is analogous to the human brain. The human brain has possibly the most powerful neural network. There are roughly 100 billions of neurons in our brain that form the central nervous system. Each neuron is connected to another 10,000 neighbouring neurons.

Figure 3.1 shows the image of a biological neuron. A neuron is an electrically excitable biological cell that accepts, processes, and transmits information through electrical and chemical signals. It works as an electrical cable that has three parts: the neuron cell body, the dendrites, and the axon. The electrical impulse arrives on the dendrites, gets processed on the cell body, and then moves to another neuron along the axon. At the end of the axon, the contact with the dendrite of the next neuron is made through a synapse. In order to enable us to react to a sudden change to our environment, the ascending neurons transport stimuli to the central nervous system, and in return, the descending neurons send the signal to the body muscles to react.

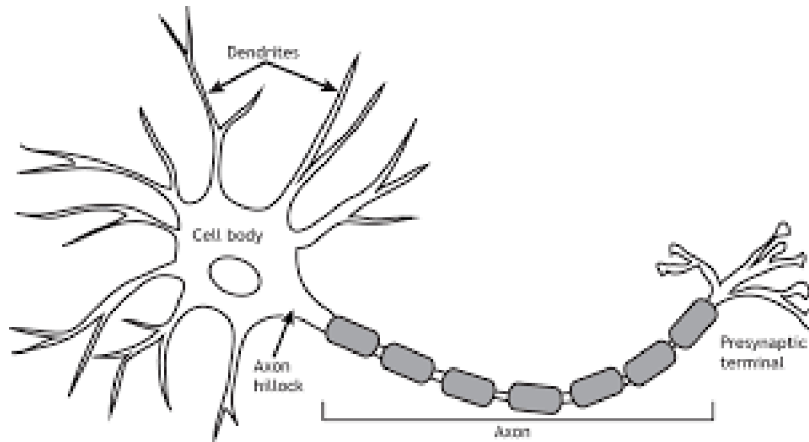


Figure 3.1: A biological neuron

Similar to the human neural network, an artificial neural network also consists of hundreds and thousands of artificial neurons. They are also called as nodes or units of the ANN. The neurons are internally connected to propagate information. Figure 3.2 shows the structure of an artificial neuron that can be considered as the core building block of an ANN:

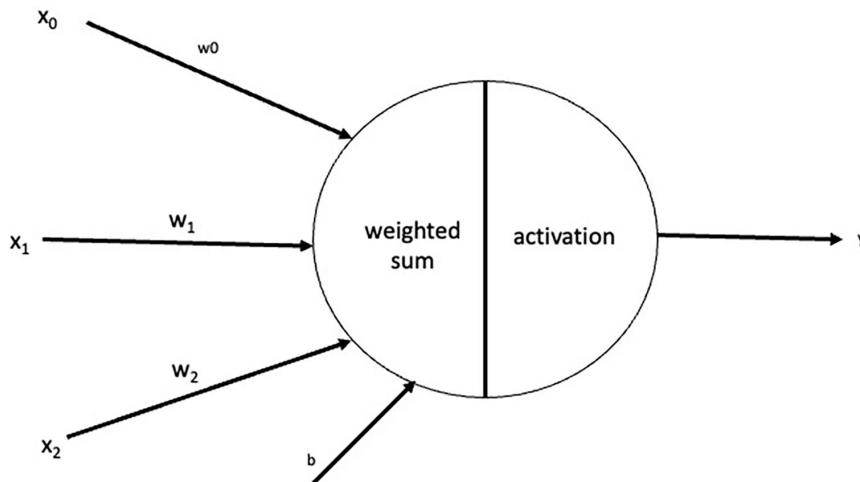


Figure 3.2: Architecture of a single neuron in an artificial neural network

The neuron in our example has three inputs, x_0 , x_1 , and x_2 . Each of them is associated with the corresponding weights: w_0 , w_1 , and w_2 , respectively. There can be an additional bias term, b . The weights determine the importance of each input applied to the neuron. The weight values and the bias are the trainable parameters in the neuron. Each neuron performs two operations. The inputs are multiplied by their

respective weights and are summed up together with the bias term to form the parameter z , where.

$$z = \sum_{i=0}^2 w_i \cdot x_i + b$$

Subsequently, z is applied to a function $f()$ to produce the output of the neuron, y .

$$y = f(z)$$

The function, $f()$, is called the **activation function** of the neuron. It typically decides whether a particular neuron remains active or inactive in the neural network. In general, non-linear activation functions are used. Later in this chapter, we will learn more about different activation functions used in neural networks.

Now, let us understand the structure of an artificial neural network. A neural network is a multi-layered structure where each layer contains several neurons. The output of a neuron in one particular layer goes as input to all the neurons in the following layer. Neural networks are used in machine learning for both classification and regression purpose. *Figure 3.3* shows the architecture of a multi-layered neural network for binary classification:

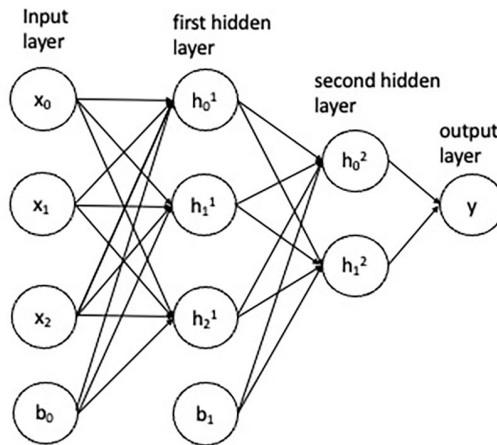


Figure 3.3: A neural network architecture

The multi-layered neural network shown in *figure 3.3* has four layers. The first layer is called the **input layer**, which represents the input feature. In *figure 3.3*, the input layer of the network takes three-dimensional feature as input and an additional bias term. Then comes two intermediate layers, which are termed as the **hidden layers** followed by the **output layer**. The hidden layers contain multiple neurons, as shown in *figure 3.2*. The hidden layers are responsible for extracting complex non-linear patterns from the input that are used to determine the output.

In our neural network, the first hidden layer has three neurons, and the second hidden layer has two neurons. Of course, you can have more hidden layers and higher number of neurons in each hidden layer. A network with more hidden layers is, in general, more complex in nature. The neurons in the hidden layers take their inputs and perform some non-linear mathematical transformation through an activation function. Finally, there is an output layer with a single neuron that has two distinct output values, a binary 0 or 1 indicating two different output classes (for example, cats versus dogs).

Now, let us understand some mathematical notations. The inputs to the network in the preceding figure is represented by x_i , and the bias term is represented by b . The neurons in the first hidden layers are represented by h_j^1 and the neurons in the second hidden layer are represented by h_k^2 . Here, i , j , and k represent the data dimension in the layers. In our example, $i=3$, $j=3$, and $k=2$. The information in the ANN is propagated in one direction from the input toward the output layer. Hence, this type of network is also called as **feedforward network**. Each neuron in a hidden layer is connected with all the neurons in the next layer. This kind of layer is called as **fully connected layers** or **dense layers**, and the network is called a **fully connected neural network**. As mentioned earlier. The inputs to a neuron are multiplied by their corresponding weights and are summed up along with the bias term before being applied to the activation function, which determines the output.

The network in our example has a single neuron in the output. Hence, it can only be used for binary classification. However, the output layer can be modified to solve a multi-class classification problem (for example, cats, dogs and horses).

Binary cross entropy loss function

In *Chapter 2, Crash Course on Python and TensorFlow Basics*, we created a linear regression model for predicting the value of a variable, y , from an input variable, x . In linear regression, the prediction is unbounded continuous numbers, such as the price of a property, salary of a person, or so on. Remember, we used the **Mean Squared Error (MSE)** between the predicted and the actual values of the datapoints in the training dataset as the loss function. The loss function was minimized during training to update the learning parameters. However, in *figure 3.3*, we have considered a neural network for binary classification. A classifier solves a different problem than a regressor. Instead of predicting continuous values, it predicts discrete values which are also called as class labels. For example, a binary classifier can be used to predict whether the animal present in an image is a cat or not, or for an incoming email, whether it is spam or not. For this kind of problem, we will use a different loss function, the **binary cross entropy (BCE)** or the log loss function.

The binary cross entropy loss function looks as the following:

$$J = - \frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i))$$

Imagine we want to create a classifier that will predict whether the input image contains a cat or not. For a given input, binary cross entropy tries to predict in a probabilistic manner how good the prediction is. Suppose y is the actual label (say 1 for cat and 0 for not cat) for a given input x . We denote $p(y)$ as the predicted probability of that data point being 1 for all N numbers of points in the dataset. For example, for an input image of a cat, predicting a probability 0.94 indicates it is a good prediction. Whereas a predicted probability of 0.2 will be considered as a bad prediction. For each positive example that is, for $y = 1$, the value of $\log(p(y))$ measures the log probability of predicting it as 1, and for each negative example that is, $y = 0$, $\log(1-p(y))$ is the log probability of predicting it as 0. Since the probability value lies between 0 and 1, the logarithmic value is always negative. Hence, the minus sign is added at the beginning of the preceding equation to measure the loss function, J , as a positive value. The loss value increases as the prediction probability diverges from the actual labels and becomes small as the predicted values get closer to the actual labels. During training, the loss function is minimized using some algorithms like gradient descent via updating the neuron weights.

For a multi-class classification problem with M different target classes, the expression for the loss function becomes:

$$J = - \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M y_{ij} \log(p_{ij})$$

In classification problems, the class labels are often represented by categorical values like strings or some abrupt numbers. In machine learning, we use a technique called one-hot encoding to represent the categorical class labels into numerical values. Here, the class labels are converted into array-like structures with numerical ordering to feed them to a machine learning model. For example, we have an image dataset having four different classes of cat, dog, tiger, and horse; the one-hot encoding for the different classes can be represented as $[0,0,0,1]$, $[0,0,1,0]$, $[0,1,0,0]$, and $[1,0,0,0]$.

Neural network activation functions

Previously, we have briefly talked about activation functions. An activation function determines whether a neuron in a network will be activated or not. All neurons in

a network are not activated. A neuron can propagate its information to the next layer only when it is activated. In other words, the activation functions perform the complex mapping between the input and the output variables. The simplest form of activation function is the linear activation function that takes an input and directly passes it as output. However, a linear activation function is not very useful in a neural network as it passes everything that comes in. This is where the non-linear activation functions come into the picture.

Non-linear activation functions help the neural network to learn more complex features from the data. Please note: an activation function must be differentiable to facilitate the training of the neural network. In the next section, we will discuss how a neural network is trained by calculating the gradient of the cost function with respect to the weights. A few popular activation functions commonly used in neural networks are explained as follows:

Sigmoid activation function

It is an S-shaped non-linear activation function whose values lie between 0 and 1. It can be represented by the following equation:

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

As shown in *figure 3.4*, the output value $f(z)$ tends toward 1 as the input value z moves toward positive, and $f(z)$ tends to 0 as z moves towards negative from 0. The value of $f(z) = 0.5$ when $z = 0$. Sigmoid activation function is typically used in the output layer of a neural network for binary classification.

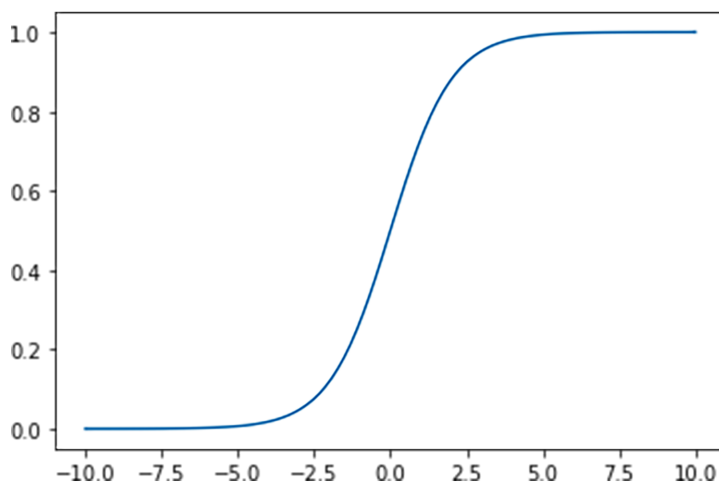


Figure 3.4: Sigmoid activation function

Tanh activation function

Hyperbolic tangent or tanh activation function is a shifted version of the sigmoid. Its output ranges between -1 and $+1$. The equation for tanh function is given by the following:

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

The corresponding plot is shown in *figure 3.5*. The function gives an output close to $+1$ for a positive input value z and returns close to -1 for a negative value of z . It returns 0 when $z = 0$. Tanh activation function can be used in the hidden layers in some neural networks. Since the output lies between -1 and $+1$, it helps in centering the data by bringing the mean close to 0 . This makes the training process easier.

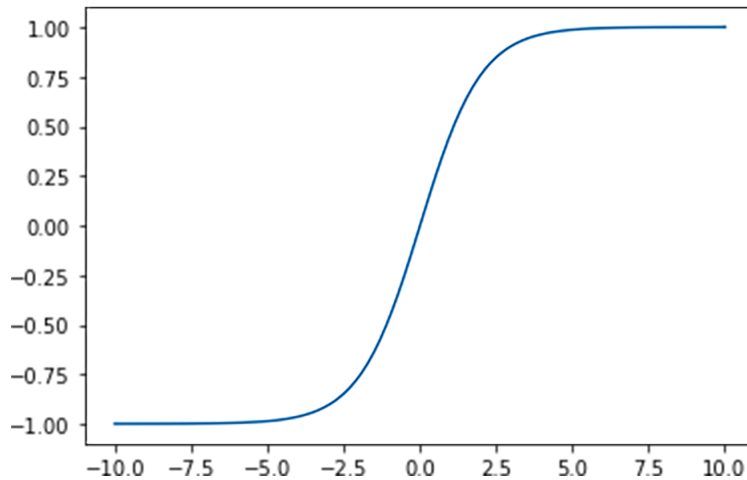


Figure 3.5: tanh activation function

ReLU activation function

One major limitation of the sigmoid and tanh activation functions is the vanishing of the gradients. These activation functions map large input values into a smaller range. Hence, a large change in the input will cause a small change in the output. As a result, the gradient of the functions is close to zero. Hence, there are situations in deep neural networks when there is virtually no update in the weights during training over the iterations. This situation is called the vanishing gradient problem. The problem can be resolved by using the **Rectified Linear Unit (ReLU)** activation function. The expression of ReLU is given by the following:

$$ReLU(z) = \max(0, z)$$

It returns the same input value as the output for a positive value of z and returns 0 otherwise. The shape of ReLU is shown in *figure 3.6*. It is the most popular activation function used in the hidden layer of deep neural networks.

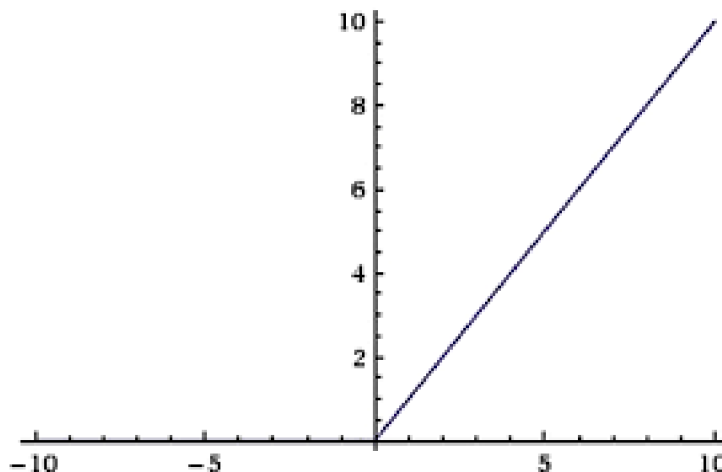


Figure 3.6: ReLU activation function

Softmax function

The softmax function takes a vector of k real values and returns a vector of k real values whose sum is equal to 1. Softmax function is typically used in the final layer of a multi-class neural network for predicting the probability of the target classes. For an input vector $z = [z_1, z_2, \dots, z_k]$, the softmax function is calculated as follows:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}}$$

Learning the neural network weights—the backpropagation algorithm

Neural networks are comprising of several interconnected layers from the input to the output. Each layer contains multiple neurons or nodes. The nodes have their inputs that are multiplied by the irrespective weights and are added up along with the bias term, which is then applied to an activation function to determine the output that propagates to the nodes in the next layer. The weight and the bias terms are the parameters of the network. A deep neural network can have several hundreds or thousands, or even millions of nodes. Now the question arises, how does a network

determine the weight values for the nodes? This is done by a learning algorithm termed as the **Backpropagation** algorithm.

The backpropagation algorithm was introduced in the 1960s, but it got popular in the 1990s in a famous paper by *David Rumelhart, Geoffrey Hinton, and Ronald Williams*, which described how backpropagation can be used for learning the weights of a neural network.

In this section, we will provide a very simple explanation of the backpropagation algorithm.

The concept of backpropagation is quite similar to the gradient descent used in *Chapter 2, Crash Course on Python and TensorFlow Basics*, to estimate the weights in linear regression. For a multi-layered neural network, the backpropagation algorithm calculates the gradient of the loss with respect to the weights in a backward direction from the output layer to the input.

Now, let us try to understand the backpropagation algorithm for a simple multi-layered feedforward neural network shown in *figure 3.7*. The network takes a single-dimensional input and passes it through two hidden layers and the output layer that predicts two classes, 1 and 0. Each hidden layer has one single node. The weights associated with the inputs to the hidden layers and the output layer are $w^{(1)}$, $w^{(2)}$, and $w^{(3)}$, respectively. The network takes an input x and passes it through all the layers to get an output $y1$. This is called **forward propagation**. The prediction error is minimized by backpropagation via adjusting the weights in different layers.

Let us assume the actual label for that input is y . The loss function is defined by $J(y,y1)$. For binary cross entropy we know that the loss function returns a very low value when the predicted value closely matches the target label. The weights are initialized by some random values. The objective of the training is to adjust the weights of the nodes at different layers in such a way that the loss function is minimized. In gradient descent, it is achieved in an iterative manner by calculating the derivative of the loss function J with respect to every weight in the network.

In a multi-layered network, the derivative of loss function with respect to a weight located in the middle of the network can be computed using the chain rule. This is given by the following:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$

The backpropagation algorithm first calculates the gradient with respect to the weights in the final layer. These are then used to calculate the gradient of the previous layers using the chain rule. This goes on all the way to the first layer.

Let us now consider the simple neural network taken in our example to understand the concept of backpropagation:

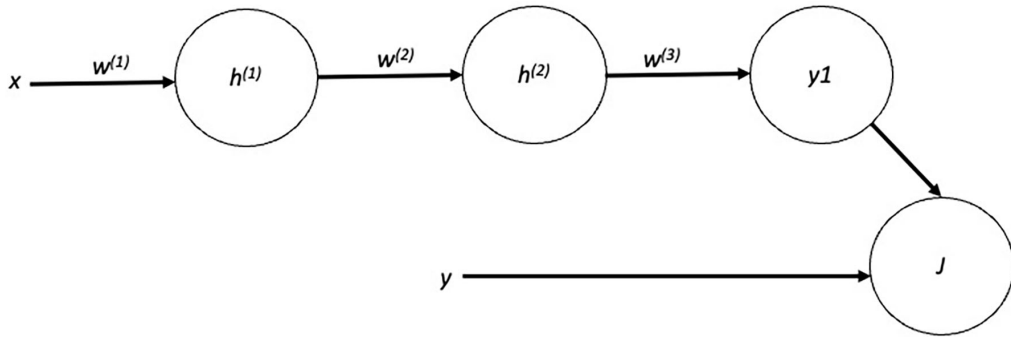


Figure 3.7: A simple neural network for backpropagation

As mentioned earlier, it has a single input x and two hidden layers. Each hidden layer is having a single node with weights $w^{(1)}$ and $w^{(2)}$, and the corresponding output $h^{(1)}$ and $h^{(2)}$. Let us assume $f()$ is the activation function for a hidden layer.

For the first hidden layer, the output can be calculated as follows:

$$h^{(1)} = f^{(1)}(w^{(1)} \cdot x)$$

For the second hidden layer:

$$h^{(2)} = f^{(2)}(w^{(2)} \cdot h^{(1)})$$

For the output layer, the predicted output is as follows:

$$y_1 = w^{(3)} \cdot h^{(2)}$$

For a binary classification problem, we measure the binary cross entropy as the loss function.

In backpropagation, we first calculate the gradients of the loss function with respect to the weight of the output layer $\frac{\partial J}{\partial w^{(3)}}$, then $\frac{\partial J}{\partial w^{(2)}}$, and finally $\frac{\partial J}{\partial w^{(1)}}$.

This can be done by applying the chain rule:

$$\begin{aligned} \frac{\partial J}{\partial w^{(3)}} &= \frac{\partial J}{\partial y_1} \cdot \frac{\partial y_1}{\partial w^{(3)}} \\ \frac{\partial J}{\partial w^{(2)}} &= \frac{\partial J}{\partial y_1} \cdot \frac{\partial y_1}{\partial h^{(2)}} \cdot \frac{\partial h^{(2)}}{\partial w^{(2)}} \\ \frac{\partial J}{\partial w^{(1)}} &= \frac{\partial J}{\partial y_1} \cdot \frac{\partial y_1}{\partial h^{(2)}} \cdot \frac{\partial h^{(2)}}{\partial h^{(1)}} \cdot \frac{\partial h^{(1)}}{\partial w^{(1)}} \end{aligned}$$

Once the gradients are calculated, the weights are updated by the following:

$$w^{(3)} = w^{(3)} - \alpha \frac{\partial J}{\partial w^{(3)}}$$

$$w^{(2)} = w^{(2)} - \alpha \frac{\partial J}{\partial w^{(2)}}$$

$$w^{(1)} = w^{(1)} - \alpha \frac{\partial J}{\partial w^{(1)}}$$

Here, α is termed as the learning rate. The weight values are initialized by random numbers at the beginning of the training. The backpropagation algorithms update them to the optimum values to reduce the loss function. The weights are updated through a number of epochs on the entire training dataset. The training is assumed to be completed when the loss function apparently reaches local minima, that is, the loss value does not significantly reduce further. At this stage, the weights are saved. The network with the saved weight can be used in the test phase, which can take unknown data as input for making predictions.

In this section, we have explained the backpropagation in the simplest way assuming a single-dimensional input and single node in each of the hidden layers. The concept can be logically extended for large complex neural networks containing multiple nodes in each layer. In that case, all the inputs to a particular node are first multiplied by their respective weights and are summed up, and eventually applied to the activation function to get the output.

Introduction to Convolutional Neural Network

So far, we have discussed the key aspects of artificial neural networks. Although multi-layered feedforward neural networks have been successfully used in various machine learning applications, they still have scalability issues in handling large input data in the form of image or video files. For example, suppose we want to design an image classifier using a single-layered feedforward neural network where the inputs are grayscale images of a resolution of 128×128 pixels. That means each image can be considered as a matrix of dots having 128 columns and 128 rows. The dots are called as pixels, and their value typically ranges between 0 and 255. For a grayscale image, a 0 pixel value represents the color black, 255 represents white, and the values in between represent various shades of gray. A value close to 255 indicates the pixel is closer to white. A grayscale image has a single channel. That means the

actual dimension of the image is $128 \times 128 \times 1$. For color images, the number of channel is 3 (Red, Green, and Blue).

Recall the structure of the feedforward neural network in *figure 3.3*. The input to the network is a 1D vector, not a matrix. Hence, an input image applied to that type of network first needs to be reshaped into a vector having $128 \times 128 \times 1 = 16,384$ elements. That means the input layer needs to have 16,384 nodes. Now, suppose we have a single dense hidden layer with 500 nodes. Remember, each node in the input layer is connected with all the nodes in the following hidden layers. So, it requires $16,384 \times 500 = 8,192,000$ connections and corresponding weight values between the input and the first hidden layer. In a practical scenario, we may need to add multiple hidden layers for a better performance. Hence, the size of the network exponentially increases along with the number of parameters. It takes lots of time to train such a huge network.

A convolutional neural network or CNN is a special type of neural network structure that automatically processes the input data to extract relevant features that are applied to the fully connected dense layers for classification. The main advantage of CNN is that you do not need to do much pre-processing on the input data, as CNN can take care of that. CNNs are popularly used in image processing, video processing, and computer vision applications. CNN has its own set of grid-like arrangements called filters that can extract the relevant features from the input via convolution. In computer-based digital image processing, we use various filters to control the brightness or contrast of an image, and to detect various edges inside it. A CNN tries to automate those steps, which are later used as features for performing a classification or a regression.

During training, a CNN itself discovers the best suitable filters for feature extraction for the specific task. As a result, the whole feature extraction process is automated, and it does not require hand-crafted feature computation. A major advantage of CNN and other deep learning algorithms is they can automatically extract the most relevant features from large unstructured datasets during training. This is particularly important on sophisticated computer vision problems like face recognition and other applications when you do not have enough domain knowledge to manually extract the key features from the input.

In 1989, *Lecun et al.* proposed the first CNN architecture that was trained using backpropagation for handwritten digit classification. CNN started getting more and more attention in the early 2010s when they outperformed other machine learning models in the annual competition of **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)** in the detection and classification of objects of various categories from millions of input images. The CNN architecture evolved with

different deep architectures, namely, AlexNet (2012), VGGNet (2014), GoogleLeNet (2014), and ResNet (2015).

Architecture of a CNN

The architecture of CNN varies depending on the application. However, a typical CNN comprises the following layers:

- Input layer
- Convolutional layer
- Pooling layer
- Fully connected layer or dense layer
- Output layer

Input layer

The input layer represents the input to the CNN. The input data can be 1D (time-series data), 2D (images), or 3D (videos). However, CNNs are primarily used for image processing-related applications. For digital images, the input shape is represented in $Width \times Height \times nChannel$ format, where $nChannel$ denotes the number of input channels. For grayscale images, $nChannel = 1$, and for color RGB images, $nChannel = 3$.

Convolutional layer

Convolutional layers are the heart of a CNN that are responsible for extracting relevant information from the input (image). Convolutional layers are composed of multiple filters or kernels that scan the entire image by performing convolution operations. For a 2D input image, the convolution filters are also required to be of 2D. Convolution has three operations, addition, multiplication, and shifting. Supposing our input image has a dimension of 128×128 , we want to perform the convolution using a 3×3 filter. The process is done in the following manner:

1. Place the 3×3 filter on the top left corner of the input image.
2. Multiply the pixel values with the corresponding filter value and add them up and return it as the central value for the scanned region.
3. Shift the filter to the right in horizontal and then in vertical direction.
4. Repeat Steps 2 and 3 until it scans the entire image.

In image processing, the convolution operation transforms an image by applying a filter over each pixel and its local neighbour. The size and the values of the filter determine the transformation.

Let us take an example to understand the convolution process more clearly. Suppose we have an input grayscale image of resolution $5 \times 5 \times 1$. In a simple way, let us assume the pixels form a 5×5 matrix of the following values, shown in *figure 3.8*:

30	30	30	0	0
30	30	30	0	0
30	30	30	0	0
30	30	30	0	0
30	30	30	0	0

Figure 3.8: Input matrix of shape 5×5

We also have a 3×3 filter with the following kernel values, shown in *figure 3.9*, to perform the convolution operation:

1	0	-1
1	0	-1
1	0	-1

Figure 3.9: The 3×3 kernel for convolution

Now, let us see a step-by-step manner of how to compute 2D convolution on that image using the filter:

1. In the first step of convolution, we place the filter on the top left corner of the input. Do an element-wise multiplication at each cell, sum up the multiplied values, and store the value in the top left corner of the output matrix. The scanned region is shown on the left, and the corresponding output in the right of *figure 3.10*:

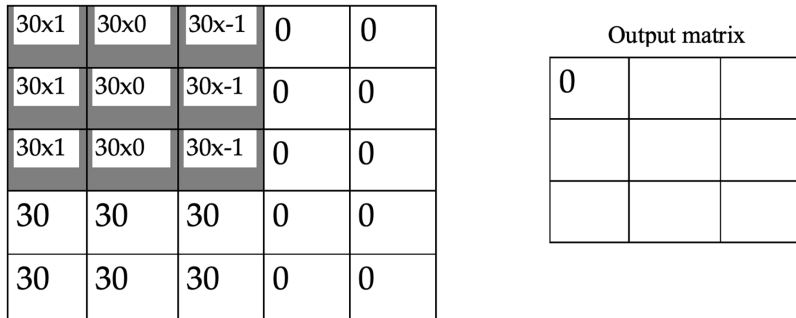


Figure 3.10: Step 1

- In Step 2, we shift the filter one pixel right to the input and repeat the operation in Step 1. The resulting value is stored in the second cell of the output matrix. Check out figure 3.11:

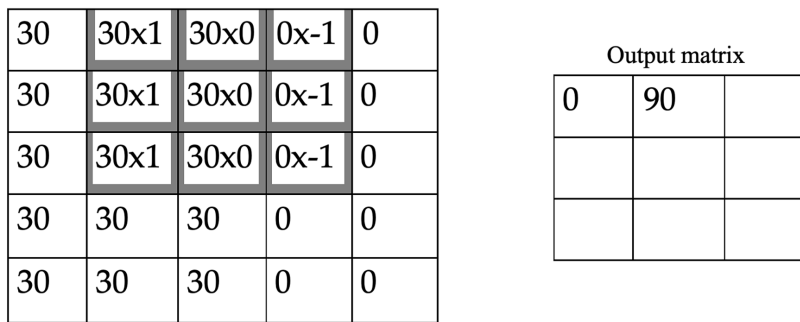


Figure 3.11: Step 2

- In the next step, we again shift one pixel right and repeat Step 1. The resulting value is stored in the third cell of the output matrix. This leads us to the following figure, figure 3.12:

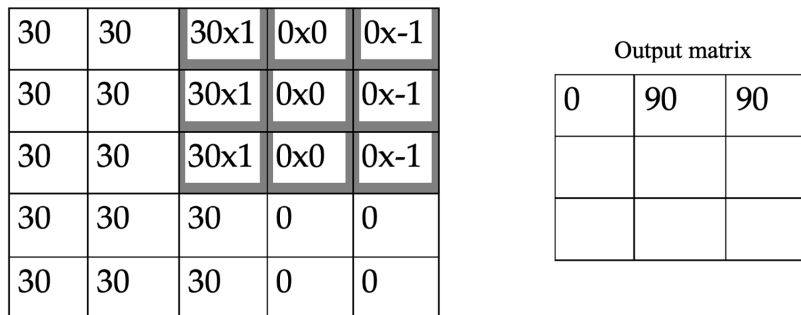


Figure 3.12: Step 3

4. In Step 4, the filter is moved one pixel down vertically. The output is stored in the first column of the second row, as shown in *figure 3.13*:

30	30	30	0	0
30x1	30x0	30x-1	0	0
30x1	30x0	30x-1	0	0
30x1	30x0	30x-1	0	0
30	30	30	0	0

0	90	90
0		

Figure 3.13: Step 4

5. In Step 5, we again move one pixel right. This leads us to *figure 3.14*:

30	30	30	0	0
30	30x1	30x0	0x-1	0
30	30x1	30x0	0x-1	0
30	30x1	30x0	0x-1	0
30	30	30	0	0

0	90	90
0	90	

Figure 3.14: Step 5

6. In Step 6, we again move one pixel right. This leads to *figure 3.15*:

30	30	30	0	0
30	30	30x1	0x0	0x-1
30	30	30x1	0x0	0x-1
30	30	30x1	0x0	0x-1
30	30	30	0	0

0	90	90
0	90	90

Figure 3.15: Step 6

7. In Step 7, we move one pixel down vertically. This leads us to *figure 3.16*:

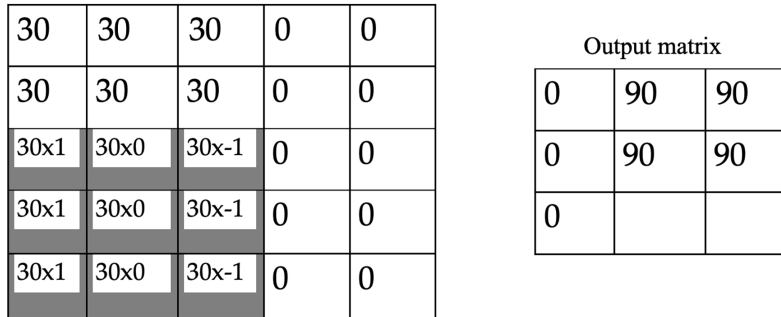


Figure 3.16: Step 7

8. In Step 8, we move one pixel right. This leads us to *figure 3.17*:

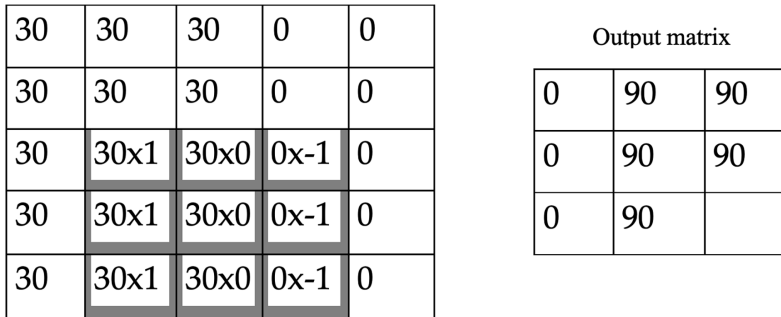


Figure 3.17: Step 8

9. In the final step, we move one pixel right to scan the entire image. This leads us to *figure 3.18*:

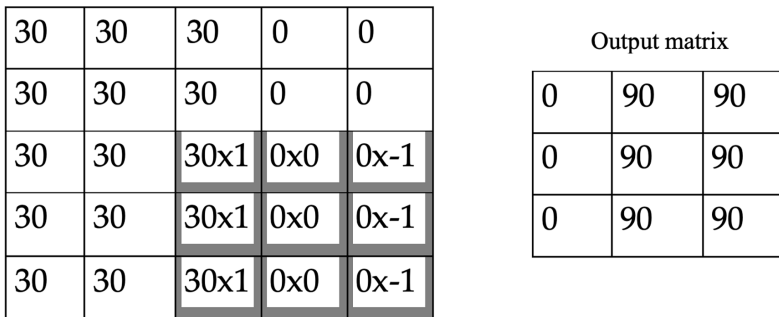


Figure 3.18: Step 9

Now, let us try to understand the impact of convolution on the image. Recall our input grayscale image in *figure 3.8*. it should visually look like *figure 3.19*:



Figure 3.19: Colour map of the input matrix

Here, the pixels with zero values represent black, and the non-zero pixels are gray. Since the non-zero values are much smaller than 255, the gray shades will be closer to black. Now after the convolution operation, the output matrix (Refer to *figure 3.18*) will look like *figure 3.20*:



Figure 3.20: Colour map of the output matrix after convolution

It can be seen from the preceding example that the output shape gets reduced after the convolution operation. For an input of $M \times M$, if we have the kernel dimension $f \times f$ of the filter for convolution, the output dimension becomes $(M-f+1) \times (M-f+1)$.

The convolution operation in a CNN is primarily responsible for simple image processing tasks like edge detection. By adjusting the weights of the filters, we can determine various horizontal or vertical edges in an image. A deep CNN can have multiple convolutional layers along with multiple filters at each level. The first few

layers are typically responsible for simple edge detection. The deeper layers extract more complex patterns specific to the application, like the extraction of human faces, locating whiskers of a cat, and so on, which are later used as discriminating features for making a classification. Remember, the kernel values are the trainable parameters in a CNN. We start with some random numbers as initial values and learn them on the training dataset.

Zero padding

In the previous example, we have seen that the convolution is done by scanning each pixel of the input image by placing the filter and shifting it in order to scan the entire image. However, not all pixels are scanned with equal intensity. It is evident that the pixels located around the center of the image are scanned for several times during shifting of the filter, whereas the pixels at the corners of the image are scanned only once. Suppose we are going to implement an object classification task. Now, if our target object is located at one of the corners that region will be scanned lesser during convolution. There is a popular technique in CNN where we add pixels with zero values around the original input image to increase its dimensionality. This is called zero padding, which ensures that all pixels in the original image are scanned more than once. It also ensures that the dimension of the output matrix remains identical to the original input even after convolution. In the previous example, if we apply p number of pixels at all sides of the image for zero padding, the new output dimension will be $(M+2p-f+1) \times (M+2p-f+1)$. If we wish to keep the output dimension identical to the input, the value of p will be $p = (f-1)/2$.

Strided convolution

In the previous example of convolution, we have shifted the filter by one pixel at every step in a horizontal and vertical direction. Hence, the stride length is 1. However, the stride length can be more than 1. If we select the stride length as s , the output dimension becomes $[(M+2p-f)/s+1] \times [(M+2p-f)/s+1]$. Strided convolution can be used to reduce the computational load on high-resolution images.

In the preceding examples, we have not considered the number of channels. We have assumed a single filter having a kernel dimension of $f \times f$ for convolution. There can be multiple filters as well. In the previous example, if the input image has a single channel (that is, input dimension $M \times M \times 1$), and we use k filters for convolution with zero padding and stride, the output dimension becomes $[(M+2p-f)/s+1] \times [(M+2p-f)/s+1] \times k$.

Once the convolution operation is done, the output matrix is converted to a non-linear transformation (mapping) by applying to an activation function. The ReLU activation function is commonly used in CNN.

Pooling layer

The pooling layer performs a down-sample operation for dimensionality reduction of the input. This is particularly important to reduce the computational load when we work on high-resolution image data. Pooling is applied to the output of the convolutional layers by sliding a rectangular filter of some size and calculating the maximum or average of the region of the input. Max-pooling and average-pooling operations are commonly used in CNN. In max-pooling, the pooling filter scans the input, and the maximum value corresponding to the scanned area is cropped as the representative value of that region. Similarly, in average pooling, the average value of the scanned area is marked as a representative value of the scanned area.

Suppose we have 6×6 dimensional input data shown as follows, where we apply pooling using a 2×2 filter. The corresponding outputs for max-pooling and average-pooling are shown as follows. The color-coding is used to show the scanned area in the input by the 2×2 pooling window and the corresponding pooled values in the output. The output dimension becomes 3×3 .

This can be seen in *figures 3.21* and *3.22*:

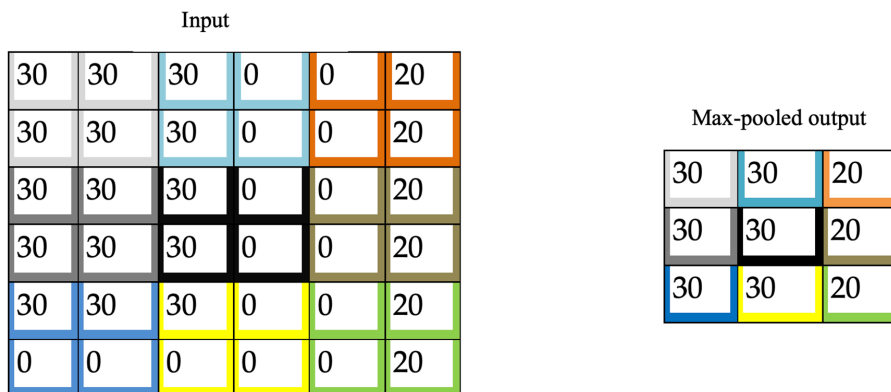


Figure 3.21: Example of max-pooling

Input					
30	30	30	0	0	20
30	30	30	0	0	20
30	30	30	0	0	20
30	30	30	0	0	20
30	30	30	0	0	20
0	0	0	0	0	20

Average-pooled output		
30	15	10
30	15	10
15	7.5	10

Figure 3.22: Example of average-pooling

Fully connected layer or dense layer

A fully connected layer or a dense layer is similar to the hidden layers of a feedforward neural network. The output of the convolutional and the pooling layers are flattened to form a 1D vector and applied to the fully connected layer. Each input to the layer is connected to all the nodes of the fully connected layer. There can be more than one fully connected layer in a deep CNN architecture. The nodes are activated by non-linear activation functions like ReLU or tanh.

Output layer

The output layer uses a sigmoid activation function for binary classification or a softmax activation function for multi-class classification. The output function implements a loss function like binary cross entropy (for binary classification) or categorical cross entropy (for multi-class classification). The network is trained end-to-end to minimize the loss function using backpropagation, by therefore learning the parameters in different layers.

Putting them all together

Now, we have learnt about the different layers of a CNN. Let us now see how a complete CNN looks like. Figure 3.23 shows the end-to-end architecture of a typical CNN for image classification. The dimension of different layers and the corresponding output dimensions are duly mentioned for a better understanding.

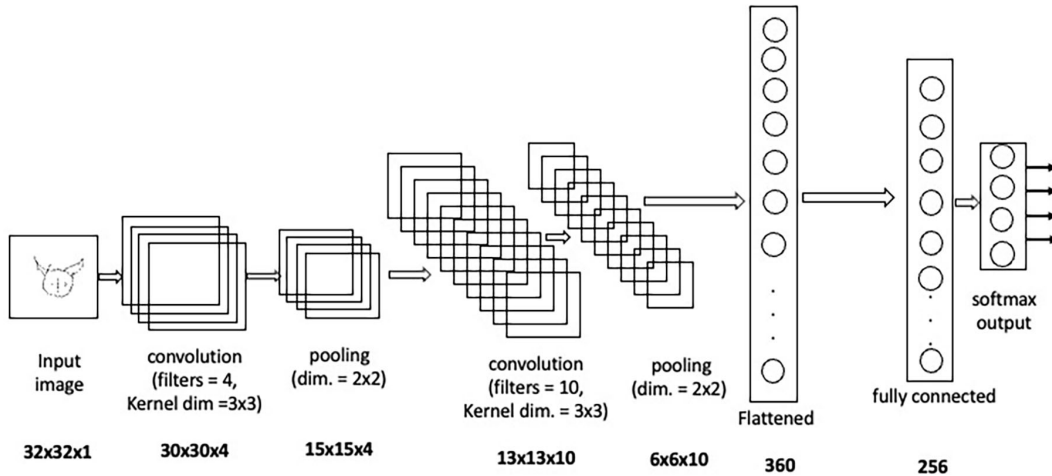


Figure 3.23: End-to-end architecture of a CNN

The architecture contains multiple convolution and pooling layers. The input image has a dimension of $32 \times 32 \times 1$. First, the input image is applied to a convolutional layer with four filters and kernel dimension 3×3 . The output dimension after convolution becomes $30 \times 30 \times 4$. The output is then passed through a pooling layer having 2×2 pooling window for dimensionality reduction. Next comes a second convolutional layer with 10 filters with a kernel dimension of 3×3 and an associated pooling layer. The output dimension becomes $6 \times 6 \times 10$. The output is flattened and applied to a fully connected layer having 256 nodes, followed by the output layer for predicting four output classes using a softmax function.

In a CNN, the kernels of the filters in the convolutional layers and the weight values of the nodes in the fully connected layers are the trainable parameters. The weights are initialized by random numbers. There are associated bias terms in all layers. Then we apply backpropagation to update the weights and bias in different layers in such a way that the loss function is minimized.

Neural network hyperparameters

A neural network, more specifically, a deep neural network has multi-layered architecture. Even a simple feedforward network can have many hidden layers in between the input and the output layer. A typical deep CNN architecture has more than one convolutional layer with associated pooling layers to generate a high-dimensional non-linear feature map which is applied to fully connected layers, followed by the output layer. In general, a deeper network can learn more detailed features than a shallower network. But, this may often cause an undesired

performance. It is highly possible that a very deep network having many intermediate layers will fail on an unseen test dataset despite performing well on the training set. This condition is called an overfit, which is a common problem one might face while training a deep neural network.

There is a set of parameters whose values can be controlled during training to determine the performance of a network. They are called network hyperparameters. Hyperparameters are tuneable, and they heavily influence network performance. A neural network can have various hyperparameters. A few of them are as follows:

Number of layers

A shallower network is easy to train. The loss function easily converges, but it may often fail to learn the properties of the training dataset. Hence, it fails on both training and test data. This condition is called an underfit. Similarly, a deeper network can learn more important features, but there is always a chance of an overfit. The number of layers is a critical hyperparameter that determines the performance of a neural network. You may start with a shallower network and gradually add more layers to it to check the performance improvement.

Learning rate

The learning rate during training is another important hyperparameter. It is a factor that determines the speed at which the network weights are updated in each iteration. While minimizing the loss function using gradient descent, it is assumed that the loss function has at least one optimum minima value, and gradient descent tries to reach that through iterations. However, in practice, the loss function can be a non-monotonous function, and it can have many sub-optimum local minima points. The learning rate can be between 0 and 1. If a very small learning rate is selected, the network will take many iterations to minimize the loss, and the learning process may be stuck. Similarly, if a very large learning rate is selected, the weight updates will be so big that the loss function might never reach the local minima but converge to suboptimal minima.

Dropout

Dropout is a popular way to ensure a large network is not overfitted. Many nodes in a deep network are often redundant. Using dropout, we can simply remove some nodes in each layer of a network. Which nodes have to be removed are determined in a random manner. During training, we specify a dropout amount and the layers where the dropout will be applied. We pass a hyperparameter called *keep probability*

(p) which assigns a dropping probability of $1-p$ to the nodes. Those nodes are removed randomly from training. Dropout is a very efficient way to train deep neural networks that avoid overfitting.

Regularization

This is another simple strategy of avoiding overfitting by controlling the weight values. Two popular regularization techniques used in neural networks are L1 and L2 regularization. The L1 regularization tries to shrink the weight values to zero, whereas the L2 regularization tries to minimize the weight values to non-zero values. The regularization terms are added to the original loss function ($J(w)$) to form a modified loss function, which is minimized during training. The corresponding equations for L1 and L2 regularization are as follows:

$$\text{Loss_L1} = J(w) + \lambda \sum |w_i|$$

$$\text{Loss_L2} = J(w) + \lambda \sum w_i^2$$

In L1 regularization, the absolute values of the weight are summed up and added to the loss function. In L2 regularization, the square value of the weights is summed up and added with the loss function. The parameter, λ is another hyperparameter that determines the importance of regularization in the loss function.

Choice of optimization algorithm

We have only discussed about gradient descent as the optimization algorithm to minimize the loss function. However, there are other popular optimization algorithms like stochastic gradient descent, RMSProp, and Adam that provides adaptive learning rate. The choice of optimization algorithm greatly influences the network performance.

Mini-batch size

Another important hyperparameter is the mini-batch size. While training a neural network on a large dataset, we often break it into smaller mini-batches. In each iteration, we compute the gradient of the loss of one mini-batch and update the weights. We can significantly reduce the training time by breaking large datasets into smaller min-batches.

Other popular hyperparameters are as follows:

- Number of epochs, that is, how many times you cover the entire dataset to update the weights
- How the initial weight values are set
- Network activation functions, and so on

A few hyper-parameters specific to CNN are as follows:

- Number of convolutional layers
- Number of output filters in each layer
- Filter dimension for convolution
- Pooling window dimension, and so on

Conclusion

Neural networks are a set of popular machine learning algorithms that mimic the learning of the human brain. In this chapter, we have briefly covered the key aspects of neural networks. We have started with the concept of simple multi-layered feedforward neural network and gradually moved on to **Convolutional Neural Network (CNN)**, a popular deep neural network architecture commonly used in modern image processing and computer vision applications. CNNs and related networks are heavily used in modern TinyML applications, and we will also implement few of them in subsequent chapters in our TinyML projects. One thing to remember is that deep learning architectures are large in size and often resource-hungry. Hence, they need to be significantly optimized depending on the capacity of the target platform, which is particularly evident in TinyML applications. In the upcoming chapter, we will learn to use TensorFlow to create our first neural network architecture for a real-world application. In *Chapter 5, Model Optimization Using TensorFlow*, we will see how a large neural network can be compressed to run on tiny edge devices and other smaller devices for TinyML applications.

Key facts

- Neural networks are powerful machine learning algorithms that mimic the learning mechanism of the human brain.
- A neural network is a multi-layered feedforward architecture containing multiple neurons or nodes that propagate processed information from input to output.

- The activation function determines which set of neurons in a neural network will be activated.
- Each input to a node is associated with a weight. Weights are the trainable parameters in a neural network.
- The neural network learns its weight values during training using the backpropagation algorithm.
- A CNN is a popular deep neural network architecture commonly used in image processing and computer vision applications.
- The core layers in a CNN are the convolutional layer, pooling layer, fully connected layer, and output layer.
- The convolution filters are responsible for extracting the relevant patterns from the input through convolution filters without manual pre-processing.

Further reading

1. Roberts, Daniel A., ShoYaida, and Boris Hanin. *The Principles of Deep Learning Theory: An Effective Theory Approach to Understanding Neural Networks*. Cambridge University Press, 2022.
2. Weidman, Seth. *Deep learning from scratch: building with Python from first principles*. O'Reilly Media, 2019.
3. Rivas, Pablo. *Deep Learning for Beginners: A beginner's guide to getting up and running with deep learning from scratch using Python*. Packt Publishing Ltd, 2020.

Join our book's Discord space

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

<https://discord.bpbonline.com>



CHAPTER 4

Experiencing TensorFlow

Introduction

In the previous chapter, we briefly discussed the key aspects of **Artificial Neural Networks** (ANN) with a focus on **Convolutional Neural Network** (CNN). In this chapter, we will implement our first neural network architecture to solve a real-world problem: the classification of handwritten digits on a publicly available database. We will primarily use TensorFlow to implement the neural networks. In *Chapter 2, Crash Course on Python and TensorFlow Basics*, we briefly discussed the basics of TensorFlow. To summarize, TensorFlow is an end-to-end open-source software platform containing libraries for implementing machine learning applications. TensorFlow has been extremely popular in recent days in designing large and complex deep learning architectures. In TensorFlow, when a machine learning model is defined, it internally creates a dataflow graph comprising a series of connected nodes. Each node represents a mathematical operation, and each connection represents a multi-dimensional array, also called as tensor. Although TensorFlow 2 allows eager execution, the graph execution mode has made it faster, flexible, and robust. You can easily execute a TensorFlow 2 program in graph mode. In graph mode, a model can be restored without the original Python code and can even be deployed into a device without the Python environment.

One of the biggest benefits of using TensorFlow is the level of abstraction. It takes care of the major details of most of the underlying algorithms in a machine learning or a deep learning application, for example, backpropagation. Hence, writing programs in TensorFlow is super easy as you mostly need to focus on your application logic. TensorFlow applications can run on almost any target environment, such as local desktops, remote servers running Windows, Linux, macOS X, smartphone devices running Android and iOS, or Linux-based single board computers. TensorFlow contains additional libraries to convert a machine learning model into C++ equivalent libraries to run on selected microcontrollers. Hence, you can use TensorFlow to create your TinyML applications. Throughout this book, we will primarily use Keras to implement the machine learning models. Keras is a high-level set of Python APIs in TensorFlow, which is popularly used in the rapid prototyping of various neural network models. Fortunately, both TensorFlow and Keras are readily available in Colab. So, you can easily start writing your program in Colab without installing any extra libraries. In this chapter, we will primarily focus on creating end-to-end neural network models using TensorFlow. The later chapters will focus on optimizing the neural networks to create deployable TinyML applications.

Structure

In this chapter, we will discuss the following topics:

- Keras and TensorFlow
- Classification of handwritten digits using a feedforward neural network
 - Data processing
 - Model implementation
- Implementation of a Convolutional Neural Network
- Evaluation metrics in the classification model

Objectives

The objective of this chapter is to have experience the power of TensorFlow in implementing neural network models. In *Chapter 2, Crash Course on Python and TensorFlow Basics*, we created a simple linear regression model using TensorFlow. In this chapter, we will learn how TensorFlow can be used to create large neural network applications. Remember, we implemented all the necessary functions for linear regression in *Chapter 2, Crash Course on Python and TensorFlow Basics*. However,

a major advantage of TensorFlow is the availability of various functionalities to create machine learning models. In this chapter, we will primarily use the TensorFlow inbuilt functions to implement all the major building blocks of the neural networks. For this, we will use Keras, a specially designed set of TensorFlow APIs for quick implementation of neural network models in Python. This chapter can be broadly divided into two subparts. We will first implement a simple feedforward artificial neural network comprising two dense layers for the classification of handwritten numerical digits on a freely available public database, the MNIST database. Later, we will implement our first CNN model that yields an improved classification performance on the same dataset. We will also briefly talk about some metrics used for the performance evaluation of machine learning algorithms.

Keras and TensorFlow

Keras is an open-source high-level neural network library written in Python, which runs on top of TensorFlow. Keras offers simple, flexible, and powerful APIs for implementing neural network applications. It was primarily developed for rapid experimentation and prototype designing. With Keras, you can create your deep learning architecture using a few lines of code. Keras provides a high level of abstraction for developing and shipping of machine learning applications. You can create almost all popular deep learning architectures by calling the APIs without even implementing the complex underlying functionalities. Keras comes as a high-level set of APIs directly integrated with TensorFlow 2, which can be directly called from TensorFlow 2. It enables you to exploit the full advantage of the scalability and cross-platform capabilities of TensorFlow. Hence, Keras can be run on GPUs, TPUs, and also on mobile platforms.

The Keras architecture has three main categories:

1. models
2. layers
3. core modules

Every neural network is a Keras **model**, and every model is a composition of Keras **layers** (such as the convolutional layer, pooling layer, dense layer, and so on). The **core modules** contain different activation functions, loss functions, regularization parameters, and so on. In *figure 4.1*, a simple ANN structure is illustrated, which we will implement in Keras:

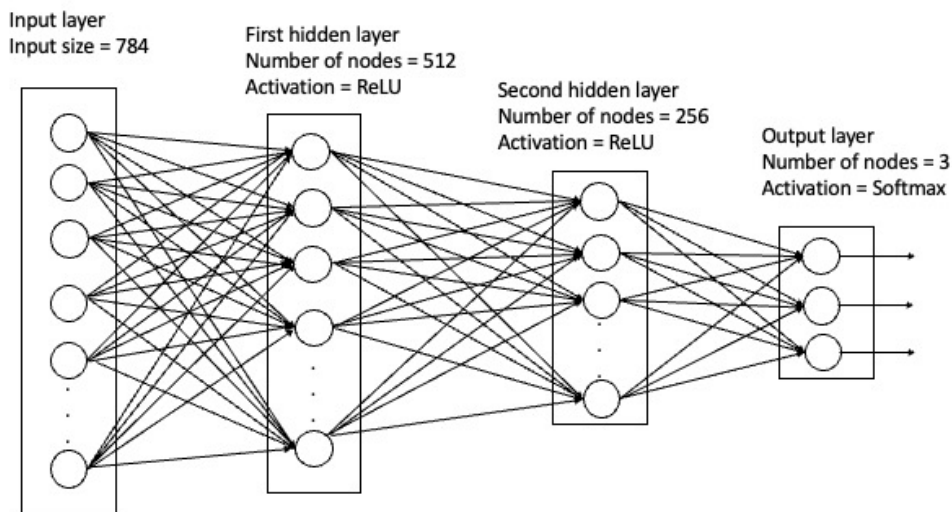


Figure 4.1: Sample ANN structure

In the preceding figure, we show a simple multi-layered feedforward ANN structure for a three-class classification problem. The input has a feature dimension of 784. Next comes two fully connected hidden layers with 512 and 256 number of nodes, respectively, having ReLU activation function. Finally, we have the fully connected output layer with three nodes for prediction using a softmax activation function. Most of the neural networks covered in this book will fall under the **Sequential model**. It is a linear composition of Keras layers. Different Keras layers can be stacked in a Sequential model using the `add()` function to define the architecture.

As mentioned earlier, Keras APIs are directly integrated with TensorFlow 2. Since TensorFlow 2 is preinstalled in Colab, you do not need to install them separately. If you want to install TensorFlow on your system for offline code execution, you may refer to the official website of TensorFlow¹.

Now, let us implement the ANN structure shown in *figure 4.1* as a Keras Sequential model.

Let us open a new notebook in Colab and save it as a new project. We will start writing our program by importing the necessary APIs and then defining a Sequential model. Refer to the following code:

```
>>>from tensorflow.keras.models import Sequential

    model = Sequential()
```

1 <https://www.tensorflow.org>

Next, we will add three dense layers. In order to add a layer to a sequential model, we need to call `model.add()` and provide the details of that layer. In the first layer of any Keras model, we must provide the shape of the input data. Please see the following code:

```
>>from tensorflow.keras.layers import Dense

    model.add(Dense(512, activation = 'relu', input_shape = (784,)))

    model.add(Dense(256, activation = 'relu'))

    model.add(Dense(3, activation = 'softmax'))
```

The ANN comprises three fully connected dense layers, including two hidden layers, and the output layer. In each dense layer, we have mentioned the number of nodes and the activation function. We have used the ReLU activation in the hidden layers and softmax in the output layer. Once the model is defined, we can see a model overview by calling `model.summary()`. Execute the following command in a new cell.

```
>>model.summary()
```

It will print the following output in *figure 4.2*, which states our model is a sequential model. It displays the tensor shape at the end of different layers and also the number of trainable parameters in the network.

```
Model: "sequential"
-----
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 512)	401920
dense_1 (Dense)	(None, 256)	131328
dense_2 (Dense)	(None, 3)	771

```
-----
Total params: 534,019
Trainable params: 534,019
Non-trainable params: 0
-----
```

Figure 4.2: Output of model.summary() in Keras

Once our model is defined, we need to configure the learning process by setting the training parameters, for example, the loss function. Then we can fit the model on training data to initiate the learning process that will give us the trained model. Subsequently, the model can be evaluated on test data. We will learn more about these steps once we implement a real model.

Classification of handwritten digits using a feedforward neural network

In the previous section, we have learnt how to define a simple neural network in TensorFlow using Keras APIs. Now, we will develop our first neural network application to solve a real-world application of classifying handwritten numerical digits. The first thing you require in order to implement a machine learning model is an appropriate dataset. For this application, we will use a publicly available database, the MNIST database.

MNIST is a popular entry-level database for machine learning beginners and researchers for doing experiments. It contains labelled images of handwritten digits. The images in the database are of relatively smaller dimensions, and hence, require minimum pre-processing. Because of smaller dimensions, it also takes comparatively lesser time to train and evaluate machine learning models, making the dataset ideal for performing various experiments. The images in the MNIST database are divided into training and test sets. The training set has 60,000 image examples, and the test set has 10,000 images of handwritten digits. All the images have been size-normalized and centered in fixed-size images. Hence, you can directly work on them without much pre-processing. The images are available in grayscale with dimensions of 28×28 . The images are handwritten numerical digits from 0 to 9, which are fully annotated. Hence, there are ten different class labels in the database. MNIST is readily available in Keras. Hence, you can easily load the database with a single function call.

Go back to your Colab notebook. We will load the database using Keras using the following lines of code:

```
>>from tensorflow.keras.datasets import mnist

#load the dataset

(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

On executing the preceding code snippet, the MNIST database will be loaded into your workspace. The variables `X_train` and `X_test` will store the images corresponding to the training and test set, and `y_train` and `y_test` store the values corresponding to class labels. Now, let us check about the shape of the variables. Execute the following code in a cell:

```
>>print(' shape of training data ',X_train.shape)

print('shape of training labels ',y_train.shape)
```

```

print('shape of training data ',X_test.shape)
print('shape of training labels ',y_test.shape)
print('type of X_train ',type(X_train))
print('type of y_train ',type(y_train))

```

The code segment will show the shape of different data variables in the training and test set along with the datatype, as shown in *figure 4.3*:

```

shape of training data (60000, 28, 28)
shape of training labels (60000,)
shape of training data (10000, 28, 28)
shape of training labels (10000,)
type of X_train <class 'numpy.ndarray'>
type of y_train <class 'numpy.ndarray'>

```

Figure 4.3: Details of the MNIST database

It can be seen that the images in the MNIST database are stored as NumPy arrays. Each image sample in the training and test set has a dimension of 28×28 pixels of 8-bit unsigned integers (uint8). Hence, each pixel value ranges between 0 and 255. The variables `y_train` and `y_test` contain a single-valued number between 0 and 9 corresponding to the class labels of the images.

Now, let us plot a few sample images from the training set for a better understanding of the database. The following code snippet randomly selects nine images from the training set and plots them in grayscale.

```

>>import numpy as np

import matplotlib.pyplot as plt

from numpy import random

plt.rcParams['figure.figsize'] = (4,4)

for i in range(9):

    plt.subplot(3,3,i+1)

    num = random.randint(0, len(X_train))

    plt.imshow(X_train[num], cmap='gray')

    plt.title('Class {}'.format(y_train[num]))

plt.tight_layout()

```

The output of the preceding code in one run is shown in *figure 4.4*. Since the samples are drawn at random, you can expect different sets of output each time you execute the code. Apart from plotting the sample images, the corresponding class labels are also shown. The digits are located centrally in the images in white with a dark background. An important thing to note is there can be multiple different drawing patterns for a particular digit in the database. See the two examples of the digit 2 in *figure 4.4*. They look completely different. This is absolutely normal as human handwriting widely varies for different persons. It is always a good idea to introduce diversity in the training set in order to create a generalized machine learning model so that it does not fail on unseen test images. When you create your own dataset, it is always recommended to have some levels of variation in the training set so that it covers the full range of the data belonging to a particular class label.

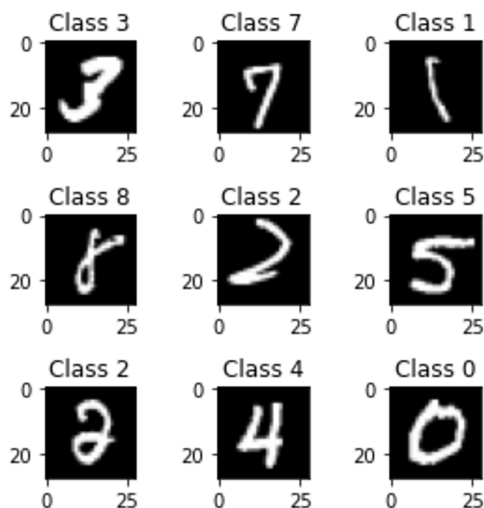


Figure 4.4: Plotting random samples from MNIST

Now, let us create our first neural network to create a classifier on MNIST. As mentioned earlier, we will first create a simple feedforward ANN classifier and later design a CNN structure.

Data processing

As discussed in *Chapter 3, Gearing with Deep Learning*, an ANN-based on dense layers, takes 1D data vectors as input. Hence, the pixel values need to be reshaped from a 28×28 matrix to a vector of $28 \times 28 = 784$ values. Each point in the vector represents a feature, which goes to the network as the input. We will do one basic pre-processing on the input images in both training and test set. The pixel values will be scaled

down between 0 and 1. This is done by dividing the pixel values by 255. The process is called data normalization, which helps in reducing the scales of the input feature values to ensure a better and faster training. After the pre-processing, the input images will be converted into 32-bit floating point numbers from 8-bit unsigned integers. The pre-processing code is shown as follows:

```
>>#reshape the training and test data into 1D tensors

X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)

# convert into 32-bit floating point numbers
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

X_train /= 255                    # normalize the input
X_test /= 255
```

Since we have ten different target classes to predict, the output layer of the ANN will have ten nodes. We also need to modify the class labels from their integer values into one-hot encoded vector in order to compute the cross entropy loss during training. Keras has an inbuilt function, `to_categorical()`, that converts the class labels into the equivalent one-hot vectors. Refer to the following code:

```
>>from tensorflow.keras.utils import to_categorical

num_class = 10

print('label of 100th instance in training data: ', y_train[100])
print('label of 500th instance in test data: ', y_test[500])

y_train = to_categorical(y_train, num_class)
y_test = to_categorical(y_test, num_class)

print('label of 100th instance in training data one hot encoded: ',y_
train[100])

print('label of 500th instance in test data one hot encoded: ', y_
test[500])
```

The output of the preceding code segment is shown in *figure 4.5*. In the code example, we have taken the 100th sample from the training data, which has a class label of 5, and the 500th sample from the test data, which has a class label of 3, and shown how the output looks after one-hot encoding.

Note: One-hot encoding can be applied on string class labels as well, for example, an image dataset containing labelled images for cats, dogs, and horses. We will explore that in *Chapter 6, Deploying My First TinyML Application*.

```
label of 100th instance in training data: 5
label of 500th instance in test data: 3
label of 100th instance in training data one hot encoded: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
label of 500th instance in test data one hot encoded: [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
```

Figure 4.5: One hot encoding of the class labels

Model implementation

Once we are done with loading and pre-processing of the data, we can define our neural network architecture.

A Keras model has the following four stages:

1. **Defining the model:** We define a Sequential model and add the necessary layers.
2. **Compiling the model:** We configure the model for training by defining the loss function to be minimized, the optimizer that minimizes the loss function, and a performance metric to internally evaluate the performance.
3. **Fitting the model:** Here, we fit the model on the actual training data for a given number of epochs to update the training parameters. We will get the model at the end of training.
4. **Evaluation:** Once you have the model, you can evaluate on unseen test data.

We will now implement a simple feedforward ANN. Our ANN has a single input layer and two hidden dense layers, followed by the output layer for classification. The two hidden layers have 512 and 128 nodes, respectively. The ReLU activation function is used in the hidden layers. The output layer has 10 nodes for classifying 10 different classes using a softmax function. As mentioned before, we will define a Sequential model and add all the layers. Remember, in the first layer, we must provide the dimension of the input data. In our case, the input has $28 \times 28 = 784$ values. The following code:

```
>>>from tensorflow.keras.models import Sequential
```

```

from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(512, activation = 'relu', input_shape = (784,)))
model.add(Dense(128, activation = 'relu'))
model.add(Dense(num_class, activation = 'softmax'))

```

Execute `model.summary()` to get an overview of the model, the output shape at each layer, and the number of trainable parameters. Refer to *figure 4.6*:

```

Model: "sequential_4"

```

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 512)	401920
dense_5 (Dense)	(None, 128)	65664
dense_6 (Dense)	(None, 10)	1290

```

Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0

```

Figure 4.6: Summary of the ANN model for the classification of handwritten digits

Now, we can train our model. We will first configure our model for training by specifying the loss function and the optimizer, and the evaluation metric. Since we are dealing with a multi-class classifier problem, we will use a loss function readily available in Keras, the *categorical cross entropy* loss function. We will use Adam as an optimizer to minimize the loss, which is a very popular choice in deep neural networks owing to its adaptive learning rate. We will use the default learning rate. Finally, we provide classification accuracy as a metric to judge our model performance during training. Now, we will execute the function `model.compile()` to configure the process. Refer to the following code:

```

>>model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

```

Now, we are ready to train our model. We will call the function `model.fit()`. The training data and the labels are supplied as input parameters. We also specify two more hyperparameters, the **mini-batch** size, and the number of **epochs** for training. The mini-batch size specifies how much of the data is used to compute the loss function and update the weights in one iteration. We specify a mini-batch size of

128, that is, the weights will be updated once on every chunk of 128 image samples randomly selected from the training data. In a complete epoch, we completely traverse the entire training data once which is divided into into multiple mini-batches. We specify 10 epochs for training. Now, let us train our model by executing the following code:

```
>>model.fit(X_train, y_train, batch_size=128, epochs=10, verbose=1)
```

The output is shown in *figure 4.7*:

```
Epoch 1/10
469/469 [=====] - 4s 8ms/step - loss: 0.2375 - accuracy: 0.9309
Epoch 2/10
469/469 [=====] - 4s 8ms/step - loss: 0.0876 - accuracy: 0.9734
Epoch 3/10
469/469 [=====] - 4s 8ms/step - loss: 0.0559 - accuracy: 0.9834
Epoch 4/10
469/469 [=====] - 4s 8ms/step - loss: 0.0394 - accuracy: 0.9879
Epoch 5/10
469/469 [=====] - 4s 8ms/step - loss: 0.0271 - accuracy: 0.9916
Epoch 6/10
469/469 [=====] - 4s 8ms/step - loss: 0.0217 - accuracy: 0.9928
Epoch 7/10
469/469 [=====] - 4s 8ms/step - loss: 0.0172 - accuracy: 0.9944
Epoch 8/10
469/469 [=====] - 4s 8ms/step - loss: 0.0197 - accuracy: 0.9935
Epoch 9/10
469/469 [=====] - 4s 8ms/step - loss: 0.0127 - accuracy: 0.9958
Epoch 10/10
469/469 [=====] - 4s 8ms/step - loss: 0.0122 - accuracy: 0.9959
<keras.callbacks.History at 0x7feb00e19f90>
```

Figure 4.7: Training of the ANN model

Setting the parameter `verbose=1`, we will allow to print the necessary information during training in the output console. As shown, it prints the execution time, the cross entropy loss value, and the overall classification accuracy on the training data at the end of every epoch. The classification accuracy measures the fraction of data that are correctly predicted by the classifier. Suppose you have 100 instances for prediction in your dataset. Your classifier has correctly detected 98 out of them. In that case, the classification accuracy will be 98%. A higher accuracy indicates a more accurate model. It can be observed that the cross entropy loss tends to decrease with the increase in epochs, and as a result, the classification accuracy improves. This is the intended scenario, as with every epoch, we are getting closer to a local minima of the loss function. At the end of 10 epochs, we get a classification accuracy of 99% on the training data.

We have a model that gives around 99% classification accuracy on the training data. Next, we need to see how the model performs on the unseen test data. We will call the function `model.evaluate()`. It takes two NumPy arrays as input, the image

data in the test set, and the corresponding labels. It returns an array having two elements. The first element is the loss on the test data, and the second element is the classification accuracy. Refer to the following code, for which the output is shown in *figure 4.8*:

```
>>score = model.evaluate(X_test, y_test)

print('loss on test data: ', score[0])

print('accuracy on test data:', score[1])

313/313 [=====] - 1s 2ms/step
loss on test data: 0.09871130436658859
accuracy on test data: 0.9771999716758728
```

Figure 4.8: Prediction of test data

It can be observed that with a simple ANN, we can achieve a classification accuracy of around 98% on the MNIST test set. Please note in this example, we have trained our ANN for only 10 epochs. Since we worked on a fairly simple dataset, the loss function got minimized in 10 epochs. In a practical scenario, you may require to train a model for several hundreds of epochs to eventually minimize the loss.

As an exercise, you can change various hyperparameters of the ANN, such as the number of hidden layers, number of nodes in the layers, number of epochs, mini-batch size, learning rate, and so on, and check the impact on the classification performance.

Implementation of a Convolutional Neural Network

In the previous section, we have designed a simple feedforward neural network in TensorFlow using Keras. We have trained the model on the MNIST database for handwritten digit classification. In this section, we will design a slightly more complex neural network, a Convolutional Neural network or CNN, to solve the same problem. In *Chapter 3, Gearing with Deep Learning*, we have briefly discussed about the basic architecture of a CNN. It typically has five layers.

1. Input layer
2. Convolutional layer
3. Pooling layer
4. Fully connected layer
5. Output layer

Similar to the feedforward neural network, we will first define our CNN architecture and then configure it for training. One big advantage of CNN over a simple ANN is that a CNN can automatically extract relevant features from the image. Hence, the input images can be directly applied to a CNN without reshaping them into 1D vectors.

The architecture of the CNN we are going to implement for handwritten digit classification is shown in *figure 4.9*. The architecture has the following layers:

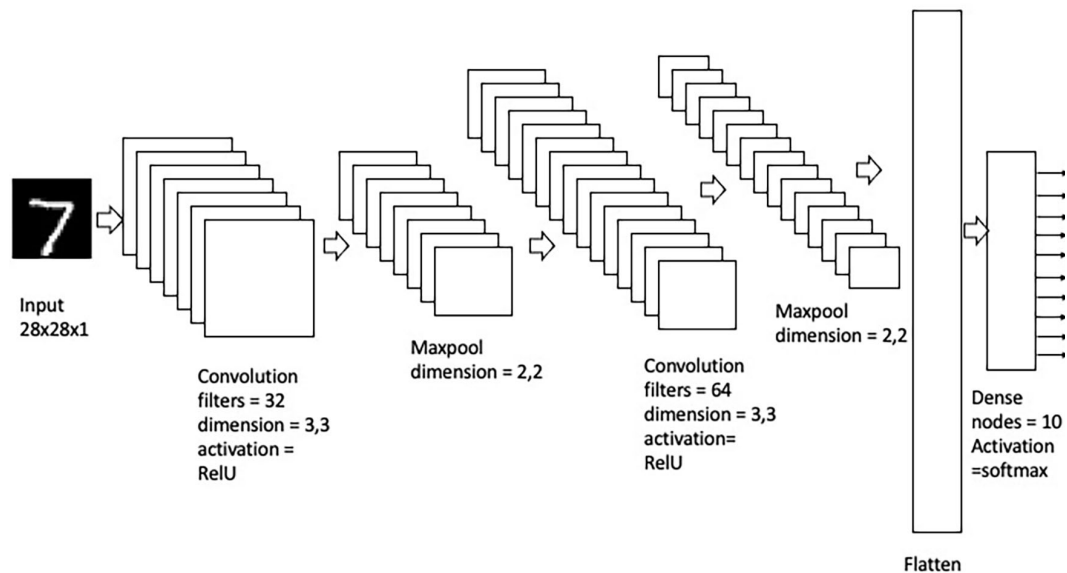


Figure 4.9: CNN architecture for classification of handwritten digits

The different layers are as follows:

1. **Input layer:** Input dimension of $28 \times 28 \times 1$ for grayscale images
2. **Convolutional layer:** Kernel dimension of 3×3 , number of output channels = 32, activation = ReLU
3. **Maxpool layer:** 2×2 pooling window
4. **Convolutional layer:** Kernel dimension of 3×3 , number of output channels = 64, activation = ReLU
5. **Maxpool layer:** 2×2 pooling window
6. **Flatten layer:** To convert into a 1D vector
7. **Dense layer:** 10 nodes with a softmax activation function for prediction

Keras has inbuilt APIs for almost all the layers that form the basic building blocks of a CNN. Hence, you hardly need to write any code to implement the fundamental layers of a CNN.

We will now implement the model. As mentioned earlier, the model will be trained and evaluated on the MNIST database. In a simple feedforward ANN comprising only dense layers, we need to reshape the input images into 1D tensors. Thanks to the convolution filters of a CNN, we can automatically extract relevant features from the input images. Hence, we do not to restructure the input images into 1D data. Since the images in the MNIST database are in grayscale, we need to specify the input channel as 1. We perform a reshaping operation on the 2D NumPy array to represent them in the format of *image width* × *image height* × *number of input channels*. This is the required dimension of the input image to apply to a CNN structure. In our case, the dimension of the images will be 28 × 28 × 1. The pixel values are normalized between 0 and 1. Similarly, the class labels are also converted into one hot encoded vectors.

```
>>#load the dataset

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.astype('float32') # change integers to 32-bit
floating point numbers

X_test = X_test.astype('float32')

X_train /= 255 # normalize the input
X_test /= 255

X_train = X_train.reshape(-1, X_train.shape[1], X_train.shape[2], 1)
X_test = X_test.reshape(-1, X_test.shape[1], X_test.shape[2], 1)

y_train = to_categorical(y_train, num_class)
y_test = to_categorical(y_test, num_class)
```

Now, we can define CNN architecture. As mentioned earlier, most of the building blocks are readily available in Keras. We will briefly discuss about few of the functional layers commonly used to implement a CNN.

Conv2D: The Conv2D layer performs 2D convolution operations on images. Although 2D convolution is the popular operation, Keras supports 1D and 3D convolution operations too. Conv2D takes a 2D tensor (for example, image) as input, performs convolution operation based on the kernel dimension provided in the input argument, and returns the output tensor. When Conv2D is used as the first layer in the model, you need to specify the dimension of the input data as an extra argument. For 2D convolution, the dimension of the input is a tuple in the form of (image width, image height, number of channels); for example, in our example, the input shape is (28,28,1). A few other input arguments for Conv2D are as follows:

- **filters:** Number of output filters, a whole integer value
- **kernel_size:** A tuple specifying the width and height of the kernel for convolution
- **strides:** A tuple that specifies the stride length, default is (1,1)
- **padding:** *“valid”* if zero padding is not used or *“same”* if zero padding is used, default is *“valid”*
- **activation:** specifies the activation function
- **use_bias:** *True* and *False*, default is *True*
- **kernel_initializer:** specifies how the kernel values are initialized; the default is *“glorot_uniform”*
- **bias_initializer:** specifies how the bias terms are initialized; the default is *“zeros”*

In most cases, we only need to specify a few of the input parameters, such as the *“filters,” “kernel_size,” “padding,”* and *“activation”*, and use the default values for the rest.

MaxPooling2D: It performs 2D maxpool operation on the input. A few of its input arguments are as follows:

- **pool_size:** A tuple to specify the pool window size
- **strides:** Specifies the stride length for pooling; default is *None*
- **padding:** Default is *“valid”*

Flatten: It reshapes the input tensor to an equivalent 1D tensor so that it can be applied to a dense layer. For example, if the input has a dimension of (batch_size, 10, 10, 2), the output dimension after flattening will be (batch_size, 200).

Dense: These are regular, fully connected layers used in neural networks. We have already used them in the previous section. A few of the input arguments are as follows:

- **units:** Specifies the number of nodes
- **activation:** The activation function
- **use_biase:** Default is True
- **kernel_initializer:** Default is “*glorot_uniform*”
- **bias_initializer:** Default is “*zeros*”

Now let us define our CNN. We first import the necessary layers in Keras. Then we define a Sequential model and add all the necessary layers. As shown in *figure 4.9*, the CNN we are going to define has a pair of **Conv2D** layers with associated **Maxpooling2D** layers. The kernel dimension and the number of output channels are shown in the figure. We will use the ReLU activation function in the Conv2D layers. The output of the final convolutional layer is reshaped using the **Flatten** layer. In order to mitigate the scope of overfit, we apply 50% of dropout. Then comes the final dense layer having 10 nodes along with a softmax activation function for prediction. The complete code to define the network is shown as follows:

```
>>from tensorflow.keras.models import Sequential

    from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
    Dropout, Dense

    model = Sequential()

    model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_
    shape = (28,28,1)))

    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))

    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())

    model.add(Dropout(0.5))

    model.add(Dense(num_class, activation='softmax'))
```

Similar to the previous example, we have provided the shape of the input images as an argument in the first convolutional layer. Now, let us have a close look at the structure of the model and its parameters. As mentioned, we can get an overview of the model by executing the command `model.summary()`. The output is shown in *figure 4.10*:

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_2 (MaxPooling 2D)	(None, 13, 13, 32)	0
conv2d_3 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_3 (MaxPooling 2D)	(None, 5, 5, 64)	0
flatten_1 (Flatten)	(None, 1600)	0
dropout_1 (Dropout)	(None, 1600)	0
dense_1 (Dense)	(None, 10)	16010

=====
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
=====

Figure 4.10: Summary of the CNN for classifying handwritten digits

Carefully look at the shape of the tensor at the output of different layers. Since the default padding in the `Conv2D()` is *valid*, the tensor dimension shrinks after each convolution operation. If the input tensor has a dimension of $W \times H$ and the corresponding kernel dimension is $k \times k$, the output dimension becomes $(W-k+1) \times (H-k+1)$ after convolution. We increase the number of filters in the second convolutional layer to extract more detailed features. The subsequent pooling operation is performed to reduce the data dimension. The output is flattened into a single vector and applied to the dense layer for classification.

Now, let us compile the model. We will keep the same set of parameters we used in our earlier ANN.

```
>>model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

Now, we can train our CNN. We will call `model.fit()` and pass the training data as input arguments. We will keep the number of epochs and batch-size similar to what we have used in the earlier example. Execute the following code:

```
>>history = model.fit(X_train, y_train, batch_size=128, epochs=10)
```

Refer to *figure 4.11* for the output:

```
Epoch 1/10
469/469 [=====] - 42s 87ms/step - loss: 0.3341 - accuracy: 0.8993
Epoch 2/10
469/469 [=====] - 41s 87ms/step - loss: 0.1038 - accuracy: 0.9687
Epoch 3/10
469/469 [=====] - 42s 89ms/step - loss: 0.0797 - accuracy: 0.9751
Epoch 4/10
469/469 [=====] - 41s 87ms/step - loss: 0.0657 - accuracy: 0.9798
Epoch 5/10
469/469 [=====] - 40s 86ms/step - loss: 0.0579 - accuracy: 0.9821
Epoch 6/10
469/469 [=====] - 41s 87ms/step - loss: 0.0518 - accuracy: 0.9844
Epoch 7/10
469/469 [=====] - 41s 87ms/step - loss: 0.0461 - accuracy: 0.9851
Epoch 8/10
469/469 [=====] - 41s 87ms/step - loss: 0.0433 - accuracy: 0.9860
Epoch 9/10
469/469 [=====] - 41s 87ms/step - loss: 0.0424 - accuracy: 0.9870
Epoch 10/10
469/469 [=====] - 40s 86ms/step - loss: 0.0389 - accuracy: 0.9877
```

Figure 4.11: Training of the CNN

On executing the preceding code, we can see how the network behaves on the training data with increasing epochs. Similar to the previous example, the loss function decreases, and the classification accuracy increases. We again get around 99% classification accuracy on the training data at the end of 10 epochs. The variable *history* stores all the information related to the loss value and accuracy at different epochs. We can plot the training history using **Matplotlib** library. See the following code for plotting the training performance of the model:

```
>>import matplotlib.pyplot as plt

plt.subplot(2,1,1)

plt.plot(history.history['loss'], '*-')

plt.xlabel('epochs')

plt.ylabel('loss')

plt.subplot(2,1,2)

plt.plot(history.history['accuracy'], '*-')

plt.xlabel('epochs')

plt.ylabel('accuracy')
```

The code output is shown in *figure 4.12*:

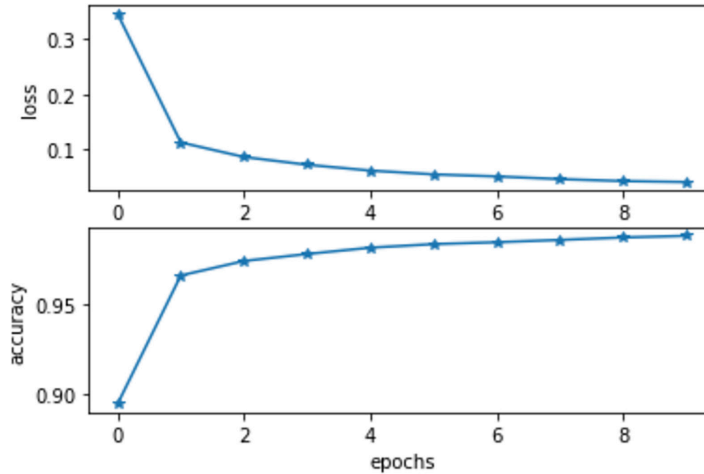


Figure 4.12: Plotting the loss and accuracy versus epochs

Finally, let us evaluate our model on the test data by executing the following code:

```
>>score = model.evaluate(X_test, y_test)
    print('loss on test data: ', score[0])
    print('accuracy on test data:', score[1])
```

Refer to *figure 4.13* for the output:

```
313/313 [=====] - 3s 8ms/step - loss: 0.0284 - accuracy: 0.9903
loss on test data: 0.02837059646844864
accuracy on test data: 0.990299997138977
```

Figure 4.13: Evaluation of test data

Using CNN, we get around 99% classification accuracy on the test set. It can be recalled that earlier, we got 98% classification accuracy using the feedforward ANN network. Hence, we conclude that the CNN marginally outperforms the ANN on the MNIST database. You are strongly encouraged to see how the CNN performance by tweaking various network hyperparameters. You may add more convolution and pooling layers to the existing CNN. You may also modify the kernel dimension and the number of filters. We may add more dense layers or even change the dropout rate. Apart from that, you may also modify the learning rate and mini-batch size and may use more number of epochs for training of the network and check the model performance.

It can be observed that both the feedforward ANN and the CNN result in more than 98% classification accuracy on the test set only after 10 epochs of training. Remember, MNIST is a relatively simple database where the images are very small in size. Moreover, the digits are centrally located in the images and do not have any complex, ambiguous background. Hence, the network requires a relatively less number of epochs for training. In a practical scenario, it may take several hundreds of epochs in order to minimize the loss function.

Finally, let us go back and check the model summary for the ANN and the CNN in *figures 4.6* and *4.10*, respectively. Our ANN model resulted in a total of 468,874 trainable parameters, whereas the CNN generates only 34,826 trainable parameters, which is much lesser than the ANN. A Keras model can be saved by using the `model.save('filename')` function. In TensorFlow, we save the model files in H5 file format. It can be seen that the size of the ANN model is more than 5.5 MB in the computer hard disk, whereas the size of the CNN model is around 459 KB, which is more than 10× smaller than the ANN model. Hence, we can conclude that with CNN, we get a much smaller yet more accurate model than an ANN.

Evaluation metrics in classification models

In this example, we have measured classification accuracy as a metric for evaluating the performance of a neural network classifier. However, classification accuracy cannot always properly justify the true performance of a classifier model. Let us take a different example. Suppose we have designed a classifier for detecting a rare type of lung cancer from chest X-ray images. Suppose we have tested our model on a skewed population of 200 people, where only five people have the disease, and others are normal. Now, if our classifier predicts all test subjects as normal, it will correctly predict all 195 normal subjects, despite not detecting a single diseased case. So, we will get a classification accuracy : $\frac{195}{200} \times 100 = 97.5\%$. It sounds very good in terms of the accuracy value. But in real life, it is a poor classifier because it predicts everything as a single class label. Hence, classification accuracy cannot be an optimum metric for this kind of problem. In this section, we will briefly talk about certain metrics popularly used in evaluating the performance of a machine learning model.

- **Confusion matrix**

Confusion matrix shows the classification performance of a model in tabular format in terms of four parameters, **True Positive (TP)**, **True Negative (TN)**, **False Positive (FP)**, and **False Negative (FN)**. Suppose you have designed a binary classifier for identifying images of cats. Your classifier gives an output

1 if it predicts a cat in the image and returns 0 if no cat is detected. True positive measures how many 1s predicted by the classifier are correct based on their actual labels. Similarly, true negative measures how many predicted 0s (that is, no cat) are actually correct. False positive measures how many predicted 1s are actually 0; that is, the classifier predicted that the image contains a cat, but actually, there is no cat. Finally, false negative measures how many predicted 0s are actually 1; that is, the classifier has predicted that the image does not contain any cat, but there are actually cats in the images.

For a binary classifier, the confusion matrix is shown in *figure 4.14*:

		Predicted Class	
		1	0
True Class	1	True Positive (TP)	False Negative (FN)
	0	False Positive (FP)	True Negative (TN)

Figure 4.14: Confusion matrix for a binary classifier

Confusion matrix can also be derived for both binary and multi-class classifiers. For multi-class classifiers, the parameters TP, TN, FP, and FN are measured in one-vs-all strategy. Where we split the problem into multiple binary classification problems per class.

- **Classification report**

With the confusion matrix, we can define a number of metrics to evaluate the classification performance.

Recall or Sensitivity: It measures how many positive samples have been correctly predicted by the classifier. It is given by the following equation:

$$Recall = \frac{TP}{TP + FN}$$

Precision: It measures out of all predicted positive samples by the classifier how many are actually correct. It is given by the following:

$$Precision = \frac{TP}{TP + FP}$$

Specificity: It measures the fraction of negative samples that are correctly predicted.

$$\text{Specificity} = \frac{TN}{TN + FP}$$

Classification accuracy: It is a measure of the fraction of all correctly predicted samples by the classifier.

$$\text{Classification accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

F1 score: This is a weighted average of precision and recall. It is calculated as the harmonic mean of precision and recall. Refer to the following equation:

$$F1 \text{ score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

F1-score takes into account both precision and recall in a single metric. Hence, it is widely considered as a better performance metric compared to classification accuracy, particularly to evaluate on imbalanced datasets.

Conclusion

In this chapter, we have developed our first neural network application using TensorFlow for the classification of handwritten digits. We have particularly used the Keras APIs integrated with TensorFlow, which gives us quick interfaces to implement large deep learning architectures from scratch with a high level of abstraction. We have initially implemented a simple feedforward ANN and then a CNN model using the publicly available MNIST database. It can be observed that the CNN offers a smaller yet more accurate model compared to the ANN. Now the question arises, is the CNN model small enough to run on tiny microcontrollers? The simple answer is No. In this chapter, we have implemented a fairly simple CNN with only two convolutional layers. Although it worked pretty well in our case because of the simplicity of the database, it will most certainly not work on complex practical datasets where the images are large in size and are noisy in general.

In practice, a CNN architecture may have many layers which may generate several thousands of trainable parameters. Hence, the model size will significantly increase. Such a model might not be suitable for TinyML applications. You need to compress the original model before deploying it on a smaller target device. However, model compression often causes a negative impact on performance. Hence, we need to maintain a trade-off between model size and accuracy while compressing a model. This is called model optimization. In the upcoming chapter, we will go one step closer to our first TinyML application by converting a large neural network model into a much smaller yet optimized model using TensorFlow.

Key facts

- Keras offers sets of APIs on TensorFlow for quick implementation of neural networks in Python.
- Keras provides a high level of abstraction.
- There are four stages in a Keras model: model definition, compiling the model, fitting on the training data, and evaluation of test data.
- MNIST is a publicly available database containing labelled images of handwritten digits for benchmarking machine learning model.
- A CNN can outperform a traditional ANN with much lesser number of trainable parameters.

Join our book's Discord space

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

<https://discord.bpbonline.com>



CHAPTER 5

Model Optimization Using TensorFlow

Introduction

In *Chapter 4, Experiencing TensorFlow*, we developed our first neural network model for the classification of handwritten digits. TensorFlow offers readily available Python interfaces under Keras for quick implementation of complex neural network architectures. A neural network designed for practical applications can have several thousands of trainable parameters, resulting in large models. We developed a fairly simple **Convolutional Neural Network (CNN)** in the previous chapter that had only two convolution layers. Moreover, the network was intended to process very small images having dimensions of 28×28 pixels. Even then, the network had around 35K trainable parameters, and it required 459 kilobytes of memory space to store the model. In a real-world scenario, you may require to design more complex networks that may have millions of trainable parameters. Such a model may require several megabytes of memory space. Large deep learning algorithms typically run on powerful desktops or data centers or are hosted in the cloud. In that case, we typically do not bother about the hardware resource like memory space or computation capacity, or power consumption to run those large deep learning models. However, the scenario is completely different in TinyML. Here, our target platforms are smaller edge devices and microcontrollers, which are severely resource constrained. For example, a typical microcontroller has only 256 kilobytes

of computational memory and only 64 megahertz of CPU speed. Hence, a standard deep neural network cannot be directly deployed on a microcontroller to implement a TinyML application.

In the previous chapters, we learned the basic concepts of TensorFlow to create machine learning models. TensorFlow also offers a set of libraries for compressing and optimizing large machine learning models. **TensorFlow Lite** is a specially designed open-source library for deploying machine learning and deep learning models on smaller devices like mobile phones, microcontrollers, and other IoT and edge devices. With **TensorFlow Lite Converter**, you can convert a TensorFlow model into a compressed **FlatBuffers** format that is optimized for mobile and edge devices. FlatBuffers is an efficient cross-platform serialization library for various programming languages such as C, C++, JAVA, JavaScript, Python, and so on. It represents hierarchical data in a flat binary buffer so that the serialized data can be accessed directly without unpacking while providing forward/backward compatibility. It has very efficient memory management, resulting in a faster inference speed on the target device.

A TensorFlow Lite model is represented by `.TFLITE` file extension. A model generated by the TensorFlow Lite Converter is much smaller than the corresponding TensorFlow model and also runs much faster with a minimum performance impact. TensorFlow Lite supports multiple platforms such as Android, iOS, and Linux-based single board computers like Raspberry Pi and also some microcontrollers. It also supports a number of programming languages, including C, C++, JAVA, Python, and so on. In addition, TensorFlow offers the **TensorFlow Model Optimization Toolkit**, a set of software tools that can reduce model complexity for ease of deployment and faster execution on edge devices. For example, a 32-bit floating point model can be quantized into a pure integer-based model, which is not only 4× smaller than the original model but also runs much faster on the target device.

Structure

In this chapter, we will discuss the following topics:

- Experiencing TensorFlow Lite
- TensorFlow Model Optimization Toolkit
 - o Quantization
 - Post-training quantization
 - Quantization-aware training

- o Weight pruning
- o Weight clustering
- o Collaborative optimization

Objectives

In this chapter, we will learn different techniques about how a base TensorFlow model can be effectively compressed in a step-by-step manner for deploying on smaller target devices through practical coding examples. We will primarily learn to use **TensorFlow Lite** and **TensorFlow Model Optimization Toolkit**. First, we will create a baseline Convolutional Neural Network (CNN) model using TensorFlow, which will be trained and evaluated on Fashion MNIST, another popular benchmark machine learning database that contains grayscale images of different fashion products. Next, we will convert the baseline model into a compressed TensorFlow Lite model, that can be further converted into equivalent libraries for deploying on various target platforms such as Android and iOS-based smartphones, Linux-based single board computers or even microcontroller units to make inferences. We will also explore various optimization techniques offered by TensorFlow through **TensorFlow Model Optimization Toolkit**, including quantization, weight pruning, and weight clustering. We will optimize the baseline model, and experience the impact of different optimization techniques on both model size and classification performance.

Experiencing TensorFlow Lite

TensorFlow Lite or TFLite, a lighter version of TensorFlow, is an open-source framework for on-device machine learning. It enables TensorFlow models to run on mobile, embedded, and IoT devices in a seamless manner. TFLite was developed and open-sourced by Google. Some of its key advantages are as follows:

- TFLite converts a baseline TensorFlow machine learning model into an efficient and portable format called **FlatBuffers**, which is an efficient serialization cross-platform library.
- Flatbuffers are easily accessible, extremely memory efficient, faster, flexible, and generate tiny code with small headers, which is easy to integrate.
- TFLite models have a decreased inference time. This implies a reduced latency, which is essential for real-time on-device machine learning.
- TFLite is extremely user-friendly. It supports various hardware platforms

and programming languages.

However, it is important to note that TFLite models are particularly designed for model inferencing on tiny edge devices. Although you can rarely train a model on a mobile smartphone or a Raspberry Pi, on-device model training is very difficult for microcontrollers. Training of a machine learning model is far more computationally expensive and is typically done on a desktop computer or a cloud server. Moreover, in most of the cases, you do not need to retrain your model very often. Hence, offline training is the preferable approach. The main objective of TensorFlow Lite is make real-time inferences on smaller edge devices, which may run 24x7.

Now, let us explore how to create a TFLite model from a base TensorFlow model through a practical example. In *Chapter 4, Experiencing TensorFlow*, we developed our first CNN model for classifying handwritten digits. In this chapter, we will create another similar CNN architecture, although for a different application. The CNN we are going to design will classify images of various fashion products. For this application, we will use the Fashion MNIST database. Fashion MNIST is another popular open-access machine learning database containing labelled images for various fashion products. The database is quite similar with MNIST in terms of image size and structure of training and test splits. It has 60,000 training and 10,000 test examples. Each example is a 28×28 grayscale image. The training and test examples are associated with a numerical number as labels corresponding to the following product code:

- 0: T-shirt/top
- 1: Trouser
- 2: Pullover
- 3: Dress
- 4: Coat
- 5: Sandal
- 6: Shirt
- 7: Sneaker
- 8: Bag
- 9: Ankle boot

Now, we will design a base CNN model in TensorFlow to classify different fashion

products on Fashion MNIST. Create a new Colab notebook and save it as a project. Connect to a Python runtime.

We will first import the necessary libraries:

```
>>import numpy as np

    from numpy import random

    import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.datasets import fashion_mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dropout, Dense

from tensorflow.keras.utils import to_categorical

from tensorflow.keras.optimizers import Adam
```

Next, we will load the database in our work directory. Similar to MNIST, the Fashion-MNIST database is also readily available with Keras. We can import it easily with the following line of code:

```
>>(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
```

It will load the training and the test data and their labels. Now, let us plot a few random samples from the database for quick visualization. Refer to the following code. It will show the images and the corresponding labels.

```
>>plt.rcParams['figure.figsize'] = (5,5)

for i in range(9):

    plt.subplot(3,3,i+1)

    num = random.randint(0, len(X_train))

    plt.imshow(X_train[num], cmap='gray')
```

```
plt.title('Class {}'.format(y_train[num]))  
plt.tight_layout()
```

The code output at one execution is shown in *figure 5.1*:

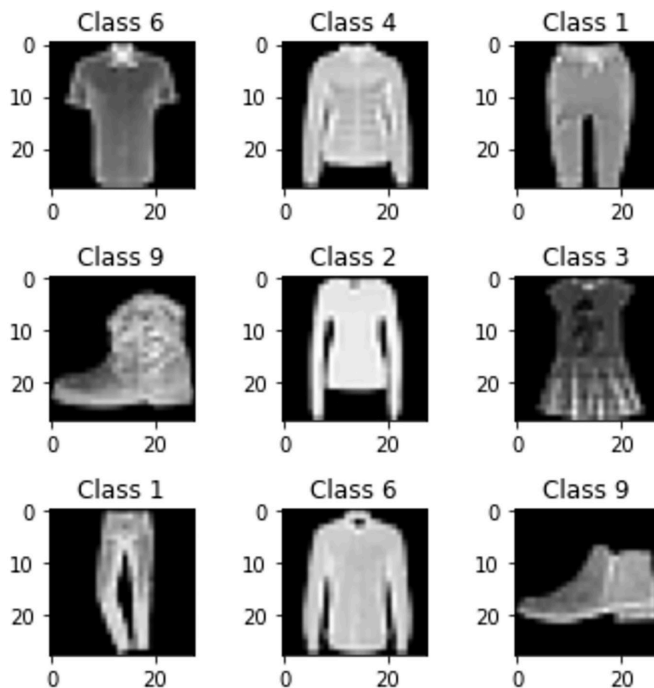


Figure 5.1: Sample Images from the Fashion-MNIST database

Similar to *Chapter 4, Experiencing TensorFlow*, we will apply some basic pre-processing on the images. The pixel values of the images are rescaled between 0 and 1 and the dimension is reshaped in (width, height, 1) format. We also convert the class labels into one hot encoded vector. Refer to the following code:

```
>>num_class = 10  
  
X_train = X_train.astype('float32') # change integers to 32-bit  
floating point numbers  
  
X_test = X_test.astype('float32')
```

```

X_train /= 255                                # normalize the input
X_test /= 255
# reshape the input as per input to CNN
X_train = X_train.reshape(-1, X_train.shape[1], X_train.shape[2], 1)
X_test = X_test.reshape(-1, X_test.shape[1], X_test.shape[2], 1)
#convert the output labels to one hot vector
y_train = to_categorical(y_train, num_class)
y_test = to_categorical(y_test, num_class)

```

Now, we will define the CNN, which will be our baseline model. For this application, we will again define a fairly simple architecture having only two convolutional layers. The number of convolution filters used in the two convolutional layers are 32 and 64, respectively. The kernel size is taken as 3×3 . After each convolution, a maxpool layer is applied. The output feature map is flattened and applied to a dense layer having 100 nodes, followed by the final dense layer with 10 nodes for prediction of different classes using a softmax activation function. We also apply dropout before the dense layers. As discussed in *Chapter 4, Experiencing TensorFlow*, we will create a Sequential model under Keras and add the necessary layers to define our model. Refer to the following code:

```

>>model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_
shape = (28,28,1)))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dropout(0.5))

model.add(Dense(100, activation = 'relu'))

model.add(Dense(num_class, activation='softmax'))

```

You can see the model details by executing `model.summary()` on Colab. Refer to *figure 5.2* for an overview of the model, which has around 180,000 parameters:

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_2 (MaxPooling 2D)	(None, 13, 13, 32)	0
conv2d_3 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_3 (MaxPooling 2D)	(None, 5, 5, 64)	0
flatten_1 (Flatten)	(None, 1600)	0
dropout_1 (Dropout)	(None, 1600)	0
dense_2 (Dense)	(None, 100)	160100
dense_3 (Dense)	(None, 10)	1010

```

=====
Total params: 179,926
Trainable params: 179,926
Non-trainable params: 0
=====

```

Figure 5.2: Summary of the baseline CNN

Now, it is time to configure the network for training. We will define an Adam optimizer with a custom learning rate of 0.002 to compile our model. The categorical cross entropy loss will be minimized during training. Refer to the following code:

```
>>opt = Adam(lr=0.002)

model.compile(loss='categorical_crossentropy', optimizer = opt,
metrics=['accuracy'])
```

Let us now train the model. We will call `model.fit()` and pass the necessary arguments. The batch size is selected as 128, and the model will be trained for 15 epochs on the training data using the following command:

```
>>model.fit(X_train, y_train, batch_size=128, epochs=15)
```

On executing the preceding command, the network will start getting trained. The intermediate classification performance on the training data will be printed after each epoch of training. You can expect a classification accuracy of around 80% on the training set at the end of the first epoch and which will improve substantially. The model will produce around 92% accuracy on the training set at the end of 15 epochs. You can train for more epochs and check the model performance.

Next, let us evaluate the model performance on the test set by executing the following code:

```
>>score = model.evaluate(X_test, y_test)

    print('accuracy on test data:', score[1])
```

You can see that the model will give around 91% accuracy on the test set. With this, we have created and evaluated the base model. Next, we will convert it into an equivalent TFLite model. Before that, first, save the baseline model in our work directory in .H5 format.

```
>>model.save('baseline_model.h5')
```

You can locate the model file by clicking the **File** icon on the left side of your Colab notebook. The model size is around 2.1 megabytes. Remember that the file is only temporarily stored in the workspace of your Colab notebook. Remember, all saved files and variables in Colab are deleted as soon as you disconnect the runtime. You can right-click on it and click on **Download** in order to save model file on your host computer for offline access. The saved H5 model file can be loaded in Keras by calling the function `load_model()`. Refer the below code example, where we load the saved model as `baseline_model`:

```
>>from tensorflow.keras.models import load_model

    baseline_model = load_model('baseline_model.h5')
```

Now, let us convert the model into a TFLite model. TensorFlow has all the inbuilt APIs for doing the conversion. For TensorFlow 2, We can do the conversion using `tf.lite.TFLiteConverter`. We will call the `from_keras_model()` method under the `tf.lite.TFLiteConverter` class and pass the baseline model as a function argument. The model will be converted to the equivalent TFLite model using the `convert()` method. See the following code:

```
>>converter = tf.lite.TFLiteConverter.from_keras_model(baseline_model)

    tflite_model = converter.convert()
```

We can further convert the baseline model to a TensorFlow graph using `tf.function`, which contains all the computational operations, variables, and weights. This can be achieved by exporting the model as a concrete function. Finally, the concrete function is converted into a TFLite model using the method `from_concrete_functions()`. See the following code:

```
>># export model as a concrete function
```

```

func = tf.function(baseline_model).get_concrete_function(
    tf.TensorSpec(baseline_model.inputs[0].shape, baseline_model.
        inputs[0].dtype))
# serialized graph representation of the concrete function
func.graph.as_graph_def()

# converting the concrete function to TFLite
converter = tf.lite.TFLiteConverter.from_concrete_functions([func])
tflite_model = converter.convert()

```

Now, let us save the TFLite model in the file system. We will import the **Pathlib** library and store the model, as shown in the following code:

```

>>import pathlib

tflite_models_dir = pathlib.Path('/content/tflite_models/')
tflite_models_dir.mkdir(exist_ok=True, parents=True)
tflite_model_file = tflite_models_dir/'model.tflite'
tflite_model_file.write_bytes(tflite_model)

```

The first line of the preceding code will create a directory **tflite_models** in the Colab workspace. Our TFLite model will be stored in that directory with **.TFLITE** file extension. The code will also print the size of the TFLite file in the console. In our case, the model size will be around 722 kilobytes. Recall, the baseline H5 model size is 2.1 megabytes.

Now, we have a TFLite model. We will use it to make inferences in Python. This can be done by using the **tf.lite.Interpreter** class.

The following steps are done to make an inference in TensorFlow Lite:

1. First, we create an instance of the **Interpreter** class. It takes the path containing the **.TFLIE** file as an input.
2. Allocate memory to the Interpreter by calling the function **allocate_tensors()**.
3. After memory allocation, call **get_input_details()** and **get_output_details()** to get some details about the input and the output tensor.

4. Now, we are ready to make inferences. Get an image from the test data and reshape it according to the desired input shape to the model.
5. Set the input tensor by copying the input data. Use the method `set_tensor()`.
6. Invoke the interpreter to make an inference by calling `Interpreter.invoke()`.
7. Get the value of the output tensor.
8. Convert it into the predicted class label.

Refer to the following code example. Here, we predict on the entire test set using the `Interpreter` to obtain the overall classification accuracy.

```
>>tflite_model_file = 'tflite_models/model.tflite'

interpreter = tf.lite.Interpreter(model_path=tflite_model_file)
interpreter.allocate_tensors()

input_index = interpreter.get_input_details()[0]['index']
output_index = interpreter.get_output_details()[0]['index']

pred_list = []
for images in X_test:
    input_data = np.array(images, dtype=np.float32)

    input_data = input_data.reshape(1, input_data.shape[0], input_data.
    shape[1], 1)

    interpreter.set_tensor(input_index, input_data)
    interpreter.invoke()

    prediction = interpreter.get_tensor(output_index)
    prediction = np.argmax(prediction)
    pred_list.append(prediction)

accurate_count = 0
for index in range(len(pred_list)):
```

```
if pred_list[index] == np.argmax(y_test[index]):  
    accurate_count += 1  
accuracy = accurate_count * 1.0 / len(pred_list)  
  
print('accuracy = ', accuracy)
```

The accuracy obtained by executing the preceding code should be similar to the baseline TensorFlow model. With this, we have learnt to convert a baseline model into an equivalent TFLite model. However, the model size still remains 722 kilobytes, which may not be suitable for deploying on smaller edge devices. As mentioned earlier, the RAM size of a typical microcontroller can be fewer than 512 kilobytes. Hence, the model needs to be further reduced. Remember, we have used a fairly simple CNN architecture having only two convolutional layers. In practice, you may need to design more complex networks, having more layers which will have more trainable parameters, resulting in a larger model. Compressing of a model can have a negative impact on its performance. Hence, we need to focus on an optimized model compression. Fortunately, TensorFlow provides a software package called the **TensorFlow Model Optimization Toolkit** that can effectively optimize large deep neural networks without a significant performance drop. We will learn more about different optimization techniques offered by TensorFlow in the next section through coding examples.

TensorFlow Model Optimization Toolkit

The TensorFlow Model Optimization Toolkit consists of a set of libraries for the effective optimization of large neural networks. The primary goal of optimization is to enable a large machine learning model to seamlessly run on smaller edge devices having restricted hardware resources in terms of memory and computational capacity. They also need to consume lower battery power on the target hardware. Such applications are particularly useful in scenarios where we require continuous 24×7 monitoring, for example, machine condition monitoring in large industries, on-device cardiac health monitoring systems, smart voice assistant devices, and so on. A few popular model optimizations techniques are as follows:

- Lowering the precision of model weights and activations
- Reducing some of the lesser important parameters in the model
- Updating the model topology

TensorFlow Model Optimization Toolkit provides a number of inbuilt optimization functionalities, such as model quantization, weight pruning, and clustering which

can effectively compress large neural network models. Remember, each optimization technique comes with a compromised model performance, which needs to be fine-tuned. In TensorFlow model optimization, we take a baseline model as input, apply the desired quantization techniques, and then fine-tune the model performance via retraining, that eventually converts into a much smaller model for deployment. The following sections will cover various optimization techniques provided by TensorFlow Optimization Toolkit in some detail. Content of following sections are strongly influenced by the examples provided in the official website of TensorFlow for model optimization. We have used some of the readily available functions from those examples. Interested readers can refer the official website to learn more¹.

Quantization

Quantization is an optimization strategy used to lower the precision of a machine learning model. Both model weights and activation outputs can be quantized in the process. Integer-based quantization is particularly common in TinyML. It converts the weights and activation outputs from the original 32-bit floating point numbers to the nearest 8-bit fixed-point numbers. As a result, the model size is reduced by a factor of 4. The resulting model also has a faster inference speed. Quantization is particularly common in low-powered microcontroller devices, as many of them do not have floating-point units in the hardware. TensorFlow supports two types of quantization: **post-training quantization** and **quantization-aware training**.

Post-training quantization

Post-training quantization is the simplest form of quantization that is performed on a baseline floating-point TensorFlow model. Once the baseline model is trained with sufficient accuracy, the model weights and activation outputs are quantized to 8-bit precision. As expected, quantization of the model parameters might introduce errors in the model performance. A major drawback of post-training quantization is that it never compensates for the quantization error. Hence, it is important to check the model performance after post-training quantization before deployment in order to ensure that the performance is within acceptable limit.

In the following example, we will apply post-training quantization to our baseline CNN model. It is important to remember that the size of the baseline TFLite model is 722 kilobytes. It is fairly simple to perform integer quantization in TFLite. We do not require any new package. Once the instance for **TFLiteConverter** is created, we need to add the following line of code before converting it into the TFLite model:

```
>>converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

¹ https://www.tensorflow.org/model_optimization/guide/get_started

The complete code for post-training quantization is as follows:

```
>>converter = tf.lite.TFLiteConverter.from_keras_model(baseline_model)
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    tflite_model_ptq = converter.convert()
```

Now, save the model as a `.TFLITE` file by executing the following code. The quantized model, `model_ptq.tflite` will be saved under the same `tflite_models` directory created earlier in the Colab workspace.

```
>>tflite_models_dir = pathlib.Path('/content/tflite_models/')
    tflite_models_dir.mkdir(exist_ok=True, parents=True)
    tflite_model_file = tflite_models_dir/'model_ptq.tflite'
    tflite_model_file.write_bytes(tflite_model_ptq)
```

The quantized model size is around 188 kilobytes, which is roughly 4x smaller than the base TFLite model, which has a size of 722 kilobytes. Let us now evaluate the quantized model performance on the test set. Execute the following test code:

```
>>tflite_model_file = 'tflite_models/model_ptq.tflite'
    interpreter = tf.lite.Interpreter(model_path=tflite_model_file)
    interpreter.allocate_tensors()

    input_index = interpreter.get_input_details()[0]['index']
    output_index = interpreter.get_output_details()[0]['index']

    pred_list = []
    for images in X_test:
        input_data = np.array(images, dtype=np.float32)

        input_data = input_data.reshape(1, input_data.shape[0], input_data.
            shape[1], 1)

        interpreter.set_tensor(input_index, input_data)
        interpreter.invoke()
        prediction = interpreter.get_tensor(output_index)
        prediction = np.argmax(prediction)
```

```
pred_list.append(prediction)

accurate_count = 0
for index in range(len(pred_list)):
    if pred_list[index] == np.argmax(y_test[index]):
        accurate_count += 1
accuracy = accurate_count * 1.0 / len(pred_list)

print('accuracy = ', accuracy)
```

The classification accuracy will be similar to the baseline model. So, we can conclude that using post-training quantization, we get a 4× smaller model with a similar accuracy.

Quantization-aware training

In post-training quantization, we take a pre-trained model and convert the weights and activation output into 8-bit integers. One major disadvantage is that we do not fine-tune the model after quantization. In most of the cases, lowering the precision of the model weights will introduce a loss called quantization error. This can have a negative impact on model performance. Quantization-aware training tries to minimize the loss via backpropagation by retraining the model for few epochs. By doing this, it mitigates the impact of quantization error to some extent.

Let us experience quantization-aware training with a concrete example. We will again quantize the baseline model. Before that, we need to install the necessary software package, which is not readily installed with TensorFlow. Execute the following command in Colab:

```
>>pip install -q tensorflow-model-optimization
```

It will install all the necessary libraries for you. Next, we will load the baselined CNN model to perform quantization.

```
>>baseline_model = load_model('baseline_model.h5')
```

Now, let us apply quantization-aware training. First, import the necessary libraries.

```
>>import tensorflow_model_optimization as tfmot
```

Next, we will create an instance of `tf.quantization.keras.quantize_model` to define the quantization-aware model. We will use the inbuilt function, `quantized_model()`. Refer to the following code:

```
>>quantized_model = tfmot.quantization.keras.quantize_model
```

```
q_aware_model = quantized_model(baseline_model)
```

Since quantization-aware training needs a model retraining for fine-tuning the performance, we need to configure the model parameter, which will be quite similar to the base model. Refer to the following code:

```
>>q_aware_model.compile(optimizer='adam',
                        loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
                        metrics=['accuracy'])
```

Here, we have used the default learning rate for retraining. In general, retraining is preferred to be done at a much smaller learning rate than the learning rate used to train the baseline model.

The model summary is shown in *figure 5.3*:

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
quantize_layer (QuantizeLayer)	(None, 28, 28, 1)	3
quant_conv2d_2 (QuantizeWrapperV2)	(None, 26, 26, 32)	387
quant_max_pooling2d_2 (QuantizeWrapperV2)	(None, 13, 13, 32)	1
quant_conv2d_3 (QuantizeWrapperV2)	(None, 11, 11, 64)	18627
quant_max_pooling2d_3 (QuantizeWrapperV2)	(None, 5, 5, 64)	1
quant_flatten_1 (QuantizeWrapperV2)	(None, 1600)	1
quant_dropout_1 (QuantizeWrapperV2)	(None, 1600)	1
quant_dense_2 (QuantizeWrapperV2)	(None, 100)	160105
quant_dense_3 (QuantizeWrapperV2)	(None, 10)	1015

```

=====
Total params: 180,141
Trainable params: 179,926
Non-trainable params: 215
=====

```

Figure 5.3: Summary of the model after applying quantization-aware training

All layers in the model are prefixed by “**quant**”. Note that the resulting model is only quantization-aware but not yet quantized. The floating point model weights and activations are rounded to mimic integer values. Before converting them into full-integer, we will retrain to fine-tune the model. Since we already have a baseline model, retraining can be done on a subset of the training data. However, in this example, we will use the entire training data. See the following code, where we retrain our quantization-aware model for two epochs:

```
>>q_aware_model.fit(X_train, y_train, batch_size=500, epochs=2, validation_split=0.1)
```

The preceding code randomly selects 10% of training data as a validation set and evaluates the model performance on the validation data at each epoch. In this way, we can keep a track on the model’s accuracy. Once the training is done, we will quantize the model into an 8-bit integer and save as a TFLite model. This part of the code is similar to what we have done earlier in post-training quantization. Refer to the following code to convert and save the new model:

```
>>converter = tf.lite.TFLiteConverter.from_keras_model(q_aware_model)
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    tflite_model_qat = converter.convert()

    tflite_models_dir = pathlib.Path('/content/tflite_models/')
    tflite_models_dir.mkdir(exist_ok=True, parents=True)
    tflite_model_file = tflite_models_dir/'model_qat.tflite'
    tflite_model_file.write_bytes(tflite_model_qat)
```

Upon executing, you can see that the resulting model `model_qat.tflite` has a size of 188 kilobytes which is quite similar to what we achieved via post-training quantization. Now, let us examine the model performance on the test set. Execute the following code:

```
>>tflite_model_file = 'tflite_models/model_qat.tflite'
    interpreter = tf.lite.Interpreter(model_path=tflite_model_file)
    interpreter.allocate_tensors()

    input_index = interpreter.get_input_details()[0]['index']
```

```
output_index = interpreter.get_output_details()[0]['index']

pred_list = []
for images in X_test:
    input_data = np.array(images, dtype=np.float32)

    input_data = input_data.reshape(1, input_data.shape[0], input_data.
    shape[1], 1)

    interpreter.set_tensor(input_index, input_data)
    interpreter.invoke()
    prediction = interpreter.get_tensor(output_index)
    prediction = np.argmax(prediction)
    pred_list.append(prediction)

accurate_count = 0
for index in range(len(pred_list)):
    if pred_list[index] == np.argmax(y_test[index]):
        accurate_count += 1
accuracy = accurate_count * 1.0 / len(pred_list)

print('accuracy = ', accuracy)
```

You can see that the model will produce a classification accuracy above 92% on the test set. So, we conclude that with quantization-aware training, you can get slightly higher accuracy compared to post-training quantization.

In the preceding example, we have applied quantization-aware training to the entire model. However, you may not want to quantize all the layers. A fully quantized model can often be less accurate compared to the baseline model, even after retraining. To mitigate that risk, the critical feature extraction layers are often not quantized in a deep neural network. For example, you may prefer to quantize only the first few convolutional layers of a CNN.

In the following code, we will quantize only the dense layers of the baseline CNN. We will first define a function `apply_quantization(layer)` to define which layers will be quantized. Next, we will use `tf.keras.models.clone_model` to apply quantization to the dense layers by calling the function. Refer to the following code:

```
>>def apply_quantization(layer):
    if isinstance(layer, tf.keras.layers.Dense):
        return tfmot.quantization.keras.quantize_annotate_layer(layer)
    return layer

annotated_model = tf.keras.models.clone_model(
    baseline_model,
    clone_function=apply_quantization,
)

q_aware_model_dense = tfmot.quantization.keras.quantize_
apply(annotated_model)

q_aware_model_dense.summary()
```

The model summary is shown in *figure 5.4*:

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_2 (MaxPooling 2D)	(None, 13, 13, 32)	0
conv2d_3 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_3 (MaxPooling 2D)	(None, 5, 5, 64)	0
flatten_1 (Flatten)	(None, 1600)	0
quant_dropout_1 (QuantizeWrapperV2)	(None, 1600)	1
quant_dense_2 (QuantizeWrapperV2)	(None, 100)	160105
quant_dense_3 (QuantizeWrapperV2)	(None, 10)	1015

Figure 5.4: Model summary after applying quantization-aware training to dense layers

You can see only the dense layers are prefixed by “**quant.**” Similar to the previous example, the model can now be trained and converted to a quantized TFLite model. You can also evaluate the model performance on the test set. Since only the dense layers are quantized, the resultant model will be less compressed than a fully quantized model. Remember that in many practical applications, you may need to opt for selective quantization in order to mitigate a performance drop and maintain a balanced trade-off between model size and accuracy.

Weight pruning

Weight pruning is another popular model optimization technique that zeros out some of the less significant model weights. The pruned elements are trimmed from the model to introduce sparsity. Such sparse models are easy to compress and occupy lesser memory space in the target device. During inference, the zero weights are skipped, resulting in an improved latency due to lesser mathematical operations.

Similar to quantization-aware training, weight pruning also requires retraining. The pruned elements are zeroed out, and they do not take part in the backpropagation for learning the model weights. The most common pruning criteria is the **magnitude-based weight pruning**. In this process, the connections in the network having absolute values of the weights below a certain threshold are set to zero. Weights having very small absolute values typically have less contribution in the model outcome. Zeroing them will have a minimal impact on the overall performance. Introducing a higher amount of sparsity results in a more compressed model but may cause a negative impact on model performance. Hence, a trade-off between model size and accuracy needs to be maintained.

In the following example, we will apply magnitude-based weight pruning to our baseline model. The necessary functionalities lie in the class `tfmot.sparsity.keras.prune_low_magnitude`. We will load the baseline model from the saved .H5 file. Next, we will define a pruning schedule, where we start by applying 40% of sparsity to the baseline model, then gradually increase the sparsity, and eventually stop at 75% of model sparsity in two epochs. We define a polynomial function to introduce the sparsity in a step-wise manner during training. The step-size is defined based on the size of the training data. Refer to the following code for more details:

```

>>prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude

batch_size = 128

epochs = 2

validation_split = 0.1 # 10% of training set will be used for
validation set.

Num_samples = X_train.shape[0] * (1 - validation_split)

end_step = np.ceil(num_samples / batch_size).astype(np.int32) * epochs

# Define model for pruning.
Pruning_params = {
    'pruning_schedule': tfmot.sparsity.keras.
    PolynomialDecay(initial_sparsity=0.40,
                    final_
                    sparsity=0.75,
                    begin_step=0,
                    end_step=end_step)
}

model_for_pruning = prune_low_magnitude(baseline_model, **pruning_
params)

```

Now, we will compile the model for training.

```

>>model_for_pruning.compile(optimizer='adam',
                             loss=tf.keras.losses.CategoricalCrossentropy(from_
logits=True),
                             metrics=['accuracy'])

```

The model summary is shown in *figure 5.5*:

Layer (type)	Output Shape	Param #
prune_low_magnitude_conv2d_4 (PruneLowMagnitude)	(None, 26, 26, 32)	610
prune_low_magnitude_max_pooling2d_4 (PruneLowMagnitude)	(None, 13, 13, 32)	1
prune_low_magnitude_conv2d_5 (PruneLowMagnitude)	(None, 11, 11, 64)	36930
prune_low_magnitude_max_pooling2d_5 (PruneLowMagnitude)	(None, 5, 5, 64)	1
prune_low_magnitude_flatten_2 (PruneLowMagnitude)	(None, 1600)	1
prune_low_magnitude_dropout_2 (PruneLowMagnitude)	(None, 1600)	1
prune_low_magnitude_dense_4 (PruneLowMagnitude)	(None, 100)	320102
prune_low_magnitude_dense_5 (PruneLowMagnitude)	(None, 10)	2012
=====		
Total params: 359,658		
Trainable params: 179,926		
Non-trainable params: 179,732		

Figure 5.5: Model summary after applying weight pruning

Note the prefix “**prune_low_magnitude**” at each layer. Now, the model will be trained for two epochs on the training set. For the training, we need to use a Keras callback that will update the pruning wrappers with the optimizer step. We will again use 10% of the training data for internal validation. See the following code:

```
>>import tempfile

log_dir = tempfile.mkdtemp()

callbacks = [

    tfmot.sparsity.keras.UpdatePruningStep(),

    tfmot.sparsity.keras.PruningSummaries(log_dir=log_dir)

]
```

```

model_for_pruning.fit(X_train, y_train,
                      batch_size=batch_size, epochs=epochs, validation_
                      split=validation_split,
                      callbacks=callbacks)

```

Once trained, we can convert the model using a TFLite converter and save it in the directory. Remember that before we convert the model, we need to apply `strip_pruning()` to restore the original model with sparse weights. Refer to the following code:

```

>>model_for_export = tfmot.sparsity.keras.strip_pruning(model_for_
pruning)

converter = tf.lite.TFLiteConverter.from_keras_model(model_for_export)
tflite_model_pruned = converter.convert()

tflite_models_dir = pathlib.Path('/content/tflite_models/')
tflite_models_dir.mkdir(exist_ok=True, parents=True)
tflite_model_file = tflite_models_dir/'model_pruned.tflite'
tflite_model_file.write_bytes(tflite_model_pruned)

```

Now, it is time to evaluate the performance of the pruned model `model_pruned.tflite`. We will use the same evaluation code to classify the model on the test set.

```

>>tflite_model_file = 'tflite_models/model_pruned.tflite'

interpreter = tf.lite.Interpreter(model_path=tflite_model_file)
interpreter.allocate_tensors()

input_index = interpreter.get_input_details()[0]['index']
output_index = interpreter.get_output_details()[0]['index']

pred_list = []
for images in X_test:

```

```
input_data = np.array(images, dtype=np.float32)

input_data = input_data.reshape(1, input_data.shape[0], input_data.
shape[1], 1)

interpreter.set_tensor(input_index, input_data)
interpreter.invoke()
prediction = interpreter.get_tensor(output_index)
prediction = np.argmax(prediction)
pred_list.append(prediction)

accurate_count = 0
for index in range(len(pred_list)):
    if pred_list[index] == np.argmax(y_test[index]):
        accurate_count += 1
accuracy = accurate_count * 1.0 / len(pred_list)

print('accuracy = ', accuracy)
```

On executing the preceding code, we will see that the pruned model will give around 90% accuracy on the test set, which is slightly lesser compared to quantization-based optimization. A possible reason for lower accuracy may be due to the high amount of sparsity applied to the model, which is 75% in our case. You can try to reduce the amount of sparsity in the previous code example and check the performance. Finally, we will measure how much the model is compressed by using sparsity. We will define a function `get_zipped_model()` that saves the model as a compressed ZIP file and print the size of the zipped model. Refer to the following code:

```
>>def get_gzipped_model(file):
    import os
    import zipfile

    _, zipped_file = tempfile.mkstemp('.zip')
```



```

with zipfile.ZipFile(zipped_file, 'w', compression=zipfile.ZIP_
DEFLATED) as f:

    f.write(file)

return os.path.getsize(zipped_file)

```

Now, let us check the model size after compressing it as ZIP files. We will first print the zipped size of the original TFLite model, followed by the pruned model.

```
>>print('Size of compressed baseline model: %.2f bytes' % (get_
gzipped_model('tflite_models/model.tflite')))
```

```
>>print('Size of zipped pruned TFlite model: %.2f bytes' % (get_
gzipped_model('tflite_models/model_pruned.tflite')))
```

You can see that the size of the original TFLite model after compression is around 671 kilobytes, whereas the size of the pruned model is around 245 kilobytes.

In the previous example, we applied weight pruning to the entire model. Similar to quantization-aware training, we can prune only a few selected layers. The following code example will prune only the dense layers of the model. We will define a function `apply_pruning()` and pass the dense layers to be pruned.

```

>>def apply_pruning(layer):

    if isinstance(layer, tf.keras.layers.Dense):

        return tfmot.sparsity.keras.prune_low_magnitude(layer)

    return layer

model_for_pruning = tf.keras.models.clone_model(

    baseline_model,

    clone_function=apply_pruning)

```

You can now train the model and evaluate it on the test set. As expected, the model will be larger than the fully pruned model, but you will likely to get a better classification accuracy.

Weight clustering

Another efficient optimization technique is weight clustering which aims to reduce the number of unique weight values in a model. We only need to store the unique weight values in the model, which requires less memory space to store in the target device. Clustering algorithms are used in machine learning to deal with unlabelled data. Clustering is an unsupervised machine learning approach that divides data points into groups (clusters) such that all data points having similar statistical properties are grouped together. We can roughly assume that all members inside one cluster belong to the same category. In weight clustering, the weight values in each layer of a model are divided into a given number of clusters. All members in a particular cluster are represented by a single value. For example, the cluster centroid can be used as the representative value for all the weights in that cluster. Suppose you have 100 weights in a layer. You divide them into 10 different clusters. Then, all the weights in that layer can be represented by 10 unique values corresponding to the 10 cluster centroids. You can achieve a more compressed model by using a lesser number of clusters.

In the following example, we will apply weight clustering using the function `tfmot.clustering.keras.cluster_weights`. Load the baseline model and execute the following code:

```
>>cluster_weights = tfmot.clustering.keras.cluster_weights

    CentroidInitialization = tfmot.clustering.keras.CentroidInitialization

    clustering_params = {
        'number_of_clusters': 16,
        'cluster_centroids_init': CentroidInitialization.KMEANS_PLUS_PLUS
    }

    clustered_model = cluster_weights(baseline_model, **clustering_params)
```

Here, we divide the weights of each layer into 16 clusters. The cluster centroids are learnt using K-Means algorithm. The centroids are initialized by the K-Means++ algorithm. Similar to quantization-aware training and weight pruning, we again need to fine-tune the model to optimize the performance. So, we will retrain it on the entire training set. Refer to the following code. Here, we retrain with a much smaller learning rate compared to the base model.

```
>>opt = tf.keras.optimizers.Adam(learning_rate=1e-5)
clustered_model.compile(
    loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
    optimizer=opt,
    metrics=['accuracy'])
```

A summary of the model is shown in *figure 5.6*:

Layer (type)	Output Shape	Param #
cluster_conv2d (ClusterWeights)	(None, 26, 26, 32)	624
cluster_max_pooling2d (ClusterWeights)	(None, 13, 13, 32)	0
cluster_conv2d_1 (ClusterWeights)	(None, 11, 11, 64)	36944
cluster_max_pooling2d_1 (ClusterWeights)	(None, 5, 5, 64)	0
cluster_flatten (ClusterWeights)	(None, 1600)	0
cluster_dropout (ClusterWeights)	(None, 1600)	0
cluster_dense (ClusterWeights)	(None, 100)	320116
cluster_dense_1 (ClusterWeights)	(None, 10)	2026
=====		
Total params: 359,710		
Trainable params: 179,990		
Non-trainable params: 179,720		

Figure 5.6: Model summary after applying weight clustering

We will now retrain the model for two epochs.

```
>>clustered_model.fit(
    X_train,
    y_train,
    batch_size=128,
```

```
epochs=2,  
validation_split=0.1)
```

Once the model is fine-tuned, we will convert it into a TFLite model and save it in the directory. Similar to weight pruning, we need to apply the `strip_clustering()` method before in order to restore the model with clustered weights. Refer to the following code:

```
>>model_for_export = tfmot.clustering.keras.strip_clustering(clustered_  
model)  
  
converter = tf.lite.TFLiteConverter.from_keras_model(model_for_export)  
tflite_model_clustered = converter.convert()  
  
tflite_models_dir = pathlib.Path('/content/tflite_models/')  
tflite_models_dir.mkdir(exist_ok=True, parents=True)  
tflite_model_file = tflite_models_dir/'model_clustered.tflite'  
tflite_model_file.write_bytes(tflite_model_clustered)
```

Finally, execute the following code to evaluate the clustered TFLite model on the test set:

```
>>tflite_model_file = 'tflite_models/model_clustered.tflite'  
interpreter = tf.lite.Interpreter(model_path=tflite_model_file)  
interpreter.allocate_tensors()  
  
input_index = interpreter.get_input_details()[0]['index']  
output_index = interpreter.get_output_details()[0]['index']  
  
pred_list = []  
for images in X_test:  
    input_data = np.array(images, dtype=np.float32)
```

```

input_data = input_data.reshape(1, input_data.shape[0], input_data.
shape[1], 1)

interpreter.set_tensor(input_index, input_data)
interpreter.invoke()
prediction = interpreter.get_tensor(output_index)
prediction = np.argmax(prediction)
pred_list.append(prediction)

accurate_count = 0
for index in range(len(pred_list)):
    if pred_list[index] == np.argmax(y_test[index]):
        accurate_count += 1
accuracy = accurate_count * 1.0 / len(pred_list)

print('accuracy ', accuracy)

```

The clustered model will give around 91% accuracy on the test set, which is very close to the accuracy reported by the baseline model as well as the quantized model.

Now, once again, we will again call our previously defined function `get_gzipped_model()` to get the size of the zipped cluster model. Execute the following code:

```
>>print('Size of zipped clustered TFlite model: %.2f bytes' % (get_gzipped_model('tflite_models/model_clustered.tflite')))
```

The compressed model size is around 130 kilobytes. Remember in the previous example, the size of the compressed TFLite model is 245 kilobytes after pruning. Thus, we have achieved a smaller model by weight clustering along with a better model performance.

Like other optimization techniques, weight clustering can be applied to a few selected layers of a model. The following code applies weight clustering to the dense layers of the baseline model.

```
>>def apply_clustering(layer):
    if isinstance(layer, tf.keras.layers.Dense):
```

```

    return cluster_weights(layer, **clustering_params)

return layer

clustered_model = tf.keras.models.clone_model(
    baseline_model,
    clone_function=apply_clustering,
)

```

A brief summary of different optimization techniques in terms of model accuracy and size is provided in *table 5.1*.

Table 5.1: Impact of various optimization techniques on the CNN model on Fashion_MINIST

Model description	Accuracy on the Fashion-MNIST test set	TFLite Model size
Baseline model (no optimization)	0.91	722 kilobytes
Post-training quantization	0.91	188 kilobytes
Quantization aware training	0.92	188 kilobytes
Weight pruning	0.90	245 kilobytes (after zipped)
Weight clustering	0.91	130 kilobytes (after zipped)

Collaborative optimization

So far in this chapter, we have discussed three different optimization techniques, quantization, weight pruning, and weight clustering. All the optimization techniques have their own advantages and disadvantages. Now a question may arise, whether we can apply more than one optimization technique to one model. The answer is yes. In collaborative optimization, we can apply multiple quantization techniques one after another. This often helps in achieving the best model performance on the target platform in terms of accuracy, size, and inference latency.

Collaborative optimization can have various combinations based on your choice. In this section, we will apply sparsity-preserving clustering to the baseline model as an example of collaborative optimization. In this approach:

1. First, the baseline model will be pruned to add sparsity.
2. Next, we will apply weight clustering to the pruned model, ensuring the model sparsity is preserved.

3. Finally, we will apply post-training quantization and convert the model into a full integer TFLite model.

We will start implementing by loading the baseline model.

```
>>baseline_model = load_model('baseline_model.h5')
```

In the first stage, we will fine-tune the baseline model by applying sparsity. We will define a pruning schedule to apply constant 50% sparsity and retrain the model. As mentioned earlier, we need to apply a Keras callback during the training. See the following code:

```
>>import tensorflow_model_optimization as tfmot

prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude

pruning_params = {
    'pruning_schedule': tfmot.sparsity.keras.ConstantSparsity(0.5,
        begin_step=0, frequency=100)
}

callbacks = [
    tfmot.sparsity.keras.UpdatePruningStep()
]

pruned_model = prune_low_magnitude(baseline_model, **pruning_params)

pruned_model.summary()
```

Now, we can compile our model and train. We will use an Adam optimizer with a smaller learning rate. The training will be done for three epochs on the entire training set. A small portion of data containing 10% of the training data is used as a validation set. Execute the below code:

```
>>opt = tf.keras.optimizers.Adam(learning_rate=1e-5)

pruned_model.compile(
    loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
    optimizer=opt,
```

```
metrics=['accuracy'])
```

```
pruned_model.summary()
```

```
pruned_model.fit(X_train, y_train, batch_size=128, epochs=3, validation_
split=0.1, callbacks=callbacks)
```

Recall our previous example of weight pruning. Once the training is done, we need to apply the `strip_pruning()` method to restore the original model with sparse weights. The same operation will be done here. We will also create a clone of the model for future usage. Refer to the following code:

```
>>stripped_pruned_model = tfmot.sparsity.keras.strip_pruning(pruned_
model)
```

```
# make a cloning of the model
```

```
stripped_pruned_model_copy = tf.keras.models.clone_model(stripped_
pruned_model)
```

```
stripped_pruned_model_copy.set_weights(stripped_pruned_model.get_
weights())
```

Now, we will apply sparsity-preserving clustering to the pruned model. We will use eight clusters in each layer. The cluster centroids are initialized by the K-Means++ algorithm. Refer the below code:

```
>>from tensorflow_model_optimization.python.core.clustering.keras.
experimental import (
```

```
    cluster,
```

```
)
```

```
cluster_weights = tfmot.clustering.keras.cluster_weights
```

```
CentroidInitialization = tfmot.clustering.keras.
CentroidInitialization.KMEANS_PLUS_PLUS
```

```
cluster_weights = cluster.cluster_weights
```



```
clustering_params = {
    'number_of_clusters': 8,
    'cluster_centroids_init': CentroidInitialization.KMEANS_PLUS_PLUS,
    'preserve_sparsity': True
}
```

```
sparsity_clustered_model = cluster_weights(stripped_pruned_model_copy,
**clustering_params)
```

Note that while defining the clustering parameters, we have passed the parameter *preserve_sparsity* as “True” to preserve the model sparsity achieved in the previous step. Now, we will again train the model for another three epochs.

```
>>sparsity_clustered_model.compile(optimizer='adam',
                                   loss=tf.keras.losses.CategoricalCrossentropy(from_
                                   logits=True),
                                   metrics=['accuracy'])

sparsity_clustered_model.fit(X_train, y_train, batch_size=128,
                             epochs=3, validation_split=0.1)
```

Once trained, we will restore the model by applying the `strip_clustering()` method.

```
>>stripped_sparsity_clustered_model = tfmot.clustering.keras.strip_
clustering(sparsity_clustered_model)
```

Finally, we will apply post-training quantization on the sparsity-preserved clustered model and convert it into a full integer TFLite model to save in the file system.

```
>>converter = tf.lite.TFLiteConverter.from_keras_model(stripped_sparsity_
clustered_model)

converter.optimizations = [tf.lite.Optimize.DEFAULT]

sparsity_clustered_quant_model = converter.convert()

tflite_models_dir = pathlib.Path('/content/tflite_models/')

tflite_models_dir.mkdir(exist_ok=True, parents=True)
```

```
tflite_model_file = tflite_models_dir/'model_sparsity_clustered_quant.tflite'
```

```
tflite_model_file.write_bytes(sparsity_clustered_quant_model)
```

Now, we have created and saved the TFLite model, `model_sparsity_clustered_quant.tflite`. Let us evaluate the model performance on the entire test set:

```
>>tflite_model_file = 'tflite_models/model_sparsity_clustered_quant.tflite'
interpreter = tf.lite.Interpreter(model_path=tflite_model_file)
interpreter.allocate_tensors()

input_index = interpreter.get_input_details()[0]['index']
output_index = interpreter.get_output_details()[0]['index']

pred_list = []
for images in X_test:
    input_data = np.array(images, dtype=np.float32)

    input_data = input_data.reshape(1, input_data.shape[0], input_data.shape[1], 1)

    interpreter.set_tensor(input_index, input_data)
    interpreter.invoke()
    prediction = interpreter.get_tensor(output_index)
    prediction = np.argmax(prediction)
    pred_list.append(prediction)

accurate_count = 0
for index in range(len(pred_list)):
    if pred_list[index] == np.argmax(y_test[index]):
        accurate_count += 1
accuracy = accurate_count * 1.0 / len(pred_list)

print('accuracy = ', accuracy)
```

You will get around 90.5% accuracy on the test set. It is important to remember that in this approach, we have applied 50% sparsity to the baseline model, followed by weight clustering using only eight clusters. You can get a more accurate model by applying a reduced model sparsity and a higher number of clusters in each layer.

Finally, we will call the function `get_gzipped_model()` to obtain the model size after compression.

```
>>print('Size of zipped sparsity preserved clustered TFlite model: %.2f bytes' % (get_gzipped_model('tflite_models/model_sparsity_clustered_quant.tflite')))
```

The compressed model size is around 61 kilobytes. The size of the baseline unoptimized TFLite model was 722 kilobytes. Recall that in our previous examples, the size of the compressed pruned model was 245 kilobytes, and the size of the compressed clustered model was 130 kilobytes. So, we can achieve a much smaller model using collaborative optimization.

In this example, we have shown only one type of collaborative optimization, the sparsity-preserving clustering. For experimental purposes, you can change the sequence of different optimization techniques on the base model and check the corresponding impact on model size and accuracy.

Conclusion

Deep learning models are often large in size. A standard deep learning model needs to undergo certain optimization steps in order to effectively deploy on resource-constrained hardware. Model optimization is a critical part in TinyML. In this chapter, we have experienced how a large neural network can be effectively optimized using TensorFlow libraries. We have learnt about TensorFlow Lite, a specially designed TensorFlow library to convert TensorFlow models into a portable FlatBuffers format for efficiently deploying on a plethora of smaller edge devices and microcontrollers. We have also learnt how a TensorFlow model can be effectively compressed within an accepted range of performance drop using the TensorFlow Optimization Toolkit. TensorFlow Optimization Toolkit supports various optimization algorithms such as quantization, weight pruning, and clustering. Different optimization techniques have different impacts on the model performance. In this chapter, we have created a baseline CNN model for classifying different fashion products on a public database. Subsequently, we have applied various optimization techniques on the baseline model and evaluated their impact on model size and accuracy. In general, model compression often comes with a performance drop. In order to get the optimum performance, the compressed model requires to fine-tune via retraining.

Now, we have some basic knowledge in optimizing a machine learning model to deploy as a TinyML application. However, we have done all the model designing as well as model inference on Colab notebook using cloud connectivity. In the upcoming chapter, we will deploy a compressed TensorFlow model on a real edge device to make inference.

Key facts

- TensorFlow Lite comprises a set of tools for compressing machine learning models to run on low-powered edge devices.
- A TensorFlow Lite model is converted into a portable FlatBuffers format, and the resulting file is stored in **.TFLITE** format.
- TensorFlow Lite supports various platforms, such as Android, iOS, embedded Linux, and microcontrollers. It also supports diverse programming languages such as C, C++, JAVA, Python, and so on.
- With TensorFlow Optimization Toolkit, we can efficiently compress a large machine learning model with a minimum performance loss.
- The precision of the model weights and activation can be reduced from 32-bit floating points to 8-bit integers using quantization, resulting in a 4x smaller and much faster model.
- Weight pruning introduces sparsity to a model by zeroing out some insignificant weights.
- Weight clustering reduces the number of unique weight values in a model.
- The optimized models often require retraining to fine-tune their performance.
- In collaborative optimization, we can incorporate multiple optimization steps into a model that often results in a better performance.

Join our book's Discord space

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

<https://discord.bpbonline.com>



CHAPTER 6

Deploying My First TinyML Application

Introduction

By now, we have become familiarized with TensorFlow in building machine learning models. Machine learning, particularly the neural network models are resource hungry. Whereas, TinyML applications are intended to run on smaller edge devices with limited hardware resources in terms of processing power and inbuilt memory. In *Chapter 5, Model Optimization Using TensorFlow*, we saw how a large TensorFlow model can be effectively optimized into a much smaller TFLite model for smaller edge devices. However, in all previous chapters, we have used the Colab notebook to develop our machine learning models. Colab is a browser-based development notebook where we actually run our Python scripts on a remote server connected via the internet, which is equipped with a powerful **Central Processing Unit (CPU)**, **Graphics Processing Unit (GPU)**, or **Tensor Processing Unit (TPU)** to perform heavy computation. Although the previous chapter discussed how to compress a large TensorFlow model into a much smaller TFLite model, we still trained the models and made inferences on Colab. In this chapter, we will create our first real-world TinyML application for image classification and deploy it on an edge device to make inferences. For this project, we will use Raspberry Pi as our target device to run the application. As discussed in *Chapter 1*, Raspberry Pi is a commercially

available small low-cost **Single Board Computer (SBC)** popularly used in edge computing and various **Internet of Things (IoT)** applications. In terms of processing power, it is quite comparable to modern smartphones. Unlike microcontrollers, it has a dedicated Linux-based operation system that enables it to be a fully-fledged computer with limited computing capacity. You can easily interact with a Raspberry Pi with standard input/output devices. It also supports a wide range of programming languages. You can even run Python programs on it. Moreover, duly optimized TensorFlow models are well-supported to run on Pi. Although the main goal of this book is to deploy TinyML applications on microcontrollers which only support low-level programming language equivalent to C/C++, this chapter will give you a feel of how to create a practical machine learning model from scratch and to deploy it on an edge device. Since Raspberry Pi comes with a Python interpreter, you can readily deploy your model without converting in another programming language. In later chapters, we will learn how to further convert a TFLite model into the equivalent libraries for deploying on microcontrollers.

Structure

In this chapter, we will discuss the following topics:

- The MobileNet architecture
 - Depthwise separable convolution
- Image classification using MobileNet
 - Brief introduction to transfer learning
 - Implementing MobileNet using transfer learning
 - Creating an optimized model for smaller target device
 - Evaluation of the model on the test set
- Introduction to Raspberry Pi
- Getting started with the Pi
 - Installing the operating system
 - Setting up the Pi
 - Remotely accessing the Pi
- Deploying the model on Raspberry Pi to make inference

Objectives

In this chapter, we will create our first TinyML project on a commercial edge device, the Raspberry Pi, for on-device image classification. The project will involve the following steps. We will first design a Convolutional Neural Network (CNN) for image classification. The model will be trained and evaluated on the CIFAR-10 database, another popular open-access database containing various types of low-resolution color images for different types of living and nonliving things such as cats, dogs, automobiles, ships, and so on.

In all previous chapters, we defined our own CNN structures for classification. However, a simple CNN may not give the desired output on non-trivial databases such as CIFAR-10. In this chapter, we will use a new CNN-based deep learning architecture, MobileNet, which is a specially designed neural network to run on low-powered edge devices. Rather than training the model from scratch, we will use the weights of MobileNet already trained on another dataset and modify only the last few layers of the architecture by training on CIFAR-10. The concept is called transfer learning, where a model pretrained on one dataset can be marginally trained on another similar dataset for only few epochs but still gives very good classification accuracy on the second dataset. The chapter has two parts. In part one, we will train and evaluate our MobileNet and convert it into the equivalent TFLite model. This part of the project will be done on Colab. In the second part of the project, the TFLite model will be deployed on a Raspberry Pi device to make inferences on offline images. We will learn in a step-by-step manner to set up the Raspberry Pi by installing the necessary software to run a TensorFlow model.

The MobileNet architecture

MobileNet is an efficient and portable Convolutional Neural Network (CNN) architecture designed for low-powered, resource-constrained mobile and embedded devices. The architecture was proposed by *Andrew Howard* and colleagues in 2017 in a paper, “**MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.**” Instead of performing standard convolution operations, MobileNets are based on an architecture that performs depthwise separable convolution, an efficient convolution operation that results in a much lighter model. The architecture maintains an efficient trade-off between latency and accuracy using two model hyperparameters, width multiplier and resolution multiplier so that the developers can choose the right model size for their applications.

Depthwise separable convolution

The MobileNet model is based on depthwise separable convolution. A depthwise separable convolution layer consists of two separate operations, a depthwise convolution and a pointwise convolution. Suppose you have a color input image of $64 \times 64 \times 3$, that is, having a tensor shape of $(64, 64, 3)$, and you want to perform the convolution with 32 filters having a kernel size of 3×3 . You have also applied zero padding to the input. Hence, the desired output tensor shape after convolution will be $(64, 64, 32)$. Under standard convolution, there will be an element-wise multiplication between the input and the filters. Since there are 32 filters, for every pixel in the input, there will be $3 \times 3 \times 32$ multiplication. The total number of multiplications required to cover the image will be: $(64 \times 64 \times 3) \times (3 \times 3 \times 32) = 35,38,944$, which is definitively quite a big amount of mathematical operations.

Under depthwise separable convolution, we perform the following steps:

1. First, we apply depthwise convolution. Unlike standard convolution, depthwise convolution is performed to a single channel at a time. The kernel size for this will be $3 \times 3 \times 1$. So, for all three input channels, the number of multiplications will be: $(64 \times 64) \times (3 \times 3 \times 1) \times 3 = 1,10,592$.
2. Next, we will perform pointwise convolution using 1×1 kernel on all three input channels. So, kernel size will be $1 \times 1 \times 3$. Since we have 32 output channels, the number of required multiplication for all the pixels will be: $(64 \times 64) \times (1 \times 1 \times 3) \times 32 = 3,93,216$

So, the total number of operations are : $1,10,592 + 3,93,216 = 5,03,808$. In comparison to standard convolution, the total number of mathematical operations are reduced by a factor of 7. The number of trainable parameters is also reduced by a similar factor, resulting in a smaller model size and faster inference speed.

The architecture of MobileNet is quite similar to a standard deep CNN. The first layer of the architecture is generally a standard convolution layer. Subsequently, there are multiple depthwise separable convolution layers with associated batch normalization and ReLU activation layers. Depending upon the target application, the architectural topology can be modified. The final layer contains a softmax function for classification.

Image classification using MobileNet

In this project, we will design a MobileNet architecture in TensorFlow for image classification on a real-world edge device. We will primarily use the CIFAR-10 database to train and evaluate our model. As mentioned earlier, the project will be

implemented in two parts. Initially, we will define our network, train, and internally evaluate the model, and then convert it to the equivalent TFLite model. This part of the project will be done in Colab. In the second part, we will deploy the TFLite model to a Raspberry Pi device to make inferences. Note: Training a machine learning model is computationally expensive. In TinyML, we mostly train our model on a powerful server or cloud using the power of GPU or TPU. Since training is a one-time job in most applications, we can afford to train offline. Certain applications like biometric authentication require more frequent training, which is required to be done on-device. In the upcoming chapter, we will implement a project that requires on-device training.

Like MNIST and fashion-MNIST, CIFAR-10 is another open-access computer vision database popularly used for the evaluation of object detection algorithms. The original database was collected by *Alex Krizhevsky*, *Vinod Nair*, and *Geoffrey Hinton*. It is a subset of 80 million tiny images, comprising 60,000 color images with 32×32 resolution that are categorized into 10 different classes. Each class has 6,000 examples. There are 50,000 training images and 10,000 test images in the database. The numerical values corresponding to different class labels are as follows:

- 0—Airplane
- 1—Automobile
- 2—Bird
- 3—Cat
- 4—Deer
- 5—Dog
- 6—Frog
- 7—Horse
- 8—Ship
- 9—Truck

Create a new Colab notebook to implement the project and connect to a Python runtime. For this project, we strongly recommend to utilize the power of GPU for a faster training. Go to the **Runtime** tab on your notebook, click on **Change runtime type**, and select **GPU** from **Hardware accelerator**. Similar to MNIST and Fashion-MNIST databases, CIFAR-10 is readily available under TensorFlow datasets. So, we can load it using a single line of code.

Let us first import the necessary libraries.

```
>>import numpy as np
```

```
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout
```

Now, let us load the database by executing the following command:

```
>>(trainX, trainY), (testX, testY) = cifar10.load_data()
```

Next, we will check the number of images in the training and test set and their dimension. Execute the following code:

```
>>print('training set ', '\nData :', trainX.shape, '\nLabel :', trainY.
shape)
    print('\ntest set', '\nData :', testX.shape, '\nLabel :', testY.shape)
```

The training set has 50,000 images, and the test set has 10,000 images. Each color image has a dimension of $32 \times 32 \times 3$.

Now, let us plot a few random sample images from the training set along with their class labels. Execute the following code to randomly select and print nine images from the training set:

```
>>from numpy import random

plt.rcParams['figure.figsize'] = (7,7)

for i in range(9):
    plt.subplot(3,3,i+1)
    num = random.randint(0, len(trainX))
    plt.imshow(trainX[num], cmap='gray')
    plt.title("Class {}".format(trainY[num]))

plt.tight_layout()
```

The code output on one execution is shown in *figure 6.1*:

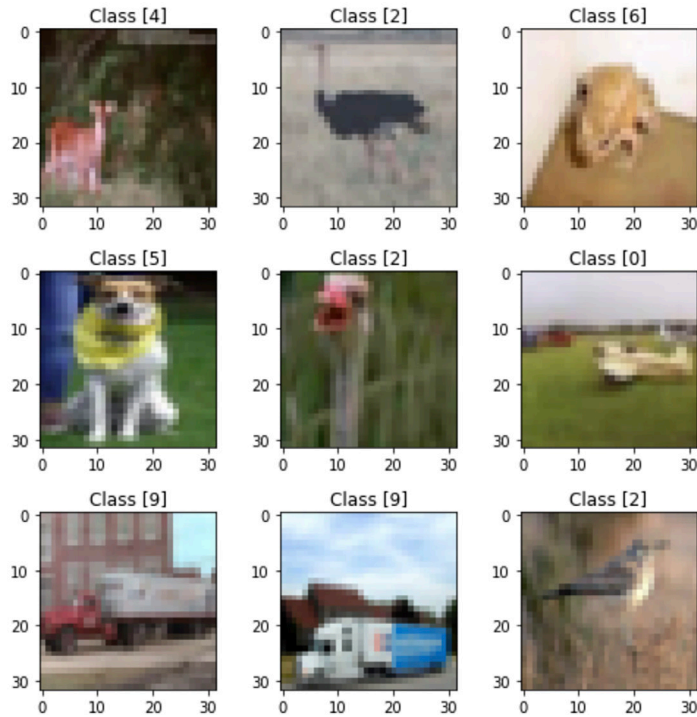


Figure 6.1: Sample images from the CIFAR-10 database

It is evident that CIFAR-10 is a complex dataset compared to MNIST. The images are collected in more practical scenarios. Many of those images have ambiguous background. Now, we will build our MobileNet model for classification. But before doing that, we need to convert the class labels to one hot encoded vector. We will also do a pre-processing step to scale down the pixel values of the images between 0 and 1. Refer to the following code to perform the operations:

```
>>from tensorflow.keras.utils import to_categorical

trainX = trainX.astype('float32') # change integers to 32-bit floating
point numbers

testX = testX.astype('float32')

trainX /= 255 # normalize the input

testX /= 255
```

```
trainY_one_hot = to_categorical(trainY)
```

```
testY_one_hot = to_categorical(testY)
```

Now, we can define our MobileNet. A major advantage of TensorFlow is that it contains a number of popular deep learning models along with the model weights, pretrained on large databases. You can load any of these models along with the pretrained weights using a single function call and can readily use them in your applications. In machine learning, we can reuse the weights of a pretrained model obtained on one database to evaluate on other related databases with a minimum retraining. The concept is called **transfer learning**. It has been very popular in modern days' data science and machine learning applications, where you do not always have a sufficient amount of labelled training data to train your model from scratch.

Brief introduction to transfer learning

The concept of transfer learning is quite analogous to the human learning process. If you know how to drive a car on Indian roads, you can quite easily adjust your skill to driving on the roads of the USA. However, there are different driving rules in the USA compared to India. For example, in India, we drive on the left side of the road, whereas in the USA, they drive on the right side of the road. However, you can adopt to these rules based on your primary driving skill. Similarly, in transfer learning, a model trained to perform one task can transfer its knowledge to perform other similar tasks. Transfer learning is particularly useful when you do not have sufficient labelled data to optimally train your model. For example, suppose you want to design a classifier to classify your own handwritten digits. But you have a relatively smaller training dataset of your own handwriting containing around 500 examples. It might be difficult to optimally train a neural network from scratch on such a smaller dataset. In one approach, you may think to train your network on a public dataset like MNIST that contains thousands of handwritten digits. However, you might not get the optimum performance on evaluating the model on your own handwritten dataset as the images in the two datasets might have different properties in terms of image resolutions, backgrounds, and writing styles. The problem can be solved using transfer learning. You can take the weights of a neural network trained on MNIST database and incrementally adjust a few of the weights on your small training dataset. This not only saves training time but also mitigates the dependency on application-specific datasets.

In a deep CNN architecture, the convolutional layers in the first few layers usually try to learn simple features like horizontal and vertical edges from the input images. The shape-related information are learnt in the middle layers, and finally, the detailed application-specific features are learnt in the later layers. By using transfer learning, the pretrained model weights for the first few and middle layers can be reused which are responsible for extracting generic features. We only modify the weights of the deeper layers on the target dataset based on the actual application. In summary, you can use transfer learning under the following scenarios:

- You do not have enough training data to train a model from scratch.
- You do not have sufficient infrastructure to train a model from scratch.
- There already exists a pretrained network for a similar application.

Implementing MobileNet using transfer learning

In this project, we will reuse the weights of MobileNet pretrained on the ImageNet database and apply transfer learning to train on CIFAR-10. Although CIFAR-10 has sufficient training data, with transfer learning, we can get a good model with fewer number of epochs that significantly reduces the training time. The ImageNet project is a large visual database designed for object recognition research. The images in ImageNet are organized according to WordNet hierarchy. All categories are derived from an ontology root in WordNet, and most of them are subsets of physical entity. There are more than 14 million hand-annotated images in more than 20,000 categories in ImageNets. Since 2010, ImageNet has been organizing an annual contest asking for software programs to classify objects and scenes.

Now, let us understand the model we are going to design for our application. We will use the top layers of the pretrained MobileNet model. We will also define a *base model* containing the final few dense layers that will be added to the pretrained MobileNet. We will also apply quantization-aware training to reduce the model precision, which will give us a smaller model for the target device.

In TensorFlow, we can directly load the MobileNet weights pretrained on the ImageNet database using a single function call. See the following code:

```
>>from tensorflow.keras.applications import MobileNet

pretrained_model = MobileNet(include_top=False, weights='imagenet',
                              input_shape=(32,32,3))
```

Carefully note the preceding function arguments. We have loaded the weight values pretrained on ImageNet. We have passed the input shape according to the images in CIFAR-10. The parameter `include_top = False` indicates that we are not taking the final layer of the pretrained MobileNet. We will define the final few dense layers as per our application. Execute the command `pretrained_model.summary()` on Colab to get the details of different layers of the MobileNet. The model has more than 3 million parameters. Output dimension of the final layer is (1, 1, 1024).

Now, we will define a function for the *base model* containing the final few layers. The base model will be added to the pretrained MobileNet model to define our complete model. Refer to the following code:

```
>>def base_model():  
    model = Sequential()  
    model.add(Dropout(0.3, input_shape = (1, 1, 1024)))  
    model.add(Flatten())  
    model.add(Dense(128,activation='relu'))  
    model.add(Dropout(0.3))  
    model.add(Dense(10,activation='softmax'))  
    return model
```

As shown, the base model contains dropout and dense layers. The input shape has to be equal to the shape of the output layer of the pretrained MobileNet. Since there are 10 different classes in CIFAR-10, the final dense layer will have 10 nodes.

Creating an optimized model for a smaller target device

Now, we will define the complete model. We will apply quantization-aware training during training to get a compressed model. See the following code:

```
>>pip install -q tensorflow-model-optimization  
  
import tensorflow_model_optimization as tfmot  
  
pretrained_model.trainable = True
```

```

basemodel = base_model()

q_pretrained_model = tfmot.quantization.keras.quantize_
model(pretrained_model)

q_base_model = tfmot.quantization.keras.quantize_model(basemodel)

original_inputs = tf.keras.layers.Input(shape=(32, 32, 3))

y = q_pretrained_model(original_inputs)

original_outputs = q_base_model(y)

model = tf.keras.Model(original_inputs, original_outputs)

```

The baseline model is appended to the pretrained MobileNet to define the end-to-end model. By setting `pretrained_model.trainable = True`, we ensure the pretrained weights will be modified during training. If the parameter is set as `False`, only the weights of the base model will be updated. The function `tfmot.quantization.keras.quantize_model()` is individually applied on both the pretrained MobileNet and the baseline models for creating an integer-based model. Summary of the final model is shown in *figure 6.2*:

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 32, 32, 3)]	0
mobilenet_1.00_224 (Functional)	(None, 1, 1, 1024)	3241000
sequential (Sequential)	(None, 10)	132506
=====		
Total params: 3,373,506		
Trainable params: 3,339,466		
Non-trainable params: 34,040		

Figure 6.2: Model summary for image classification on CIFAR-10

The model has around 3.73 million parameters. Now, we will compile and train the model on the training set of CIFAR-10. The categorical cross entropy loss function will be minimized using an Adam optimizer, and the training will be done for 20 epochs. Ten percent of data from the training set is selected as a validation set to evaluate the model performance after each epoch.

```
>>model.compile(loss = 'categorical_crossentropy', optimizer = 'adam',
                metrics = ['accuracy'])

model.fit(trainX, trainY_one_hot, batch_size = 100, epochs = 20,
          validation_split = 0.1)
```

It will take around 20 minutes to train the model on Colab using GPU as a runtime. By the end of 20 epochs, you can expect around 81% accuracy on the validation set. Now, it is time to evaluate your model on the test set using the following code:

```
>>model.evaluate(testX, testY_one_hot)[1]
```

You will get a classification accuracy similar to what was achieved on the validation set (around 81%). Now, we will quantize it into an integer-based model, convert into the equivalent TFLite model and save it in our Colab workspace. Refer to the following code.

```
>>converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model_qat = converter.convert()

import pathlib
tflite_models_dir = pathlib.Path('/content/tflite_models/')
tflite_models_dir.mkdir(exist_ok=True, parents=True)
tflite_model_file = tflite_models_dir/'model_qat.tflite'
tflite_model_file.write_bytes(tflite_model_qat)
```

The preceding code will create a directory `tflite_models` in your workspace and save the model `model_qat.tflite` in the directory. The compressed model has a size of 3.65 megabytes. It is strongly recommended to download the TFLite file to your host machine for offline access, as it can be deleted from Colab if the runtime is disconnected. In order to download a file from Colab, right-click on it and select **Download**. We will need this file later once we will deploy it on a Raspberry Pi for inference.

Evaluation of the model on the test set

Now, we have a training model. Let us execute it on the test set for performance evaluation. Refer to the following code to print the TFLite model accuracy on the test set:

```
>>tflite_model_file = 'tflite_models/model_qat.tflite'

interpreter = tf.lite.Interpreter(model_path=tflite_model_file)
interpreter.allocate_tensors()

input_index = interpreter.get_input_details()[0]['index']
output_index = interpreter.get_output_details()[0]['index']

pred_list = []

for images in testX:
    input_data = np.array(images, dtype=np.float32)

    input_data = input_data.reshape(1, input_data.shape[0], input_data.
    shape[1], 3)

    interpreter.set_tensor(input_index, input_data)
    interpreter.invoke()
    prediction = interpreter.get_tensor(output_index)
    prediction = np.argmax(prediction)
    pred_list.append(prediction)

accurate_count = 0
for index in range(len(pred_list)):
    if pred_list[index] == np.argmax(testY_one_hot[index]):
        accurate_count += 1
```

```
accuracy = accurate_count * 1.0 / len(pred_list)
```

```
print('accuracy = ', accuracy)
```

The model will yield around 81% accuracy on the test set. With this, we finish the first part of our project of creating the TFLite model on Colab. In the following sections, we will see how this model can be deployed on a Raspberry Pi for image classification.

Introduction to Raspberry Pi

Raspberry Pi is a series of low-cost small **Single Board Computers (SBC)** popularly used in scientific research in edge analytics, IoT, and also for educational purpose. Originally developed by Raspberry Pi Foundation in association with Broadcom, the first version of the device was launched in the year 2012. Raspberry Pi has a Linux-based operating system, and it supports multiple programming languages such as C, C++, and Python. It also hosts a number of input/output pins to programmatically control external electronic components such as servo motors, temperature sensors, and so on. That is why Raspberry Pi has been extremely popular in physical computing, digital electronic system designing, edge analytics, and IoT applications. Moreover, it has a full support to run sizable TensorFlow models. The current generations of Raspberry Pi have the following versions, Zero, 1, 2, 3, and 4.

Throughout this book, we will use the Raspberry Pi 3 Model B+ to deploy our projects. However, you should be able to run them on other Pi models as well, except Pi Zero.

Figure 6.3 features the Raspberry Pi 3 Model B+:



Figure 6.3: Raspberry Pi 3 Model B+

The Raspberry Pi 3 Model B+, shown in *figure 6.3* has a dimension of 85.6 mm × 56.5 mm, which is similar to the size of a credit card. It comes with a quad-core 1.4 gigahertz CPU and 1 gigabytes of RAM. However, you cannot just start working with a Pi as soon as you have the device. You need to procure few other components as well. Raspberry Pi does not contain any inbuilt memory for data storage. Hence, you need a micro-SD card in order to store the operating system and all your programs. You also need a monitor along with an HDMI cable to connect to your Pi for display. You also need a USB keyboard and mouse as input devices. Finally, you need a micro-USB cable and adaptor for the power supply. The recommended input voltage is 5 Volt, and the recommended current is 2 Ampere.

Figure 6.4 features a labelled diagram of Raspberry Pi. Although the actual layout of the board may vary in various versions of Raspberry Pi, you will mostly find all the major components in all the versions:

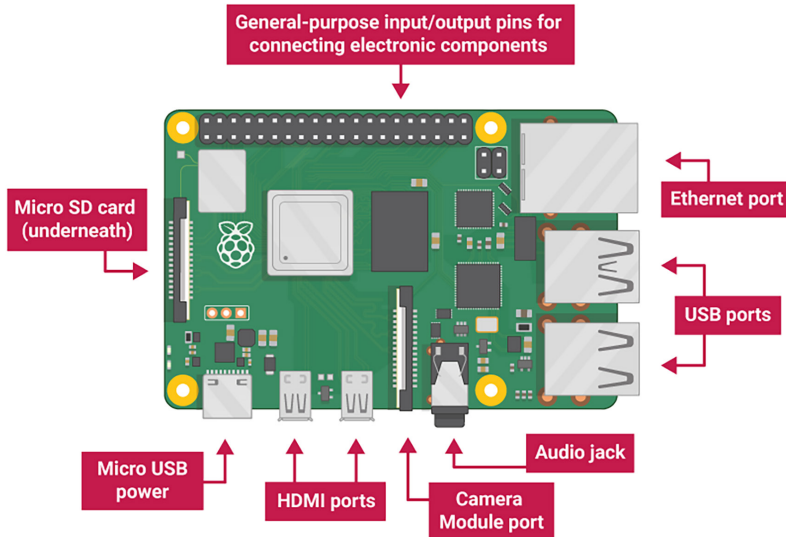


Figure 6.4: Labelled diagram of a Raspberry Pi1

- **Processor:** Raspberry Pi 3 Model B+ comes with a powerful 1.4 gigahertz quad-core ARM Cortex-A53 64-bit processor with 512 kilobytes of shared memory cache. It has 1 gigabytes of RAM.
- **SD card slot:** Raspberry Pi does not have any inbuilt permanent memory. It has a memory card slot where you need to insert a micro-SD card to store the operating system and all other relevant files, including your programs.

1 <https://projects.raspberrypi.org/en/projects/raspberry-pi-getting-started/2>

- **USB ports:** There are 4 USB ports where you can connect peripheral devices such as keyboard and mouse. You can also connect a USB mass storage device to your Pi for file transfer.
- **HDMI port:** Raspberry Pi comes with an HDMI port to connect a monitor for displaying the output.
- **Camera module port:** Raspberry Pi has a dedicated port to connect a camera called the Pi camera. In the upcoming chapter, we will implement a project where we will connect a camera to the Pi for real-time video capturing and analysis.
- **Power Supply:** Raspberry Pi 3 Model B+ comes with a micro-USB connector for an external power supply. It requires a steady 5V power supply. Remember, Pi does not have any physical ON/OFF switch. The device turns on as soon as you connect it to the power supply and turns off when it is disconnected.
- **Ethernet port:** Raspberry Pi has an Ethernet port. It is also equipped with Bluetooth and can be connected to a network via wireless LAN.
- **Audio jack:** You can connect a headphone.
- **GPIO pins:** A powerful feature of Raspberry Pi is the **General-Purpose Input/Output (GPIO)** pins. These are programmatically configurable digital ports where you can interface with various electronic components to your Pi for acquiring data in real-time. The modern Raspberry Pi has 40 pins. GPIO pins can be used to perform a variety of alternative functions. For more details on GPIO pins, you are encouraged to go through the official documentation of Raspberry Pi.

Getting started with the Pi

In this section, we will see how to set up a Raspberry Pi 3 Model B+. As mentioned earlier, you cannot just work with a standalone Pi. You need a few more components, which are as follows:

- A micro-SD card containing the OS.
- A micro-USB power cable.
- A monitor and an HDMI cable. If your monitor does not have an HDMI port, you need a VGA-HDMI adaptor.
- USB keyboard and mouse.
- An internet connection, ethernet, or Wi-Fi.

Installing the operating system

The Raspberry Pi has a dedicated Linux-based operating system called the Raspberry Pi OS (previously known as Raspbian). You can download the OS from the official website². It provides you all the necessary instruction for installing the OS. You are strongly recommended to download the latest version of the OS. Check for all download options on the website. We recommend you to download the OS as an imager format, which is easy to install to a micro-SD card from any host computer or laptop. Your Pi will be fully operational as soon as you insert the micro-SD card in the slot and power it on.

Open a browser on your computer and go to the official website of Raspberry Pi to download the OS. Remember, your host computer needs to have the facility of writing to an SD card. You need to insert the micro-SD card into the card reader slot of the host machine, and it should be in FAT32 format. It is recommended that your SD card has at least 4 gigabytes of free memory space. Once the imager is downloaded, you need to install the application on your computer. It will guide you to write the OS to the micro-SD card. Now, open the application. It will open the window shown in *figure 6.5*.

Select **Raspberry Pi OS (32 bit)** as the operating system and the location of your micro-SD card as your storage location. Clicking the Settings icon lets you set various options, such as providing a username and password for your Pi, setting of the time zone and geolocation, configuring the wireless LAN, and so on. Now click on the **Write** button. The OS will start installing on your micro-SD card. It may take several minutes to complete the installation.

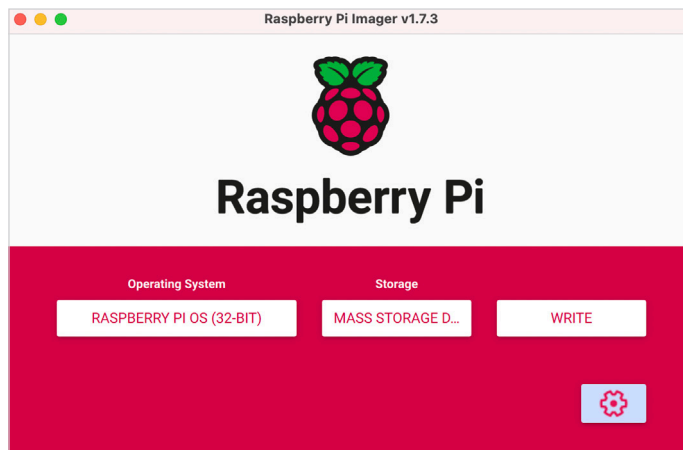


Figure 6.5: Installing Raspberry Pi OS from the imager

² <https://www.raspberrypi.com/software/>

Once the writing is completed, you will be prompted to eject the micro-SD card from the host machine. Your SD card is now ready to be inserted into the Pi.

Setting up the Pi

Now it is time to set up the Pi. First, you need to insert the micro-SD card loaded with the OS into the dedicated card slot of the Pi. Next, connect the peripheral devices, such as the keyboard and mouse, to the Pi via USB ports. Finally, connect the monitor to the Pi via an HDMI port. Now, insert the micro-USB cable into the Pi for the power supply. For power supply, you can use an AC adopter or a power bank or even use the USB port of a desktop. Remember, your Pi requires a steady power supply of 5 Volt. You should connect all the peripheral devices to the Pi before connecting to the power supply.

When you turn on the power supply for the first time, the Pi will do some internal configuration and automatically restart. If everything goes fine, you can expect a screen, as shown in *figure 6.6*, upon restarting.

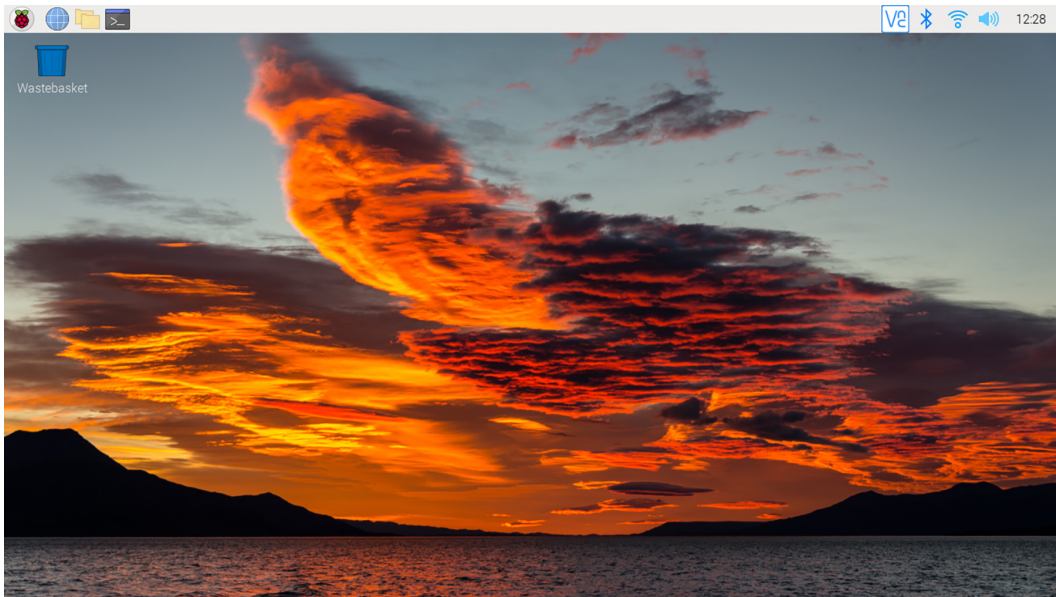


Figure 6.6: Raspberry Pi desktop welcome page

Figure 6.6 shows the home page for one certain version of the OS. Depending upon your installed version, you may see a different home page.

Now, it is time to do some configurations. Follow the steps:

1. First, you need to connect to the internet. You can either use an ethernet cable or connect it to a wireless network.
2. Click on the Raspberry Pi icon at the top left corner of your desktop.
3. Select **Preference** → **Raspberry Pi Configuration**. You will get a dialog box.
4. Click on **Interfaces**, as shown in *figure 6.7*. You will find a set of options. From there, you need to enable **SSH**, **VNC**, **SPI**, **I2C**, and **Serial Port**. Enabling SSH and VNC allows you to remotely access the Pi from another desktop or laptop. SPI, I2C, and Serial port are various ways to communicate with external electronic components such as the camera, OLED display, and temperature sensors to the Pi.
5. Now, restart the Pi.

For first-time users, you should update the software. Open a terminal by clicking on the terminal icon located at the top of the desktop. Execute the following commands. It will take some time to update the software.

```
>>sudo apt-get update
```

```
>>sudo apt-get upgrade
```

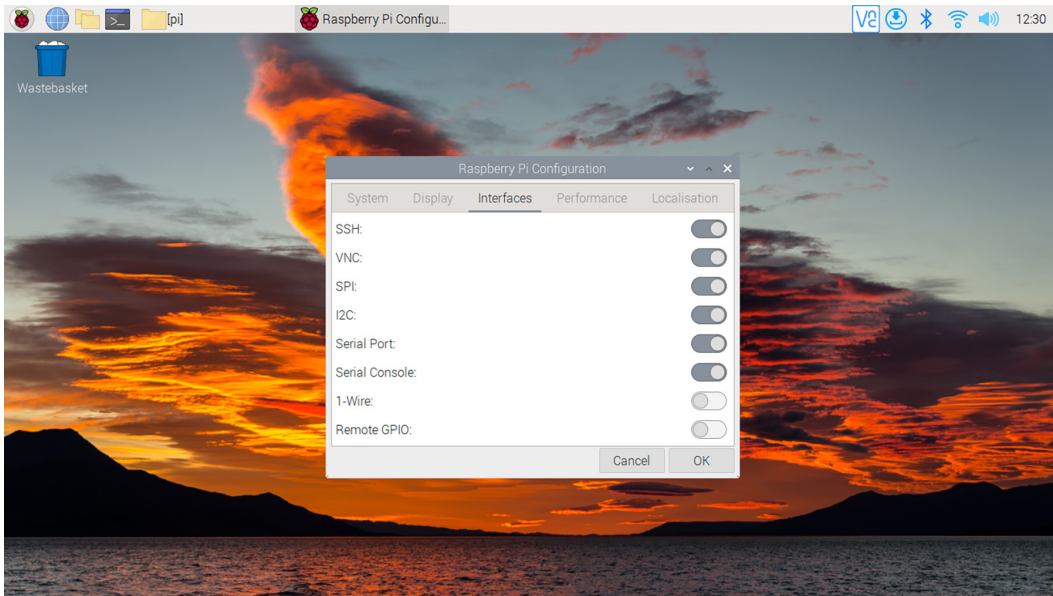


Figure 6.7: Configuring your Raspberry Pi

Remotely accessing the Pi

In the previous section, we have successfully set up the Raspberry Pi. We have connected a number of input/output devices, such as keyboard, mouse, and monitor to the Pi, to act as a standalone system for computing. However, Raspberry Pi is primarily used as a low computing edge device for low-powered, 24×7 processing. The addition of extra peripheral devices consumes more power. In practical applications, Raspberry Pi devices are often not connected to any input/output peripheral devices, and they are controlled remotely. There are various ways of remotely accessing a Raspberry Pi. In a simple way, Raspberry Pi can be controlled from an external device using the **secured shell (SSH)** protocol.

SSH is a secured client-server protocol. The remote machine you need to connect, and control should have the SSH server installed. On the other hand, your host machine should have an SSH client. When the client machine connects to the server over SSH, it can be controlled as a local machine from the client to execute commands remotely. Details of the SSH protocol are beyond the scope of this book. In short, the SSH server has a dedicated **Transmission Control Protocol (TCP)** port over which it monitors the network waiting for an SSH client to connect. The client first needs to establish a secured connection by issuing SSH commands. It requires an authentication process. Once the connection is established, you can control the remote machine.

Make sure your Raspberry Pi and the host machine are in the same network. In order to connect to a Raspberry Pi, you need to enable SSH from the Raspberry Pi configuration window. Refer to *figure 6.7* for details. The SSH client is preinstalled on machines having Linux and macOS X. However, you need to install the client separately on Windows-based computers. You can download and install PuTTY, which is a free SSH and telnet client for Windows. Execute the following command in a terminal of the host machine in order to initiate a remote connection with the Pi.

```
>>ssh [username]@[server_ip_hostname]
```

You need to know the username and the IP address of the Pi. Once the command is executed, you will be prompted to enter the password for the Pi. Once the verification is done, a secured connection will be established, and you will get control of the Pi on your host machine. Type **exit** on the command line, and the remote connection with the Pi gets disconnected.

Deploying the model on Raspberry Pi to make inference

So far, we have set up our Pi and installed the OS. We have also seen how a Pi can be accessed from a remote machine to execute commands. Now, we will set up our Pi to classify images based on the TensorFlow Lite model we created using MobileNet. Although Raspberry Pi has enough computational power, you still cannot effectively train very large neural networks on it. It is commonly used as a low-powered edge device to make inferences. We train our model on a more powerful machine having accelerated hardware like GPU and TPU and convert it into an optimized TFLite model to make on-device inferences.

As expected, you need to install a few software on your device to run a TFLite model. As mentioned earlier, Raspberry Pi supports Python programming language, and the Python interpreter is already installed with the Raspberry Pi OS. Open a terminal in your Pi and type the following command, as shown in *figure 6.8*, to check the version of Python:

```
>>python --version
```

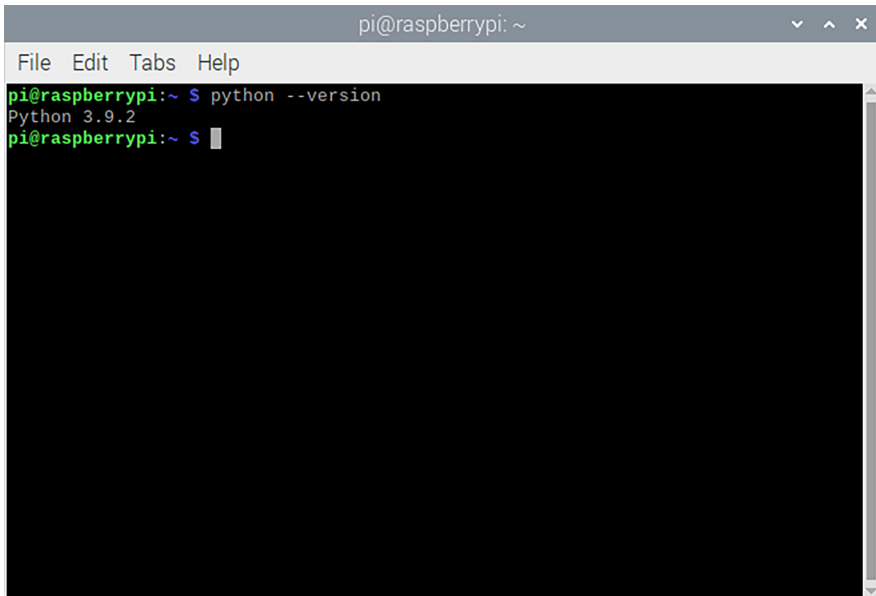
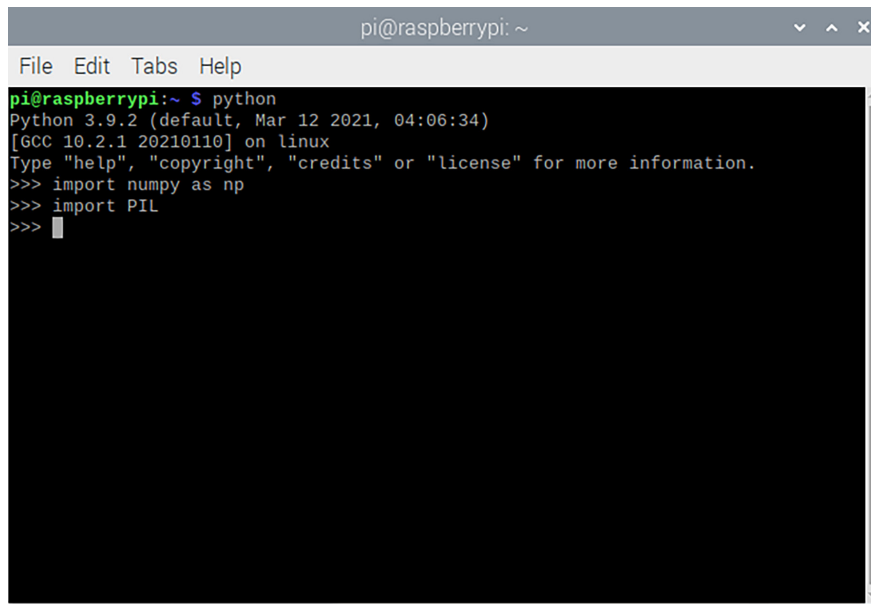
A screenshot of a terminal window titled 'pi@raspberrypi: ~'. The window has a menu bar with 'File', 'Edit', 'Tabs', and 'Help'. The terminal shows the command 'python --version' being entered and executed, resulting in the output 'Python 3.9.2'. The prompt 'pi@raspberrypi:~ \$' is visible before and after the command.

Figure 6.8: Checking the Python version in Raspberry Pi

Figure 6.8 shows the Python version installed on the Pi. The recent Raspberry Pi OS comes with Python 3. Make sure your Pi contains the similar version of Python which you used to train your model. You also need to install a few more libraries. The NumPy library is preinstalled in Pi. We need the **Python Imaging Library (PIL)** for reading image files, which is also preinstalled.

Open a terminal on Pi and type **Python**; you will be inside the Python console, as shown in figure 6.9:



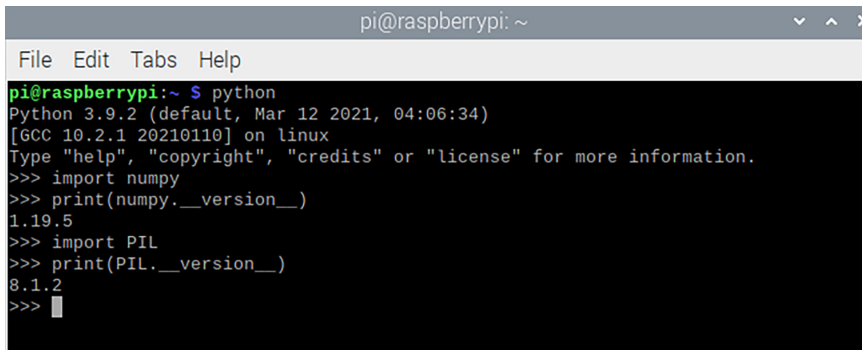
```
pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ python
Python 3.9.2 (default, Mar 12 2021, 04:06:34)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> import PIL
>>> 
```

Figure 6.9: Opening Python console in Raspberry Pi

You can execute all your Python commands and scripts here. Type the following set of commands to check the version of NumPy and PIL library installed in your Pi.

```
>>>import numpy
>>>print(numpy.__version__)
>>>import PIL
>>>print(PIL.__version__)
```

As shown in *figure 6.10*, the version of NumPy and PIL installed in the Pi will be printed in the console.



```

pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ python
Python 3.9.2 (default, Mar 12 2021, 04:06:34)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> print(numpy.__version__)
1.19.5
>>> import PIL
>>> print(PIL.__version__)
8.1.2
>>>

```

Figure 6.10: Checking the version of NumPy and PIL library

Finally, we need to install the TensorFlow library. Remember, our only purpose here is to make on-device inference. We are not going to train a model on the Pi. Hence, we are not going to install the full TensorFlow. Instead, we will only install **tfLite-runtime**, the official TensorFlow Lite library for running TensorFlow models on mobile devices and embedded platforms. It enables on-device inference on smartphones powered by Android, iOS, and also Linux-based SBCs like Raspberry Pi.

The easiest way of installing **tfLite-runtime** on the Pi is by using the **pip** command. Open a terminal and type the following command:

```
>>pip install tfLite-runtime
```

The necessary libraries will be installed. Now, open the Python console and type the following command to access the **tf.Lite.Interpreter** class.

```
>>>from tfLite_runtime.interpreter import Interpreter
```

If you have successfully installed **tfLite-runtime**, the command will be executed without showing an error. As we know, an instance of **Interpreter** is required to make inferences using the TFLite model.

Now, we will write a Python script to classify an offline image from the CIFAR-10 database on Raspberry Pi using the TensorFlow Lite model, **model_qat.tflite** that we created earlier in the chapter. Before we do so, let us first create some sample image files from the test set of the CIFAR-10 database and save them in JPG format. The images will be our test files to evaluate the model.

Go back to the Colab project and execute the following code. It will load the CIFAR-10 database once again; select 10 random samples from the test set and save them in JPG format in the workspace.

```
>>from numpy import random

    from PIL import Image

    (trainX, trainY), (testX, testY) = cifar10.load_data()

    for i in range(10):
        num = random.randint(0, len(testX))
        im = Image.fromarray(testX[num])
        im.save('sample' +str(i+1)+'.jpg')
```

You need to download these images from Colab Workspace to your host computer. Now, let us write the Python script to make an inference. The script is very similar to the earlier inference scripts we created in *Chapter 5, Model Optimization Using TensorFlow*. Let us import the necessary libraries first.

```
>>import sys

    import numpy as np

    from PIL import Image

    from tfLite_runtime.interpreter import Interpreter
```

The script we are going to write will be executed on the command line of Pi. It will take a single input argument, the full path of the image that we want to classify. To make an inference, we first need to create an instance of the **Interpreter** class using the TFLite model and allocate the tensor. Refer to the following code:

```
>>tfLite_model_file = 'model_qat.tflite'

    interpreter = Interpreter(model_path=tfLite_model_file)

    interpreter.allocate_tensors()

    input_index = interpreter.get_input_details()[0]['index']
    output_index = interpreter.get_output_details()[0]['index']
```

Now, we will read the image file passed through the command line and do some pre-processing. We will resize the image according to the images in the CIFAR-10 database, which is, 32×32 . This step is required so that we can evaluate our model on images taken from other sources. Next, each pixel value will be divided by 255 for normalization. Refer to the following code:

```
>>Im = Image.open(sys.argv[1])
    Im_resized = Im.resize((32, 32))
    Im = np.asarray(Im_resized)

    Im = Im/255
```

Next, the image will be reshaped according to the input of the model, that is, a tensor of shape (1, 32, 32, 3). We will allocate tensor for input and output. Finally, we will call the method `Interpreter.invoke()` to make the inference.

```
>>input_data = np.array(Im, dtype=np.float32)

    input_data = input_data.reshape(1, input_data.shape[0], input_data.
    shape[1], 3)

    interpreter.set_tensor(input_index, input_data)
    interpreter.invoke()

    prediction = interpreter.get_tensor(output_index)
    prediction = np.argmax(prediction)
```

The output of the preceding code segment will be the numeric class label which needs to be mapped to the string value of the actual class. We will define a list containing the class names ordering as per the assigned numerical values. Then we will call the particular list element based on the prediction label. Refer to the following code:

```
>>labels = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog',
'Frog', 'Horse', 'Ship', 'Horse']

    # get the predicted label as a string
    print(labels[prediction])
```

So, the complete inference script is as follows:

```
import sys

import numpy as np

from PIL import Image

from tfLiteRuntime.interpreter import Interpreter

labels = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog',
          'Horse', 'Ship', 'Horse']

tfLite_model_file = 'model_qat.tflite'

interpreter = Interpreter(model_path=tfLite_model_file)
interpreter.allocate_tensors()

input_index = interpreter.get_input_details()[0]['index']
output_index = interpreter.get_output_details()[0]['index']

Im = Image.open(sys.argv[1])
Im_resized = Im.resize((32, 32))
Im = np.asarray(Im_resized)

Im = Im/255

input_data = np.array(Im, dtype=np.float32)

input_data = input_data.reshape(1, input_data.shape[0],
                                input_data.shape[1], 3)

interpreter.set_tensor(input_index, input_data)
interpreter.invoke()

prediction = interpreter.get_tensor(output_index)
```

```
prediction = np.argmax(prediction)
print(labels[prediction])
```

Now, open a text editor on your host computer and copy and paste the preceding script. Make sure the code indentation is properly maintained. Now, save the file and name it as `image_classifier.py`. You need to transfer the following files to the Pi.

- The script, `image_classifier.py`
- The model file, `model_qat.tflite`
- The JPG files you created for testing your script (sample1.jpg, sample2.jpg, and so on)

You can transfer the preceding files to the Pi using a USB mass storage device. However, if the Pi is not connected to any input/output peripheral devices such as a keyboard, mouse, or monitor, we can rely on SSH. Using SSH commands, we can remotely access the Pi from a host computer, transfer all the necessary files, and execute the Python script for image classification.

Secure Copy Protocol (SCP) is a file transfer protocol based on SSH that provides secure file transfer between two machines connected over the network. You can either transfer a file from your host machine to a remote server or can copy a file from a remote server to your local machine. SCP runs on port 22. While transferring data, both the files and the password are encrypted. SCP commands are included in computers running Linux and macOS X. However, you need to install it on Windows. If you have installed PuTTY, it includes PSCP, an SCP for Windows. On the Pi, you just need to ensure that SSH is enabled.

Execute the following command on the terminal of your host machine in order to send a file from your host computer to the Pi:

```
>>scp [source_file] [username]@[server_ip_hostname]:[destination_location]
```

Now, let us make an inference on the Pi remotely. Perform the following steps one after another:

1. Power on the Pi and make sure it is connected to the network. Make sure your Pi and host computer are on the same network.
2. On your host machine, open a terminal and access the Pi remotely using the IP address. Suppose, in our case, the username is `pi` and the IP address is `192.168.29.38`. Then execute the following command on the host machine:

```
>>ssh pi@192.168.29.38
```

3. You will be prompted for the password for your Pi. On entering the correct password, you will get access to the Pi from the host machine. In the Pi, we will create a directory, **image_classification**, under Desktop. Execute the following command on the host machine:

```
>>mkdir Desktop/image_classification
```

4. We will store all our files in this directory.
5. Now, we need to transfer the Python script, the TFLite model file, and the test image files to the Pi. Open another terminal on your host machine and type the following command to transfer the python script, **image_classifier.py**:

```
>>scp image_classifier.py pi@192.168.29.38:~/Desktop/image_classifier
```

6. You will be again asked for the password. Upon verification, the Python script will be transferred to the Pi and saved in the directory. Follow the above procedure to transfer the model file and the test image files.
7. Once all the necessary files are successfully transferred, we can make an inference. Go back to the previous terminal on the host machine which is accessing the Pi using SSH. First, go to the directory **image_classifier** by issuing the following command:

```
>>cd Desktop/image_classifier.
```

8. Now, execute the following command to run the Python script.

```
>>python image_classifier sample1.jpg
```
9. It will take **sample1.jpg** as input, run the inference script and print the predicted class name on the terminal. You can test the script on another image by changing the filename and check the performance. Since, our model has reported around 81% classification accuracy on the test set, you can expect most of the test samples to be correctly predicted.
10. Finally, you can check the performance of the model on images downloaded from other sources as well. You can download or collect sample color images of various objects upon which the model has been trained, such as airplanes, cars, dogs, ships, and so on, and check the model performance. Do not worry about the dimension of the images, as the Python script will readjust them into 32×32 pixels. You will be surprised to see that the model can correctly classify many of those images downloaded from external source, like the one shown in *figure 6.11*. Make sure that the target object is clearly visible in the images used and they do not have a very complex background.



Figure 6.11: Sample test images from an external source to be tested by our classifier

Conclusion

Congratulations! You have successfully deployed your first TinyML application on a Raspberry Pi. The main objective of this chapter was to get the feeling of creating a real-world TinyML application from scratch and deploying it on a commercial edge device. We have created a deep learning model for image classification and optimized it to run on low-powered edge devices. In this chapter, we have learnt about MobileNet, a highly optimized, powerful CNN-based deep neural network architecture for mobile devices and edge devices. MobileNet exploits depthwise separable convolution that performs the convolution operation with fewer mathematical operations. Remember that the target devices such as smartphones, Raspberry Pi, or microcontrollers often do not have sufficient hardware resources to train large deep learning models. We have trained our model on Colab using the power of GPU and CPU and then converted the model into the equivalent TFLite model. The model has been later used to classify offline images on the Pi. In this chapter, we have created a fairly simple application to begin with developing TinyML. In the upcoming chapter, we will implement a more complex and practical project, identifying a person from a live video feed captured using a camera.

Key facts

- TensorFlow models can be deployed on smaller edge devices like Raspberry Pi or microcontrollers to make inferences.
- Neural network models are primarily trained on a powerful computer. The edge devices are meant to make inferences.
- MobileNet is a CNN structure specially designed for low-powered mobile devices.

- MobileNet exploits depthwise separable convolution, an efficient way of performing convolution with reduced mathematical operations.
- Transfer learning focuses on applying the knowledge gained while solving one task to a related task.
- Transfer learning is often useful if we do not have enough data to train a model.
- Raspberry Pi is a Single Board Computer for IoT applications
- Raspberry Pi has a dedicated Linux-based operating system, it can run Python scripts. You can even execute TensorFlow models on it.

Join our book's Discord space

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

<https://discord.bpbonline.com>



CHAPTER 7

Deep Dive into Application Deployment

Introduction

In *Chapter 6, Deploying My First TinyML Application*, we built our first TinyML application for image classification. We used the pretrained MobileNet architecture to define our model. Later in the chapter, the model was deployed on Raspberry Pi, a commercial tiny edge device for image classification. We used the TensorFlow Lite library for the compression and optimization of the model. Although our machine learning model was trained on the publicly available CIFAR-10 database, it could successfully detect similar types of images from other sources as well. The objective of the previous chapter was to learn the different stages involved in creating an end-to-end TinyML application through a practical example. In this chapter, we will create a slightly complex but more practical application, identifying a person in real-time from a facial image extracted from the live video stream recorded by a camera. This application will also be deployed on the Raspberry Pi.

On-device person identification has plenty of applications in our everyday life. We all know the importance of user privacy and security in the modern digital world. Biometric-based user authentication has become immensely popular in modern cyber-physical systems. In biometric authentication, some unique signatures of your physical properties are used to automatically identify yourself in order to give you certain access to some information and facilities which are solely owned by

you, therefore, restricting the access of others. Fingerprint, retinal and facial images are particularly used for biometric verification purposes. These features are unique for every person. The key advantage of biometric-based authentication is they are easy to use but difficult to imitate. You do not need to remember a password to authenticate yourself. Take the example of your smartphone. Although one can unlock the phone with a passcode, many of us still prefer to unlock our device by some type of biometric verification like fingerprint scanning or face recognition. By this, it is ensured that an unauthorized person cannot have access to our personal device. Among various biometric techniques, authentication by face recognition is particularly popular. It is more unobtrusive compared to fingerprint or retina-based authentication. Instead of placing the finger on the sensor for scanning or focusing on a camera that scans the retina, one just passively needs to look at the camera to get identified.

Now let us understand how biometric recognition works. We can take the example of user identification via fingerprint scanning on the smartphone to unlock the device. It is a classic example of applied machine learning, which requires a dedicated on-device training process. Recall, when you set up your phone for the first time, you are asked to set up your fingerprint to use it as an identification system to unlock your device. You are prompted to place your finger at different angles on the dedicated fingerprint sensor of your phone. This is nothing but providing data to the system to train. An algorithm runs in the backend that extracts the relevant features from the fingerprint such as loops, whorls, and arches in order to train a model that is specific to your finger. Of course, the training process is quite different because you need to train a model on a small amount of data. Once your phone is set up, each time you place your finger on the sensor, it checks the similarity score of the features to identify you in order to unlock your device.

In this chapter, we will create a lightweight end-to-end application for on-device face recognition and deploy it on Raspberry Pi. However, the implementation of the project will be slightly different. Unlike previous examples, this project will be entirely implemented on the Pi, including the training. We need to remember a few things in mind before implementing the project. The on-device inference latency needs to be very low so that the recognition happens in real-time. Moreover, our application should be robust enough to deal with various ambient lights and backgrounds. Finally, if you are the target person for recognition, no other person should be recognized as you by the system. We will discuss more about these challenges when we start implementing the project. Once developed, the face-recognition application can be used as a user authenticator to perform a number of interesting projects. You can create your own novel applications and can deploy them in your everyday life. For example, you can create an application for automatically turning on a smart

light in your room as soon as you are detected by the camera installed at the room entrance, which communicates with the Pi. Similarly, using the face recognition application at the entry point, you can design an automated system that detects the authorized persons and automatically grants their access to a premise while stopping others from entering.

Structure

In this chapter, we will discuss the following topics:

- System requirement
- The face recognition pipeline
- Setting up the Raspberry Pi for face recognition
 - The Raspberry Pi camera module
 - Installing the necessary libraries
- Implementation of the project
 - Data collection for training
 - Model training
 - Real-time face recognition

Objectives

In this chapter, we will create an end-to-end application for on-device person recognition from a live video stream on a Raspberry Pi. The application will capture a video stream using a camera connected to the Pi, analyze the image frames to recognize the target person as soon as he/she appears in front of the camera. Similar to the previous project, this project will also be implemented in Python. However, the implementation will be slightly different. In previous projects, we trained our machine learning models on a different machine hosted on the cloud, not on the Pi. To be more specific, our neural network models were trained and optimized using the Colab notebook powered by the enormous infrastructure provided by Google. The resulting model was used in the Pi only to make inferences. However, in this project, we will perform both model training and recognition on the same device. Take the example of the image classification project we implemented in *Chapter 6, Deploying My First TinyML Application*. The training was a one-time job. We collected hundreds and thousands of images from a public dataset to train a neural network. In such applications, we need to retrain rarely only if we want to improve the model performance or decide to add a new type of object for detection. The classifier network was large and complex with thousands of trainable parameters; hence, it

was not possible to train on the Pi. However, the scenario is different in the case of person identification. The training is very much specific to the person you want to detect. Here, we need to train on a much smaller dataset, as getting a large number of training samples for one person is not feasible. Moreover, you may frequently need to add/remove a person in order to update the list of target persons to be recognized by the system. Hence, training/retraining can be more frequent. That is why we prefer to do the training on the Pi itself.

Another key differentiator comes in getting the data. In all previous chapters, we relied on publicly available datasets for training. However, as you can understand, for face recognition, we need to create our own dataset for every single person we want to recognize. In all previous examples, we used large neural networks, particularly Convolutional Neural networks (CNN), to design the machine learning model. In this project, we will use a much simple approach. Although complex, deep neural network architectures exist for face recognition, they can be difficult to train on the Pi.

Face recognition is a complex task that involves lots of complex image processing and computer vision algorithms at various stages for detection and recognition of facial properties. Rather than implementing every single image processing algorithm from scratch, we will use readily available Python libraries to efficiently perform most of the image processing operations, such as face detection from an image, background separation, relevant feature extraction, model creation, and so on, and primarily focus on implementing the end-to-end pipeline. We will also focus on executing the project close to real-time for a better user experience.

System requirement

In this project, we are dealing with real-time image processing on an input video stream. Hence, we must need a camera that is compatible with the Raspberry Pi. We will use the Raspberry Pi camera, an off-the-shelf camera module specially designed for the Pi. Moreover, since we need to collect our own data in order to train on them, we will also require the standard input/output peripheral devices to be connected to the Pi. This will be required for visualization aspects.

The following components are required to implement this project:

- Raspberry Pi 3 Model B+
- Raspberry Pi camera module
- The micro-USB power cable for the Pi
- AC power adopter or a power bank for power supply

- Standard peripheral input/output devices such as a monitor with HDMI cable, keyboard, and mouse

The face recognition pipeline

A high-level architectural diagram for the face recognition pipeline is shown in *figure 7.1*. Look at the different blocks in the pipeline carefully for a better understanding of different steps involved in the project. Like any machine learning application, this project has two distinct parts: training and testing. The training is an offline process to obtain a learning model that is specific to the target persons you want to recognize by your system. Once we get the necessary image files for training, we will extract the relevant features to train the machine learning model. We will use the model to recognize the person in real-time. The testing process has some similarities to training, but it will be a real-time operation. We will extract image frames from the live video stream captured by the camera and compute the same set of features we did during training. The features, together with the model, will determine whether the target person is present in the image frame or not. In case you want to update the list of persons you want to recognize, you need to retrain your model.

Training phase

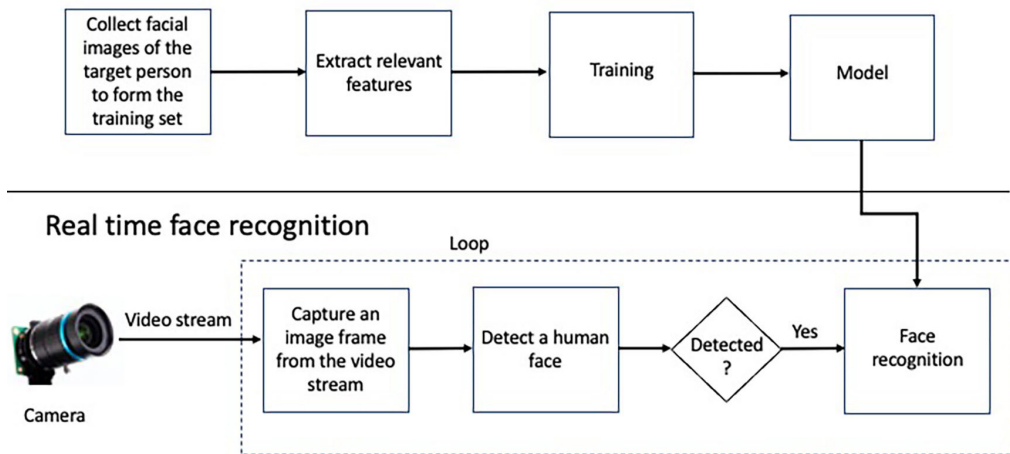


Figure 7.1: The face recognition pipeline

As we all know, getting the right data is the key thing to train any machine learning model. In all previous projects, we trained our models on large public datasets. In this project, we need a good number of frontal facial images of the person we want to recognize. As expected, we cannot get a readily available public dataset for that.

Hence, we will collect a small dataset of our own for training. We will implement a Python script to capture frontal facial images using a Pi camera connected to the Raspberry Pi to collect the relevant training data. Once the data is collected, we will process it for relevant feature extraction in order to create our learning model. We will use standard open-source libraries for most of the image-processing tasks, which will be duly discussed later. Once the model is created, we will use it for our face recognition task.

Now, let us understand the process involved in testing where we actually perform the face recognition task in real-time. The Pi camera is programmed to capture video in real-time. We will extract image frames from the video stream inside an infinite loop and analyze each frame. We will perform two tasks in the face recognition process. Our first job will be to detect whether a human face is present in the extracted frame. If a human face is detected, then only we will go to the next step to determine whether the detected face belongs to the target person based on a similarity matching of the features. This entire processing needs to be very close to real-time in order to get a better user experience.

Setting up the Raspberry Pi for face recognition

In the previous section, we have briefly discussed about the face recognition pipeline. Now, we will start implementing the actual project. A detailed explanation of all the underlying algorithms responsible for face recognition is beyond the scope of this book. As mentioned earlier, rather than implementing every single image processing algorithm involved in the application, we will use some highly optimized open-source libraries to perform most of them and primarily focus on implementing the end-to-end pipeline. We will start implementing the project by installing the camera module, followed by installing the necessary software libraries.

The Raspberry Pi camera module

The Raspberry Pi camera is a commercially available low-cost, high definition digital camera module board that can capture good-quality images when connected to the Pi. We will use the 5 megapixel camera, which is capable of recording good quality video. Although the recorded image quality is inferior to the USB-based Webcam, it

can be reliably used for various applications, including face recognition. *Figure 7.2* features the Raspberry Pi camera module as it is and when it is connected to the Pi:

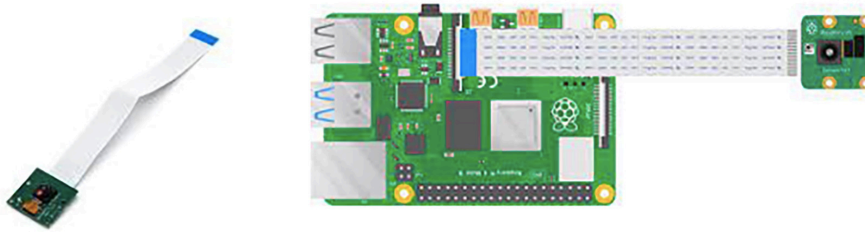


Figure 7.2: The Raspberry Pi camera module (left) and when connected to the Pi (right)

The camera module is shown in *figure 7.2* (left). We strongly recommend the readers procure the Raspberry Pi camera module in order to follow the rest of the chapter detailing the project implementation. As shown, the camera module has two parts, the green board containing the camera lens and the necessary electronic circuitry for image capturing and the white ribbon for connecting the camera module to the Raspberry Pi board. Connecting the camera to the Pi is fairly simple. As mentioned in the previous chapter, the Raspberry Pi comes with a camera port located in between the audio port and the HDMI port, as shown in *figure 7.2* (right).

Do the following steps to connect the Pi camera. First, power off the Pi. Next, pull up the plastic clip covering the camera port of the Raspberry Pi and gently insert the ribbon of the camera module inside. Make sure that the blue end of the ribbon is facing toward the USB port and the LAN port of the Raspberry Pi device, as shown in the preceding figure. Now, connect the monitor, keyboard, and mouse with the Pi. Finally, power on the device.

Once powered on, you first need to configure your Raspberry Pi in order to detect the camera. This is a one-time job. Open a terminal on the Pi and type the following command:

```
>>sudo raspi-config
```

You will be prompted by the following window shown in *figure 7.3*. We need to perform the following steps:

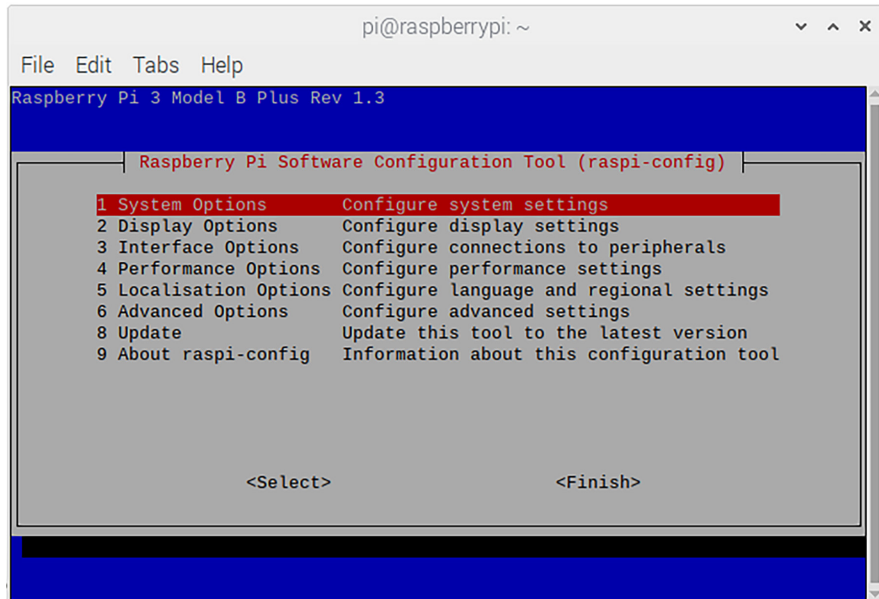


Figure 7.3: GUI on entering `rsapi-config` command on Raspberry Pi terminal

Select Interface Options (the third option in the figure) from the list, and the following window shown in *figure 7.4* will appear:

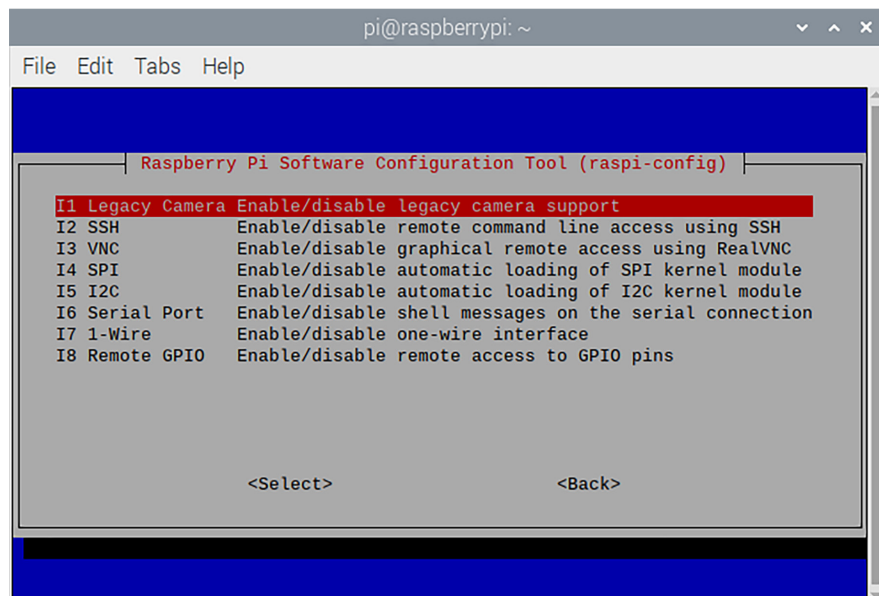


Figure 7.4: Selecting `raspi-config` under Interface Options

Select the first option from the list to enable the camera. The following window, shown in *figure 7.5*, will appear:

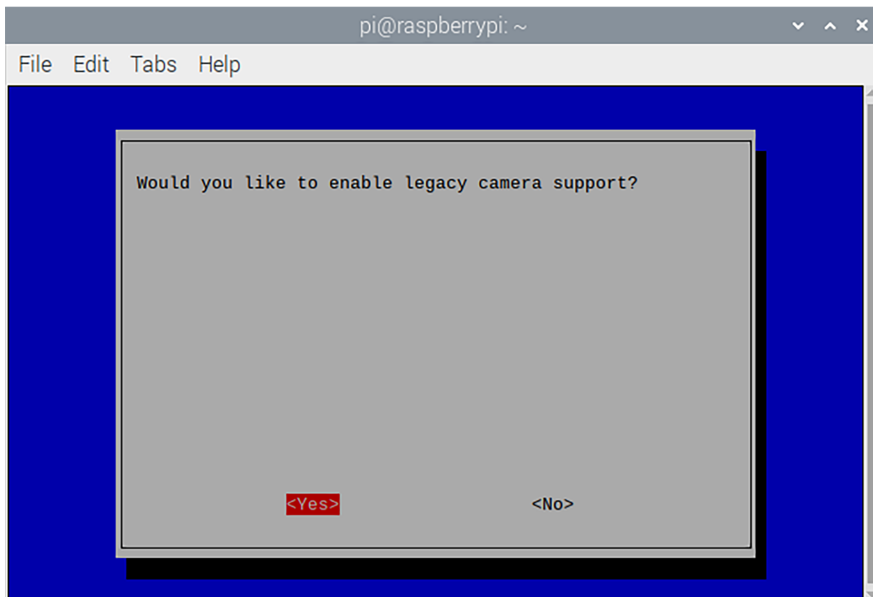


Figure 7.5: Enabling the camera

Select **Yes** to enable the camera. Now, restart the Pi to fully enable the camera. Once restarted, type the following command in a terminal to check the camera status. Refer to *figure 7.6*:

```
>>vcgencmd get_camera
```

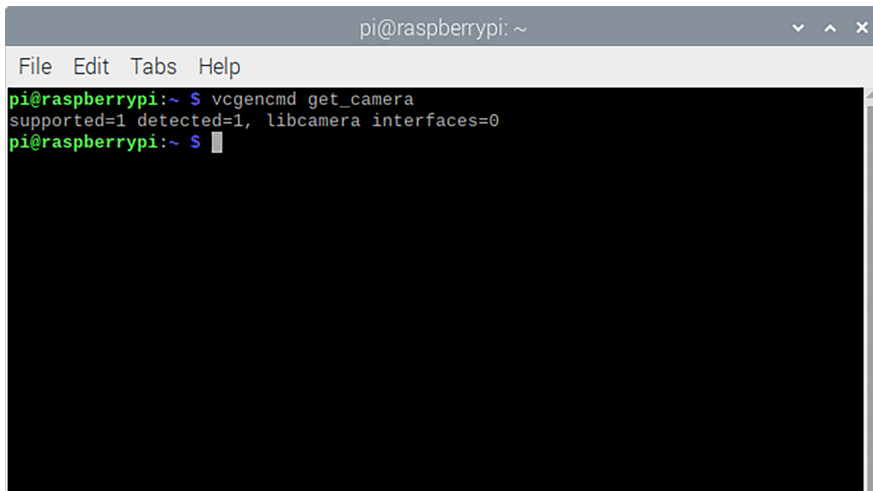


Figure 7.6: Checking camera status on Raspberry Pi

If you find the two parameters **supported = 1** and **detected = 1** as the command output, then your camera is successfully configured. Otherwise, the camera has not been detected. Make sure that the white ribbon in the camera module is properly connected to the Pi. If everything goes well, type the following command to test the camera:

```
>>raspistill -o test.jpg
```

It usually takes few seconds to turn on the camera. Once the camera is on, a display window will open, showing the camera preview. The window will remain open for few seconds and close automatically. The last frame of the video will be saved in the home directory of your Pi with a file name **test.jpg**.

Installing the necessary libraries

Now, we have successfully set up the Raspberry Pi camera. Next, we need to install the necessary software packages to implement the project. We will first install the **picamera** package. This is a Python library for the Raspberry Pi camera module containing optimized APIs to efficiently capture images and videos. Open a terminal on your Pi and enter the following command to install the package:

```
>> sudo apt-get install python3-picamera
```

Wait till the package is completely installed. Next, we will install **OpenCV**. It is a large and popular open-source library for image processing and computer vision. It contains more than 2,500 highly optimized algorithms related to image and video processing operations. Apart from that, there are standard libraries in OpenCV to detect objects such as human faces from images and videos. OpenCV is highly scalable and has an interface for all major programming languages such as C++, JAVA, and Python and supports almost all major operating systems such as Windows, Linux, macOS X, and also the Raspberry Pi OS. Enter the following command on the terminal to install OpenCV for Python:

```
>>sudo apt install python3-opencv
```

Now, we will install another package called **imutils**. It contains a series of OpenCV convenience functions for doing some basic image processing operations like image translation, rotation, resizing, and so on. You must install OpenCV first in order to install the *imutils* package. Enter the following command in the terminal:

```
>>pip install imutils
```

Next, we will install a Python package containing readily available optimized libraries for face recognition. The library is built with advanced machine learning

algorithms with a benchmark face detection accuracy of 99.38%. Using the library, you just need few lines of codes with high level of abstraction to easily detect and recognize human faces from an input image. Type the following command to install the package:

```
>>pip install face-recognition
```

Finally, enter the following command that will install the required dependencies:

```
>>sudo apt-get install libatlas-base-dev
```

With these, you have installed all the necessary libraries and dependencies for the project. Restart your Pi to make everything work.

Implementation of the project

We have successfully set up our system in the previous section. Now, it is time to implement the actual project. The entire face recognition task can be broadly divided into the three following parts:

- Data collection for training
- Model training
- Real-time face recognition

We will discuss each part of the project in some detail for implementation.

Data collection for training

As we know, data collection is the most important job in machine learning to train a good working model. In the previous examples, we developed machine learning models for the classification of handwritten digits or various image objects such as dogs, cats, cars, and so on. For those applications, we have the availability of rich open-access datasets such as MNIST or CIFAR-10. For this project, our training dataset requires facial images of the person we wish to recognize by the face recognition system. Hence, we will create a small dataset containing few headshot images of the target persons for training. A headshot can be defined as a portrait photograph of the frontal face. As part of this project, we will implement a Python script for data collection. It will capture a video using the Pi camera. Whenever a certain key is pressed by the user, the video frame at that moment will be captured and stored in the file system as a digital image.

Now, let us understand the script. It will be executed on the command line. You need to enter the name of the person as an input argument for whom you wish to collect

the data. Now, let us write the script. We will start by importing the libraries:

```
>>import os
    import sys
    import cv2
    from picamera import PiCamera
    from picamera.array import PiRGBArray
    from time import time
```

We need one more step. We will create a parent directory **dataset** in the same path where the Python script is located. The name of the persons we want to collect the data for will be obtained by the script from the input argument provided by the user in the command line, and accordingly, a new directory will be created under the parent directory. All headshots for that person will be stored under that directory. Refer to the following code:

```
>>name = sys.argv[1]
    # get parent directory
    parent_path = os.getcwd()
    # create the directory for image storing
    path = os.path.join(parent_path+'/dataset', name)
    isExist = os.path.exists(path)
    print(isExist)
    if isExist==False:
        os.mkdir(path)
```

In this preceding script, the person's name is obtained from the command line and stored in the variable **name**. Next, we check whether a directory having the same person's name already exists or not and create it accordingly.

Now, we will start capturing the video stream using the Pi camera. We will use the readily available functions from the **PiCamera** module. See the following code.

```
>>cam = PiCamera()
    cam.resolution = (320, 240)
```

```
cam.framerate = 10
```

```
rawCapture = PiRGBArray(cam, cam.resolution)
```

First, we create a **PiCamera** object. We need to mention the video resolution and the frame rate to capture the raw stream. The function **PIRGBArray()** defines a three-dimensional RGB NumPy array for camera capture. We provide the video resolution as 320×240 , which is good enough to run the face recognition task in real-time on the Pi. In general, a higher video resolution ensures a better image quality which is good for face recognition, but it comes with a penalty of more computational load. The video frame rate is given 10 frames per second.

Now, we will write the main loop. It will call an infinite while loop to capture image frames from the continuous video stream. When executed, a new window is created displaying the camera preview. In each iteration within the loop, the program will wait for 1 millisecond for a key press. If the user presses the *space* key, it saves the current frame as a JPEG image in the directory corresponding to the name of the person. We make a system call to get the current time in milliseconds and use the same to assign a unique filename to each of the JPEG images stored in the directory. On hitting the *escape* key, the program exits from the loop and closes all windows. The code segment is shown as follows:

```
>>while True:

    for frame in cam.capture_continuous(rawCapture, format='bgr',
        use_video_port=True):

        image = frame.array

        cv2.imshow('Press Space to take a photo', image)

        rawCapture.truncate(0)

    k = cv2.waitKey(1)

    #rawCapture.truncate(0)

    if k%256 == 27: # ESC pressed

        break

    elif k%256 == 32:

        # SPACE pressed
```

```

        img_name = path+'/sample_{}'.format(int(time() * 1000)) +
        '.jpg'

        #img_name = 'image_{}'.format(img_counter) + '.jpg'

        cv2.imwrite(img_name, image)

        print('{} written!'.format(img_name))

        #img_counter += 1

    if k%256 == 27:

        print('Closing... ')

        break

cv2.destroyAllWindows()

```

Now, we have the entire script. Open a text editor in the Pi and copy and paste all the code segments. Be careful to maintain the indentation while creating the script. Save the file as **headshots_data_collection.py**. Create a folder named **Face_Recognition_Project** under the home directory and paste the Python file there. This is your main project directory. Make sure that the parent directory, **dataset** is also located in the main project directory parallel to the Python script.

Now, open a terminal on the Pi and get into the **Face_Recognition_Project** directory using the **cd** command. Then, enter the following command to execute the script.

```
>>python headshots_data_collection.py person_A
```

Replace the string **person_A** with your name if you wish to collect your own data. It will take a few seconds to turn on the camera. Wait for the display window to open, and the video preview will appear. Now, gently hold the camera with the white ribbon and bring it close to your face to capture headshot images. An ideal headshot should be similar to the photographs used in the identity cards. You should directly look at the camera, and your face should cover 70%–80% of the frame. Make sure that your frontal face is clearly visible. The camera should not be too close or far away from the face. Adjust your position based on the camera preview. Make sure the room where you are sitting has sufficient ambient light, and you should have a clearly separable background.

Stay still in the position and hit the *space* key while looking at the camera to capture and save a headshot image. Now, go back to the **dataset** directory. You

will find a directory of your name where the image file is stored. Follow the same procedure to capture 10–15 headshot images. For a better recognition accuracy, you should introduce some variations in the captured image by slightly bending your face towards your right and left during the capture. You can also have different backgrounds while capturing the data. When you have enough data, press the *escape* key to close the script.

Now, you can start training the model that will be specifically for you. Additionally, you can follow the same procedure to take another person's data so that your system can recognize two different persons at the same time.

Model training

In this section, we will learn how to train a model on the captured data for face recognition. The training is a one-time process. We only need to retrain only if you need to add or remove a person or have new training images to improve the model accuracy. We will primarily use the **OpenCV** and the **face_recognition** library to implement the training. Our training code performs the following operations:

- Get the input images from the training directory one-by-one and do some basic pre-processing.
- Locate the faces in each image using the **face_recognition** library.
- Compute a list of 128-dimensional encoding for each unique face in the image. These are used as discriminating features for the person.
- Dump the encoding in a pickle file which is the training model for our case.

Now, let us start implementing the code. Here are the required libraries:

```
>>from imutils import paths
import face_recognition
import pickle
import cv2
import os
```

Next, we will access the directory where the training images are stored. We will define two lists, one for storing the names of different persons for training and the other for storing the encodings (features) computed for each face.

```
>>datapath = list(paths.list_images('dataset'))
```

```
Names = []
```

```
Encodings = []
```

Now, we will start processing the training images in a loop. Refer to the following script:

```
>>for (i, data) in enumerate(datapath):  
    name = data.split(os.path.sep)[-2]  
    image = cv2.imread(data)  
    rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  
  
    boxes = face_recognition.face_locations(rgb, model='hog')  
  
    # compute the facial embedding for the face  
    encodings = face_recognition.face_encodings(rgb, boxes)  
  
    # loop over the encodings and append to the list  
    for encoding in encodings:  
        Encodings.append(encoding)  
        Names.append(name)
```

Let us understand what is happening inside the loop. First, we access the training images for each person. One image file is read at a time inside the loop. We use the `imread()` function available in the OpenCV module to read an image. By default, OpenCV reads the color channels of an image in the order of **Blue, Green, Red (BGR)**. We need to convert it into an RGB image for further processing. Next, we call the `face_locations()` method available in the `face_recognition` library that identifies and locates all human faces in the image. The function uses the “hog” model for face recognition. It is to mention that the library also supports more complex face recognition algorithms via CNN. Although slightly more inaccurate than the CNN-based approach, the hog model is extremely lightweight, and hence, is particularly suitable for our application.

Now, let us briefly understand how the algorithm works. The hog model uses a **histogram of oriented gradient (hog)** and a linear **support vector machine (SVM)**

classifier for face detection. The hog is popularly used in object detection in computer vision. It works in the following way:

In hog, an input image is first divided into small connected cells. Then it computes the histogram for each cell. It then normalizes the result using a block-wise pattern, and returns a descriptor for each cell. A histogram can be assumed to be similar to a bar graph that groups the image pixel values into different ranges. When plotted, the histogram comprises connected bars of different heights. The horizontal axis of each bar represents different ranges of pixel values, and the height of the bar represents how many pixels are falling in that range. With histogram, we can estimate the density of the pixel values in the image. The feature vector is computed by combining the histograms for all the cells, which is then used for deciding the presence or absence of faces in the image using an already trained SVM classifier. The function **face_locations()** returns the (x,y) coordinates of all the bounding boxes corresponding to the faces detected in the input image.

Next, we call the **face_encoding()** function to compute the facial embedding which is the distinguishing marker for the face. Finally, we loop over the encodings to append them to the list.

When encodings are computed on all training images for a person, we dump the encodings in a pickle file that works as the training model. Pickle is a Python module used for serializing and deserializing Python object structures. See the following code:

```
>>print('Dumping in pickle')
    data = {'encodings': Encodings, 'names': Names}
    f = open("encodings.pickle", 'wb')
    f.write(pickle.dumps(data))
    f.close()
```

Now, copy all the code segments discussed in the section and paste them into a text editor in Pi and name it as **train_model.py**. Save the file inside the main project directory **Face_recognition_Project**.

Now, execute the script on a terminal using the following command:

```
>>python train_model.py
```

The program will parse all the training images one after one, process each of them to obtain the face encodings, and finally create the pickle file, **encodings.pickle**, which will be stored in the same directory.

Real-time face recognition

Now, we have the training model. We are ready to implement the testing part for on-device face recognition. Refer to the face recognition pipeline in *figure 7.1*. The recognition task involves the following steps:

- A live video stream is captured using the Pi camera. It will be displayed in a new video window.
- Image frames are extracted from the video stream in an infinite loop. Each image will be individually analyzed.
- For each frame, we will first check whether the frame has a human face or not. If a face is detected, then only it will determine whether the detected face matches the target person(s) upon which the model had been created.
- If the target person is detected, the name will be shown, and it will print unknown if we do not get a match.

Now, we will implement the code. Let us import the necessary libraries.

```
>>import face_recognition
import imutils
import pickle
import time
import cv2
from picamera import PiCamera
from picamera.array import PiRGBArray
```

Now, let us perform the initialization tasks. We will first load the training model which is stored in the project directory as a pickle file to obtain the encoding information of the trained faces along with the name of the person(s). We will create a variable **currentname**, which will store the name of the last person detected by the algorithm. If the detected face does not match with any of the faces in the training set, the variable will be set as *unknown*. The default value of the variable **currentname** is set as *unknown*. See the following code:

```
>>data = pickle.loads(open('encodings.pickle', 'rb').read())
currentname = 'unknown'
```

Now, we will start capturing a video stream from the camera. This part of the code is similar to what we did during data collection. We will create a **PiCamera** object and

specify the video resolution and the frame rate to capture in an RGB array. Here is the code:

```
>>cam = PiCamera()

    cam.resolution = (320, 240)

    cam.framerate = 10

    rawCapture = PiRGBArray(cam, cam.resolution)
```

Now, we will implement the main loop. It is an infinite loop. Inside the loop, one image frame will be grabbed from the video stream at a time for processing. The program will wait for 1 millisecond for a user key press. The live video will be displayed in a window. It will exit the loop if the *escape* key is pressed.

On every frame, the following set of operations are performed.

1. First, we call the `face_location()` method from the `face_recognition` module to detect and locate the faces inside the images.
2. If a face is detected, it will provide the (x, y) coordinates of the boundary box of the face. The boundary box of the detected face will be marked by a square in the image window
3. Next, we will compute the facial encodings of the detected face using the `face_encodings()` method.
4. Finally, we will check whether the extracted encodings match with our known encodings stored in the pickle file.
5. In case of a match, the name of the person will be printed on top of the boundary box. Otherwise, we will print *unknown* on top of the boundary box.

Here goes the full code snippet corresponding to the main loop:

```
>>while True:

    for frame in cam.capture_continuous(rawCapture, format='bgr', use_
        video_port =True):

        #get a frame from the video

            image = frame.array

            rawCapture.truncate(0)

            k = cv2.waitKey(1) & 0xFF

            #rawCapture.truncate(0)
```

```
    if k%256 == 27: # ESC pressed
        break
# detect faces in the frame
    boxes = face_recognition.face_locations(image)
# compute the facial embeddings for each face bounding box
    encodings = face_recognition.face_encodings(image, boxes)
    names = []

    for encoding in encodings:
        # try to match each face in the input image to our known
        # encodings
        matches = face_recognition.compare_faces(data
            ['encodings'],
            encoding)
        name = 'unknown' #if face is not recognized

        # check to see if we have found a match in our training file
        if True in matches:
            matchedIdxs = [i for (i, b) in enumerate(matches) if b]
            counts = {}

            # loop over the matched indexes for each recognized
            # face
            for i in matchedIdxs:
                name = data['names'][i]
                counts[name] = counts.get(name, 0) + 1

            # determine the recognized face with the largest
            # number of votes
```

```
name = max(counts, key=counts.get)

#If someone in your dataset is identified, print their
name on

if currentname != name:
    currentname = name
    print(currentname)

# update the list of names
names.append(name)

# loop over the recognized faces
for ((top, right, bottom, left), name) in zip(boxes, names):
    # draw the predicted face name on the image
    cv2.rectangle(image, (left, top), (right, bottom),
        (0, 255, 225), 2)
    y = top - 15 if top - 15 > 15 else top + 15
    cv2.putText(image, name, (left, y), cv2.FONT_HERSHEY_
SIMPLEX,
        .8, (0, 255, 255), 2)

cv2.imshow('window', image)

if k%256 ==27 : # ESC pressed
    print("Closing...")
    break

cv2.destroyAllWindows()
```

Copy all the code segments discussed in this section and paste them into a text editor. Name the file `face_recognition_camera.py` and save it in the main project directory. That means your project directory will contain the following:

1. All three Python files—`headshots_data_collection.py`, `train_model.py`, and `face_recognition_camera.py`.
2. The `dataset` folder contains the training images.
3. The pickle file, `encodings.pickle`, correspond to the training model.

Open a terminal, and type the following command to execute the script for face recognition:

```
>>python face_recognition_camera.py
```

The program will take for around 5–10 seconds to perform the initializations. A new window will appear to show the camera preview. Now, bring the camera close to your face using the ribbon so that it can capture your headshot images. It will mark your face with a box and print your name. Similarly, it will be able to detect any other persons as well if your application is trained on them. For all other persons, the detected faces will be marked as unknown. In order to get the optimum performance, your face position should be similar to the training images. Since there are plenty of image processing operations involved in the recognition process, the on-device real-time performance might get slightly compromised. You can expect a frame rate of 2–3 frames per second in the recognition process on a Raspberry Pi 3 Model B+. It will run much faster on a Raspberry Pi 4 device. One more thing to remember is the video resolution. In our application, the video resolution is 320×240 pixels. Although a lesser video resolution ensures a much faster performance, the recognition accuracy may drop due to inferior image quality. You are encouraged to analyze the trade-off between model accuracy and real-time performance by varying the video resolution.

Conclusion

In this chapter, we have implemented a practical TinyML application for real-time on-device person recognition from facial images on the Raspberry Pi. We have primarily relied on publicly available libraries to implement the project. The lightweight hog model, readily available in the `face_recognition` library, is used for face detection. Note that the modern deep learning-based face recognition approaches are often more accurate but are computationally expensive; hence, they are difficult to implement on a Raspberry Pi for real-time recognition.

Person recognition has several practical use cases in our everyday life in terms of user authorization and authentication. There are plenty of such applications that you can deploy on your own. For example, you can automatically turn on and turn off the smart light of the room based on person authentication. You can also deploy the face recognition application at the front door of your house so that it can send messages prompting their names as soon as the known persons appear in front of the camera. You are encouraged to create your own application on top of the face recognition project and deploy for fun.

So far, we have used Raspberry Pi as the target device to deploy our TinyML applications. It is basically a miniaturized computer with limited hardware resources, but it still has sufficient computational power to run sizable deep learning models. Remember that real TinyML applications are primarily meant for deploying on much smaller devices, such as microcontrollers, which are more constrained in terms of hardware and software. In the upcoming chapter, we will explore how a machine learning model can be further optimized to deploy on such devices.

Key facts

- Person recognition has several applications in modern digital systems for automatic on-device user authentication.
- The Raspberry Pi camera is a dedicated hardware module for capturing images and video on the Raspberry Pi.
- The face-recognition module is a freely available Python package for automatic face detection.
- The face recognition project we have implemented in this chapter has three parts: data collection, model training, and real-time person recognition.
- The application requires on-device training for a better user experience.
- In this project, we have used the histogram of oriented gradients (hog) to extract features, which is a simple yet efficient way of face recognition.
- The hog-based models are only reliable in detecting the frontal face. Hence, while executing the project, make sure you directly look at the camera while capturing the training data and also for recognition. Also, make sure the room has sufficient ambient light.

Join our book's Discord space

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

<https://discord.bpbonline.com>



CHAPTER 8

TensorFlow Lite for Microcontrollers

Introduction

So far, we have successfully implemented two TinyML projects. In *Chapter 6, Deploying My First TinyML application*, we developed an optimized Convolutional Neural Network (CNN) model for offline image classification. In the upcoming chapter, we implemented another application for real-time on-device face recognition from a live video feed recorded by a camera. Both our applications were deployed on Raspberry Pi, a commercially available single board computer. Despite its smaller form factor and lower power consumption than a standard desktop computer, Raspberry Pi has a powerful processing unit and enough computation memory space to store sizeable deep learning models to run complex applications. As a result, we hardly faced any real challenge in porting our application to make on-device inferences. However, it is worth noting that TinyML applications are primarily intended to be implemented on even smaller devices, such as microcontrollers, which have limited computing capacity compared to a Raspberry Pi and only a few kilobytes of computing memory.

In *Chapter 1, Introduction to TinyML and its Applications*, we briefly talked about microcontrollers. Microcontrollers are tiny integrated circuits specifically designed to execute a certain task in a repetitive manner. Typically, a microcontroller is composed of a processor, memory, and input/output peripherals, all assembled

into one chip. Microcontrollers are commonly used in large industrial machines, vehicles, electronic appliances, and medical equipment to perform some kind of automation. Usually, a microcontroller reads the data from a sensor device through some interfaces and processes it internally either to produce an output or to trigger another microcontroller. Despite their limited capabilities, microcontrollers have the advantage of consuming very low power. Consequently, they can remain active for extended periods of time even if they are continuously performing their designated tasks. For example, the electronic thermometer found in many households contains a simple microcontroller that reads the temperature sensor, converts it into the required scale, and displays the value. Similarly, in large industrial machines in factories, hundreds or thousands of microcontrollers are installed to continuously measure the machine vibration, load, and so on 24×7 in order to generate alarms when necessary. In general, microcontrollers are primarily fitted into another machine to do some measurements of that host machine. Hence, they need to be small in size so that they can be easily fitted. Second, they need to draw extremely low power from the source so that they remain active for several weeks without charging or replacing the battery. As a result, the computational capability of a microcontroller is extremely low.

Although a microcontroller may appear similar to a single board computer like Raspberry Pi, they have several significant differences. Both have a processor, memory, and input/output peripherals. But this is the only similarity between them. Let us now understand the difference. Raspberry Pi is identical to a complete computer system with a dedicated operating system that enables it to run multiple programs concurrently. For instance, one can surf the Web, listen to music and even write a Python program. On the other hand, microcontrollers are dedicated to one application, and they typically do not have an operating system. They also have fewer resources. For example, a typical microcontroller has a few hundred kilobytes of RAM, whereas a Raspberry Pi has several gigabytes of RAM. Microcontrollers also have limited processing capabilities. Modern deep neural networks can be large in size. As demonstrated by the TensorFlow Lite model used for image classification in Chapter 6, *Deploying My First TinyML Application*, which had a size of 5.6 megabytes. Although it was a fairly simple deep learning model, it would not fit in a microcontroller without further optimization. A second key difference is that you cannot directly write codes to a microcontroller. Microcontrollers do not have an operating system or a compiler. In order to program a microcontroller, we have to install an environment on our computer which communicates with the microcontroller. We write and compile our program on the host computer and then upload it into the target microcontroller for execution.

In previous chapters, we implemented all our machine learning projects in Python which is a high-level programming language. Fortunately, Raspberry Pi comes with a Python interpreter. However, most microcontrollers can only be programmed in a dedicated low-level language similar to C/C++. You might wonder how we can convert a large TensorFlow model written in Python into an equivalent application to efficiently run on microcontrollers. TensorFlow provides **TensorFlow Lite (TFLite) for Microcontrollers**, a set of libraries specially designed to run machine learning models on tiny microcontroller devices. The core runtime is highly compressed and just fits in 16 kilobytes on an ARM Cortex M3 processor, which is a popular choice in many microcontrollers. Furthermore, it does not require any operating system support, any specific C/C++ libraries, or dynamic memory allocation.

TensorFlow Lite for Microcontrollers is written in C++ and requires the 32-bit platform to operate. It supports a number of microcontroller-based development boards, including Arduino Nano 33 BLE Sense, SparkFun Edge, Adafruit EdgeBadge, Espressif ESP32-DevKitC, and many others. In this book, we will use the Arduino Nano 33 board to deploy our TinyML applications. In this chapter, the focus is on creating a simple machine learning application to modulate the brightness of a Light Emitting Diode (LED) according to a sinusoidal function and deploy it on Arduino Nano. In the upcoming chapter, we will create a more complex application of keyword detection via speech processing.

Structure

In this chapter, we will discuss the following topics:

- Arduino Nano 33 BLE Sense
 - Setting up the Arduino Nano
- First TinyML project on the microcontroller—modulating the potentiometer
 - Required components
 - Connecting the circuit
 - Read potentiometer to control the brightness of the LED
 - Creating a TensorFlow model to modulate the potentiometer reading
 - Inference on Arduino Nano using TensorFlow Lite for Microcontrollers

Objectives

In this chapter, we will learn to develop our first neural network application for a tiny microcontroller device. We will particularly use the open-source library TensorFlow Lite for Microcontrollers to implement our project. The main objective of this chapter is to be familiar with the procedure of implementing a neural network from scratch in TensorFlow to eventually deploy it on a microcontroller to make on-device inferences. Our application will be deployed on Arduino Nano 33 BLE Sense, which is a recommended microcontroller board for TinyML applications. The application we are going to implement is fairly simple. The microcontroller will read the linearly varying electrical voltage provided by an external potentiometer as its input, convert the voltage values as per a halfwave sinusoid function using a simple neural network, and controls the brightness of an LED accordingly. Remember, due to its resource constraints, you cannot even train a small neural network on a microcontroller. We will train the neural network in Colab using TensorFlow. The model will be converted into a TFLite model. This part of the project will be implemented in Python. We will then convert the model into an equivalent C/C++ library for microcontrollers. Next, we will write an inference application for Arduino using a programming language that is very similar to C/C++. In this chapter, we will learn in detail how to set up the Arduino to implement TinyML applications. As a prerequisite, this chapter assumes that you have some fundamental knowledge of programming in C/C++ and also have some basic idea of electronic circuit designing.

Arduino Nano 33 BLE Sense

The Arduino Nano 33 BLE Sense is a tiny microcontroller-based development board by Arduino. It is a 3.3V AI-enabled board of the Arduino family in the smallest available form factor. The processor of the microcontroller is powerful enough to run TensorFlow Lite and is highly recommended for deploying TinyML applications. It also comes with **Bluetooth Low Energy (BLE)** module that can be used in IoT applications. The board has an nRF52840 processor that comes with 64 megahertz clock speed, 256 kilobytes of Static RAM (SRAM), and 1 megabyte of flash memory. The board has 14 digital I/O pins, out of which there are eight analog input pins to connect to external electronic components and sensors for data acquisition. It has a 3.3V operating voltage, and it draws a 10 mA current per I/O pin, which is an extremely low power, even for an Arduino. Furthermore, the microcontroller board comes with a number of embedded sensors such as a 9-axis **Inertial Measurement Unite (IMU)** sensor, humidity, and temperature sensor, barometric sensor, proximity

sensor, and microphone, so that one can readily create a number of practical applications without any additional circuitry. It supports standard communication protocols such as UART, I2C, and SPI, among others, to interact with external circuits and sensors for data transfer. Arduino Nano has a micro-USB port to connect to a laptop or a desktop for power supply and file / data transfer via serial communication. Be very careful while working with the Nano. You should never apply more than 3.3V to the pins. Connecting a higher voltage signal may damage the board. *Figure 8.1* features the Arduino Nano 33 BLE Sense Board:

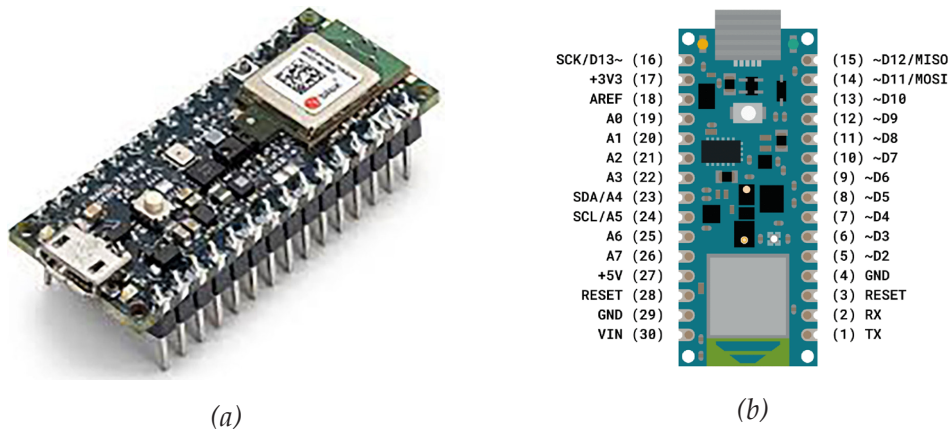


Figure 8.1: (a) The Arduino Nano 33 BLE Sense Board with header and (b) its pin-out diagram

Figure 8.1(a) shows the image of the Arduino Nano 33 BLE Sense microcontroller board with its header pins soldered. The pin-out diagram is shown in *figure 8.2(b)*. The board is available for procurement from the official website of Arduino¹ and also from standard e-commerce platforms selling embedded hardware. The official Arduino website provides more details about the Arduino Nano board along with its technical specifications. Apart from the Arduino board, you will also require a micro-USB to USB cable to connect your Arduino to a host computer. Make sure the cable is enabled for both power supply and data transfer. In the official Arduino store, you can procure the Arduino Nano board as part of a complete package called the Arduino Tiny Machine Learning Kit², which is a ready-to-use development kit containing all the necessary components to design and deploy simple TinyML applications. The kit comes with the Arduino Nano board, a micro-USB cable, a shield for mounting the Arduino Nano, and an Arduino camera to implement real-world image and video-related applications.

1 <https://docs.arduino.cc/hardware/nano-33-ble-sense>

2 <https://store-usa.arduino.cc/products/arduino-tiny-machine-learning-kit>

Setting up the Arduino Nano

Now, let us set up the Arduino Nano for programming. Arduino microcontrollers support a high-level programming language that is syntactically very similar to C/C++. Remember, we cannot directly write programs on most of the microcontrollers, including Arduino. We write all the codes on a host computer, compile them, and then upload them to the microcontroller for execution. Arduino provides an **Integrated Development Environment (IDE)** for writing programs, which needs to be installed on the host computer beforehand. In this section, we will learn to setup the Arduino Nano device in detail. First, connect the micro-USB port of the Arduino Nano to the cable and the other end of the cable to the USB port of the host computer. Once connected, a green LED next to the micro-USB port will be ON, indicating your Arduino Nano is successfully connected and powered.

In order to start programming, we need to install the Arduino IDE on our computer. The IDE supports all major operating systems, such as Windows, Linux, and macOS X. Visit the official website to download and install the IDE³. Although Arduino IDE 2.0 is the recent major release of the IDE, which is faster and more feature-rich, we will use the classical Arduino IDE 1.8.x to implement all the projects. You can expect to seamlessly run all the projects covered in this book, even if you use version 2.0 of the IDE. Follow the detailed instruction provided in the official website to download and install the Arduino IDE depending on the operating system of your host computer. In case you face any difficulty, the Arduino community has a forum⁴ to discuss issues regarding the installation of the IDE and other troubleshooting, which you are encouraged to visit.

Once the installation is completed, open the IDE on your computer. You will see a window similar to what is shown in *figure 8.2*.

3 <https://www.arduino.cc/en/software>

4 <https://forum.arduino.cc>



Figure 8.2: Arduino IDE homepage

The layout of Arduino IDE contains the following components:

- The white text editor in the middle is where you write your code.
- The black console at the bottom is where the error message and other information are shown during compilation and code uploading.
- A toolbar with few buttons is located on top of the text editor to perform various operations, including saving your work, compiling, and uploading programs.

Now, we need to install some libraries specific to the Arduino Nano 33 board. To do so, follow the given steps:

1. Disconnect the Arduino board. On the IDE, go to **Tools** → **Board** → **Board Manager**. A new window will open.
2. Type **Mbed OS** in the search space of the window.
3. Select **Arduino Mbed OS Nano Boards** and click on **Install**.

Mbed OS is an open-source low-powered operating system for Cortex-M boards used in Arduino Nano. It provides an abstraction layer for the microcontrollers so that application developers can write C/C++ applications on any Mbed-enabled board. Refer to *figure 8.3*. It will download and install the core for the Arduino Nano 33 BLE Sense. Make sure you install the latest version of the software.

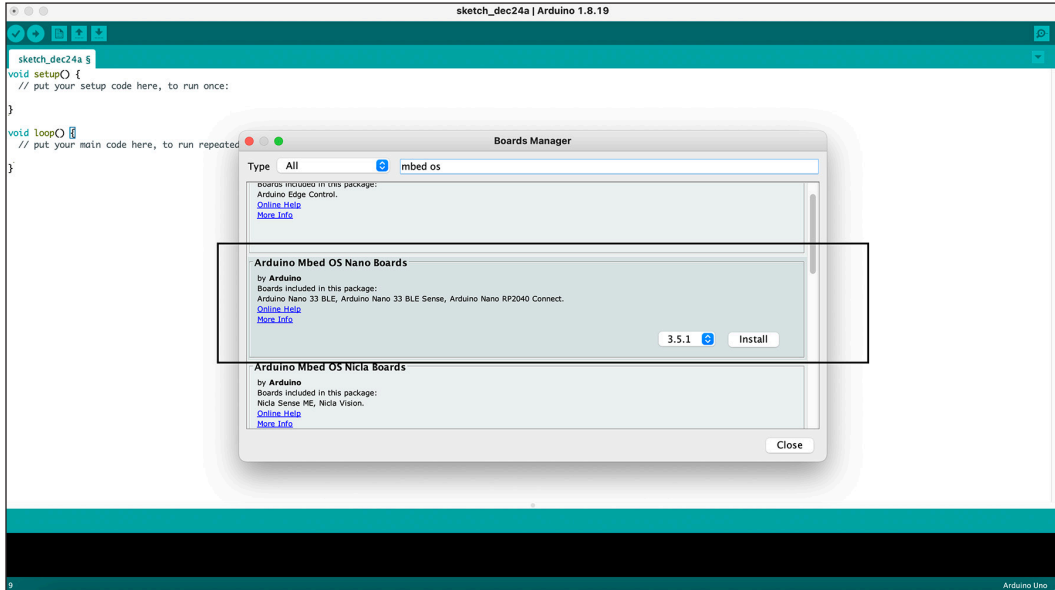


Figure 8.3: Installing the Mbed OS core for Nano boards

Once the necessary library is installed, close the IDE. Now, connect your Arduino Nano board to your computer’s USB port via the micro-USB cable. Wait for the green LED on the Arduino to be ON. Now, open the IDE again. You need to perform two important steps. First, go to **Tools** → **Board** → **Arduino Mbed OS Nano Boards** → **Arduino Nano 33 BLE** to select the Arduino Nano board. Next, go to **Tools** → **Port**, check, and select the port corresponding to the board.

For Windows machines, the port will be showing something like `<com14>` (**Arduino Nano BLE**). For Linux, the port will look something like `/dev/ttyACM0`, and for macOS X, the port will be something like `/dev/cu.usbmodem11201` (**Arduino Nano BLE**). Of course, the final numeric value might vary at your end. Note carefully, every time you connect your Arduino to your computer for programming, ensure that you have selected the board and also the proper port at the beginning. Your program will not be uploaded if the board and the port is not recognized by the host machine, and it will throw an error.

Now, let us execute a simple program on the Nano board. Arduino IDE comes with plenty of example codes. We will execute one of them. Go to **File** → **Example** → **01.Basics** → **Blink**. A new code window will appear, which is shown in figure 8.4. The code inside the text editor looks very similar to a C/C++ program. The program description and the expected output are mentioned at the beginning of the code. This is possibly the simplest Arduino program for beginners. It turns on the inbuilt

LED of the Arduino for one second and then turns it off for another one second, which keeps repeating.

Refer to *figure 8.4*:

```

Blink
/*
Blink

Turns an LED on for one second, then off for one second, repeatedly.

Most Arduinos have an on-board LED you can control. On the UNO, MEGA and ZERO
it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN is set to
the correct LED pin independent of which board is used.
If you want to know what pin the on-board LED is connected to on your Arduino
model, check the Technical Specs of your board at:
https://www.arduino.cc/en/Main/Products

modified 8 May 2014
by Scott Fitzgerald
modified 2 Sep 2016
by Arturo Guadalupi
modified 8 Sep 2016
by Colby Newman

This example code is in the public domain.

https://www.arduino.cc/en/Tutorial/BuiltInExamples/Blink
*/

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000); // wait for a second
  digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW
  delay(1000); // wait for a second
}

```

Figure 8.4: Code to blink an LED on Arduino

Now, let us have a code walkthrough. Arduino programs are termed as **sketches**. Every Arduino sketch must have two void functions, **setup()** and **loop()**. The **setup()** function is used to configure the Arduino device for the program. The function is executed only once at the beginning of the program. We typically perform all initialization tasks inside it. In the preceding code example, we initialize the digital pin of the inbuilt LED of the Arduino inside **setup()**. The function **loop()** is

an infinite loop that executes forever once the `setup()` function is executed. We put all our programming logic inside this function. In the preceding code example, we first turn on the LED by setting the voltage level as **HIGH** to the digital pin of the LED. Then, we wait for a second, turn off the LED by setting the voltage of the digital pin to **LOW**, and then again wait for a second. The whole thing keeps repeating.

Now, we will compile the sketch. Click on the *tick* button located on the top of the text editor in the IDE to compile. Refer to *figure 8.4*. In case of a compilation error, the error message will be printed in the console at the bottom of the text editor. Once it is successfully compiled, click on the *right arrow* button next to the tick button to upload the sketch to the Arduino Nano. Make sure you have selected the correct board and the port before uploading the code. You will see the log messages in the console. Once the program is successfully uploaded, you will find that the yellow LED of the Nano board, next to the micro-USB port and opposite to the green LED, will start blinking. It will turn on for a second and then turn off for another second, and the process will continue.

Remember, a program uploaded to a microcontroller executes forever as long as the power supply is ON. If you disconnect the power by removing the cable and reconnecting again, it will start executing the program as soon as the power supply is restored. You can upload a new program to remove the existing program. A simple way to remove an existing program is to upload an empty sketch to Arduino. To create an empty sketch, create a new sketch, compile and upload the following program:

```
void setup()
{
}

void loop()
{
}
```

It will delete the existing program and upload a new program that does nothing. You can again upload a fresh sketch.

Now, we have successfully set up our Arduino Nano and executed our first program on it. In the next section, we will learn to deploy our first TinyML application on it.

First TinyML project on the microcontroller—modulating the potentiometer

Our first TinyML application on Arduino Nano is a fairly simple project, which is intended to guide you through different steps of implementing and deploying a TinyML project from scratch on a real-world microcontroller. We will specifically use the TensorFlow Lite for Microcontrollers package to implement the project.

Let us understand the problem statement before implementing it. In this project, we are going to modulate the voltage output of a linear potentiometer according to a halfwave sinusoidal curve. You might have read about potentiometers in your undergraduate courses on electrical engineering. A potentiometer converts mechanical displacement into an electrical output. A potentiometer is basically a three-terminal variable resistor in which the resistance can be manually varied to control the current flow in the circuit. They are commonly used as voltage dividers in electrical circuits. The circuit diagram of a potentiometer is shown in *figure 8.5 (a)*. In simple words, a potentiometer works by varying the position of a sliding contact across a uniform resistance. As shown, the entire input voltage is applied across the whole length of the resistance. The output voltage is the voltage drop between the fixed and the sliding contact. As we move the sliding contact from one end of the resistance to another, the output voltage increases. Potentiometers are commercially available in the form of a small passive electrical component shown in *figure 8.5 (b)*. It has three pins and a manual rotating shaft. The two outer pins are connected to the **voltage input (Vin)** and the **ground (GND)**, respectively. You can connect any of one of them to the input voltage and the other to the ground. The analog output voltage is obtained from the central pin. You need to rotate the shaft manually to change the resistance. As a result, the output voltage also varies.

Refer to *figure 8.5*:

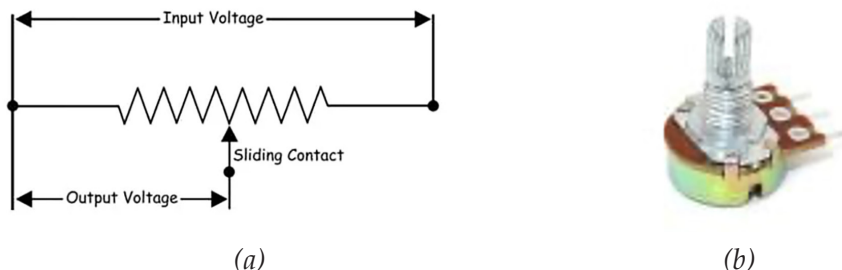


Figure 8.5: (a) Circuit diagram of a potentiometer (b) a 1-kiloohm potentiometer

In this project, we will use the potentiometer to linearly divide the source voltage ranging between 0 V and 3.3 V. The voltage output by the potentiometer will be converted to a halfwave sinusoid to modulate the LED.

We will implement our project in the following three steps:

1. First, we will connect the potentiometer to our Arduino Nano board and implement a baseline program that reads the potentiometer value and applies it to the inbuilt LED of the Arduino Nano to control its brightness. As expected, the brightness of the LED will vary linearly as we rotate the potentiometer shaft to increase the voltage.
2. Next, we will create a machine learning model to convert the linear potentiometer values according to halfwave sinusoid. We will implement a simple neural network that takes inputs within a linearly spaced range of values and maps them into a halfwave sinusoid. The network model will be developed in TensorFlow. This part of the project will be implemented in Colab in Python. We will then convert the TensorFlow model into the equivalent C++ library for Arduino.
3. Finally, we will implement the inference program on the Arduino IDE. The program will receive the potentiometer value as input and convert it using the model and modulate the intensity of the LED according to the model output. This part of the project will be implemented using the APIs provided by TFLite for Microcontrollers.

Required components

We need certain components in order to implement the project. Apart from the Arduino board, we require a potentiometer that will be connected to the Arduino Nano to get the input data. We strongly recommend to use a breadboard to make the necessary circuit connections. In this way, you do not need to solder the components. You will require to have the following components to implement the project:

- **Arduino Nano 33 BLE Sense:** Make sure that the header pins are properly soldered to the board so that it can be easily inserted to a breadboard to make necessary connections. Refer to *figure 8.1 (a)*.
- Micro-USB to USB cable for power supply.
- 1-kiloohm linear potentiometer.
- Breadboard.
- Few male-to-male jumper wires to make necessary connections.

Connecting the circuit

A breadboard is a solderless device for the temporary designing of prototype electronic circuits for various experimental purposes in the laboratory environment. Different components of an electronic circuit can be easily interconnected by inserting their pins into the appropriate holes of a breadboard and also by using jumper wires. Hence, you do not have to solder them in your circuit and can easily remove the circuit components. *Figure 8.6* features the layout of a breadboard.

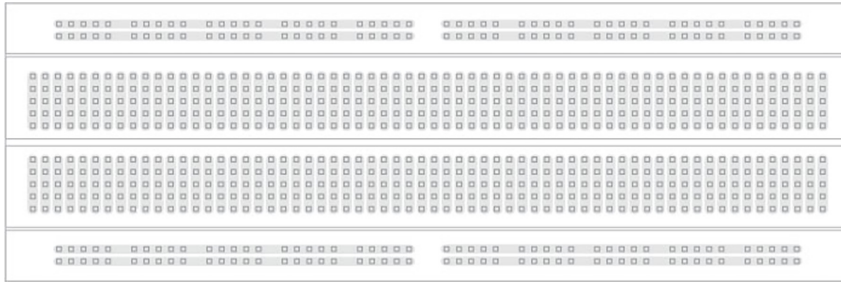


Figure 8.6: Layout of a breadboard; the interconnectivity is highlighted

The highlighted regions in the preceding figure indicate how different holes are interconnected inside the breadboard. There is a split in the middle of the breadboard. The holes on either side of the split are totally disconnected. The holes in the top two rows and the bottom two rows in a breadboard are connected horizontally. The remaining holes are connected vertically. All connected holes are internally shorted.

Now, let us learn how to place the Arduino Nano on the breadboard. Place it in such a way that the pins at one half of the Nano are inserted in one side of the board, and the remaining pins are on the other side of the breadboard. That means the pins at each side of the Nano are inserted at either end of the split. Refer to *figure 8.7*. The connection ensures that no two pins are internally connected. Make sure the pins are fully inserted in the breadboard holes. Take proper care while inserting or removing the Nano so that the pins are not damaged in the process.

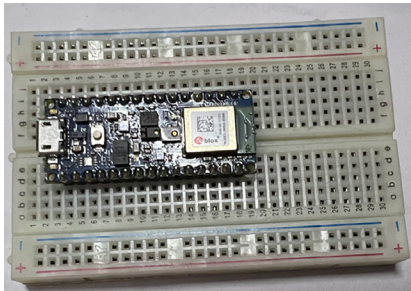


Figure 8.7: Placing the Arduino Nano on a breadboard

Now, follow *figure 8.8* to connect the potentiometer to the breadboard. Make the following connections using jumper wires:

- The pin at one end of the potentiometer connects to the GND of the Arduino Nano (pin number 29. Refer to *figure 8.1 (b)*).
- The pin at the other end of the potentiometer connects to the Vin (3.3 V) of the Nano (pin number 17).
- The central pin of the potentiometer connects to the analog pin A0 (pin number 19) to read the potentiometer value as input.

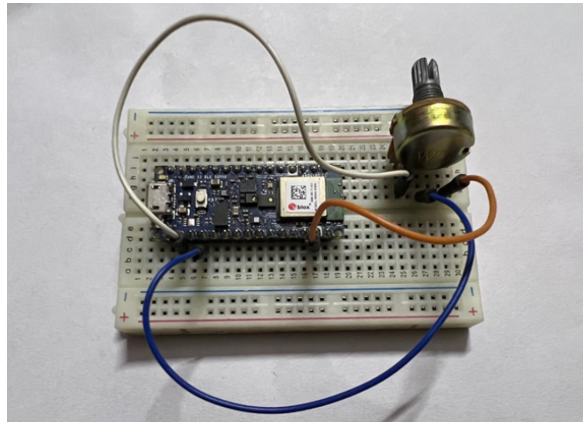


Figure 8.8: Connecting the potentiometer to the Arduino Nano

Finally, connect the Arduino to the host computer via the micro-USB port using the cable for power supply. Open the Arduino IDE on your computer. First, go to **Tools** → **Board** to select the Arduino Nano board, and then go to **Tools** → **Port** to select the port corresponding to the Nano.

Read potentiometer to control the brightness of the LED

Now, we will implement a program to read the potentiometer value and also use the same to control the brightness of the inbuilt LED of the Nano. Open the Arduino IDE. Go to **File** → **New**. A new sketch for an empty program will open. Copy and paste the following code into the text editor:

```
void setup() {  
    // initialize serial communication at 9600 bits per second:  
    Serial.begin(9600);  
    pinMode(LED_BUILTIN, OUTPUT);  
}
```



```
}

void loop() {
  // put your main code here, to run repeatedly:
  // read the input on analog pin 0:
  int i ;
  int cnt = 20;
  int sum = 0;
  for (i = 0; i<cnt; i++)
  {
    int sensorValue = analogRead(A0);
    sum = sum + sensorValue;
    delay(1);          // delay in between reads for stability
  }
  int avg_value = sum/cnt;
  int outputValue = map(avg_value, 0, 1023, 0, 255);
  // print out the brightness value
  Serial.println(outputValue);
  analogWrite(LED_BUILTIN, outputValue);
  //delay(1);          // delay in between reads for stability
}
```

Let us now understand what is happening in the code. We first establish a serial communication between the Arduino and the host computer on port 9600. We will listen to that port to get the data stream (the potentiometer reading) sent by the Arduino to the host machine. Next, we initialize the LED inside the `setup()` function. The main program goes inside the function `loop()`. Here, we first read the voltage value provided by the potentiometer from the A0 analog pin. The analog pins in Arduino have 10-bit **Analog to Digital Converter (ADC)** to convert the analog voltage reading into an equivalent digital value for the microcontroller. Hence, the readings will be between 0 and 1023. Even in a stable position, the potentiometer reading can occasionally fluctuate due to circuit noise. To get a stable value, we take the average of every 20 readings and store that value in the variable, `avg_value`. Next, we map the value, `avg_value` between 0 and 255 in order to safely apply it to an LED. The mapped value is assigned to the variable `outputValue`. Here, 0 and 255 correspond to the minimum and maximum brightness for the LED. The function

`Serial.println()` prints the variable `outputValue` to the serial port so that it can be accessed by the host computer. Finally, we assign that value to the LED pin to control its brightness.

Now, save the preceding sketch with a suitable name, compile the code, and upload it to the Arduino Nano. Once the code is uploaded, slowly rotate the manual shaft of the potentiometer to vary the output voltage. Note that the polarity of the potentiometer depends on the way you connect the two terminal pins to the source voltage and the ground. If you follow the connection diagram shown in *figure 8.8*, the output voltage will increase as you rotate the shaft clockwise, and will decrease if the shaft is rotated anti-clockwise. The yellow LED will be brighter as you rotate it clockwise, and the brightness will reduce otherwise. On the IDE, go to **Tools** → **Serial Plotter**. A new window will appear, which plots the live data stream sent to the host computer by the Arduino Nano on serial port. In the preceding program, we are printing the variable `outputValue` value in the serial port, which will be plotted in the form of a live graph. Now, fully rotate the manual shaft from one end to another in the clockwise direction, wait for a moment, and then go back to the original position by rotating it anti-clockwise. You will see that the plotted values will linearly increase from 0 to 255 as you go clockwise and then eventually return back to zero as you go anti-clockwise. Refer to *figure 8.9*:

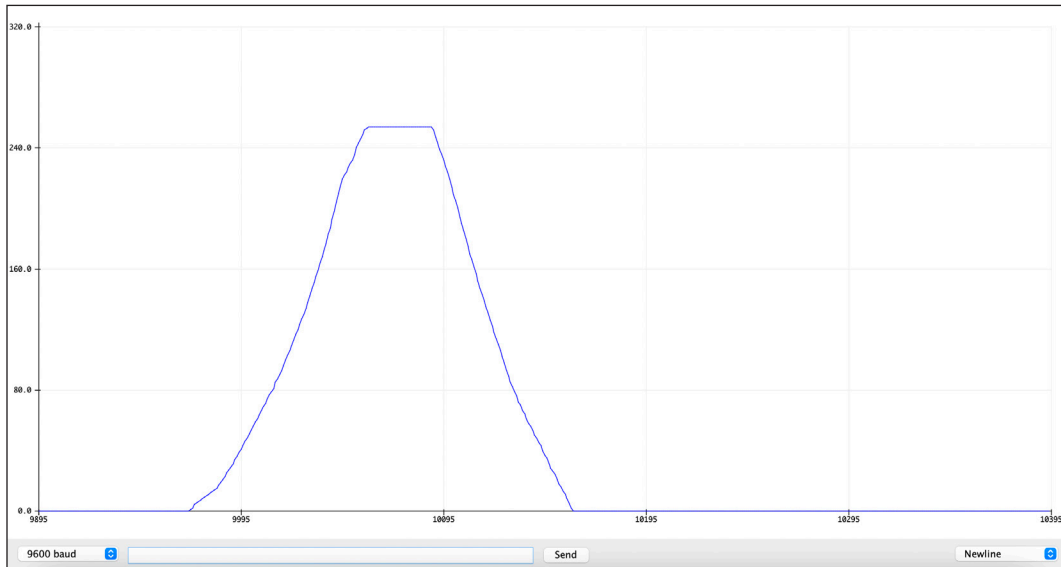


Figure 8.9: Serial plotter output

The more uniformly you rotate the shaft, the better triangular shape will be plotted. With this, we have successfully implemented our baseline code to read the potentiometer value and control the LED brightness linearly. Now, we will

implement the actual project to modulate the linear potentiometer readings according to a halfwave sinusoid. In the next section, we will implement a simple neural network model in Colab to convert a series of linearly spaced inputs into a halfwave sinusoid. The model will later be used by our Arduino to modulate the potentiometer readings.

Creating a TensorFlow model to modulate the potentiometer reading

In the previous section, we have created the baseline program to control the brightness of an LED based on the voltage provided by a potentiometer. In this section, we will create a simple neural network that takes an input value and converts it into a halfwave sinusoid. Mathematically speaking, we will create a neural network that will take a value x as input and return an output $f(x)$. If the range of x is in between 0 and T , then $f(x)$ is given by the following:

$$f(x) = \sin\left(\frac{\pi}{T} \cdot x\right)$$

The value of $f(x)$ will increase from 0 with x . It reaches the maximum value of 1 at $x = T/2$. Then, it starts decreasing and again reaches 0 at $x = T$. We will develop the neural network model on Colab using Python and TensorFlow and then convert it into an equivalent TFLite model. Subsequently, the TFLite model will be converted into an equivalent C++ library for Arduino.

Note that in all previous projects, we have developed neural network models to solve classification problems, such as classifying images or determining whether a target person is present in an image or not. In this project, we are going to solve a regression problem.

Go to Colab on your browser and create a new notebook and save it with an appropriate name. As usual, we will start by importing the libraries.

```
>>import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

import numpy as np

from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt

import math
```

Now, we will obtain the relevant dataset to train and evaluate the network. Simulating the dataset for this problem is fairly simple. We will generate 1,000 linearly spaced data points between 0 and 1, which will be the input of the neural network. The output values will be computed by applying them to the halfwave sinusoid function. Refer to the following code that generates the data and plots:

```
>>num_sample = 1000

x = np.random.uniform(low=0, high=1, size=num_sample).astype(np.float32)

# Shuffle the values to ensure they are not in order
np.random.shuffle(x)

y = np.sin(math.pi*x).astype(np.float32)

plt.plot(x, y, '.')

plt.xlabel('x values (input)')

plt.ylabel('y values (output)')
```

The program will generate a nice-looking curve of a halfwave sinusoid, shown in *figure 8.10*. The input and the output are stored in the variables x and y , respectively. The output y increases from 0 with x , reaches the maximum value at $x = 0.5$, and then again decreases to eventually reach 0 at $x = 1$.

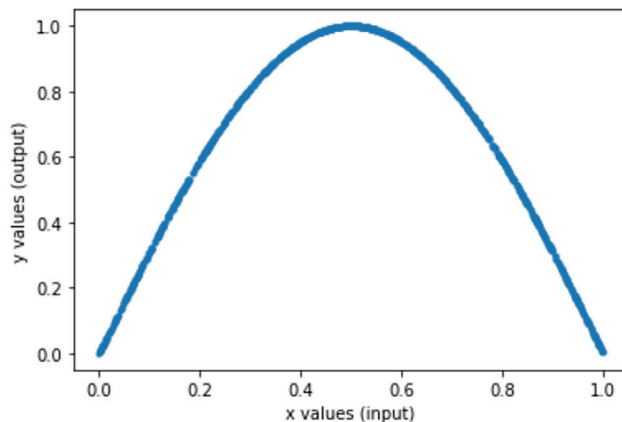


Figure 8.10: Plotting the generated data

In most real-world applications, the inputs and outputs cannot be modeled by using a simple mathematical equation, as the output is often influenced by external noise. This is the reason we use machine learning to estimate the relationship between

inputs and outputs empirically in a data-driven way. In this application, we will add some low-amplitude random noise to corrupt the output. The noisy output data is plotted in *figure 8.11*.

```
>>y= y+ 0.05 * np.random.randn(*y.shape)
plt.plot(x, y, '.')
plt.xlabel('x values (input)')
plt.ylabel('y values (output noisy)')
```

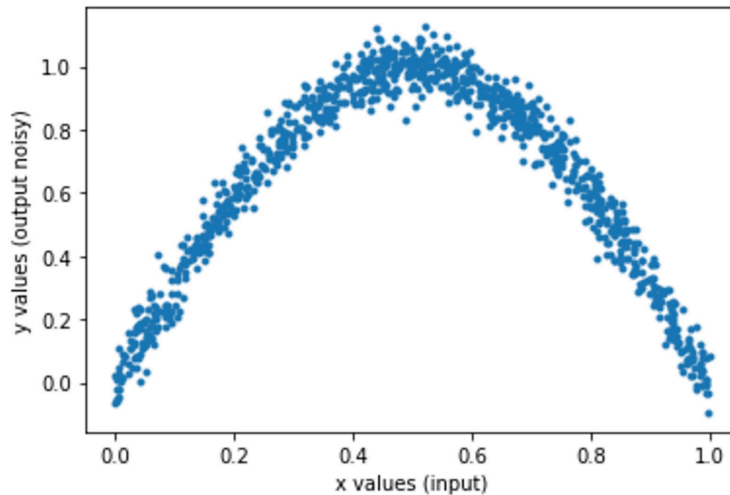


Figure 8.11: Plotting the dataset after the addition of noise

Now, for a given value of x , you cannot get the accurate value of y by applying it to a halfwave sinusoid function. Machine learning can help you to find a suitable relationship between them.

Before creating the machine learning model, we will first split the dataset into training and test sets. We randomly select 75% of data for training and the remaining 25% of data for evaluation purposes. We will use the `train_test_split()` function under the scikit-learn package for that.

```
>>x_train, x_test, y_train, y_test = train_test_split(x, y, train_size=0.75)
```

Next, we will define our neural network architecture. We will use a very simple neural network comprising two hidden dense layers having 8 and 16 nodes, respectively, followed by the final dense layer for prediction. Refer to the following code:

```
>>model = Sequential()
    model.add(Dense(8, activation='relu', input_shape=(1,)))
    model.add(Dense(16, activation='relu'))
    model.add(Dense(1))
```

Now, we will compile the model for training. Since we are dealing with a regression problem, we will reduce the **mean-squared error** between the predicted and the actual output values as the loss function. An Adam optimizer will be used to minimize the loss.

```
>>model.compile(optimizer='adam', loss='mean_squared_error')
```

Finally, we will train the network. The **batch_size** is taken as 50, and we will train the network for 500 epochs. The test set is used for validation:

```
>>history = model.fit(x_train, y_train, epochs=500, batch_size=50,
validation_data=(x_test, y_test))
```

Wait for the model to be completely trained. Now, we will use the model to predict on the test set. We will also visually compare between the actual and predicted values. Refer to the following code:

```
>>test_loss = model.evaluate(x_test, y_test)

y_test_pred = model.predict(x_test)

plt.clf()
plt.plot(x_test, y_test, 'b.', label='actual values')
plt.plot(x_test, y_test_pred, 'r.', label='predicted')
plt.legend()
plt.xlabel('x values')
plt.ylabel('y values')
plt.show()
```

The actual and the predicted values are plotted in blue and red, respectively. As shown in *figure 8.12*, the predicted values closely match the actual output values.

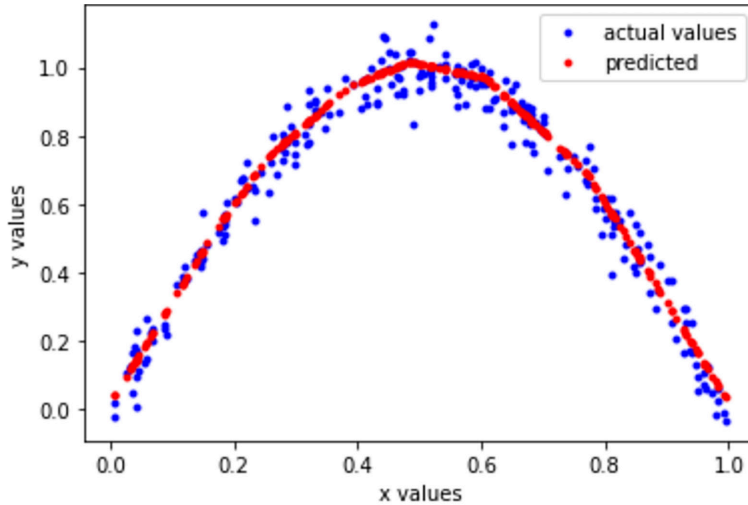


Figure 8.12: Comparison between predicted and actual values

Our baseline model is now created. We will next convert it into an optimized TFLite model. This part of the code is similar to what we did in our previous projects. We will create a `TFLiteConverter` object for the conversion and save the resulting model in the Colab workspace. Refer to the following code:

```
>>converter = tf.lite.TFLiteConverter.from_keras_model(model)
    model_no_quant_tflite = converter.convert()

import pathlib

tflite_models_dir = pathlib.Path('/content/tflite_models/')
tflite_models_dir.mkdir(exist_ok=True, parents=True)
tflite_model_file = tflite_models_dir/'model.tflite'
tflite_model_file.write_bytes(model_no_quant_tflite)
```

The resulting TFLite model size is around 2,588 bytes which is small enough to fit the Arduino Nano having 256 kilobytes of SRAM and 1 megabyte of flash memory. Since we are dealing with a very simple neural network, we are not applying any optimization technique while converting it into the TFLite model. However, you can try standard techniques like quantization-aware training or post-training-quantization at your end and check the impact on model size and performance.

Finally, we will evaluate the TFLite model on the test set. We are already familiar with this part of the program. We will create an **Interpreter** object using the TFLite model, allocate the tensor for the data and call the **Interpreter.invoke()** function to make inferences on the entire test set. The program will also plot the predicted and actual values. Refer to the following code:

```
>>x_test = x_test.reshape((x_test.size, 1))

  tflite_model_file = 'tflite_models/model.tflite'

  # Initialize the TFLite interpreter

  interpreter = tf.lite.Interpreter(model_path=tflite_model_file)

  interpreter.allocate_tensors()

  input_index = interpreter.get_input_details()[0]['index']
  output_index = interpreter.get_output_details()[0]['index']

  prediction_list = []

  x_test = x_test.reshape((x_test.size, 1))
  x_test = x_test.astype(np.float32)

  for i in range (len(x_test)):
    interpreter.set_tensor(input_index, [x_test[i]])
    interpreter.invoke()
    prediction = interpreter.get_tensor(output_index)[0]
    prediction_list.append(prediction)

  plt.clf()

  plt.plot(x_test, y_test, 'b.', label='actual values')

  plt.plot(x_test, np.array(prediction_list), 'r.', label='TFLite
  predicted')

  plt.legend()
```



```
plt.xlabel('x values')
plt.ylabel('y values')
plt.show()
```

It will generate the following curve, as shown in *figure 8.13*. It is evident that the TFLite model behaves similar to the base TensorFlow model.

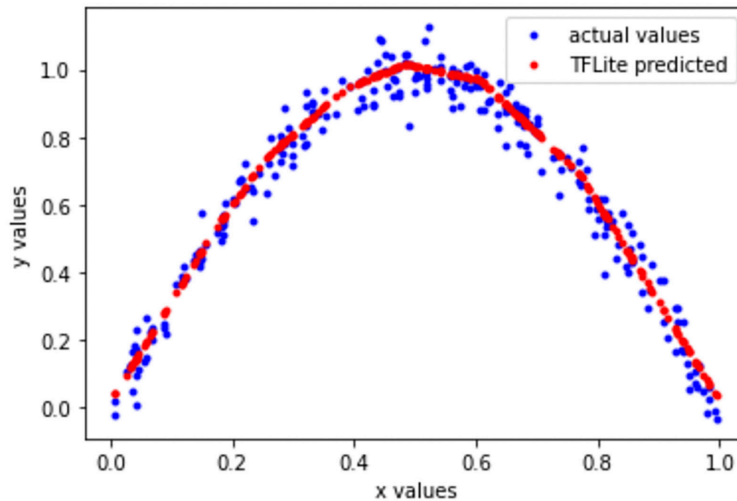


Figure 8.13: Comparison between predicted and actual values by the TFLite model

Now, we have the TFLite model, which can be called in Python but is not yet accessible by microcontrollers. We need to convert it into an equivalent C++ library for microcontrollers. We will do the conversion using the `xxd` command. On Unix-like operating systems, the `xxd` command creates a hex dump of a given input file. Execute the following lines of code in Colab that first installs `xxd` and then convert the TFLite model file into a C++ file.

```
>># Install xxd
!apt-get update && apt-get -qq install xxd
# Convert to a C source file for Microcontrollers model
!xxd -i tflite_models/model.tflite > model.cc
```

Once the code is executed, you will find a new file named `model.cc` generated in your Colab workspace, which will be used by your Arduino program to make inferences. Right-click on the file and download it to your host machine. With this, we have successfully created our neural network model. In the next section, we will see how this model file can be used in our program to make on-device inferences on

an Arduino Nano in order to modulate the LED brightness according to a halfwave sinusoid wave.

Inference on Arduino Nano using TensorFlow Lite for Microcontrollers

In this section, we will develop the program for Arduino Nano to make inferences using a TensorFlow model. As mentioned earlier, TensorFlow Lite for Microcontrollers is a specially designed, highly optimized library for implementing machine learning and deep learning applications on microcontrollers. The key objective of this section is to get familiarized with the library APIs to implement a TinyML application.

The program we are going to implement will read the analog voltage obtained from the potentiometer as input, convert the value according to halfwave sinusoid using the neural network model we created in the previous section, and finally, modulate the LED brightness accordingly. To begin, first, connect the Arduino to your computer and open the Arduino IDE. Now, we need to install the necessary library. TFLite for Microcontrollers is written in C++ 11, and hence, can be imported into any C++ project. The framework is readily available as a standalone Arduino library on Mbed OS. Go to **Tools** → **Manage Libraries** to open the **Library Manager**. Search for **arduino_tensorflow_lite** in the search window. The original Arduino TensorFlow Lite library is removed from **Library Manager** at the time of writing this book. The readers can download the **Harvard_TinyMLx** library, which contains a version of the TensorFlow Lite library for Arduino Nano. Refer to *figure 8.14*. Install the library.

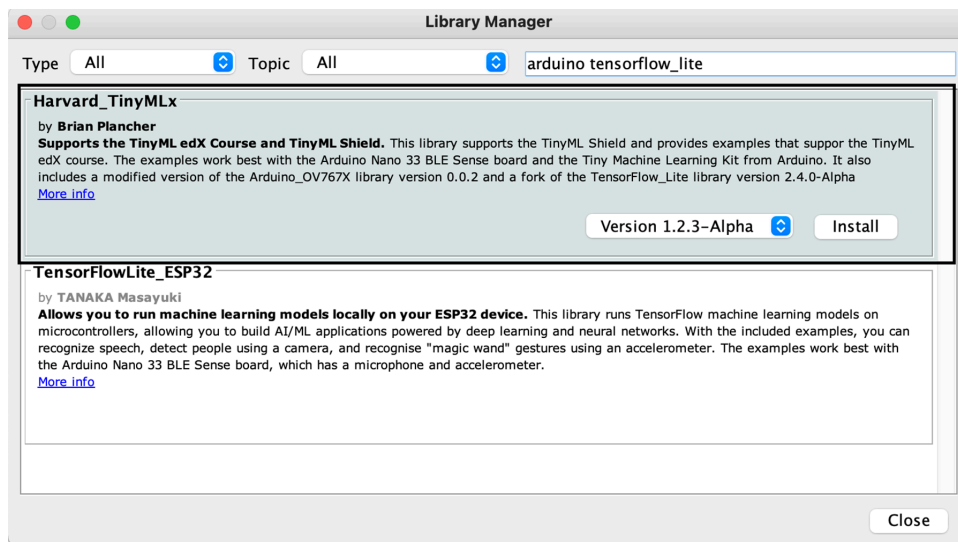


Figure 8.14: Installing TensorFlow Lite library for Arduino Nano 33 BLE Sense

Now, you are ready to implement TinyML projects on your Arduino. Go to **File** → **New** on your IDE to open a new sketch and save it with a suitable project name.

Similar to a C/C++ program, we will begin our code by including the necessary files containing the required functionalities for TensorFlow Lite into our code.

```
#include <TensorFlowLite.h>
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
```

Next, we will include the header file corresponding neural network model that we created in the previous section.

```
#include "model.h"
```

As expected, we need to create two files, **model.cpp** and **model.h**, in the project workspace. Click on the down arrow located at the right side of the toolbar of the IDE and select **New Tab** from the dropdown menu. Create two blank files and save them as **model.cpp** and **model.h**. Go to **model.cpp** and define the following two variables.

```
alignas(8) const unsigned char g_model[] = {
};

const int g_model_len = ;
```

Next, open the downloaded model file, **model.cc** in a text editor. Copy the entire hex code inside the curling braces next to the variable **unsigned char tf_lite_models_model_tf_lite[]** and paste inside the curling braces next to **g_model** in **model.cpp**. Next, go to the last line of the downloaded model file to copy the integer value assigned to the variable **tf_lite_models_model_tf_lite_len** and assign it to the variable **g_model_len**. Finally, open the header file **model.h** and enter the following code. Save the file

```
#ifndef TENSORFLOW_LITE_MICRO_POTENTIOMETER_HELLO_WORLD_MODEL_H_
#define TENSORFLOW_LITE_MICRO_POTENTIOMETER_HELLO_WORLD_MODEL_H_
extern const unsigned char g_model[];
extern const int g_model_len;
#endif //
```

Now, we have included the model files in our program. Go back to the main sketch. We will first define the data structures corresponding to the model, the model input, the model output, and an error reporter defined as per TFLite for Microcontrollers. Next, we will define a TensorFlow interpreter. We will also need to allocate a dedicated memory space called a *Tensor Arena*, which is the working space of the interpreter to perform the inference operation. Assigning the memory space is a very critical task in creating any application on the microcontroller. You need to assign and adjust this carefully, depending upon your model length, input and output size, and other constraints of the applications and also the capacity of the microcontroller. Refer to the following code:

```
namespace {  
    tflite::ErrorReporter* error_reporter = nullptr;  
    const tflite::Model* model = nullptr;  
    tflite::MicroInterpreter* interpreter = nullptr;  
    TfLiteTensor* model_input = nullptr;  
    TfLiteTensor* model_output = nullptr;  
  
    constexpr int kTensorArenaSize = 5 * 1024;  
    uint8_t tensor_arena[kTensorArenaSize];  
}
```

Now, we will write the **setup()** function to initialize different variables. Refer to the following code:

```
void setup() {  
  
    Serial.begin(9600);  
    pinMode(LED_BUILTIN, OUTPUT);  
  
    // Set up logging (will report to Serial, even within TFLite functions)  
    static tflite::MicroErrorReporter micro_error_reporter;  
    error_reporter = &micro_error_reporter;  
}
```

```
// Read and map the model into a usable data structure
model = tflite::GetModel(g_model);
if (model->version() != TFLITE_SCHEMA_VERSION) {
    error_reporter->Report("Model version does not match Schema");
    while(1);
}

static tflite::AllOpsResolver resolver;
// Create an interpreter to run the model to make inference
static tflite::MicroInterpreter static_interpreter(
    model, resolver, tensor_arena, kTensorArenaSize,
    error_reporter);
interpreter = &static_interpreter;

// Allocate memory from the tensor_arena for the model's tensors
TfLiteStatus allocate_status = interpreter->AllocateTensors();
if (allocate_status != kTfLiteOk) {
    error_reporter->Report("AllocateTensors() failed");
    while(1);
}

// Assign model input and output buffers to pointers
model_input = interpreter->input(0);
model_output = interpreter->output(0);

}
```

Now, go through the preceding code snippet carefully. We are performing the following steps in the preceding lines of code.

1. Initialize the LED pin.
2. Setup a logger for error reporting.
3. Loaded the training model and map it into a suitable data structure as per TFLite for Microcontrollers library.
4. An **AllOpResolver** instance is declared so that the interpreter can access the machine learning operations to run the model. Currently, TFLite for Microcontrollers supports only few such operations. Note that with **AllOpResolver**, we include all possible operations in a neural network, such as convolution, dense, and so, which are supported by TFLite for Microcontrollers. This process obviously consumes more memory. However, we can afford this in our application because we are dealing with a simple neural network containing only two dense layers only. Alternatively, we can include only the required operations to reduce memory consumption.
5. Create a **MicroInterpreter** object to make inference.
6. Allocate memory from the defined **Tensor Arena** to the model's tensors.
7. Assign model input and output buffers to pointers. In this application, we have a single-valued input and output.

Once all variables are set up, we can implement the **loop()** function to execute the actual operation. In this loop, we read the potentiometer value from pin A0. Similar to the baseline code, it calculates an average of 20 successive readings to get a stable value. The value is mapped between 0 and 1 and is copied to the model's input tensor. Next, we run **MicroInterpreter->Invoke()** to make the inference. The output is obtained from the output tensor. Since the model output is ranged between 0 and 1, the value is multiplied by 255 and then applied to the LED pin to control the brightness. The whole code inside **loop()** is as follows:

```
void loop() {
    int i;
    int cnt = 20;
    int sum = 0;
    for (i = 0; i < cnt; i++)
    {
        int sensorValue = analogRead(A0);
        sum = sum + sensorValue;
```

```
    delay(1);          // delay in between reads for stability
  }
  int avg_value = sum/20;
  // map the value between 0 and 1
  // as per the input to the neural network model created earlier
  float x_val = (float)avg_value/1023;

  // Copy value to input tensor
  model_input->data.f[0] = x_val;

  // Run inference
  TfLiteStatus invoke_status = interpreter->Invoke();
  if (invoke_status != kTfLiteOk) {
    error_reporter->Report("Invoke failed on input: %f\n", x_val);
  }

  // Read predicted y value from output buffer (tensor)
  float y_val = model_output->data.f[0];

  // Translate to a PWM LED brightness
  int brightness = (int)(255 * y_val);
  analogWrite(LED_BUILTIN, brightness);

  // Print the brightness value
  Serial.println(brightness);
}
```

Copy and paste all the preceding codes and save the sketch. Now, Compile and upload the program to the Arduino Nano. It may take some time to compile it for the first time. Wait till the sketch is uploaded. Now, slowly rotate the shaft of the

potentiometer from one end to another, either clockwise or anti-clockwise, and note the changes in the LED brightness. You can see that the yellow LED is initially OFF. The brightness will increase as the shaft is rotated (that is, the output voltage of the potentiometer increases). The brightness will be maximum when the shaft is somewhere in the middle of the entire range. The brightness will start reducing as you rotate further. The LED will be OFF again when the shaft is completely rotated. We print the variable, **brightness** that controls the LED brightness on the serial port. Now, go to **Tools** → **Serial Plotter** to visualize the plot. Rotate the shaft completely from one end to another. You will see a waveform similar to a halfwave sinusoid will be plotted. Refer to *figure 8.15*.

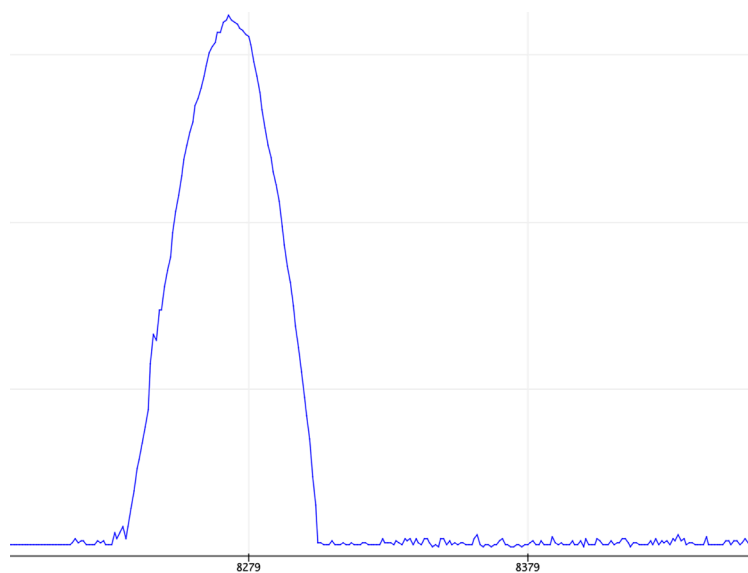


Figure 8.15: Model output at the serial plotter

Conclusion

In this chapter, we have implemented our first TinyML application on Arduino Nano 33 BLE Sense, a commercially available microcontroller unit. We have specifically used TensorFlow Lite for Microcontrollers, a highly optimized machine learning library for microcontrollers to implement the project. Arduino Nano BLE 33 Sense is a microcontroller-based development board commonly used in implementation of tiny machine learning applications. With little effort, the application developed in this chapter can also be implemented on other commercial microcontroller boards such as SparkFun Edge, Raspberry Pi Pico, and so on. Remember, the key objective of this chapter was to become familiar with the Arduino device and also with the APIs provided by TensorFlow Lite for Microcontrollers for end-to-end implementation of

a real TinyML project comprising a small neural network. In the upcoming chapter, we will implement a more complex project of keyword recognition from human speech on Arduino Nano using a more complex Convolutional Neural Network.

Key facts

- Microcontrollers are tiny electronic circuits to perform certain computational tasks. They are severely resource-constrained in terms of memory and computation power.
- TensorFlow Lite for Microcontrollers is a highly optimized software library written in C++ for implementing TinyML applications on microcontrollers.
- TensorFlow models trained on Colab can be converted into an equivalent C++ library to make inferences on microcontrollers.
- Arduino Nano BLE 33 Sense is a popular development board for experimenting with TinyML.
- Arduino IDE is a specially designed environment to program the Arduino microcontroller.
- An Arduino program source code must contain two key functions, **setup()** to initialize the program environment and **loop()** to execute the main program logic in a repeated manner.
- Before uploading a program to Arduino, always make sure to select the right board and port in the IDE.

Join our book's Discord space

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

<https://discord.bpbonline.com>



CHAPTER 9

Keyword Spotting on Microcontrollers

Introduction

The previous chapter described how to successfully deploy a machine learning model on Arduino Nano 33 BLE Sense, a commercially available microcontroller unit for TinyML applications. Recall the different steps involved in the project. First, we collected the necessary dataset for the application and created the machine learning model in Python on Colab. Next, we converted the model into an equivalent TFLite model which was then eventually exported as an equivalent C/C++ library for the target microcontroller. In general, microcontrollers have much lesser memory space and lower computational capacity than modern smartphones or single board computers like Raspberry Pi. Hence, each time we implement a project, we have to be very careful in optimizing our model specifically for the target hardware device where the model will be deployed. In the previous chapter, we created a simple neural network application that modulates the brightness of an LED. Our focus was on using **TensorFlow Lite for Microcontrollers**, a library specifically designed for running machine learning models on low-powered microcontrollers. The main objective of the previous project was to become familiar with the steps involved in creating a TinyML application from scratch to eventual deployment on a microcontroller. In this chapter, we will implement a more complex project of real-time keyword spotting on microcontrollers.

Keyword spotting is a widely used frontend application in the voice assistant devices such as modern smart speakers. The smart speakers have been extremely popular in recent times. We all are aware of the voice assistant devices such as Amazon's Alexa, Google Home, or Apple's Siri. They are intended to recognize and comprehend our voice commands and act accordingly. Amazon's Alexa and Google Home are available in the form of smart speakers, as shown in *figure 9.1*.

The voice assistant is also available on smartphones. For example, Siri is a voice-driven virtual assistant that is readily available on Apple's iPhone, iPad, and MacBook devices powered by iOS or macOS. A similar voice assistant service is also available from Google on the Android platform. Cortana is another popular voice assistant by Microsoft. Through voice assistant services you can perform a variety of tasks via speech commands, whether you want to play music from your playlist, check weather forecast, add an item to your online shopping cart, or control other smart devices in your home, like lights and TVs. Your smart speakers can handle it all.

Figure 9.1 features two commercially available smart speakers, Amazon Echo, and Google Home:



Figure 9.1: Commercially available smart speakers (a) Amazon Echo, enabled by Alexa and (b) Google Home

The underlying technology of the voice assistant devices is a perfect example of how edge AI and IoT communicate together. Recall how you operate your voice-assisted device on your smartphone. Suppose you want to ask your device about tomorrow's weather forecast. What do you do?

You do not ask the question directly without addressing your device. Instead, you first call your device by a dedicated proper noun, which is termed as a **keyword** or

a **wake-up command**. Then, you ask the actual question, for example, *What's the weather in Delhi?* The keyword is different for different devices. On iPhone, you say, “Hey Siri!”; on Android-based smartphones, you say, “Ok Google!,” and on Amazon Alexa, you need to say “Alexa!” as the dedicated keyword for addressing the device before your actual query. Your device first detects the dedicated keyword to wake up from a semi-passive state to an active state. Then it processes the following voice instruction asking about the weather forecast and responds accordingly. The task of identifying the dedicated keyword for the device is termed as keyword spotting.

The process of keyword spotting involves a lightweight machine learning application that runs on-device and should use minimal power. It should also have minimum latency. The sole purpose is to activate the voice assistant in order to initiate the actual voice assistant service. In this chapter, we will develop a basic keyword spotting application on the Arduino Nano microcontroller.

Structure

In this chapter, we will discuss the following topics:

- Working principles of a voice assistant
- Implementation of a keyword spotting algorithm in Python
 - Audio spectrogram
 - Designing a Convolutional Neural Network model for keyword spotting
- Introduction to Edge Impulse
- Implementing keyword spotting in Edge Impulse
- Model deployment

Objectives

This chapter will guide you through the process of creating a basic keyword spotting application on the Arduino Nano 33 BLE Sense. As mentioned in the previous chapter, the Arduino Nano device is equipped with various sensors, including a high-quality microphone, which will be readily used in this project for real-time speech recording as digitalized audio data and classification of one-second audio data directly on the microcontroller. The chapter can be broadly divided into two parts. Initially, we will design an end-to-end machine learning application in Python for keyword spotting using a **Convolutional Neural Network (CNN)** and test it on offline speech data. This will allow us to become familiar with the various steps involved in creating a basic speech recognition system. Later on, we will reimplement the keyword spotting application to execute on the actual microcontroller device. In

the previous chapter, we developed our TinyML application on Arduino IDE using TensorFlow Lite for Microcontrollers. However, this approach may not always be convenient to create real-life complex applications. In this chapter, we will introduce Edge Impulse, an easy-to-use browser-based tool for creating highly optimized machine learning models on embedded platforms. Using Edge Impulse, you can easily create a machine learning model for a wide range of target hardware devices with a minimum effort of code writing. Furthermore, depending on your hardware specifications, Edge Impulse suggests the optimum model for your application.

Working principles of a voice assistant

Before developing the actual keyword spotting application, it is important to understand how voice assistants operate. Suppose we wish to know about today's weather forecast from our smart speaker. The smart speaker comes with a high-quality microphone for audio reordering. The microphone of the smart speaker is always on and waits for the dedicated keyword. Under this scenario, it is in an inactive state and does not respond to your commands. It gets activated only when it detects the dedicated keyword. Then, only it responds to your queries. To summarize, when you ask a question, the following three things happen in the background:

- **Keyword spotting:** Your voice assistant is equipped with a powerful microphone that always passively listens for the keyword, for example, "Hey Siri" for an Apple device or "Alexa" for an Amazon Echo. Keyword spotting is the primary task performed by a voice assistant to wake up in order to answer your queries. Keyword spotting is a real-time task which needs to be performed on-device. In fact, a lightweight tiny machine learning algorithm continuously running on the microcontroller of the speaker is responsible for doing the task. The keyword spotting has to be specific to one user voice. That means your device should be activated only when called by you. Remember, when you configure a new smartphone for the first time, it asks you to configure the voice assistant service. You are asked to say the designated keyword for few times, which is recorded as the training data to create a learning model based on your voice. Once the training is done internally, the model ensures that the device will wake up only when the designated keyword is spoken by you.
- **Natural language processing:** After the device is activated, it captures your following voice instructions that contain your actual question and respond accordingly. This part is done in the cloud, which requires internet connectivity. The processing involves **Natural Language Processing (NLP)**, a dedicated branch of artificial intelligence for speech processing. It combines

computational linguistics with machine learning models to understand human language in the form of speech or text data to understand its meaning. This part of the job involves complex machine learning algorithms which cannot be done on-device. Once the voice instruction is decoded, it searches its online data repository to find a suitable answer to it.

- **Text to Speech:** Now the machine knows what to do to your query. It converts the resolution to human speech and plays back to you, which sounds similar to a human voice. On certain devices, it maps some actions, for example, turning on/off a smart light based on your instruction.

In this chapter, we will implement a simple on-device keyword spotting application on Arduino Nano 33 BLE Sense. We will create a machine learning model and train it to detect two target keywords, “on” and “off.” Since the Arduino Nano 33 BLE Sense has a built-in microphone, no additional sensors are necessary for voice recording. Our application will continuously record background sound and internally process every one-second audio recording. In the next section, we will implement an end-to-end keyword spotting algorithm in Python using TensorFlow to get a real feel of the speech recognition application that we are going to build. Subsequently, we will implement a deployable keyword spotting model using **Edge Impulse** and upload it on Arduino for real-time keyword spotting.

Implementation of a keyword spotting algorithm in Python

In this section, we will learn to implement a baseline keyword spotting project in Colab using TensorFlow. The key objective is to understand the different steps involved to implement a speech recognition system from scratch, which is the basic building block of any keyword-spotting application. Later in this section, we will use the Edge Impulse platform to develop an Arduino library for real-time on-device keyword spotting.

To begin, let us create a new project in Colab and save it with an appropriate name. The following code examples are strongly influenced by the TensorFlow tutorial on audio recognition. You may refer to the Colab notebook for more details¹. We strongly recommend to use the power of GPU for this project so that the model takes less time to train. On your Colab, go to **Runtime** → **Change runtime type**. It will open a new window, **Notebook settings**. Select **GPU** from the dropdown menu under **Hardware accelerator**. Click on **Save**. Now, click on the **Connect** button located at the right of the Colab project to connect to a online GPU runtime.

1 https://www.tensorflow.org/tutorials/audio/simple_audio

We will start by importing the necessary Python libraries.

```
>>!pip install -U -q tensorflow tensorflow_datasets
```

```
import os

import pathlib

import matplotlib.pyplot as plt

import numpy as np

import seaborn as sns

from numpy import random

import shutil

import tensorflow as tf

from tensorflow.keras import layers

from tensorflow.keras import models
```

Next, we need to get the necessary dataset to create the machine learning model. We will use the publicly available **Speech Commands** dataset in our project. The dataset was contributed by *Pete Warden*, Technical Lead of the TensorFlow Micro team at Google. The dataset contains more than 60,000 instances of one-second-long utterances of 30 different spoken keywords such as “yes”, “no”, “up”, “down”, “left”, “right”, “on”, “off,” and many more, which are stored in terms of digital audio files. The audio files are sampled at 16 kilohertz. You may refer to the paper “*Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*” by *Pete Warden* to know more about the dataset. Let us now download the dataset in our project workspace in Colab. We will create a directory called **data** in our workspace to store the dataset. The dataset will be downloaded as a compressed zip file, and the uncompressed data will be stored inside the directory. Refer to the code to perform the previously-mentioned tasks:

```
>>DATASET_PATH = 'data/'

data_dir = pathlib.Path(DATASET_PATH)

if not data_dir.exists():

    tf.keras.utils.get_file(
```



```
'speech_commands.zip',  
origin='http://download.tensorflow.org/data/speech_commands_v0.02.  
tar.gz',  
extract=True,  
cache_dir='.', cache_subdir='data')
```

Downloading and extracting this dataset will take some time due to its large size of approximately 2.5 gigabytes. Once the process is complete, you can navigate to the Files button on Colab to view the file structure in your workspace. A new directory named **data** will be created, which will contain subdirectories corresponding to all 30 keywords. Each subdirectory will contain the corresponding audio files in WAV format.

Now, let us delve into the problem we aim to solve in this project. We want to develop a keyword spotting algorithm that can detect the target keywords “on” and “off.” However, creating a binary classifier may not be sufficient, as the input audio could contain human speech with other words or may have no background audio at all.

In such cases, a binary classifier would predict either “on” or “off,” which is not the intended outcome. To deal with the scenario, we will create a multi-class classifier to detect four different class labels—“on”, “off”, “others”, and “silent”. Hence, we need to form a new dataset from the original Speech Commands dataset containing audio instances corresponding to the four target classes we are going to detect. We will create a new empty directory, **mydataset**, under the parent directory **data**. First, we will copy the two directories “on” and “off” with all their contents from the original Speech Commands dataset to the newly created directory. Next, we will create two empty directories: “others” and “silent”. Use the following code:

```
>>os.mkdir('data/mydataset')  
  
!cp -r 'data/on' 'data/mydataset'  
  
!cp -r 'data/off' 'data/mydataset'  
  
os.mkdir('data/mydataset/silent')  
  
os.mkdir('data/mydataset/others')
```

Now, we need to populate the newly created two directories with relevant files. We will randomly select a few examples from the remaining keywords in the Speech Commands dataset and mark them as others.

In the following code, we randomly selected 150 audio files from each label corresponding to different keywords available in the dataset, such as “yes”, “no”, “up”, “down”, “left”, “right”, and many others and place them in the designated directory for other types of audios. Refer to the following code:

```
>>other_labels = ['yes', 'no', 'up', 'down', 'left', 'right', 'bed',
'bird', 'cat', 'dog', 'happy', 'house', 'marvin', 'sheila', 'tree', 'wow']

    sample_per_label = 150

for label in other_labels:
    path = 'data/'+label

    f = os.listdir(path)
    num_files = len(f)
    file_indx = np.arange(num_files)
    random.shuffle(file_indx)
    for i in range(sample_per_label):
        index = file_indx[i]
        file_name = f[index]
        source = path + '/' + file_name
        destination = 'data/mydataset/others/' + file_name
        # copy only files
        if os.path.isfile(source):
            shutil.copy(source, destination)
```

Finally, we need to populate the silent folder. To maintain a class balance, we will create 2,400 silent audio files, each having a duration of one second. We will create an empty array and add a small amount of random noise to it in order to create some variation in the silent recordings and save them as WAV files. Refer to the following code:

```
>>import scipy

    from scipy.io.wavfile import write
```

```
fs = 16000
num_files = 2400

for i in range(num_files):
    sample = np.zeros(fs)
    filename = str(i*100)+ 'silent.wav'
    sample = sample + 0.1*random.randn(fs)
    scipy.io.wavfile.write('data/mydataset/silent/'+filename, fs,
        sample.astype(np.int16))
```

Similar to the Speech Command dataset, sampling rate of the silent audio files is set to 16,000 hertz, that means, every second of audio contains 16,000 sample points. Note the samples are represented in a 16-bit integer number in the audio file. Next, we need to convert the audio files into an equivalent TensorFlow dataset format that can be easily applied to a deep learning architecture. Additionally, we also need to split the dataset into training and validation sets. We will use the TensorFlow function `audio_dataset_from_directory()` for that. Refer to the following code. It will split the dataset in such a way that 80% of the data is kept for training and the remaining portion as the validation set. We also define a `batch_size` of 64 for training.

```
>>train_ds, val_ds = tf.keras.utils.audio_dataset_from_directory(
    Directory = 'data/mydataset',
    Labels = 'inferred',
    batch_size=64,
    class_names=None,
    validation_split=0.2,
    seed=0,
    output_sequence_length=16000,
    subset= 'both')
```

```
label_names = np.array(train_ds.class_names)
print()
print('label names: ', label_names)
```

Next, we will modify the dataset to drop the extra audio channel to make the overall computation lighter:

```
>>train_ds.element_spec

def squeeze(audio, labels):
    audio = tf.squeeze(audio, axis=-1)
    return audio, labels

train_ds = train_ds.map(squeeze, tf.data.AUTOTUNE)
val_ds = val_ds.map(squeeze, tf.data.AUTOTUNE)
```

Finally, we will split the validation set into two parts, an internal validation set and a test set. We will use the function `tf.data.Dataset.shard`.

```
>>test_ds = val_ds.shard(num_shards=2, index=0)
val_ds = val_ds.shard(num_shards=2, index=1)
```

Now, let us examine the dimension of the dataset. The following code extracts a batch of sample audio and the corresponding labels from the training set and prints the dimension.

```
>>for sample_audio, sample_label in train_ds.take(1):
    print(sample_audio.shape)
    print(sample_label.shape)
```

Since the batch size is 64 and each audio file is one-second-long with a sampling rate of 16,000 hertz, the dimension of the tensor `sample_audio` will be (64, 16,000).

Now, let us play a few audio files on Colab and listen to them. The following code will randomly select five audio files from the batch `sample_audio` obtained in the previous code and make them available to be played on Colab. You may execute the code for few times to randomly select different files in different runs and listen to them.

```
>>from Ipython import display

for i in range(5):
    indx = random.randint(0, sample_audio.shape[0]-1)
    label = label_names[sample_label[indx]]
    waveform = sample_audio[indx]

print('Label: ', label)
print('Audio shape: ', waveform.shape)
print('Audio playback')

display.display(display.Audio(waveform, rate=16000))
```

Now, we have the necessary data for keyword detection. We will create a machine learning classifier based on **Convolutional Neural Network (CNN)**. We know that a CNN architecture comprises a series of convolution filters that are primarily responsible for the extraction of relevant features from the input. For image data, the convolution filters perform some standard operations like edge extraction. In all previous chapters, we have applied CNN on image data. However, in this application, our inputs are audio waveforms, which is a sequence of data points collected over a period of one second. Such data are called as time-series data. A CNN may not be very effective on time-series data. In speech processing, we often represent the time-series audio signals into a spectrogram for processing. A spectrogram is an image-like format representing the time-frequency behavior of audio, which is a more realistic format to be applied to a CNN. In the following section, we will learn more about the audio spectrogram. We will also see, how the spectrogram information be used as an input to a CNN for the classification of different keywords.

Audio spectrogram

An audio signal is a combination of various frequency components. Frequency is measured in the unit of Hertz (Hz). Like any wave, a sound wave is generated due to vibrating objects like the strings of a guitar or the diaphragm of a drum that causes some sort of disturbance in the air. In human speech, the disturbance is caused by our vocal cord. If the frequency of vibration of a wave is in the range between 20 hertz and 20 kilohertz, we can hear it as an audio sound. The fundamental frequency of audio is termed as **pitch**. A high pitch sound corresponds to high frequency, and

a low pitch audio corresponds to low frequency. An audio signal can have multiple frequency components. The frequency range of an audio is termed as **spectrum**. The loudness of the audio signal at different frequency components is measured in absolute power or on a decibel scale. However, an audio signal is non-stationary in nature, which means that the frequency components and their loudness change with time. Hence, in audio processing, we often need to analyze both time and frequency information at the same time. The **spectrogram** analysis is a popular technique in audio processing that calculates the loudness or the spectral power at small time frames of the audio and also at various frequency components present in the signal.

The spectrogram is a time-frequency representation of audio. In simple words, the spectrogram displays the distribution of different frequencies present in an audio signal as it varies with time. It also represents the strength of various frequency components over time. Once computed, the spectrogram can be considered as a two-dimensional image representing both time and frequency information at the same time. It represents the time scale horizontally and the frequency scale vertically. Hence, it can be easily applied to a CNN architecture for classification.

On digital audio signals, the spectrogram is measured using a mathematical technique called **Short-time Fourier Transform (STFT)**. Here, the Signal is first broken into equal-length windows of a fixed duration. The window length taken is small enough so that there is no abrupt change in the major frequency components in successive windows. Next, we measure the spectrum for each window to determine the frequency components present in the window along with their strength. The frequency spectrum of a finite length discrete time signal is computed using the **Discrete Fourier Transform (DFT)**. The DFT measures two components, spectral amplitude, and phase. The spectral amplitude is computed for all windows and plotted over time to display the spectrogram. A detailed discussion of the underlying mathematics of DFT is beyond the scope of this book. Interested readers can go through the relevant books on digital signal processing to learn more. Now, we will implement a Python program to compute the spectrogram of an audio signal.

We will call a specially curated TensorFlow function `tf.signal.stft()` to compute the spectrogram of a time signal. As mentioned earlier, the function breaks the audio into small windows and computes DFT on each window. Here, we need to pass two parameters as input, `frame_length`, and `frame_step`. The first parameters indicate the length of the window in terms of the number of sample points. The second parameter defines the step size to proceed. By default, the STFT operation returns a set of complex numbers, which contain both the spectral amplitude and phase information. In order to get the spectrogram, we will compute the absolute

magnitude, which indicates the spectral power. Finally, we will reshape the spectrogram dimension in an image-like structure (**height x width x num_channel**) so that it can be applied to a CNN-like architecture. Refer to the following code²:

```
>>def get_spectrogram(waveform):
    # Convert the waveform to a spectrogram via STFT.
    Spectrogram = tf.signal.stft(
        waveform, frame_length=127, frame_step=64)
    # Obtain the absolute magnitude of the STFT.
    Spectrogram = tf.abs(spectrogram)

    # Add a third dimension as channel, so that the spectrogram can be
    used
    # as image-like input data with convolution layers (which expect
    # shape ('batch_size', 'height', 'width', 'channels')).

    spectrogram = spectrogram[..., tf.newaxis]
    return spectrogram
```

Now, let us plot a few sample waveforms from the audio batch, **sample_audio** we created earlier from the training dataset. We will first create a function to display the spectrogram. The spectrogram image looks like a graph. The x -axis indicates the time, and the y -axis indicates the frequency in the logarithmic scale. Refer to the following code:

```
>>def plot_spectrogram(spectrogram, ax):
    if len(spectrogram.shape) > 2:
        assert len(spectrogram.shape) == 3
        spectrogram = np.squeeze(spectrogram, axis=-1)
    # Convert the frequencies to log
    # Add an epsilon to avoid taking a log of zero.
```

² https://www.tensorflow.org/tutorials/audio/simple_audio

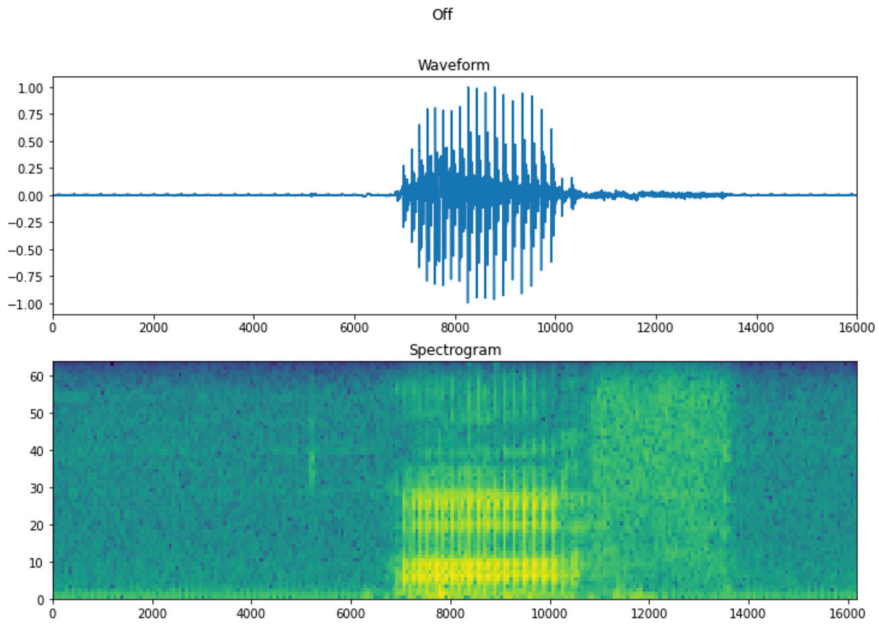
```
log_spec = np.log(spectrogram.T + np.finfo(float).eps)
height = log_spec.shape[0]
width = log_spec.shape[1]
X = np.linspace(0, np.size(spectrogram), num=width, dtype=int)
Y = range(height)
ax.pcolormesh(X, Y, log_spec)
```

Now, we will call the preceding function to plot a sample audio file and the corresponding spectrogram as a color image. Refer to the following code:

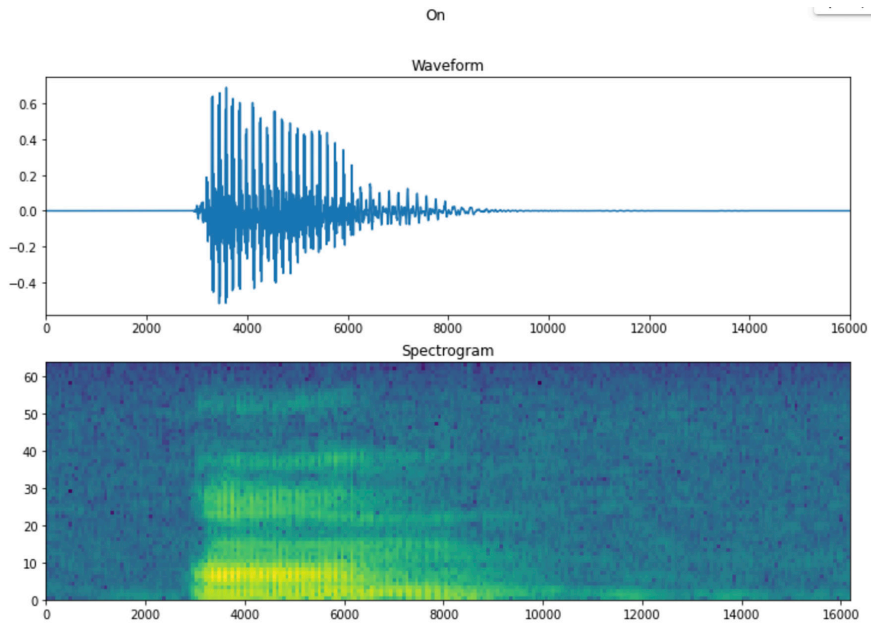
```
>>indx = random.randint(0, sample_audio.shape[0]-1)
label = label_names[sample_label[indx]]
waveform = sample_audio[indx]
spectrogram = get_spectrogram(waveform)
fig, axes = plt.subplots(2, figsize=(12, 8))
timescale = np.arange(waveform.shape[0])
axes[0].plot(timescale, waveform.numpy())
axes[0].set_title('Waveform')
axes[0].set_xlim([0, 16000])

plot_spectrogram(spectrogram.numpy(), axes[1])
axes[1].set_title('Spectrogram')
plt.suptitle(label.title())
plt.show()
```

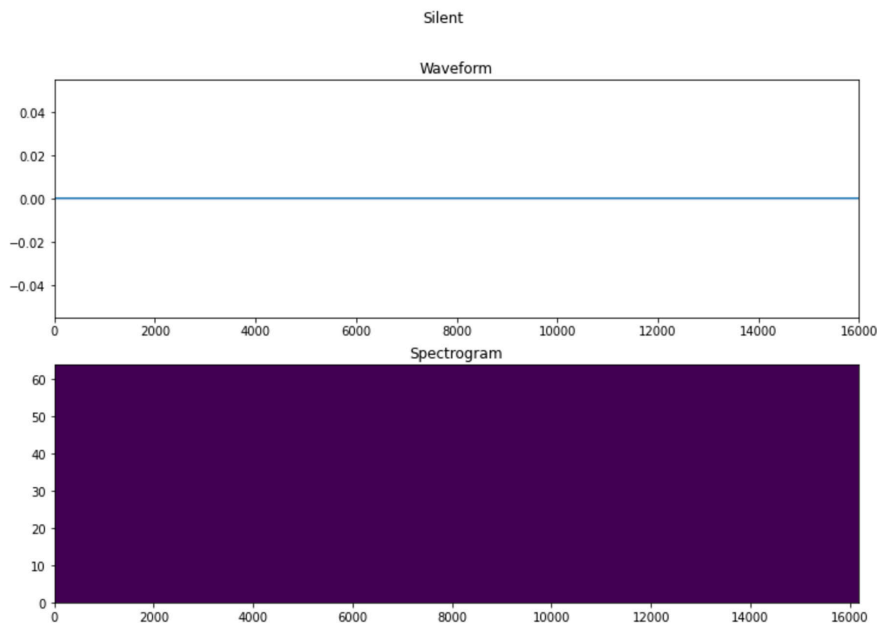
The preceding code will plot one randomly selected audio sample and the corresponding spectrogram. It will also print the audio label. Run the code for several times to get accustomed to the spectrum of different keywords we are going to detect. *Figure 9.2* shows the typical spectrogram of audio samples corresponding to “off”, “on”, and “silent” audio. Note, in all cases, the duration of the audio is one second.



(a)



(b)



(c)

Figure 9.2: Spectrogram of sample audio file (a) off, (b) on, and (c) silent

A spectrogram looks like a colored image. Let us understand the significance of different colors. A spectrogram depicts as a histogram where different colors indicate the audio power. The darker colors indicate lower spectral power, and the lighter colors indicate higher spectral power. For example, there is no power in the duration of a silent recording. As a result, the entire spectrogram is represented by dark violet color. For “off” and “on,” as shown in *figure 9.2*, the word is uttered somewhere in the middle of the recordings, as shown in the respective plots. The spectral power is the highest at the temporal location of utterance. This is represented by the yellow color in the spectrogram. The remaining silent regions in the recording are represented by appropriate darker colors.

Finally, let us check the spectrogram dimension by running the following command:

```
>>print('Spectrogram shape:', spectrogram.shape)
```

Depending upon the STFT parameters selected in the preceding code, the spectrogram dimension is (249, 65, 1). Finally, we will create a spectrogram dataset from the original audio dataset. We will also take the corresponding labels. Refer to the following function:

```
>>def make_spectrogram_ds(ds):
```

```

return ds.map(
    map_func=lambda audio,label: (get_spectrogram(audio), label),
    num_parallel_calls=tf.data.AUTOTUNE)

```

Now, we will use the preceding function to create the training, validation, and test sets with the spectrogram data:

```

>>train_spectrogram_ds = make_spectrogram_ds(train_ds)
    val_spectrogram_ds = make_spectrogram_ds(val_ds)
    test_spectrogram_ds = make_spectrogram_ds(test_ds)

```

Finally, we will perform the following operations to optimize the data reading and data parsing performance. Refer to the following code:

```

>>train_spectrogram_ds =      train_spectrogram_ds.cache().shuffle(10000).
prefetch(tf.data.AUTOTUNE)

    val_spectrogram_ds =      val_spectrogram_ds.cache().prefetch(tf.data.
AUTOTUNE)

    test_spectrogram_ds =      test_spectrogram_ds.cache().prefetch(tf.data.
AUTOTUNE)

```

Now, we have prepared our spectrogram dataset. In the following section, we will define a CNN model for keyword detection.

Designing a Convolutional Neural Network model for keyword spotting

We know that CNN is a powerful machine learning algorithms in image processing related applications. The convolutional operation is the basic building block of a CNN, which is responsible for feature extraction from the input image. A deep CNN architecture can have several convolutional layers for more detailed feature extraction, which may cause a large model due to a large number of trainable parameters. Such models might not be suitable for TinyML applications. Remember, we are planning to implement our model on an Arduino Nano. Our target microcontroller is having 256 kilobytes of RAM and 1 megabyte of flash memory. While making an inference, our entire program, along with the training model, needs to be fitted in the device RAM. There has to be memory space left for storing the recorded input audio data and other intermediate variables generated by the inference program during execution. The program also needs to run on the target hardware in real-time. Hence, we have

to be very much selective in defining the neural network architecture to create a relatively smaller yet accurate model.

The CNN we are going to use in this application will have a single convolutional layer followed by a dense layer with four nodes for the prediction of four types of audio, “off”, “on”, “others”, and “silent”. We will add dropout to the model. You will be surprised to see that the simple CNN will perform reasonably well in keyword detection. The following program defines the model:

```
>>from tensorflow.python.util.nest import flatten

    from tensorflow.python.ops.gen_nn_ops import Conv2D

    for example_spectrograms, example_spect_labels in train_spectrogram_
    ds.take(1):

        input_shape = example_spectrograms.shape[1:]

    num_classes = 4

    model = models.Sequential()

    model.add(layers.Conv2D(8, (8, 8), strides=(2, 2), padding='SAME',
    activation='relu', input_shape=input_shape))

    model.add(layers.Flatten())

    model.add(layers.Dropout(0.1))

    model.add(layers.Dense(num_classes, activation = 'softmax'))
```

Execute `model.summary()` on Colab to get the model details, as shown in *figure 9.3*:

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 125, 33, 8)	520
flatten (Flatten)	(None, 33000)	0
dropout (Dropout)	(None, 33000)	0
dense (Dense)	(None, 4)	132004
activation (Activation)	(None, 4)	0
=====		
Total params: 132,524		
Trainable params: 132,524		
Non-trainable params: 0		

Figure 9.3: The CNN architecture for keyword spotting

The model has around 132K parameters. Next, we will configure it for training by defining an Adam optimizer having a learning rate of 0.001. We will train it for 100 epochs to reduce the cross entropy loss. Refer to the following code:

```
>>model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                loss=tf.keras.losses.
                SparseCategoricalCrossentropy(from_logits=False),
                metrics=['accuracy'])

model.fit(train_spectrogram_ds, validation_data=val_spectrogram_ds,
          epochs=100)
```

Since you are connected to a GPU runtime, it will only take very less time to complete the training. After 100 epochs, you can expect a classification accuracy of around 85% on the validation set. Now, evaluate the model on the test set.

```
>>model.evaluate(test_spectrogram_ds)
```

You can expect a classification performance similar to the validation set. Now, save the model in the Colab workspace:

```
>>model.save('model.h5')
```

Now, we have the training model. We will apply **post-training quantization** to reduce the model size and save it into an equivalent TFLite model. Refer to the following code:

```
>>model = tf.keras.models.load_model('model.h5')

converter = tf.lite.TFLiteConverter.from_keras_model(model)

converter.optimizations = [tf.lite.Optimize.DEFAULT]

tflite_quant_model = converter.convert()

with open('model.tflite', 'wb') as f:
    f.write(tflite_quant_model)

tflite_models_dir = pathlib.Path('/content/tflite_models/')
tflite_models_dir.mkdir(exist_ok=True, parents=True)
tflite_model_file = tflite_models_dir/'model.tflite'
```

```
tfLite_model_file.write_bytes(tfLite_quant_model)
```

With this, the TFLite model `tfLite_quant_model.tflite` is stored in your Colab workspace. The model size is around 136 kilobytes, which can fit in the memory of an Arduino Nano. You can readily use it to evaluate on the test set. Use the following code:

```
>>tfLite_model_file = 'tfLite_models/model.tflite'

# Initialize the TFLite interpreter
interpreter = tf.lite.Interpreter(model_path=tfLite_model_file)
interpreter.allocate_tensors()

input_info = interpreter.get_input_details()[0]
input_index = input_info['index']
scale, offset = input_info['quantization']

input_index = interpreter.get_input_details()[0]['index']
output_index = interpreter.get_output_details()[0]['index']

total_count = 0.0
accurate_count = 0.0

for x, y_true in test_spectrogram_ds:
    input_shape = x.shape[1:]

    for count in range(x.shape[0]):
        temp = x[count,:]

        temp = tf.reshape(temp,(1, temp.shape[0], temp.shape[1], temp.
            shape[2]))

        interpreter.set_tensor(input_index, temp)

        interpreter.invoke()

        prediction = interpreter.get_tensor(output_index)[0]
```

```
prediction = np.argmax(prediction)

if prediction == y_true[count].numpy():
    accurate_count += 1

total_count += 1

accuracy = accurate_count/total_count

print('Accuracy : ', accuracy)
```

With this, we finish the first part of the chapter on developing a baseline keyword spotting application in Python. So far, the model has been tested on offline data. In order to deploy it on Arduino to make real-time inferences, you need to convert the TFLite model into an equivalent C++ library specific to the device. In *Chapter 8, TensorFlow Lite for Microcontrollers*, we have already done that using the command `xxd`. In that chapter, we implemented the entire inference program from scratch on the Arduino IDE for deployment. In this chapter, we will use **Edge Impulse**, a specially designed deployment platform for the easy creation of the deployable Arduino program using a simple user interface. It will internally take care of the model optimization part depending on the target hardware where the model will be deployed. We will learn more about all these in the following section.

Do not disconnect the Colab runtime. You will need to use some part of the code covered in this section later in this chapter.

Introduction to Edge Impulse

Implementing an optimized machine learning model on a microcontroller can be non-trivial. In the previous chapter, we learned how to convert a TFLite model into an equivalent library file for Arduino. Of course, the model was trained on Colab. The library file was used in the inference code which was composed on Arduino IDE. Remember, we implemented a fairly simple application in the previous chapter, and we hardly bothered about the memory space and the computing capacity of the microcontroller. However, these are critical aspects of writing programs for microcontrollers. In reality, it might be difficult to determine the most optimum machine learning architecture for a particular application that suits the requirement of the target hardware. We often rely on trial-and-error. Fortunately, there exists an open-source platform called **Edge Impulse**, which helps you to create end-to-end

TinyML applications using a simple graphical user interface. It also suggests you the most optimum model based on your requirement and target hardware device. In this chapter, we will learn about the Edge Impulse platform and use it to implement the keyword spotting project to deploy on the Arduino Nano.

Edge Impulse Studio, commonly known as Edge Impulse, is a leading cloud-based platform for developing embedded edge ML systems that can be deployed on a wide range of hardware platforms. The best part of Edge Impulse is that it provides powerful automation and low-code capabilities to build advanced machine learning applications. As a result, you can build your own application without having an in-depth knowledge of machine learning algorithms. You can create end-to-end practical applications on embedded devices for sensor analytics, audio processing, and computer vision without writing a single line of code. It has inbuilt automation tools to assist in creating highly optimized machine learning models, based on user requirement.

Edge Impulse is free for developers. In order to use it, you need to log into your account. Go to the following URL in your browser: <https://studio.edgeimpulse.com/login>

You will see the following Web page shown in *figure 9.4*:

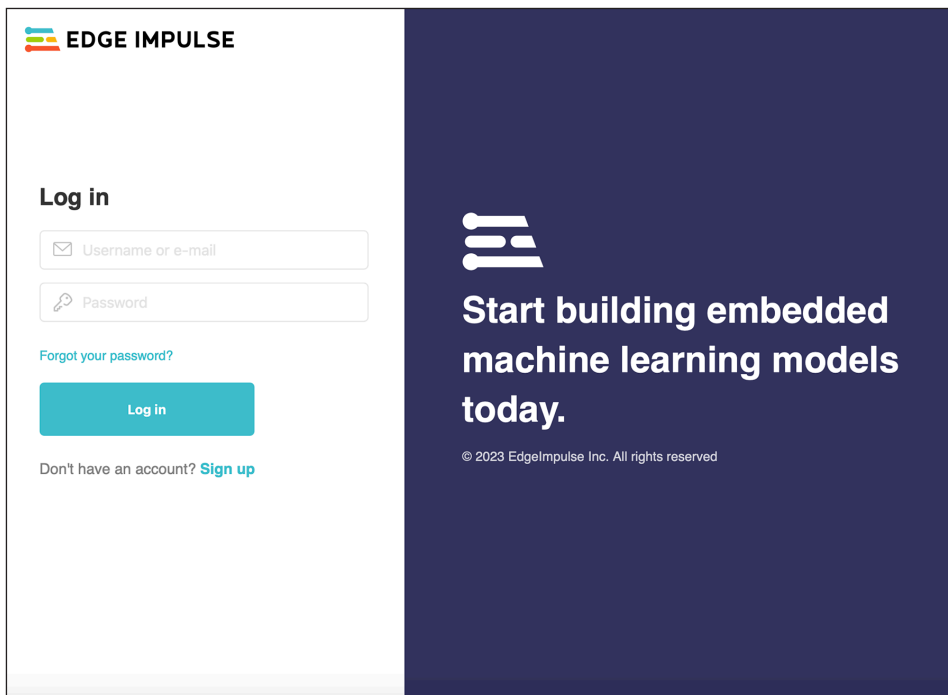


Figure 9.4: Edge Impulse Studio home page

For new users, you need to create an account by clicking on the **Sign up** option. You need to provide your e-mail id and set a username and password to create your account. Now, login to your account with the credentials. You will see a new Web page similar to *figure 9.5*. Note that the screenshots shown in the chapter are indicative. Edge Impulse frequently updates its features. You may find a different version of Edge Impulse with different properties and user interface when you actually implement the project at your end.

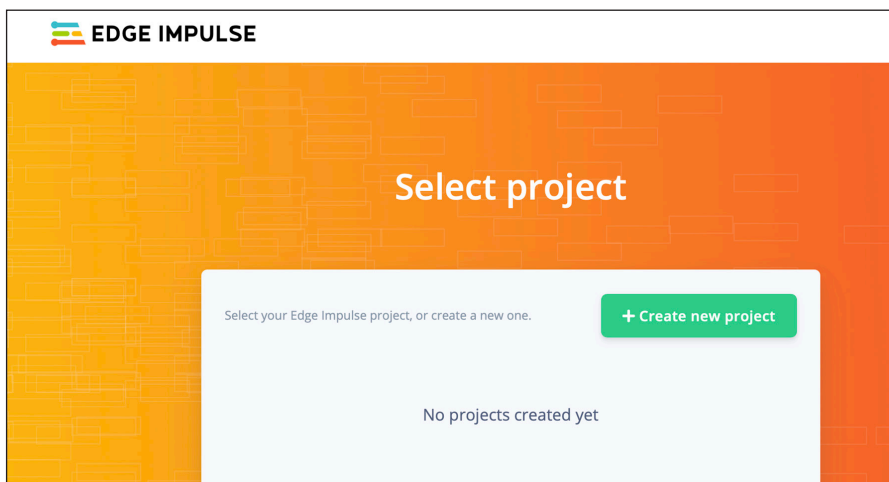


Figure 9.5: Create a project in Edge Impulse

Implementing keyword spotting in Edge Impulse

Before we start implementing the project, let us first understand how the keyword spotting application will be running on Arduino Nano 33 BLE Sense. Like any other TinyML projects, we first need the training model. In all previous chapters, we created our training models in Colab. However, this time the model will be created and optimized on Edge Impulse. You can collect your own data on Edge Impulse. The inbuilt microphone of Arduino Nano can be used for recording the audio data in real-time to train a model. The inference will happen in real-time on every one-second-long audio.

Click on **Create new project**. Give a suitable name to your project and save it. Your browser will open the main project dashboard containing various interfaces. Scroll down to the section named **Creating your first impulse**. We will mostly work here to implement various parts of the project. As shown in *figure 9.6*, the dashboard has the following parts:

- **Acquire data:** Here, you provide the necessary data to train your model. Edge Impulse supports various modes to get the data. You can either record your own data by connecting the necessary sensors to your computer, can use the inbuilt camera and microphone of your computer, or can upload an external dataset.
- **Design an impulse:** Here, you define and implement the machine learning architecture pipeline for your application. We mostly perform the following operation.
 - o First, we add a processing block that performs some kind of **digital signal processing (DSP)** on the data for feature extraction.
 - o Then, we add a machine learning block to complete the pipeline, which can be a classifier model or a regressor.
 - o Train and evaluate the model.
 - o Optimize the model based on the target hardware device
- **Deploy:** Once the model is trained, you can package your model for deploying in the target hardware. Edge Impulse supports a number of target hardware, including smartphones and a wide range of commercially available microcontrollers for which you can convert your application as a deployable package and deploy.

Refer to *figure 9.6*:

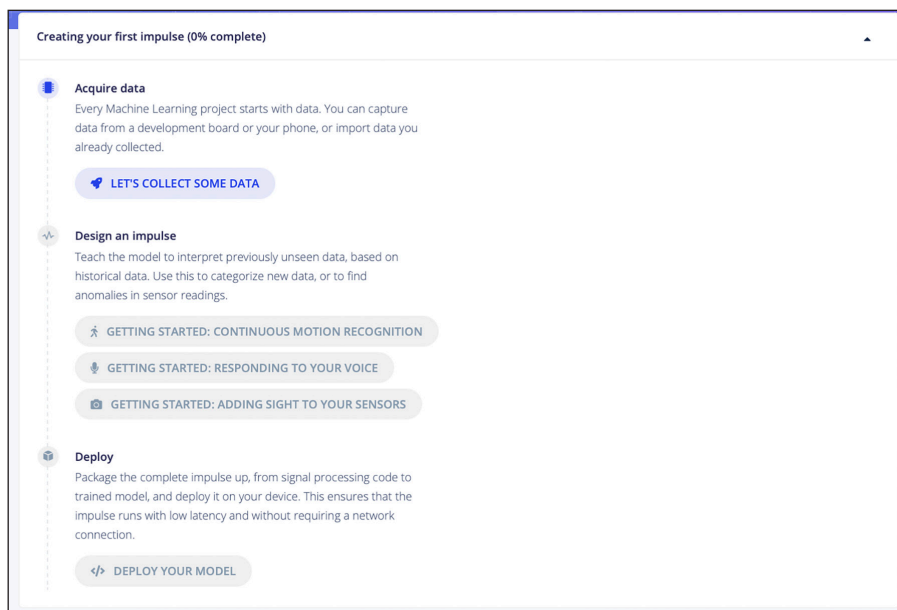


Figure 9.6: Creating your project from the Edge Impulse dashboard

Now, let us implement the project in a step-by-step manner. As expected, we will start by getting the data. Go to the project dashboard and Click on **LET'S COLLECT SOME DATA** under the section Acquire data. It will open the following window, shown in *figure 9.7*.

Edge Impulse supports various ways to record your data. It also supports a number of hardware platforms in the form of microcontroller-based development boards. Many of them come with various inbuilt sensors for real-time data recording. You can connect the supported development boards to your computer where Edge Impulse is running and configure them for real-time data collection for your project. For example, you can configure your Arduino Nano 33 to record the audio data using the inbuilt microphone to train your impulse. Similarly, you can also use your smartphone or laptop to record your voice. Finally, you can also upload external pre-recorded data for your impulse.

As per our problem statement, we need to upload labeled audio samples for all four target classes, “on”, “off”, “others”, and “silent”. For simplicity, we will primarily rely on the same Speech Commands dataset that we used in the previous section to create our baseline keyword spotting model. We will use part of that dataset to get the audio data corresponding to “on”, “off”, and “others”. Just to get a feel of how to capture real-time data on Edge Impulse, we will record the silent audio data in real-time and upload it in our training set.

Refer to *figure 9.7*:

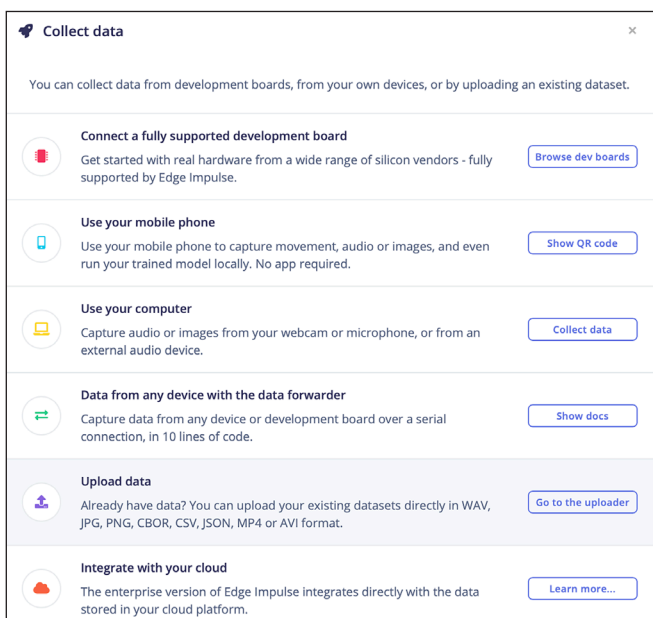


Figure 9.7: Data collection in Edge Impulse

Let us now upload the audio files corresponding to “on”, “off”, and “others”. Go back to your Colab workspace, where we implemented the baseline keyword spotting model. We will write a program on Colab that performs the following steps to create a small representative dataset to train the keyword spotting model on Edge Impulse:

1. Create another directory **mydataset** under the parent directory **data** in your Colab workspace where the original data has been stored.
2. Create three subdirectories “on”, “off”, and “others” under **mydataset**.
3. Randomly select 600 samples from each target label of the original dataset and copy them inside **mydataset** directory. We cannot select all the files from the original dataset due to restrictions in file upload size in Edge Impulse.
4. Finally, compress the entire file structure into a zipped file.

The code snippet to execute on the Colab workspace is as follows:

```
>>import shutil

os.mkdir('data/dataset')
os.mkdir('data/dataset/on')
os.mkdir('data/dataset/off')
os.mkdir('data/dataset/silence')
os.mkdir('data/dataset/others')

target_labels = ['on', 'off', 'others']
sample_per_label = 600
for labels in target_labels:
    path = 'data/mydataset/'+labels
    f = os.listdir(path)
    num_files = len(f)
    file_indx = np.arange(num_files)
    random.shuffle(file_indx)
    for i in range(sample_per_label):
        indx = file_indx[i]
        file_name = f[indx]
```

```

source = path + '/' + file_name
destination = 'data/dataset/'+labels+'/' + file_name
# copy only files
if os.path.isfile(source):
    shutil.copy(source, destination)

```

Execute the following command to save the newly created dataset as a compressed ZIP file.

```
>> !zip -r dataset.zip data/dataset/
```

Once the compressed file **dataset.zip** is created and stored in the Colab workspace, download it from Colab to your host computer and unzip the content. Now, go back to the data collection window in your Edge Impulse dashboard. Click on **Go to the uploader** under **Upload data** tab (refer to the previous *figure 9.7*). It will open a new Web page. Select Upload data. It will open the following window, shown in *figure 9.8*:

Figure 9.8: Uploading external data in edge impulse

Click on **Choose Files**. Now, go to the downloaded and unzipped folder containing your training data, and select all the audio files from the directory “off” (press *Ctrl + A*). Select the radio button **Automatically split between training and testing**. Finally, select **Enter label** under **Label** and write “off” inside the text box to set the label (refer to *figure 9.8*). It will mark all the uploaded files as off. Now, click on **Begin Upload** at the bottom. It will upload all the selected files in your Impulse. Repeat the same procedure to upload all the audio files corresponding to “on” and “others”. Make sure you label them properly before uploading.

Finally, we need to upload some silent audio clips. As mentioned earlier, we will record the silent audio on the host computer itself using the inbuilt computer microphone. Go back and click on the **Data acquisition** tab on your Edge Impulse dashboard. Click on **Show options**. It will open the data collection wizard shown in *figure 9.7*.

Make sure your computer has a microphone to record audio. Go to **Select your computer** and click on **Collect data**. Click on the tab **Collecting Audio**. It will ask for your permission for accessing the computer microphone. On granting that, you will find a new window. It will ask for certain information about your recording. Set the label for the recording as silent duration as 60 seconds. Leave the other parameters unchanged. It will split the recording randomly in 80:20 ratio for training and testing. Now, click on Start recording to record a 60 seconds long silent audio using your computer microphone. While recording, make sure there is no background noise. Once the recording is completed, the audio file will be automatically stored in your workspace. We need to perform one more step to break the 60-second-long silent audio clip into multiple one-second-long audio clips as per the required input data length for our application.

Go back to the dashboard again and click on **Data acquisition**. It will show all the audio clips uploaded and recorded for the project. Click on the three parallel dots next to the silent audio that you recorded. A new list of options will appear. Click on the split sample. Another new window will appear. Select **Set segment length (ms)**: as 1,000 and click on **Split** at the bottom of the window. It will break the large clips into one-second-long recordings.

Now, we have all the necessary data to build the keyword-spotting model. Go back to the Edge Impulse dashboard and click on **Impulse design**. Next, select **Create impulse**. It will open the following Web page shown in *figure 9.9*:

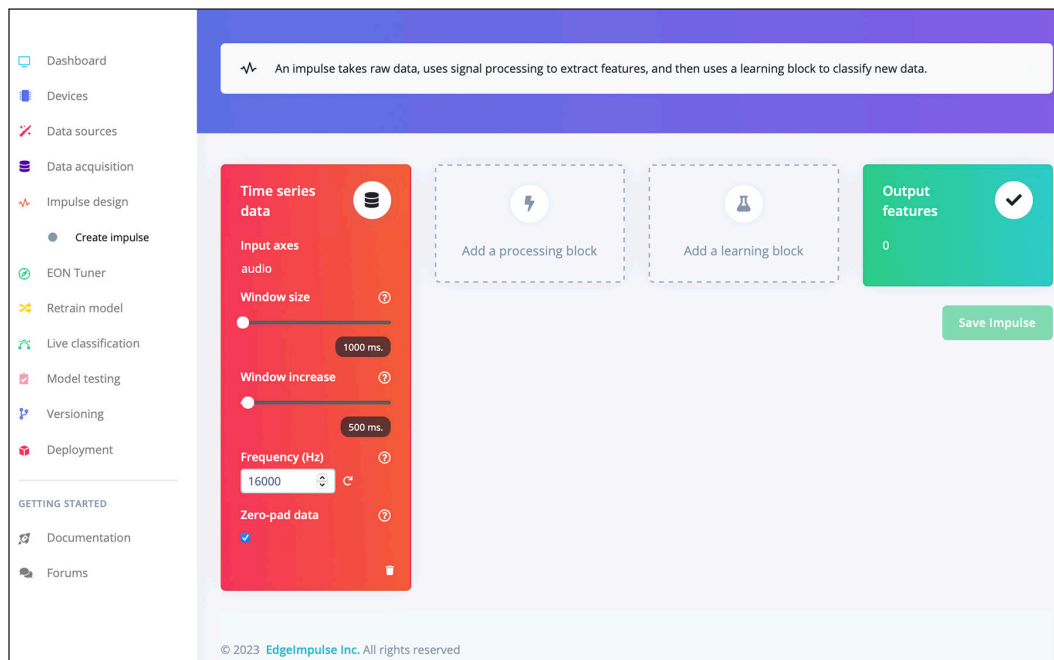


Figure 9.9: Creating an Impulse on audio data

It shows the type of data you uploaded for the project (that is, audio time-series data), the sampling frequency of the recordings, and the window length for making a prediction. You need to add two blocks to your data to create the impulse. First, you need to add a **processing block** to perform the necessary DSP operations for feature extraction. In our case, the feature extraction block can be a block to convert the time-series data into the equivalent spectrogram. Next, you need to add a **learning block**, which will be the neural network module, to perform the classification task.

Let us first add a processing block to our impulse. In our baseline keyword spotting model, we converted the raw audio files into a spectrogram, which provides a visual representation of the audio clip in the time-frequency domain. The spectrogram format is more suitable to apply to a CNN architecture for effective feature extraction. The good news is that Edge Impulse includes a number of processing operations that you can directly use in your impulse. Instead of the spectrogram, we will apply a different processing operation the **Mel Frequency Cepstral Coefficient (MFCC)**, which is particularly suitable for speech processing applications.

Click on **Add a processing block** and select Audio (MFCC) from the list. MFCC is an audio feature extraction algorithm commonly used in speech processing and speaker recognition applications. In MFCC, the frequency bands of the audio are mapped on the Mel scale, which closely approximates the human auditory system.

As a result, MFCC is considered as a better feature extraction technique compared to the spectrogram-based approach in speech processing applications. The MFCC components are calculated through the following steps:

1. Break the audio signals into small windows.
2. Calculate the spectrum of the windows via Short-Time Fourier Transform.
3. Map the spectrum power into equivalent Mel scale.
4. Calculate the logarithm of the power components at the Mel frequencies.
5. Calculate the discrete cosine transform of the logarithm powers.
6. The resulting amplitudes represent the MFCC values.

After adding the processing block for MFCC operation to the impulse, add a learning block to complete the machine learning pipeline. Click on **Add a learning block** and select **Classification**. Based on your data labels, it will automatically show the output features for your classification model. The entire pipeline is shown in *figure 9.10*:

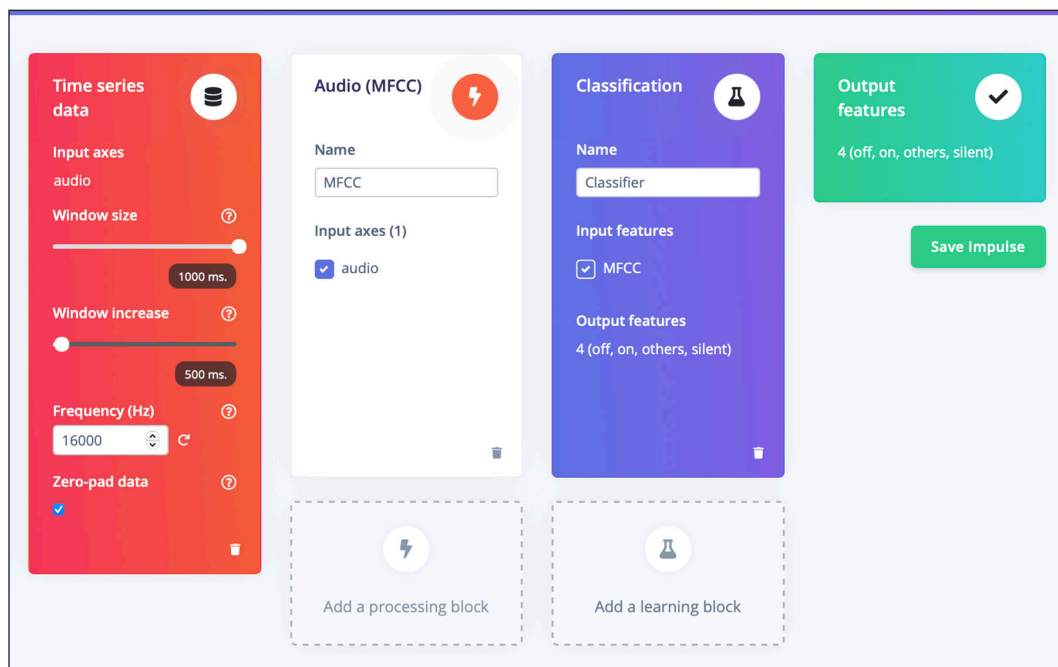


Figure 9.10: The classification pipeline on Edge Impulse

Do not forget to save the impulse. Go back to Impulse Design in the dashboard and click on MFCC just below **Create impulse**. It will show all the input parameters for feature extraction. We will use the default values of different parameters for

feature computation and will compute 13 MFCC parameters as the output of the processing block. Click on **Save parameters**. It will open a new Web page for feature computation. Click on **Generate features**. It will compute the MFCC feature on the dataset you uploaded for your application.

Next, click on **Classifier** under **Impulse Design**. Here, we will define the neural network architecture to make classification. This module gives a detailed user interface where you can add and modify different layers of a neural network architecture, such as the convolutional layer, pooling layer, dense layer, and so on, to define your own customized neural network architecture. You can also modify the parameters of different layers to achieve the optimum classification performance. Different network hyperparameters, such as the number of training cycles to train the model, learning rate, percentage of data in training set to be used for validation, and so on, can also be modified.

Now, let us define a model in Edge Impulse. For simplicity, we will select the default **1D convolutional** architecture along with the default parameters to quickly build our model. You may require to tweak the parameters to get a better classification performance. Finally, we will select the target hardware platform for the deployment of the application. Go to the top right of your Edge Impulse Web page and select the target device as **Arduino Nano 33 BLE Sense**. Now, we will create the training model. Click on **Start training**. Once the training is done, it will show the classification performance on the validation set as obtained by both the base (float32) model and the corresponding quantized (int8) model. You can play with various hyperparameters of the network and try other network architectures at your end for a better performance.

Now, we have an optimized model for the target device. Before deploying the model, we must check whether the generated model is optimized enough to run efficiently on the target device. For this, we will use the **EON Tuner** tool, readily available in Edge Impulse. EON Tuner is an automation tool that provides end-to-end optimization of a machine learning pipeline, starting from the signal processing block used for feature extraction to the actual neural network architecture for classification. It helps you select the most optimized machine learning algorithm for your application, depending upon the resource of your target hardware device. The biggest advantage of using EON Tuner is that it internally checks various combinations of signal processing algorithms and classifier architectures to automatically provide you with a suitable model for your application within a considerably shorter period of time that would otherwise take several days to build manually. Moreover, one can easily create a machine learning model without much domain expertise, which is particularly important for application developers.

Click on **EON Tuner** from your dashboard. Click on **Configure target**. Select Arduino Nano 33 BLE Sense as the target device. Finally, set the time per inference as 100 ms, which indicates the desired processing time to make a decision on a one-second-long audio recording. Click on **Save**. Now, click on **Start Eon Tuner**. It will try different combinations of feature extractors and neural network architectures and show the classification performance. It will also show the average inference time and the required memory to run it on the target device.

Once EON Tuner finishes its job, you will see a detailed performance report similar to what is shown in *figure 9.11*. It visually represents how different combinations of feature extractor (**DSP**) and neural network classifier architectures (**NN**) have performed on the data. It also shows the latency and memory used by the DSP and NN block for computation. The visual representation helps you to select the most optimized model in terms of classification accuracy, latency, and memory usage. Depending upon your requirement, you can select any of these models for deployment.

Refer to *figure 9.11*:

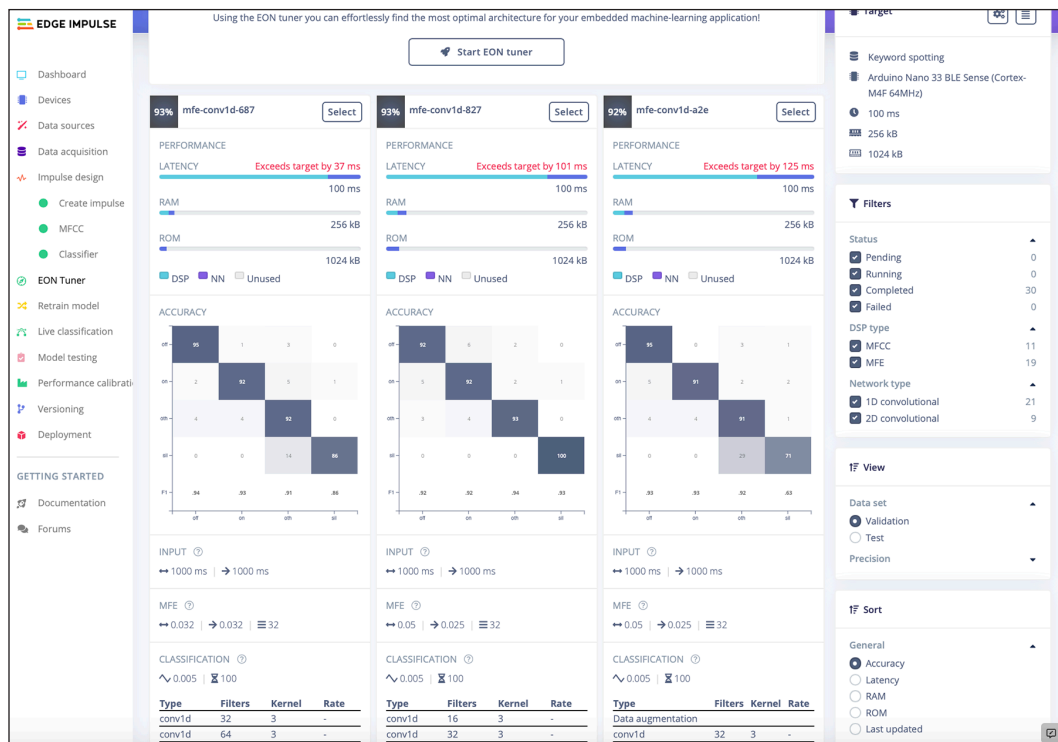
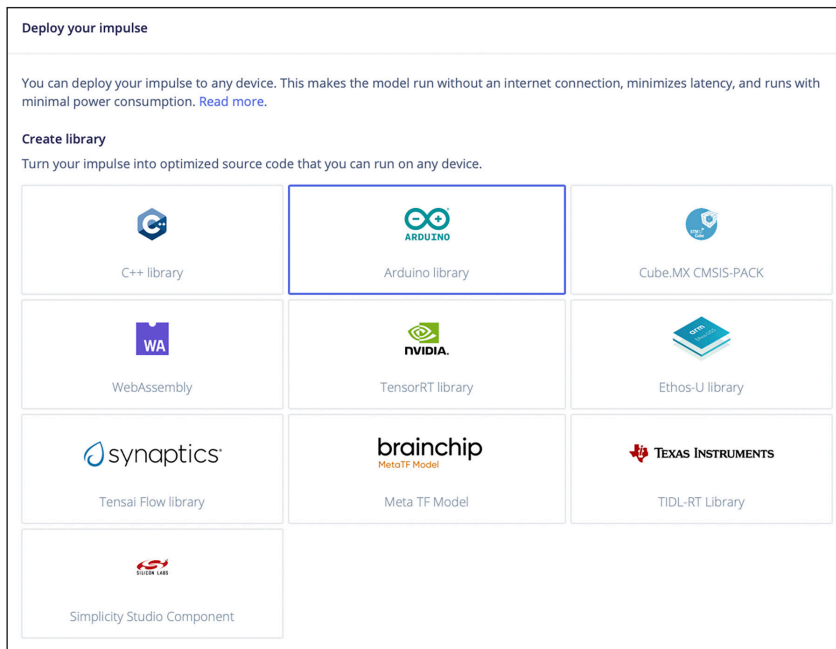


Figure 9.11: Models suggested by EON Tuner

We will select the first models from the list provided by EON Tuner, shown in *figure 9.11*. The selected model uses MFE as the processing block. Similar to MFCC, the **Mel Frequency Energy (MFE)** features are also computed from the spectrogram of the audio signal using Mel filter bank energy. Once the features are computed, the selected pipeline uses a 1D CNN model for classification. As shown in *figure 9.11*, it reports a classification accuracy of 93% with an average latency of 137 ms. Click on **Select** to select the configuration from the list to update the primary blocks of your impulse. Now go to **Retrain model** from the dashboard to retrain using the selected model architecture.

Next, go to **Model testing**. Here you can test your model performance. Click on **Classify All** to test your model on the entire test set. Additionally, you can also record live audio using your computer and test on them.

Finally, go to the **Deployment** tab. Edge Impulse allows you to deploy your model on a wide variety of hardware platforms, starting from smartphones to tiny embedded devices and microcontrollers. You can export your model as a library or can build it as a firmware. We will export the model as an Arduino library. Select **Arduino library** from the list under **Create library** tab. Make sure that you have checked **Enable EON Compiler** and selected a quantized (int8) version of the model that creates an optimized smaller version of the library. Refer to *figure 9.12*:



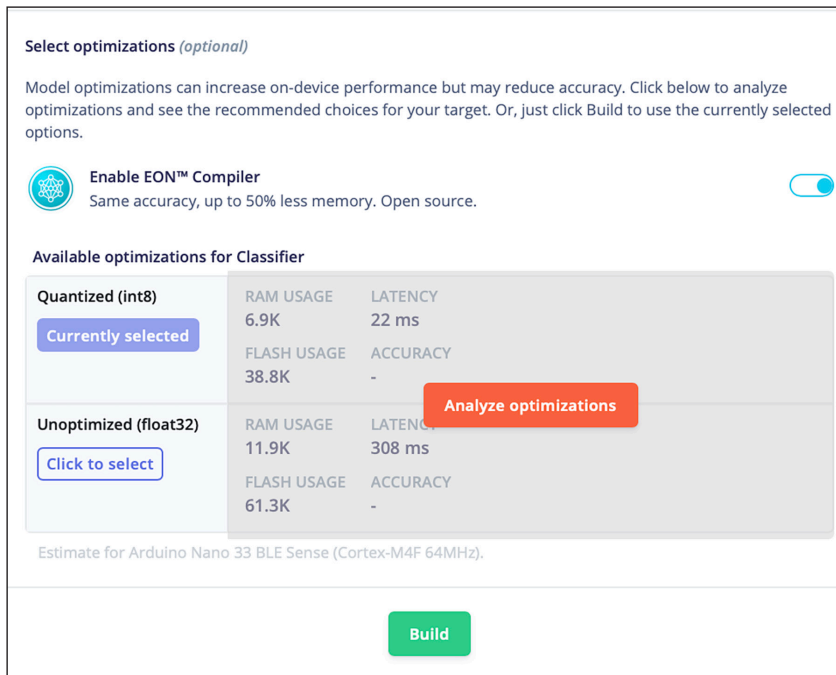


Figure 9.12: Model deployment on Edge Impulse

Click on **Analyze optimizations** to compare the performance between the optimized (int8) and the unoptimized (float32) model. Finally, click on **Build**. The library will be created and downloaded to your computer in a compressed ZIP format. We will use this library to create our keyword spotting application on Arduino Nano.

Model deployment

We are finally ready to deploy the keyword spotting application on Arduino. Connect the Arduino Nano device to your computer using the USB cable. Open the Arduino IDE. Make sure you have selected the correct board and the port to locate the Arduino device. Refer to the previous chapter for more details regarding how to connect the Nano.

Now, on your IDE, go to **Sketch** → **Include Library** → **Add .ZIP Library** and select the compressed library you created. Next, go to **File** → **Examples** and scroll down the list. You will find the **keyword spotting** project in the list. Open the project. It contains one example code for various supported microcontrollers that you can readily use. Open the sketch **nano_ble33_sense** → **nano_ble33_sense_microphone** to get the relevant code for Arduino Nano.

Look at the code carefully to understand what operations are performed. An Arduino program must have two key functions, **setup()** and **loop()**. The initialization tasks are done inside **setup()**. The following steps are done inside the **loop()** function:

1. Initially, the program pauses for 2 seconds.
2. Then it records the live audio stream for one second using the inbuilt microphone.
3. Applies the keyword spotting algorithm on the recorded data for prediction. It prints the prediction probability of the recorded audio falling in the four target classes. Higher the predicted probability, better is the chance the input data belongs to that class.
4. Repeat Steps 1–3.

Compile and upload the program on Arduino IDE. It will take some time to compile the program for the first time. Once uploaded, go to **Tools** → **Serial Monitor**. Bring the Arduino device close to your mouth. Keep a look at the instructions shown in the Serial Monitor. Say “on”, “off”, or some other keywords as soon as the Serial Monitor indicates that audio is getting recorded. It will record your voice and perform the feature extraction and classification to show the prediction probabilities for different target class labels. It will also show the time required to perform the processing and the classification tasks.

Refer to *figure 9.13*:

```

/dev/cu.usbmodem1201
Recording...
Recording done
Predictions (DSP: 84 ms., Classification: 21 ms., Anomaly: 0 ms.):
off: 0.01953
on: 0.76953
others: 0.21094
silent: 0.00000
Starting inferencing in 2 seconds...
  
```

Figure 9.13: Serial Plotter output of keyword spotting application

With these, we have successfully deployed the keyword spotting application on Arduino. You can edit the program and write your application logic to perform certain real applications, for example, turning on or turning off an LED based on your voice command.

When you deploy the model on Arduino, you might experience a poor classification performance. Remember, you have used a publicly available dataset to train your model, not your own recorded voice. Introducing your own voice in the training set may improve the classification performance. Secondly, you are using the inbuilt microphone of your Arduino Nano for recording of your voice to make inference, whereas the model is trained on a publicly available dataset where you do not have any clue on what types of microphones were used in recording. Different microphones have different frequency responses. A disparity in the quality of the recording device between the training and the test set can have a significant impact on overall classification performance. Tuning the network hyperparameters and adding your own sample voice to the training dataset can improve the model performance. You can also select and try other neural network architectures suggested by the EON Tuner and check their performance upon deployment.

Conclusion

In this penultimate chapter of the book, we have implemented our final TinyML project of real-time keyword recognition on an Arduino Nano 33 BLE Sense. Keyword spotting is an important application which is widely used as the backbone of modern voice-assisting devices such as Alexa-enabled Amazon Echo, Google Home, or Apple's Siri. The key objective of this chapter was to understand the different steps involved in implementing a real-world keyword spotting application from scratch. We have learnt two key things in the chapter. First, we have implemented a baseline keyword spotting algorithm in Colab using a CNN and evaluated it on offline data. Later in this chapter, we have learnt to create a deployable library for Arduino using Edge Impulse, a free development platform for creating ready-to-use machine learning models for embedded devices and microcontrollers. Remember, Edge Impulse frequently updates its features. The screenshots and functionalities shown in this chapter are specific to one version of Edge Impulse, which might be different in other versions. However, the readers can get used to a different version of Edge Impulse with a minimum effort.

We have also explored the EON Tuner, an automation tool that suggests the best possible machine learning model based on the constraints of the target platform. Unlike image data, audio data may not produce the desired performance if they are directly applied to a CNN architecture. Hence, you may require to apply some exhaustive processing steps on audio before applying to the neural network. Spectrogram and MFCC-based processing approaches are popularly used in audio and speech processing applications.

The classifier performance heavily depends upon the quality of the audio data and the type of sensor used to record the data. Ideally speaking, the training and the test data should be recorded using the same kind of device, as different microphones have different frequency response. You are strongly recommended to train the model on your own voice and check the classification performance. With this, we finish implementing the final project of this book. In the final chapter, we will briefly discuss some recent trends and tools of modern TinyML techniques.

Key facts

- Keyword spotting is a lightweight speech recognition task running at the device frontend of smart voice-assisted devices.
- Apart from being accurate, keyword spotting algorithms must be real-time and extremely lightweight.
- The three major parts of keyword spotting are getting the audio, processing, and classification.
- Some popular audio processing techniques for feature extraction are Spectrogram, MFCC, and MFE.
- Edge Impulse is a free tool for the easy creation of embedded machine learning applications.

Join our book's Discord space

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

<https://discord.bpbonline.com>



CHAPTER 10

Conclusion and Further Reading

Introduction

We have reached the final chapter of this book. The primary goal of the book is to offer a concise introduction to TinyML and its various aspects, enabling readers to develop straightforward and engaging TinyML projects for everyday use. Due to the swift expansion of digital data production, TinyML has become a rapidly evolving area within machine learning. The power of modern neural-network, particularly deep learning techniques, has revolutionized the AI industry. However, neural networks are computationally expensive and extremely resource hungry. A typical neural network may have thousands or even few millions of trainable parameters. Such large neural networks are primarily designed to operate on remote servers, transmitting data via the internet.

We are living in a digital world. We generate a massive amount of digital data daily, and it is not practical to transmit all of it to cloud-based servers for processing. As a result, most of the data goes unused. In TinyML, large neural networks are optimized to operate efficiently on small edge devices and microcontrollers, which have limited computing resources. In general, an optimized neural network runs faster and consumes lower battery power but often comes with reduced accuracy. While optimizing a neural network, one needs to maintain a trade-off between

inference latency and accuracy. The positive aspect is such devices are extremely low-powered, typically in milliwatt range. As a result, they can seamlessly operate 24×7 for several weeks or even years without replacing the battery. Thanks to advancements in TinyML, it is now feasible to implement on-device intelligence on billions of smart devices we use in our daily lives without relying on a cloud-based platform that would consume considerable network bandwidth to transmit user data for processing and to receive processed results. Some of the appealing benefits of TinyML include lower inference latency, better energy efficiency, reduced internet bandwidth, and cost-effectiveness. On top of that, it ensures to preserve the privacy of user data as the entire processing takes place offline without sending the data to a remote server. Implementation of TinyML applications can already be seen in large industries for machine health monitoring, agriculture, continuous patient monitoring, retail, and energy domain. We also encounter numerous TinyML applications in our daily lives, such as in our smartphones, cars and large electronic appliances like refrigerators and washing machines, which are equipped with powerful microcontrollers for edge analytics. The AI/ML market size was at USD 15.44 billion in 2021 and is expected to grow from USD 21.17 billion in 2022 to USD 209.91 billion by 2029. It can be safely assumed that the TinyML market will significantly grow in the upcoming years.

Structure

In this chapter, we will discuss the following topics:

- Brief learning summary
- TinyML best practices
- AutoML and TinyML
- Edge ML on smartphones
- Future of TinyML

Objectives

In this final chapter of the book, we will briefly recapitulate what we have learnt so far in the previous chapters of the book. We will talk about some of the best practices one should follow while developing a TinyML application. We will also briefly talk about some of the AutoML platforms that have made the development of TinyML applications much easier. We will finish the chapter by briefly discussing about the future of TinyML.

Brief learning summary

The key objective of the book was to give the readers a brief overview of TinyML as a technology and to help them getting familiar with implementing TinyML applications from scratch through practical projects. Rather than explaining every single mathematical derivation of various machine learning algorithms, we have focused more on the programming part so that the readers can get the hands-on experience to create their own applications. In *Chapter 1, Introduction to TinyML and Its Applications*, we introduced TinyML as a technology and its applications to the readers. *Chapter 2, Crash Course on Python and TensorFlow Basics*, was a crash course on Python, which is the globally accepted programming language, to implement neural network models. We learned to use Google Colab to execute Python programs connecting a Python runtime via the cloud. In the beginning of the chapter, we talked about popular Python libraries such as NumPy, Matplotlib, and so on, which are extensively used in implementing machine learning and data science projects. Later in the chapter, we discussed the basics of TensorFlow, an open-source software library for quick implementation of machine learning and neural networks.

In *Chapter 3, Gearing with Deep Learning*, we briefly discussed the key aspects of neural networks with a focus on **Convolutional Neural Network (CNN)**. CNN is a powerful neural network architecture popularly used in image processing and computer vision applications. CNNs comprise a series of convolutional filters that can automatically extract the relevant features from the input data using a set kernel with trainable weights. Raw image data can directly be applied to a CNN without much processing for effective feature extraction to make a classification. We implemented our first CNN in *Chapter 4, Experiencing TensorFlow*, for the classification of handwritten digit images. The model was implemented in TensorFlow using Keras. Keras is a Python interface of TensorFlow, popularly used for the quick implementation of neural networks. The model was trained and evaluated on the publicly available MNIST database.

In *Chapter 5, Model Optimization Using TensorFlow*, we discussed about various optimization techniques to reduce the size of neural networks so that they can effectively run on smaller edge devices. We talked about *quantization-aware training* and *post-training quantization*, which converts a floating point neural network model into an integer-based model, therefore, causing a $4 \times$ model size reduction. We also talked about other optimization techniques, such as *weight pruning* and *weight clustering*, which reduce the number of parameters in a neural network. We introduced the **TensorFlow Optimization Toolkit** to the readers, a specially designed TensorFlow library for easy implementation of various optimization techniques in Python. We also learned to convert a TensorFlow model into the equivalent TensorFlow Lite

model, which is much smaller, faster, and computationally less expensive. Such models can be efficiently deployed on small edge devices and mobile platforms, including Android and iOS devices.

In *Chapter 6, Deploying My First TinyML Application*, we created our first TinyML project from scratch and deployed it on a real edge device. In this project, we implemented a CNN classifier for object detection. We used the pretrained weights of the MobileNet architecture and optimized it using TensorFlow Optimization Toolkit. The model was evaluated on the publicly available CIFAR-10 dataset for image classification. The TensorFlow model was converted into an equivalent TensorFlow Lite (TFLite) model, which was deployed on a Raspberry Pi to make on-device inferences on offline images. Raspberry Pi is a Linux-based single board computer popularly used in AI and IoT applications. It also has a Python interpreter; hence, the inference program can be written in Python.

In *Chapter 7, Deep dive into Application Deployment*, we created a more practical application of real-time face recognition on a Raspberry Pi. We used popular open-source libraries to implement the majority of the image processing tasks and focused on implementing the end-to-end application pipeline. The application captures live video streams from a camera connected to a Raspberry Pi and runs on-device edge analytics for face recognition. The model was purposefully trained on the Pi itself so that it can be easily retrained for adding or deleting persons for detection.

In *Chapter 8, TensorFlow Lite for Microcontrollers*, we created our first TinyML application for microcontrollers. Microcontrollers are more resource-constrained compared to single board computers. They do not have any operating system. We used Arduino Nano 33 BLE Sense, a microcontroller-based development board, to deploy our Project. Arduino has its own programming language, which is similar to C/C++. There is a dedicated **Integrated Development Environment (IDE)** for Arduino which needs to be installed on the host computer to write and compile the programs in order to deploy on an Arduino device to execute. In this chapter, we implement a simple neural network application that modulates the intensity of an LED according to a sinusoid wave. TensorFlow comes with TensorFlow Lite for Microcontrollers, a highly optimized machine learning library to implement simple neural network models on microcontrollers. We trained the TensorFlow model on Colab and converted it into the equivalent TFLite model, which was then further converted into an equivalent C/C++ library for Arduino. On the Arduino IDE, we implemented the inference program using TensorFlow Lite for Microcontrollers APIs and deployed the program on Arduino Nano for real-time inference.

Finally, in *Chapter 9, Keyword Spotting on Microcontrollers*, we implemented a speech recognition application on Arduino Nano. To be specific, in that project,

we implemented an on-device keyword spotting application. Our learning in this chapter can be divided into two parts. Initially, we implemented a simple CNN-based keyword recognition system on Colab using TensorFlow to familiarize with the speech recognition application. Later, we created an end-to-end keyword detector model and deployed it on Arduino for real-time speech detection. We learned to use the **Edge Impulse** platform, which is a popular tool for quickly building complex TinyML applications on a browser. It even helps in optimizing large neural network models depending upon the resource of the target hardware with a minimum effort in code writing. One can even deploy the program created on Edge Impulse as firmware to the target microcontroller.

TinyML best practices

TinyML focuses on implementing AI and machine learning on compact edge devices. Modern machine learning, particularly deep neural network architectures, leads to sizable training models that are not only computationally demanding but also necessitate substantial storage space. In TinyML, the goal is to compress the model size to enable execution on low-powered, resource-limited edge devices. In doing so, it is essential to minimize performance degradation compared to the original model. As a result, the emphasis is on optimization rather than mere compression of a model.

Throughout this book, we have learned to create optimized TinyML applications using practical examples. However, one should pay special attention to the following points while developing a new TinyML project:

- **Know your target hardware:** Choosing the right hardware where the application will be deployed is an important factor in any TinyML application. The popular choice of target platforms includes smartphones, Raspberry Pi, Arduino Nano, ESP32, and so on. Different hardware platforms have their own pros and cons. The application developers have to judiciously decide the optimum target platform for their applications. You must have a clear idea of how much memory your application might require to make an execution, how much power it could consume, what sensors and interfaces might be required to get the data, whether you need on-device training, the approximate price range of your application at the beginning, and so on before deciding the appropriate hardware.
- **Start with a simple network architecture:** The modern TinyML applications mostly rely on neural networks such as CNN as a machine learning algorithm. CNNs have demonstrated their effectiveness in image processing and computer vision applications. Although deeper CNN architectures

are typically favored for more intricate feature extraction, simpler neural networks can also provide impressive results. We experienced in *Chapter 9, Keyword Spotting on Microcontrollers*, how a shallow CNN model performed reasonably well in keyword detection. Smaller neural networks have fewer parameters, and hence, consume very less device memory to store the model. The smaller models are inevitably faster and consume low power. Such neural networks are less prone to overfit. In the TinyML application, it is recommended to begin with a fairly simple neural network and gradually go into deeper networks through trial-and-error. Using smaller kernel dimension in the convolutional layers of a CNN also ensures lower model size and fewer mathematical operations.

- **Use a combination of optimization techniques:** We learned about various optimization techniques in *Chapter 5, Model Optimization Using TensorFlow*, to efficiently compress a neural network. Although different optimization techniques have their own advantages and limitations, a combination of various optimization techniques can often result in a better performance. In the same chapter, we implemented a collaborative optimization pipeline. We first pruned the baseline CNN by adding sparsity to it. Next, we added weight clustering to the pruned model, ensuring that the sparsity applied in the previous step was preserved. Finally, we applied post-training quantization and converted the model into a full integer. The resulting model was much smaller with a minimum performance drop, compared to the baseline model.
- **Maintain a trade-off:** In most of the TinyML applications, we first create a baseline model and then optimize it into a smaller model to deploy on the target platform. While optimization, we need to maintain a trade-off between model size and accuracy. Usually, a smaller model runs faster and consumes lesser power, but that comes with a drop in model accuracy. While optimization, you need to maintain a trade-off between model size and accuracy depending upon your target hardware.
- **Writing an optimized inference program:** In most TinyML applications, we train our model on the cloud and make inferences on the target hardware. On microcontrollers, the inference program is written in a low-level programming language similar to C/C++. Since microcontrollers are severely resource constrained, one must be very careful in writing the inference program. You must have a rough idea of how much memory is required to store the machine learning model. Apart from that, you should leave sufficient memory space to store the input data, intermediate variables, and output. Rather than declaring too many variables in your program, try

to reuse them as much as possible. Recall in *Chapter 8, TensorFlow Lite for Microcontrollers*, while implementing the inference application for Arduino using TensorFlow Lite for Microcontrollers library, we had to provide the **Tensor Arena**, a pre-defined memory space to allocate memory for the input–output and to run the interpreter. Defining memory is a critical step in embedded system programming. While there is no hard and fast rule in defining the memory space, a clear idea of the potential model size, the operations used in the model, and the input–output dimension may help in deciding.

AutoML and TinyML

Throughout this book, we have developed a number of machine learning models to solve practical problems. There is no single machine learning architecture that works well on all kinds of problems. Rather, it very much depends upon the type of application, type of your input data, what you exactly want to perform (classification or regression), and many other factors. In order to develop a good machine learning model for an application, the developer should have a good knowledge of the theories of machine learning. On top, it may also require some application-specific domain knowledge. While developing a machine learning application, one has to put major effort into deciding the model architecture that optically addresses the problem. Usually, there is no hard and fast rule in finding the best architecture for an application. We mostly rely on empirical techniques, cross-validation, and internal trial-and-error methods. You also need to tune different hyperparameters specific to the model. All these, make the whole process time-consuming. One may ask, can we automate the whole process?

In recent times, there has been plenty of attention on **Automated Machine Learning (AutoML)** techniques that enable you to quickly develop high-quality machine learning models specific to your business needs without having much expertise in relevant domains. In general, AutoML frameworks are highly optimized software suites where you give your data and, if possible, the labels as inputs. The platform runs lots of internal analytics and various combinations of models to come up with the most optimum machine learning model for you. The key advantage of AutoML is that it gives an optimum model based on your application within a short period of time and does not necessitate you to have much expertise in domain knowledge, therefore, saving significant human effort. Once the data is received, an AutoML platform performs the following tasks:

- Pre-processing of raw data, which might involve resizing images, converting color images to grayscale images, and so on.

- Data processing and feature engineering, which might involve different types of signal and image processing operations such as converting a time-series data to a spectrogram, applying different filters to an image, and so on.
- Selection of the optimum machine learning model through trial-and-error.
- Tuning of model hyperparameters.
- Selection of the evaluation matrix.
- Converting into a deployable model as per requirement.
- Result analysis.

AutoML can be very useful in creating TinyML applications. They are particularly helpful in automatically selecting the optimum model for you depending upon the resource of the target hardware. In *Chapter 9, Keyword Spotting on Microcontrollers*, we used the **Edge Impulse** platform to create the keyword detection application on Arduino Nano. We used The **EON Tuner** to find the most suitable model for the target device. The EON Tuner is a perfect example of an AutoML platform for TinyML applications. It decomposes your application into a signal processing block and a neural network architecture and analyses the input data. It suggests a suitable signal processing block along with the optimum neural network model for your application. It tries different combinations of signal processing block and neural network architectures internally from a list of available architectures to summarize the result on the user data. While summarizing, it takes care of the target platform where the model is to be deployed and reports the overall classification accuracy, expected latency, and the overall memory consumed by the signal processing block and the neural network model during the operation. The user can consider all these parameters and compare between different models to eventually select the optimum solution for the application.

With Edge Impulse and Eon Tuner, you can practically deploy a highly optimized deployable TinyML application on the desired target platform without writing a single line of code. Moreover, the user does not require much expertise in machine learning to create their applications. You can use the user interface of Edge Impulse for easy development of your application. Recall, in the previous chapter, how quickly you were able to find the optimum keyword-spotting solution for Arduino Nano using the EON Tuner that would otherwise take much longer for a manual search.

Another popular AutoML platform for TinyML is **Neuton.AI**¹. Similar to EON Tuner, Neuton is a neural network framework that helps the developer to build highly optimized tiny machine learning models and deploy them on microcontrollers and

1 <https://neuton.ai>

other embedded platforms without writing codes. Neuton is also free to use in developing exciting TinyML applications.

Edge ML on smartphones

Although our main focus in this book was to deploy machine learning models on microcontroller devices, there is plenty of attention in the industry on improving the ML experience on other edge devices, mainly smartphones. Modern smartphones are extremely powerful. Most of the high-end smartphones have dedicated **Graphics Processing Unit (GPU)** and **Neural Processing Unit (NPU)** implemented in the processor in order to use the power of advanced neural networks. You might have heard about AI chips. They are specially designed new-generation hardware accelerators for executing AI operations more efficiently. AI chips consist of **Field-Programmable Gate Arrays (FPGA)**, GPU, and **Application-Specific Integrated Circuits (ASIC)**. In short, AI chips accumulate large quantities of smaller transistors that use less energy and can run faster than big transistors. Unlike CPUs, AI chips have features that are specially designed and optimized to run AI on the processor. In 2017, Intel brought in \$1 billion from the sale of AI chips. Edge AI Chips implement AI in an edge computing environment without any connection to the cloud. By 2024, unit sales of edge AI chips are expected to exceed 1.5 billion. The key benefits of on-device AI include real-time responsiveness, improved privacy, and enhanced reliability. In 2018, Qualcomm introduced its **Qualcomm Artificial Intelligence Engine (QAIE)**, which was comprising of several hardware and software components to accelerate on-device AI-enabled user experiences on select Qualcomm® Snapdragon™ mobile platforms. The objective was to maximize the implementation of AI on mobile edge devices without a network connection.

On-device AI has been rapidly adopted by other industry leaders as well. The die shot of the chip for Samsung's Exynos 9820 processor reveals that about 5% of the total chip area is dedicated to AI processors. The Huawei Kirin 970 chip dedicates 2.1% of the die area to the NPU. Apple's A12 Bionic chip dedicates about 7% of the die area to machine learning. AI-assisted solutions are readily available in smartphones. Ultra HDR photography, Raw image processing, 4K Night videography, NLP, and HD gaming are a few examples where edge ML plays a crucial role.

Future of TinyML

TinyML is one of the fastest-growing fields of machine learning that has the potential to revolutionize the way we interact with smart devices. Being positioned at the intersection of embedded systems, machine learning algorithms, and hardware

TinyML enables smart devices to perform complex tasks without relying on cloud servers or high-power computing resources. Owing to its versatility, cost-effectiveness, low power consumption, small form factor, and compelling cost, TinyML has already seen plenty of applications in various industry sectors. In today's digital world, we generate huge amount of data every day. You will be surprised to know that less than 10% of the generated data reaches the cloud. Intelligence at the edge can revolutionize the world. The whole idea has become possible thanks to the recent advances in edge AI, more specifically, the TinyML technology.

We are yet to unleash the full potential of TinyML for the betterment of mankind. It is a real opportunity for anyone who wants to learn about this technology in order to create exciting applications for the future. Being low-cost and power efficient, TinyML applications can run 24x7 on small embedded devices. Such subsystems can be popularly used in industries for continuous monitoring of large machines as part of predictive and proactive maintenance. TinyML has also been popularly used in medical applications for continuous patient monitoring, agriculture, and also in our home appliances. TinyML is no doubt going to play a vital role to solve practical problems where traditional machine learning systems cannot work effectively owing infrastructure restriction. While conventional machine learning will evolve to solve more complex and sophisticated business problems, TinyML will be more and more popular to the end customers to solve their everyday needs. As the number of smart devices and microcontrollers is exponentially growing every day, the TinyML technology has to grow. *Pete Warden*, the Technical Lead of TensorFlow Lite, believes that TinyML will impact almost every industry in the future, including retail, healthcare, agriculture, manufacturing, and so on. *Vijay Janapa Reddi*, a professor at Harvard University and a leading researcher in TinyML, has rightly said that “the future of machine learning is tiny and bright”.

Further reading

1. Warden, Pete, and Daniel Situnayake. *Tinyml: Machine learning with TensorFlow lite on Arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.
2. Shafique, Muhammad, Theocharis Theocharides, Vijay Janapa Reddy, and Boris Murmann. “TinyML: current progress, research challenges, and future roadmap.” In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 1303-1306. IEEE, 2021.
3. Iodice, Gian Marco. “TinyML Cookbook.” (2022).
4. Ray, Partha Pratim. “A review on TinyML: State-of-the-art and prospects.” *Journal of King Saud University-Computer and Information Sciences* (2021).

5. Rajapakse, V., Karunanayake, I. and Ahmed, N., 2022. Intelligence at the extreme edge: a survey on reformable TinyML. *ACM Computing Surveys*.
6. Banbury, C.R., Reddi, V.J., Lam, M., Fu, W., Fazel, A., Holleman, J., Huang, X., Hurtado, R., Kanter, D., Lokhmotov, A. and Patterson, D., 2020. Benchmarking tinyML systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*.
7. David, R., Duke, J., Jain, A., Janapa Reddi, V., Jeffries, N., Li, J., Kreeger, N., Nappier, I., Natraj, M., Wang, T. and Warden, P., 2021. Tensorflow lite micro: Embedded machine learning for TinyML systems. *Proceedings of Machine Learning and Systems*, 3, pp.800-811.
8. Sudharsan, B., Salerno, S., Nguyen, D.D., Yahya, M., Wahid, A., Yadav, P., Breslin, J.G. and Ali, M.I., 2021, June. TinyML benchmark: Executing fully connected neural networks on commodity microcontrollers. In *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)* (pp. 883-884). IEEE
9. Han, Hui, and Julien Siebert. "TinyML: A systematic review and synthesis of existing research." In *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIIC)*, pp. 269-274. IEEE, 2022.
10. Reddi, Vijay Janapa, Brian Plancher, Susan Kennedy, Laurence Moroney, Pete Warden, Anant Agarwal, Colby Banbury et al. "Widening access to applied machine learning with TinyML." *arXiv preprint arXiv:2106.04008* (2021).

Join our book's Discord space

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

<https://discord.bpbonline.com>



Appendix

ADC	Analog to Digital Converter
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
BCE	Binary Cross Entropy
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DAC	Digital to Analog Converter
DFT	Discrete Fourier Transform
GPIO	General Purpose Input Output
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
IMU	Inertial Measurement Unit
IoT	Internet of Things
LED	LIGHT Emitting Diode
MCU	MicroController Unit
MFCC	Mel Frequency Cepstral Coefficient
ML	Machine Learning
MSE	Mean Squared Error
NLP	Natural language Processing
RAM	Random Access Memory
ReLU	Rectified Linear Unit
SBC	Single Board Computer
SCP	Secure Copy Protocol
SDK	Software Development Kit
SSH	Secure SHell
STFT	Short-Time Fourier Transform
SVM	Support Vector Machine
TFLite	TensorFlow Lite
TPU	Tensor Processing Unit
USB	Universal Serial Bus

Index

Symbols

1D convolutional architecture 261

A

activation function 61
Analog to Digital Converter (ADC) 11
Application-Specific Integrated Circuits (ASIC) 277
Arduino IDE
 layout 205
Arduino Nano
 setting up 204-208
Arduino Nano 33 BLE Sense 16, 202, 203
Artificial Intelligence (AI) 1
Artificial Neural Network (ANN) 2, 7, 58
 activation functions 63, 64
 binary cross entropy loss function 62, 63
 hidden layers 61
 input layer 61
 output layer 61
 structure 61
 theory 59, 60
Artificial Neural Networks (ANN) 85
audio spectrogram 241-247
Automated Machine Learning (AutoML) 275, 276

B

backpropagation algorithm 66-69
binary cross entropy (BCE) function 62, 63
biometric recognition 176
Bluetooth Low Energy (BLE) module 202

C

categorical cross entropy loss function 95
Central Processing Unit (CPU) 9
classification models
 evaluation metrics 105
CNN architecture
 convolutional layer 71-76
 dense layer 79
 fully connected layer 79
 implementing 79, 80
 input layer 71
 output layer 79
 pooling layer 78
 strided convolution 77, 78
 zero padding 77
CNN model
 designing, for keyword spotting 247-251
Colab Notebook 22-24
collaborative optimization 138-143
Conv2D 100

- arguments 100
- Convolutional Neural Network (CNN) 7, 58-70, 85
- architecture 71
- functional layers 99-102
- implementation 97-100
- training 102-104

D

- DataFrames 41
- deep learning 6, 7, 57, 58
- dense layers 62
- depthwise separable convolution 148
- digital signal processing (DSP) 254
- Digital to Analog Converter (DAC) 11
- Discrete Fourier Transform (DFT) 242

E

- Edge AI 8
- edge computing 7
- Edge Impulse 251-253
 - keyword spotting, implementing 253-264
 - URL 252
- Edge ML 8
 - on smartphones 277
- end-to-end Machine Learning algorithm
 - with TensorFlow 48-54
- EON Tuner tool 261, 276
- epochs 95
- evaluation metrics
 - classification accuracy 107
 - classification report 106
 - confusion matrix 105, 106
 - F1-score 107
 - precision 106
 - recall or sensitivity 106
 - specificity 106

F

- face recognition 178
 - Raspberry Pi, setting up for 180

- face recognition pipeline 179, 180
- face recognition project implementation 185
 - data collection 185-189
 - model training 189-191
 - real-time face recognition 192-196
- feedforward ANN classifier
 - data processing 92-94
 - model implementation 94-97
- feedforward network 62
- Field-Programmable Gate Arrays (FPGA) 277
- FlatBuffers 110
- for loop 30
- forward propagation 67
- fully connected layers 62
- fully connected neural network 62
- functional layers, CNN
 - Convo2D 100
 - Dense 100
 - Flatten 100
 - MaxPooling2D 100
- functions, Python 30, 31

G

- General Public License (GPL) 20
- Google Colab 16
- Graphics Processing Unit (GPU) 2, 7, 22, 277
- ground (GND) pin 209

H

- handwritten numerical digits
 - classifying, with feedforward neural network 90-92

I

- ILSVRC 70
- Inertial Measurement Unite (IMU)
 - sensor 202
- Integrated Circuit (IC) 11
- Integrated Development Environment (IDE) 22, 204

Internet of Things (IoT) 7
 challenges 8

K

Keras 87
 architecture 87
 core modules 87
 layers 87
 model 87
 ANN structure, implementing 87-89
 keyword 232
 keyword spotting 232
 audio spectrogram 241-247
 CNN model, designing for 247-251
 implementing, in Edge Impulse 253-264
 model deployment 264-266
 keyword spotting algorithm
 implementing, in Python 235-241

L

learning block 259
 loops, Python 30

M

Machine Learning
 overview 4, 5
 supervised machine learning 5, 6
 unsupervised machine learning 6
 Matplotlib 39-41
 MaxPooling2D 100
 arguments 100
 Mean Squared Error (MSE) 48, 62
 Mel Frequency Cepstral Coefficient
 (MFCC) 259
 Mel Frequency Energy (MFE) 263
 microcontroller 200
 mini-batch size 95
 MNIST 90
 MobileNet architecture 147
 depthwise separable convolution 148
 image classification 148-152

implementing, with
 transfer learning 153, 154
 model evaluation, on test set 157, 158
 optimized model, creating for smaller
 target device 154-156

N

neural network activation functions 63, 64
 ReLU activation function 65, 66
 Sigmoid activation function 64
 softmax function 66
 tanh activation function 65
 neural network hyperparameters 80, 81
 choice, of optimization algorithm 82
 dropout 81, 82
 learning rate 81
 mini-batch size 82
 number of layers 81
 regularization 82
 specific, to CNN 83
 Neural Processing Unit (NPU) 277
 neurons 58
 Neuton.AI 276
 nodes 58
 NumPy 32-36
 random number generation 37-39

O

overfit 15

P

Pandas 41, 42
 PIRGBArray() function 187
 pitch 241
 plot() function 39
 processing block 259
 Python 19-21
 advantages 20
 conditional operations 28, 29
 functions 30, 31
 libraries 32

- logical operations 28, 29
- loops 30
- Python Imaging Library (PIL) 166
- Python variables 24
 - dictionary 28
 - lists 26
 - strings 25
 - tuples 27, 28

Q

- Qualcomm Artificial Intelligence Engine (QAIE) 277
- quantization 121
 - post-training quantization 121-123
 - quantization-aware training 123-128

R

- randint() function 37
- Random Access Memory (RAM) 8
- Raspberry Pi 158
 - accessing remotely 164
 - components 159, 160
 - features 159
 - model deployment, for creating inference script 165-172
 - operating system installation 161
 - setting up 162, 163
- Raspberry Pi 3 16
- Raspberry Pi 3 Model B+
 - features 158, 159
 - setting up 160
- Raspberry Pi, for face recognition
 - libraries, installing 184, 185
 - Raspberry Pi camera module, setting up 180-184
 - setting up 180
- Read Only Memory (ROM) 11
- ReLU activation function 65, 66

S

- Secure Copy Protocol (SCP) 171

- secured shell (SSH) protocol 164
- Short-time Fourier Transform (STFT) 242
- Sigmoid activation function 64
- Single Board Computer (SBC) 10, 11
- softmax function 66
- Software Development Kits (SDK) 15
- spectrogram 242
- spectrogram analysis 242
- spectrum 242

T

- tanh activation function 65
- Tensor Arena 224
- TensorFlow 13, 42, 43, 87
 - datatypes 44, 45
 - differentiation 46
 - end-to-end Machine Learning algorithm 48-54
 - features 43
 - functions 47, 48
 - graphs 47, 48
 - tensor 44
 - variables 45
- TensorFlow Lite 13, 110, 111
 - advantages 111, 112
 - for Microcontrollers 13
 - model, creating 112-120
- TensorFlow Lite Converter 110
- TensorFlow Model Optimization Toolkit 110, 120, 121
 - quantization 121
 - weight clustering 134-138
 - weight pruning 128-133
- Tensor Processing Unit (TPU) 7, 22
- TinyML 3, 4, 8
 - best practices 273, 274
 - future 277, 278
 - hardware, for deployment 10-12
 - TinyML application, on microcontroller circuit connection 211, 212
 - inference, on Arduino Nano 222-228

- model, creating for modulating
 - potentiometer reading 215-221
 - potentiometer, modulating 209, 210
 - potentiometer, reading 212-215
 - required components 210
 - TinyML applications
 - agriculture 9
 - data acquisition 14
 - hardware and software
 - prerequisites 16, 17
 - healthcare 9
 - inference, making 16
 - model creation 14, 15
 - model deployment, at edge 15
 - model optimization and conversion,
 - for edge devices 15
 - ocean life conservation 10
 - predictive maintenance 9
 - process flow, for creating 13
 - voice-assisted devices 9, 10
 - transfer learning 147, 152, 153
 - for implementing MobileNet 153, 154
 - Transmission Control Protocol (TCP) 164
-
- ## V
-
- voice assistant, working principles
 - keyword spotting 234
 - Natural Language Processing (NLP) 234, 235
 - Text to Speech 235
 - voltage input (Vin) pin 209
-
- ## W
-
- wake-up command 233
 - weight clustering 134-138
 - weight pruning 128-133
 - magnitude-based weight pruning criteria 128
 - while loop 30

Hands-on TinyML

DESCRIPTION

TinyML is an innovative technology that empowers small and resource-constrained edge devices with the capabilities of machine learning. If you're interested in deploying machine learning models directly on microcontrollers, single board computers, or mobile phones without relying on continuous cloud connectivity, this book is an ideal resource for you.

The book begins with a refresher on Python, covering essential concepts and popular libraries like NumPy and Pandas. It then delves into the fundamentals of neural networks and explores the practical implementation of deep learning using TensorFlow and Keras. Furthermore, the book provides an in-depth overview of TensorFlow Lite, a specialized framework for optimizing and deploying models on edge devices. It also discusses various model optimization techniques that reduce the model size without compromising performance. As the book progresses, it offers a step-by-step guidance on creating deep learning models for object detection and face recognition specifically tailored for the Raspberry Pi. You will also be introduced to the intricacies of deploying TensorFlow Lite applications on real-world edge devices. Lastly, the book explores the exciting possibilities of using TensorFlow Lite on microcontroller units (MCUs), opening up new opportunities for deploying machine learning models on resource-constrained devices.

Overall, this book serves as a valuable resource for anyone interested in harnessing the power of machine learning on edge devices.

KEY FEATURES

- Gain a comprehensive understanding of TinyML's core concepts.
- Learn how to design your own TinyML applications from the ground up.
- Explore cutting-edge models, hardware, and software platforms for developing TinyML.

WHAT YOU WILL LEARN

- Explore different hardware and software platforms for designing TinyML.
- Create a deep learning model for object detection using the MobileNet architecture.
- Optimize large neural network models with the TensorFlow Model Optimization Toolkit.
- Explore the capabilities of TensorFlow Lite on microcontrollers.
- Build a face recognition system on a Raspberry Pi.
- Build a keyword detection system on an Arduino Nano.

WHO THIS BOOK IS FOR

This book is designed for undergraduate and postgraduate students in the fields of Computer Science, Artificial Intelligence, Electronics, and Electrical Engineering, including MSc and MCA programs. It is also a valuable reference for young professionals who have recently entered the industry and wish to enhance their skills.



BPB PUBLICATIONS

www.bpbonline.com

ISBN 978-93-5551-844-6



9 789355 151844 6