

EXPERT INSIGHT

TinyML

Cookbook

Combine machine learning with microcontrollers
to solve real-world problems

Second Edition

Gian Marco Iodice

«packt»

TinyML Cookbook

Second Edition

Combine machine learning with microcontrollers to solve real-world problems

Gian Marco Iodice

<packt>

BIRMINGHAM—MUMBAI

TinyML Cookbook

Second Edition

Copyright © 2023 Packt Publishing

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

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

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

Publishing Product Manager: Bhavesh Amin

Community Relations Specialist: Tejas Vijay Mhasvekar

Project Editor: Rianna Rodrigues

Content Development Editor: Bhavesh Amin

Copy Editor: Safis Editing

Technical Editor: Aneri Patel

Proofreader: Safis Editing

Indexer: Manju Arasan

Presentation Designer: Rajesh Shirsath

Developer Relations Marketing Executive: Monika Sangwan

First published: April 2022

Second edition: November 2023

Production reference: 1211123

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83763-736-2

www.packt.com

The cover image is by Monika from Pixabay (<https://pixabay.com/users/monicore-1499084/>)

tinyML is a registered trademark of tinyML Foundation and is used with permission. The author and publisher express their gratitude to tinyML Foundation for their collaboration and educational initiatives worldwide, which are vital in shaping the landscape of this field.

*I dedicate this book to Eleonora, who believed in this project from the very beginning
to help developers build healthy and thriving communities with tinyML*

Contributors

About the author

Gian Marco Iodice is an experienced edge and mobile computing specialist at Arm for **machine learning (ML)**. He is also the chair of global meetups for tinyML Foundation since 2022. He received an MSc with honors in electronic engineering from the University of Pisa (Italy), where he specialized in HW/SW co-design for embedded systems. At Arm, he leads engineering development for the Arm Compute Library, which he co-created in 2017 to run ML workloads as efficiently as possible on Arm processors. The Arm Compute Library, designed to deliver the best performance across Arm Cortex-A CPUs and Mali GPUs, is deployed on billions of devices worldwide – from servers to smartphones. In 2023, he collaborated with the University of Cambridge to integrate ML functionalities into an Arm Cortex-M microcontroller powered by algae. This ground-breaking work was showcased at the tinyML EMEA Innovation Forum in Amsterdam in June of the same year. Also, in 2023, Gian Marco contributed to developing the EdTech for Good Curation Framework. This framework, developed by UNICEF in collaboration with Arm and the Asian Development Bank (ADB), represents a significant step forward in the responsible use of technology in education because it enables public entities and international organizations to evaluate digital educational technologies, prioritizing learning outcomes and children’s safety.

I would like to give a special thanks to the thriving open-source community that has collectively transformed tinyML from a niche technology into a mature one, accessible to many developers to solve big real-world problems. This book is the result of many developers and companies around the world that have generously contributed to this field. I encountered many people during the journey that led to the creation of this book. Their support, ideas, and help to promote the first edition of the TinyML Cookbook inspired me and gave me the energy to complete this work. In alphabetical order, I would like to thank Alasdair Allan, Alessandro Grande, Angeles Cortesi, Bhavesh Amin, Carlos Hernandez-Vaquero, Christophe Favergeon, Colin Osborne, Evgeni Gousev, Felix Johnny Thomasmithibalan, Filippo Fantini, Ira Feldman, Julia Munoz, Kwadwo Dompreeh, Leandro Nunes, Massimo Banzi, Michael Hall, Olga Gorenichina, Rianna Rodrigues, Robert Wolff, Rod Crawford, Rosina Haberl, Sandro Polliatti, and Vincent Kok.

About the reviewers

Colin Osborne is a technology enthusiast and had the privilege of becoming Director of System Analysis in the Arm ML Group, as well as experimenting with ML in his own time. Before joining Arm in 2010, he was an SoC architect at Philips Semiconductors and then the newly formed NXP. He holds a master's degree in electrical and electronic engineering from the University of Surrey, UK.

Carlos Hernandez-Vaquero, Engineering Manager at Bosch (Germany), leads the Data-Centric Software Group with 10+ years in AIoT. He studied at the University of Oviedo (Spain) and the University of Aalborg (Denmark). He founded the tech communities TinyML Germany, GenAI-Gurus.com, and TensorFlow Berlin. He authored courses on tinyML and mentors at DeepLearning.ai. He is an AWS Community Builder in ML and is active in Azure and GitHub communities. Connect with him on LinkedIn or chat with his GPT digital twin, ai-mvp.com.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>

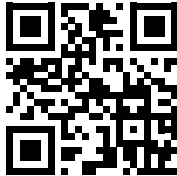


Table of Contents

Preface	xxiii
Chapter 1: Getting Ready to Unlock ML on Microcontrollers	1
Technical requirements	2
Introduction to tinyML	2
What is tinyML? • 3	
Why ML on microcontrollers? • 3	
Why run ML on-device? • 4	
The opportunities and challenges for tinyML • 5	
Deployment environments for tinyML • 5	
Join the tinyML community! • 7	
Overview of deep learning	7
Deep neural networks • 7	
Convolutional neural networks • 9	
Model quantization • 12	
Learning the difference between power and energy	13
Voltage versus current • 13	
Power versus energy • 15	
Programming microcontrollers	17
Memory architecture • 20	
Peripherals • 21	
General-purpose input/output (GPIO or IO) • 22	

<i>Analog/digital converters • 23</i>	
<i>Serial communication • 23</i>	
<i>Timers • 23</i>	
Introduction to the development platforms	24
Arduino Nano 33 BLE Sense • 24	
Raspberry Pi Pico • 25	
SparkFun RedBoard Artemis Nano • 25	
Setting up the software development environment	26
Getting ready with Arduino IDE • 26	
Getting ready with TensorFlow • 28	
Getting ready with Edge Impulse • 30	
Deploying a sketch on microcontrollers	31
Getting ready • 31	
How to do it... • 32	
There's more...with the SparkFun Artemis Nano! • 34	
Summary	35
 Chapter 2: Unleashing Your Creativity with Microcontrollers	 37
Technical requirements	38
Transmitting data over serial communication	38
Getting ready • 39	
How to do it... • 40	
There's more... • 42	
Reading serial data and uploading files to Google Drive with Python	43
Getting ready • 43	
Reading data from the serial port with pySerial • 44	
Enabling the Google Drive API • 44	
How to do it... • 50	
There's more...	55

Implementing an LED status indicator on the breadboard	55
Getting ready • 56	
<i>Prototyping on a breadboard • 58</i>	
How to do it... • 59	
There's more...with the SparkFun Artemis Nano! • 63	
Controlling an external LED with the GPIO	64
Getting ready • 64	
<i>Understanding the LED functionality • 64</i>	
<i>Introducing the GPIO peripheral • 68</i>	
<i>Using the mbed::DigitalOut function • 70</i>	
How to do it... • 71	
<i>Connecting the LED to the GPIO pin • 71</i>	
<i>Programming the GPIO peripheral in output mode • 74</i>	
There's more...with the SparkFun Artemis Nano! • 75	
Turning an LED on and off with a push-button	75
Getting ready • 76	
<i>The operating principles of the push-button • 76</i>	
How to do it... • 78	
<i>Connecting the push-button to the GPIO pin • 78</i>	
<i>Programming the GPIO peripheral in input mode • 80</i>	
There's more...with the SparkFun Artemis Nano! • 82	
Using interrupts to read the push-button state	82
Getting ready • 82	
<i>Working with interrupts using the Mbed OS API • 82</i>	
How to do it... • 84	
There's more... • 85	
Summary	86
 Chapter 3: Building a Weather Station with TensorFlow Lite for Microcontrollers	
	87
<hr/>	
Technical requirements	88

Importing weather data from WorldWeatherOnline	88
Getting ready • 89	
How to do it... • 89	
There's more... • 91	
Preparing the dataset	92
Getting ready • 92	
<i>Balancing the dataset • 92</i>	
<i>Feature scaling with Z-score • 93</i>	
How to do it... • 94	
There's more... • 98	
Training the model with TensorFlow	99
Getting ready • 100	
How to do it... • 100	
There's more... • 106	
Evaluating the model's effectiveness	107
Getting ready • 107	
<i>Evaluating the performance with the confusion matrix • 107</i>	
<i>Evaluating recall, precision, and F-score • 109</i>	
How to do it... • 109	
There's more... • 112	
Quantizing the model with the TensorFlow Lite converter	113
Getting ready • 113	
<i>Model quantization • 114</i>	
How to do it... • 118	
There's more... • 120	
Reading temperature and humidity data with the Arduino Nano	121
Getting ready • 121	
How to do it... • 122	
There's more... • 124	
Reading temperature and humidity with the DHT22 sensor and the Raspberry Pi Pico ...	124
Getting ready • 125	

How to do it... • 126	
There's more...with the SparkFun Artemis Nano! • 131	
Preparing the input features for the model inference	132
Getting ready • 133	
How to do it... • 133	
There's more...with the SparkFun Artemis Nano • 136	
On-device inference with TensorFlow Lite for Microcontrollers	136
Getting ready • 137	
How to do it... • 137	
There's more...with the SparkFun Artemis Nano! • 143	
Summary	143
 Chapter 4: Using Edge Impulse and the Arduino Nano to Control LEDs with Voice Commands	 145
Technical requirements	146
Acquiring audio data with a smartphone	146
Getting ready • 146	
<i>Collecting audio samples for KWS • 147</i>	
How to do it... • 147	
There's more • 153	
Acquiring audio data with the Arduino Nano	153
Getting ready • 154	
<i>Collecting data using a fully supported platform in Edge Impulse • 155</i>	
How to do it... • 155	
There's more • 159	
Extracting MFE features from audio samples	159
Getting ready • 160	
<i>Analyzing audio in the frequency domain • 160</i>	
<i>Extracting the Mel-spectrogram • 161</i>	
How to do it... • 164	
There's more • 166	

Designing and training a CNN	169
Getting ready • 169	
How to do it... • 169	
There's more • 174	
Tuning model performance with the EON Tuner	174
Getting ready • 174	
How to do it... • 175	
There's more • 177	
Live classifications with a smartphone	178
Getting ready • 178	
How to do it... • 178	
There's more • 181	
Keyword spotting on the Arduino Nano	182
Getting ready • 182	
<i>Learning how a real-time KWS application works • 183</i>	
How to do it... • 185	
There's more • 192	
Summary	192
 Chapter 5: Recognizing Music Genres with TensorFlow and the Raspberry Pi Pico – Part 1	 195
Technical requirements	196
Connecting the microphone to the Raspberry Pi Pico	197
Getting ready • 197	
<i>Connecting the microphone to the ADC pin • 199</i>	
How to do it... • 199	
There's more... • 203	
Recording audio samples with the Raspberry Pi Pico	203
Getting ready • 203	
<i>Recording audio with ADC and timer interrupts • 204</i>	
<i>Programming the ADC with the Raspberry Pi Pico SDK • 205</i>	

How to do it... • 205	
There's more...with the SparkFun Artemis Nano! • 211	
Generating audio files from samples transmitted over the serial	212
Getting ready • 212	
How to do it... • 212	
There's more... • 215	
Building the dataset for classifying music genres	215
Getting ready • 215	
<i>Using the GTZAN dataset for music genre classification • 216</i>	
<i>Augmenting the dataset with audio samples taken with the Raspberry Pi Pico • 216</i>	
<i>Choosing the suitable model input length • 217</i>	
How to do it... • 217	
There's more...with the SparkFun Artemis Nano! • 221	
Extracting MFCCs from audio samples with TensorFlow	221
Getting ready • 222	
Applying the Hann window • 223	
<i>Leveraging RFFT • 224</i>	
<i>Calculating the FFT magnitude • 226</i>	
<i>The Mel scale conversion • 227</i>	
<i>Computing the DCT coefficients • 230</i>	
<i>Evaluating the SRAM usage to run MFCCs • 231</i>	
How to do it... • 232	
There's more... • 239	
Summary	239
References	240
 Chapter 6: Recognizing Music Genres with TensorFlow and the Raspberry Pi Pico – Part 2	 241
Technical requirements	242

Computing the FFT magnitude with fixed-point arithmetic using the CMSIS-DSP library	243
Getting ready • 243	
<i>Evaluating the hardware capabilities of the Raspberry Pi Pico • 243</i>	
<i>Using the CMSIS-DSP Python library • 244</i>	
<i>Representing numbers in 16-bit fixed-point format • 245</i>	
How to do it... • 248	
There's more... • 253	
Implementing the MFCCs feature extraction with the CMSIS-DSP library	253
Getting ready • 254	
<i>Extracting the DCT coefficients • 254</i>	
How to do it... • 255	
There's more... • 264	
Designing and training an LSTM RNN model	266
Getting ready • 266	
<i>Time series analysis with RNNs • 266</i>	
<i>Designing a many-to-one RNN for music genre classification • 270</i>	
How to do it... • 272	
There's more... • 278	
Evaluating the accuracy of the quantized model on the test dataset	279
Getting ready • 279	
How to do it... • 279	
There's more... • 283	
Deploying the MFCCs feature extraction algorithm on the Raspberry Pi Pico	283
Getting ready • 283	
How to do it... • 284	
There's more...with the SparkFun Artemis Nano! • 293	
Recognizing music genres with the Raspberry Pi Pico	295
Getting ready • 295	
How to do it... • 296	
There's more...with the SparkFun Artemis Nano! • 301	

Summary	301
Chapter 7: Detecting Objects with Edge Impulse Using FOMO on the Raspberry Pi Pico	303
Technical requirements	304
Acquiring images with the webcam	305
Getting ready • 305	
<i>Building a dataset for FOMO • 306</i>	
How to do it... • 306	
There's more... • 310	
Designing the Impulse's pre-processing block	311
Getting ready • 311	
<i>Choosing the correct input image resolution • 312</i>	
How to do it... • 313	
There's more...with the SparkFun Artemis Nano! • 316	
Transfer learning with FOMO	316
Getting ready • 317	
<i>Behind the design of FOMO • 317</i>	
<i>Locating objects from heat maps • 318</i>	
How to do it... • 320	
There's more... • 323	
Evaluating the model's accuracy	325
Getting ready • 325	
How to do it... • 326	
There's more... • 329	
Using OpenCV and pySerial to send images over the serial interface	330
Getting ready • 330	
<i>Sending bytes over the serial with pySerial • 331</i>	
How to do it... • 332	
There's more... • 336	

Reading data from the serial port with Arduino-compatible platforms	337
Getting ready • 337	
How to do it... • 338	
There's more...with the SparkFun Artemis Nano! • 339	
Deploying FOMO on the Raspberry Pi Pico	339
Getting ready • 340	
<i>Initializing the signal_t data structure • 341</i>	
<i>Reading the output results from ei_impulse_result_t • 343</i>	
<i>Transmitting the centroid coordinates to the Python script • 344</i>	
How to do it... • 345	
There's more...with the SparkFun Artemis Nano! • 350	
Summary	351
 Chapter 8: Classifying Desk Objects with TensorFlow and the Arduino Nano	 353
Technical requirements	354
Taking pictures with the OV7670 camera module	354
Getting ready • 355	
How to do it... • 356	
There's more... • 361	
Grabbing camera frames from the serial port with Python	361
Getting ready • 361	
<i>Introducing the RGB565 color format • 362</i>	
<i>Transmitting images over the serial • 363</i>	
How to do it... • 365	
There's more... • 370	
Acquiring QQVGA images with the YCbCr422 color format	370
Getting ready • 371	
<i>Converting YCbCr422 to RGB888 • 371</i>	
How to do it... • 372	
There's more... • 375	

Building the dataset to classify desk objects	375
Getting ready • 375	
How to do it... • 375	
There's more... • 380	
Transfer learning with Keras	380
Getting ready • 380	
<i>Behind the MobileNet network design choices • 381</i>	
How to do it... • 383	
There's more... • 388	
Quantizing and testing the trained model with TensorFlow Lite	389
Getting ready • 389	
How to do it... • 389	
There's more... • 393	
Fusing the pre-processing operators for efficient deployment	393
Getting ready • 394	
<i>Image resizing with bilinear interpolation • 395</i>	
How to do it... • 397	
There's more... • 407	
Summary	407
 Chapter 9: Building a Gesture-Based Interface for YouTube Playback with Edge Impulse and the Raspberry Pi Pico	 409
Technical requirements	410
Communicating with the MPU-6050 IMU through I2C	411
Getting ready • 411	
<i>Introducing the features of the MPU-6050 IMU • 411</i>	
<i>Basics of the I2C communication protocol • 412</i>	
<i>Programming the I2C peripheral with Mbed OS • 415</i>	
<i>Accessing the I2C peripheral in Raspberry Pi Pico • 415</i>	
How to do it... • 416	
There's more...with the SparkFun Artemis Nano! • 420	

Acquiring accelerometer data	421
Getting ready • 421	
<i>The basic principles of the accelerometer • 422</i>	
How to do it... • 425	
There's more...with the SparkFun Artemis Nano! • 430	
Building the dataset with the Edge Impulse data forwarder tool	432
Getting ready • 432	
How to do it... • 433	
There's more... • 437	
Designing and training the ML model	438
Getting ready • 438	
<i>Using spectral analysis to recognize hand gestures • 439</i>	
How to do it... • 440	
There's more • 444	
Live classifications with the Edge Impulse data forwarder tool	445
Getting ready • 445	
How to do it... • 445	
There's more...with the SparkFun Artemis Nano • 447	
Developing a continuous gesture recognition application with Edge Impulse and Arm Mbed OS	447
Getting ready • 447	
<i>Managing concurrent tasks with Arm Mbed OS • 448</i>	
<i>Filtering out redundant and spurious predictions • 449</i>	
How to do it... • 450	
There's more...with the SparkFun Artemis Nano • 458	
Building a gesture-based interface with PyAutoGUI	459
Getting ready • 459	
How to do it... • 460	
There's more • 462	
Summary	462

Chapter 10: Deploying a CIFAR-10 Model for Memory-Constrained Devices with the Zephyr OS on QEMU 465

Technical requirements	466
Getting started with the Zephyr OS	466
Getting ready •	466
How to do it... •	467
There's more... •	469
Designing and training a CIFAR-10 model for memory-constrained devices	470
Getting ready •	471
<i>Replacing convolution with depthwise separable convolution •</i>	<i>471</i>
<i>Keeping the model memory requirement under control •</i>	<i>474</i>
How to do it... •	475
There's more... •	479
Evaluating the accuracy of the quantized model	479
Getting ready •	479
How to do it... •	480
There's more... •	483
Converting a NumPy image into a C-byte array	484
Getting ready •	484
How to do it... •	485
There's more... •	487
Preparing the Zephyr Project structure	488
Getting ready •	488
How to do it... •	489
There's more... •	491
Deploying the TensorFlow Lite for Microcontrollers program on QEMU	491
Getting ready •	491
How to do it... •	492
There's more... •	497

Summary	498
References	498
Chapter 11: Running ML Models on Arduino and the Arm Ethos-U55 microNPU Using Apache TVM	501
Technical requirements	502
Getting familiar with Arduino CLI	503
Getting ready • 503	
How to do it... • 503	
There's more...with the SparkFun Artemis Nano! • 507	
Downloading a pre-trained CIFAR-10 model and input test image	509
Getting ready • 509	
How to do it... • 510	
There's more... • 510	
Deploying the model with TVM using the AoT executor on the host machine	511
Getting ready	511
<i>Behind the TVM compiler • 513</i>	
<i>Code generation with TVM • 514</i>	
<i>Graph executor versus AoT executor • 515</i>	
<i>Introducing microTVM for microcontroller deployment • 516</i>	
How to do it... • 517	
There's more... • 524	
Deploying the model on the Arduino Nano	524
Getting ready • 525	
<i>Running the model inference using the TVM runtime • 527</i>	
<i>Building an Arduino sketch with the code generated by TVM • 527</i>	
How to do it... • 529	
There's more... • 535	
Deploying the model on the Raspberry Pi Pico	536
Getting ready • 536	
How to do it... • 537	

There's more...with the SparkFun Artemis Nano! • 540	
Installing the Fixed Virtual Platform (FVP) for the Arm Corstone-300	540
Getting ready • 541	
How to do it... • 542	
There's more... • 544	
Code generation with TVMC for Arm Ethos-U55	545
Getting ready • 545	
How to do it... • 546	
There's more... • 548	
Installing the software dependencies to build an application for the Arm Ethos-U	
microNPU	549
Getting ready • 549	
<i>The GNU Arm Embedded Toolchain • 549</i>	
<i>The Ethos-U driver • 549</i>	
<i>The Ethos-U platform • 550</i>	
How to do it... • 550	
There's more... • 552	
Running the CIFAR-10 model inference on the Arm Ethos-U55 microNPU	553
Getting ready • 553	
<i>An overview of memory capabilities on Corstone-300 FVP • 554</i>	
How to do it... • 556	
There's more... • 561	
Summary	561
 Chapter 12: Enabling Compelling tinyML Solutions with On-Device Learning	
and scikit-learn on the Arduino Nano and RaspberryPi Pico	563
<hr/>	
Technical requirements	564
How can we train a model on microcontrollers?	565
Getting ready • 565	
Solving the XOR and NAND problem using a shallow neural network • 566	
<i>Training a neural network with backpropagation • 568</i>	

How to do it... • 574	
There’s more...with the SparkFun Artemis Nano! • 588	
How can we deploy scikit-learn models on microcontrollers?	590
Getting ready • 590	
How to do it... • 591	
There’s more...with the SparkFun Artemis Nano • 595	
How can we power microcontrollers with batteries?	596
Getting ready • 596	
<i>Increasing the output voltage by connecting batteries in series • 597</i>	
<i>Increasing the charge capacity by connecting batteries in parallel • 597</i>	
<i>Connecting batteries to the microcontroller board • 599</i>	
How to do it... • 599	
There’s more...with the SparkFun Artemis Nano • 602	
Summary	602
References	603
 Conclusion	 605
<hr/>	
Other Books You May Enjoy	611
<hr/>	
Index	615

Preface

This book is about **tinyML**, the technology that allows smartness in a minimally intrusive way using **machine learning (ML)** on low-powered devices like microcontrollers.

This technology has been around us for many years, for example, in smartwatches, intelligent assistants, and drones, just to name a few. However, today, it is witnessing an incredible growth in all market segments because of the continued success in reducing the complexity of ML model deployment, the proliferation of low-cost devices with extraordinary computing capabilities, and the invaluable contributions from the open-source community. Therefore, tinyML is not a niche technology designed by a few people to solve a few technological problems. Instead, it is a technology in the hands of many developers to solve big real-world problems.

tinyML is an exciting field full of opportunities. With a few tens of dollars, you can give life to objects that interact with the environment smartly and transform how we live for the better. However, this field can be challenging for those unfamiliar with microcontroller programming. Therefore, this book aims to dispel these barriers and demonstrate that tinyML is for everyone through practical examples.

Whether new to this field or looking to expand your ML knowledge, this improved second edition of *TinyML Cookbook* has something for all. Each chapter is structured to be a self-contained project to learn how to use some of the key tinyML technologies, such as **Arduino**, **CMSIS-DSP**, **Edge Impulse**, **emlearn**, **Raspberry Pi Pico SDK**, **TensorFlow**, **TensorFlow Lite for Microcontrollers**, and **Zephyr**.

Your practical journey into tinyML will start with an introduction to this multidisciplinary field and get you up to speed with some of the fundamentals for deploying applications on microcontrollers. For example, you will tackle problems you may encounter while prototyping microcontrollers, such as controlling the LED light or reading the push-button state using the GPIO peripheral.

After preparing for microcontroller programming, you will focus on tinyML projects using real-world sensors. Here, you will employ the temperature, humidity, and three “V” sensors (Voice, Vision, and Vibration) to implement end-to-end smart applications in different scenarios and learn best practices for building models for memory-constrained microcontrollers.

This second edition includes new recipes featuring an **LSTM** neural network to recognize music genres and the Edge Impulse **Faster-Objects-More-Objects (FOMO)** algorithm for detecting objects in a scene. These will help you stay updated with the latest developments in the tinyML community.

Finally, you will take your tinyML solutions to the next level with **TVM**, **Arm Ethos-U55 microNPU**, **on-device learning**, and the **scikit-learn** model deployment on microcontrollers.

TinyML Cookbook is a practical book with a focus on the principles. Although most of the presented projects are based on the **Arduino Nano 33 BLE Sense** and **Raspberry Pi Pico**, this second edition also features the **SparkFun RedBoard Artemis Nano** to help you practice the learned principles on an alternative microcontroller.

Therefore, by the end of this book, you will be well versed in best practices and ML frameworks to develop ML applications easily on microcontrollers.

Who this book is for

Whether you are an enthusiast or professional with a basic familiarity with ML and an interest in developing ML applications on microcontrollers through practical examples, this book is for you.

TinyML Cookbook will help you expand your knowledge of tinyML by building end-to-end projects with real-world data sensors on the **Arduino Nano 33 BLE Sense**, **Raspberry Pi Pico**, and **SparkFun RedBoard Artemis Nano**.

While familiarity with **C/C++**, **Python** programming, and the **command-line interface (CLI)** is required, no prior knowledge of microcontrollers is necessary.

What this book covers

Chapter 1, Getting Ready to Unlock ML on Microcontrollers, provides an overview of tinyML, presenting the opportunities and challenges to bring ML on extremely low-power microcontrollers. This chapter focuses on the fundamental elements behind ML, power consumption, and microcontrollers that make this technology different from conventional ML in the cloud, desktop, or even smartphones.

Chapter 2, Unleashing Your Creativity with Microcontrollers, presents recipes to deal with the relevant microcontroller programming basics. We will deal with code debugging and how to transmit data to the Arduino serial monitor. The transmitted data will be captured in a log file and uploaded to our cloud storage in Google Drive. Afterward, we will delve into programming the GPIO peripheral using the Arm Mbed API and use a solderless breadboard to connect external components, such as LEDs and push-buttons.

Chapter 3, Building a Weather Station with TensorFlow Lite for Microcontrollers, teaches us how to implement a simple weather station with ML to predict the occurrence of snowfall based on the temperature and humidity of the last three hours. In the first part, we will focus on dataset preparation and show how to acquire historical weather data from WorldWeatherOnline. After preparing the dataset, we will see how to train a neural network with TensorFlow and quantize the model to 8-bit with TensorFlow Lite. In the last part, we will deploy the model on the Arduino Nano 33 BLE Sense and Raspberry Pi Pico with TensorFlow Lite for Microcontrollers.

Chapter 4, Using Edge Impulse and the Arduino Nano to Control LEDs with Voice Commands, shows how to develop an end-to-end **keyword spotting (KWS)** application with Edge Impulse and the Arduino Nano 33 BLE Sense board. The chapter will begin with dataset preparation, showing how to acquire audio data with a mobile phone and the built-in microphone on the Arduino Nano. Next, we will design a model based on the popular **Mel Filterbank Energy (MFE)** features for speech recognition. In these recipes, we will show how to extract these features from audio samples, train the **machine learning (ML)** model, and optimize the performance with the Edge Impulse EON Tuner. At the end of the chapter, we will concentrate on deploying the KWS application.

Chapter 5, Recognizing Music Genres with TensorFlow and the Raspberry Pi Pico – Part 1, is the first part of a project to recognize three music genres from recordings obtained with a microphone connected to Raspberry Pi Pico. The music genres we will classify are disco, jazz, and metal. Since the project offers many learning opportunities, it is split into two chapters to give as much exposure to the technical aspects as possible. Here, we will focus on the dataset preparation and the analysis of the feature extraction technique employed for classifying music genres: the **Mel Frequency Cepstral Coefficients (MFCCs)**.

Chapter 6, Recognizing Music Genres with TensorFlow and Raspberry Pi Pico – Part 2, is the continuation of *Chapter 5* and discusses how the target device influences the implementation of the MFCCs feature extraction. We will start our discussion by tailoring the MFCCs implementation for Raspberry Pi Pico.

Here, we will learn how fixed-point arithmetic can help minimize the latency and show how the CMSIS-DSP library provides tremendous support in employing this limited numerical precision in feature extraction. After reimplementing the extraction of the MFCCs using fixed-point arithmetic, we will design an ML model capable of recognizing music genres with a **Long Short-Term Memory (LSTM) recurrent neural network (RNN)**. Finally, we will test the model accuracy on the test dataset and deploy a music genre classification application on Raspberry Pi Pico with the help of TensorFlow Lite for Microcontrollers.

Chapter 7, Detecting Objects with Edge Impulse using FOMO on the Raspberry Pi Pico, showcases the deployment of an object detection application on microcontrollers using Edge Impulse and the **Faster Objects, More Objects (FOMO)** ML algorithm. The chapter will begin with dataset preparation, demonstrating how to acquire images with a webcam and label them in Edge Impulse. Next, we will design an ML model based on the FOMO algorithm. In this part, we will explore the architectural features of this novel ML solution that allows us to deploy object detection on highly constrained devices. Subsequently, we will test the model using the Edge Impulse Live classification tool and then on the Raspberry Pi Pico.

Chapter 8, Classifying Desk Objects with TensorFlow and the Arduino Nano, demonstrates the benefit of adding sight to our tiny devices by classifying two desk objects with the OV7670 camera module in conjunction with the Arduino Nano 33 BLE Sense board. In the first part, we will learn how to acquire images from the OV7670 camera module. Then, we will focus on the model design, applying transfer learning with the Keras API to recognize two objects we typically find on a desk: a mug and a book. Finally, we will deploy the quantized TensorFlow Lite model on an Arduino Nano 33 BLE Sense with the help of TensorFlow Lite for Microcontrollers.

Chapter 9, Building a Gesture-Based Interface for YouTube Playback with Edge Impulse and the Raspberry Pi Pico, teaches us how to use accelerometer measurements with ML to recognize three hand gestures with Raspberry Pi Pico. These recognized gestures will then be used to play/pause, mute/unmute, and change YouTube videos on our PC. The development of this project will start by acquiring the accelerometer data to build the gesture recognition dataset. In this part, we will learn how to interface with the I2C protocol and use the Edge Impulse data forwarder tool. Next, we will focus on the Impulse design, where we will build a spectral-features-based feed-forward neural network for gesture recognition. Finally, we will deploy the model on the Raspberry Pi Pico and implement a Python script with the PyAutoGUI library to build a touchless interface for YouTube video playback.

Chapter 10, Deploying a CIFAR-10 Model for Memory-Constrained Devices with the Zephyr OS on QEMU, demonstrates how to build an image classification application with TensorFlow Lite for Microcontrollers for an emulated Arm Cortex-M3 microcontroller. To accomplish our task, we will start by installing the Zephyr OS, the primary framework used in this chapter. Next, we will design a tiny quantized CIFAR-10 model with TensorFlow. This model will be capable of running on a microcontroller with only 256 KB of program memory and 64 KB of RAM. Ultimately, we will deploy an image classification application on an emulated Arm Cortex-M3 microcontroller through **Quick Emulator (QEMU)**.

Chapter 11, Running ML Models on Arduino and the Arm Ethos-U55 microNPU Using Apache TVM, explores how to leverage Apache TVM to deploy a quantized CIFAR-10 TensorFlow Lite model in various scenarios. After introducing Arduino CLI, we will present TVM by showing how to generate C code from an ML model and how to run it on a machine hosting the Colab environment. In this chapter, we will also discuss the **ahead-of-time (AoT)** executor, a crucial feature of TVM that can help reduce the program memory usage of the final application. Then, we will delve into running the model on the Arduino Nano 33 BLE Sense and Raspberry Pi Pico and discuss how to compile a sketch from the code generated by TVM. Finally, we will explore the model deployment on a **Micro-Neural Processing Unit (microNPU)**.

Chapter 12, Enabling Compelling tinyML Solutions with On-Device Learning and scikit-learn on the Arduino Nano and Raspberry Pi Pico, aims to answer three likely questions you might be pondering to bring your tinyML projects to the next level. The first question will delve into the feasibility of training models directly on microcontrollers. In this part, we will discuss the backpropagation algorithm to train a shallow neural network. We will also show how to use the CMSIS-DSP library to accelerate its implementation on any microcontroller with an Arm Cortex-M CPU. After discussing on-device learning, we will tackle another problem: deploying scikit-learn models to microcontrollers. In this second part, we will demonstrate how to deploy generic ML algorithms trained with scikit-learn using the emlearn open-source project. The final question we will answer is about powering microcontrollers with batteries.

To get the most out of this book

For most of the chapters, with the exception of *Chapter 10, Deploying a CIFAR-10 Model for Memory-Constrained Devices with the Zephyr OS on QEMU*, you will need a computer (either a laptop or desktop) running Linux (preferably Ubuntu 20.04+), macOS, or Windows operating systems on an x86_64 architecture. Additionally, your computer should have a minimum of two USB ports.

In *Chapter 10*, you will specifically require a computer running either Linux (preferably Ubuntu 20.04+) or macOS on an x86_64 architecture.

It is worth noting that most projects can also be developed on Macs powered by Apple silicon, such as M1 or M2 chips. However, at the time of writing, there is no support for the SparkFun RedBoard Artemis Nano on Apple silicon devices.

The only software prerequisites for your computer are:

- Python (Python 3.7+)
- A text editor (for example, gedit on Ubuntu)
- A media player (for example, VLC)
- An image viewer (for example, the default image viewer in your OS)
- A web browser (for example, Google Chrome)

During our tinyML journey, we will require different software tools to cover ML development and embedded programming. Thanks to Arduino, Edge Impulse, and Google, these tools will be in the cloud, browser-based, and have a free plan for our usage.

You can develop projects on the Arduino Nano 33 BLE Sense and Raspberry Pi Pico directly in your web browser using the Arduino Web Editor (<https://create.arduino.cc>). However, at the time of writing, the Arduino Web Editor has a limit of 25 compilations per day. Therefore, you may consider upgrading to any paid plan or using the free local Arduino IDE (<https://www.arduino.cc/en/software>) to get unlimited compilations. For those interested in the free local Arduino IDE, we have provided the instructions to install the local Arduino IDE on GitHub (https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Docs/setup_local_arduino_ide.md).

For projects involving the SparkFun RedBoard Artemis Nano, you must use the local Arduino IDE. You can find the setup instructions for developing projects on this microcontroller by following this link: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Docs/setup_sparkfun_artemis_nano.md.

The projects we will develop together require sensors and additional electronic components to build realistic tinyML prototypes and experience the complete development workflow. These components are listed at the beginning of each chapter and in the `README.md` file within the corresponding chapter folder on GitHub.

Since we will build real electronic circuits, we require an electronic components kit with at least a solderless breadboard, colored LEDs, resistors, push-buttons, and jumper wires. Don't worry if you are a beginner in electronics. You will learn more about these components in the first two chapters of this book. Furthermore, we have prepared a beginner shopping list on GitHub so you know precisely what to buy: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Docs/shopping_list.md.

Download the example code files

The code bundle for the book is hosted on GitHub at https://github.com/PacktPublishing/TinyML-Cookbook_2E.

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

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781837637362>.

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “To do so, import the `os` Python module to use its `listdir()` method, which lists all the files in a specified directory.”

A block of code is set as follows:

```
def representative_data_gen():
    data = tf.data.Dataset.from_tensor_slices(x_test)
    for i_value in data.batch(1).take(100):
        i_value_f32 = tf.dtypes.cast(i_value, tf.float32)
        yield [i_value_f32]
```

Any command-line input or output is written as follows:

```
$ arduino-cli core install arduino:mbed_nano
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Scan the **quick response (QR)** code with your smartphone to pair the device with Edge Impulse.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packtpub.com/submit-errata>, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.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 <http://authors.packtpub.com>.

Share your thoughts

Once you've read *TinyML Cookbook, Third Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?
Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781837637362>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Getting Ready to Unlock ML on Microcontrollers

Here we are – on the first step that marks the beginning of our journey into the world of tinyML.

We will start this chapter by giving an overview of this rapidly emerging field, discussing the opportunities and challenges of bringing **machine learning (ML)** to low-power microcontrollers.

After this introduction, we will delve into the fundamental elements that make tinyML unique from traditional ML in the cloud, on desktops, or even on smartphones. We will revisit some basic ML concepts and introduce new fundamental ones specific to this domain, regarding power consumption and microcontroller development. Don't worry if you are new to embedded programming. In this chapter and the next, we will provide an introduction to microcontroller programming to ensure everyone has a solid foundation to get started.

Once we have presented the tinyML building blocks, we will focus on setting up a development environment for a simple but meaningful LED application, which will officially kick off our practical journey. In contrast to what we will find in the following chapters, this chapter has a more theoretical structure to get you familiar with the concepts and terminology of this fast-growing technology.

In this chapter, we will cover the following topics:

- Introduction to tinyML
- Overview of deep learning
- Learning the difference between power and energy

- Programming microcontrollers
- Introduction to the development platforms
- Setting up the software development environment
- Deploying a sketch on microcontrollers

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- An Arduino Nano 33 BLE Sense
- Raspberry Pi Pico
- A SparkFun Redboard Artemis Nano (optional)
- A micro-USB data cable
- A USB-C data cable (optional)
- Laptop/PC with either Linux, macOS, or Windows

Introduction to tinyML

Tiny machine learning, or, as we will refer to it, **tinyML**, is a technology that is gaining huge momentum in various fields, due to its ability to enable non-intrusive smartness. tinyML is not new, as it has already facilitated consumer electronics like smart speakers and smartwatches for many years. However, recent advances in hardware and software have made it more accessible and practical than ever. Therefore, it is no longer a niche technology.

There are at least three factors that make tinyML particularly appealing: *cost*, *energy*, and *privacy*.

The first benefit given by this technology is its *cost-effectiveness*. Devices used in tinyML are typically low-cost, ranging from a few cents to a few dollars in most cases. As a result, it is an affordable technology for businesses and individuals to drive innovation.

The second unique advantage of tinyML is its ability to run ML on *low-power* platforms.

The overall goal of tinyML is to allow smartness through low-power devices. This feature enables applications to operate on compact batteries such as coin cells or even plants (https://www.youtube.com/watch?v=_xELDU15_oE) for months, contributing to tackling energy challenges sustainably.

Privacy is the other factor that makes tinyML an attractive technology. While the internet provides tremendous opportunities, there is always a concern regarding user data exposure to unauthorized parties. The risks here could concern compromised privacy or personal identity theft to commit fraud, just to name a couple. tinyML can mitigate this issue by running ML algorithms on-device without sending data to the cloud.

As you may have noticed, so far, we have discussed why tinyML has the potential to enable ubiquitous intelligence. However, what is tinyML in practical terms?

What is tinyML?

tinyML encompasses the set of ML and embedded system technologies to enable the creation of intelligent applications for low-power devices. Generally, these devices have limited memory and processing power, but they are equipped with sensors to sense the physical environment and make decisions based on ML algorithms.

In tinyML, ML and the deployment platform are not independent entities but *entities that need to know each other at best*. Building an ML architecture without considering the target device capabilities will make it challenging to deploy effective applications. On the other hand, designing power-efficient processors to expand the ML capabilities of these devices would be impossible without knowing the software algorithms involved. Therefore, *we can only bring tremendous and compelling tinyML applications to life through a delicate balance between software and hardware*.

Throughout this book, we will explore tinyML with microcontrollers as target devices. Why microcontrollers, you ask? Well, let's just say that they are the perfect match for what we want, and in the following subsection, we will tell you why.

Why ML on microcontrollers?

The first and foremost reason for choosing microcontrollers is their *popularity* in various fields, such as automotives, consumer electronics, kitchen appliances, healthcare, and telecommunications. These devices are present in our day-to-day electronic devices, and with the emergence of the **Internet of Things (IoT)**, their market growth has been exponential.

Already in 2018, the market research company IDC (<https://www.idc.com>) reported 28.1 billion microcontrollers sold worldwide. Those are impressive numbers, considering that 1.5 billion smartphones and 67.2 million PCs were sold in the same year. Therefore, tinyML is a significant milestone in the evolution of IoT devices, paving the way for the proliferation of intelligent and connected low-power devices.

The other reasons for choosing microcontrollers are their *affordability*, *ease of programming*, and *ability to run sophisticated ML algorithms*, making them suitable for a wide range of applications.

However, these devices are generally connected to the internet in the IoT space. Therefore, if we can transmit data to a trusted cloud service, why can't we delegate the ML computation to it, given its superior performance? In other words, why do we need to run ML locally?

Why run ML on-device?

In addition to privacy, as discussed earlier, there are two other reasons to run ML locally:

- **Reducing latency:** Sending data back and forth to and from the cloud is not instant and could affect applications that must respond reliably within a time frame.
- **Reducing power consumption:** Sending and receiving data to and from the cloud is not power-efficient, even when using low-power communication protocols such as Bluetooth.

The following stacked bar chart shows the power consumption breakdown for the on-board components on the Arduino Nano 33 BLE Sense board, one of the microcontroller boards employed in this book:

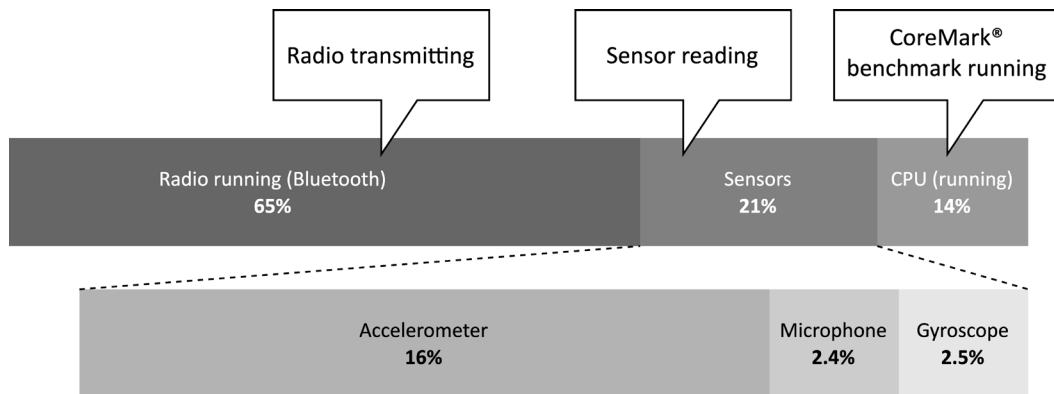


Figure 1.1: Power consumption breakdown for the Arduino Nano 33 BLE Sense board

Looking at the power consumption breakdown, we can observe that CPU computation uses less power than Bluetooth communication (14% versus 65%). As a result, it is preferable to compute more and transmit less to mitigate the risk of fast battery drain. Typically, the radio module, such as the one used for Bluetooth or other wireless communications, is the component that needs the most power in embedded devices.

Now that we know the benefits of running ML on these tiny devices, what are the practical opportunities and challenges?

The opportunities and challenges for tinyML

tinyML finds its natural home in applications where low power consumption is a critical requirement, such as when a device must operate with a battery for as long as possible.

If we think about it, we are already surrounded by battery-powered products that use ML under the hood. For example, wearable devices, such as smartwatches and fitness tracking bands, can recognize human activities to track our health goals or detect dangerous situations, such as a fall to the ground.

These products are based on tinyML for all intents and purposes because they need on-device ML on a low-power system to interpret sensor data continuously.

However, the use of battery-powered tinyML applications extends beyond wearable devices. For example, there are scenarios where we might need to monitor an environment to detect hazardous conditions, such as detecting fires to prevent them from spreading across a wide area.

There are unlimited use cases for tinyML, and the ones we briefly introduced are only a few.

However, despite the unlimited potential use cases for tinyML, some critical challenges must be addressed. The most significant challenges arise from the computational perspective of our devices, since they are often limited in memory and processing power. We work on systems with a few kilobytes of RAM and, in some cases, processors with no floating-point arithmetic acceleration. Furthermore, the deployment environment could be unfriendly. For example, environmental factors, such as dust and extreme weather conditions, could interfere during the normal execution of our applications.

As we have touched upon deployment environments briefly, let us delve deeper into them in the following subsection.

Deployment environments for tinyML

A tinyML application could live in both **centralized** and **distributed** systems.

In a **centralized** system, the application does not necessarily need to communicate with other devices. Nowadays, we interact with our smartphones, cameras, drones, and kitchen appliances seamlessly with our voices. For example, detecting the magic words “OK, Google,” “Alexa,” and so on in smart assistants is a tinyML application in every respect. In fact, this application can only run locally on a low-power system for a quick response and minimal power usage.

Usually, *centralized tinyML applications aim to trigger more power-hungry functionalities*, such as activating a media service.

In a **distributed** system, the device (that is, the **node** or **sensor node**) still performs ML locally but also communicates with nearby devices to achieve a common goal, as shown in *Figure 1.2*:

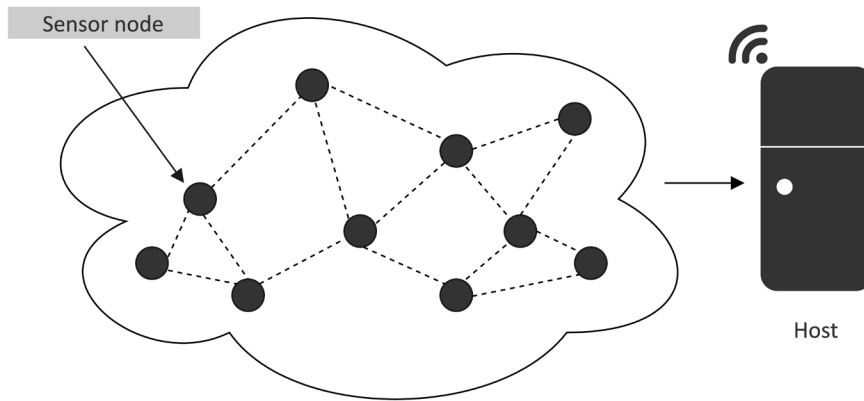


Figure 1.2: A wireless sensor network



Since the nodes are part of a network and typically communicate through wireless technologies, we commonly call the network a **wireless sensor network (WSN)**.

While this scenario may appear to conflict with the power consumption implications of transmitting data, devices may still need to collaborate to obtain meaningful knowledge about their working environment. In fact, specific applications may require a holistic understanding of the distribution of physical quantities, such as temperature, humidity, and soil moisture, rather than knowing the values from a particular node.

For example, consider an application to improve agriculture efficiency. In this scenario, a WSN could assist in identifying areas of the field that require more water than others. In fact, by gathering and analyzing data from multiple nodes across the field, a network can provide a comprehensive understanding of the soil moisture levels, helping farmers reduce their water usage. But that's not all. Efficient communication protocols are crucial for the network's lifetime. Therefore, we may think of using tinyML to make them more effective. Since sending raw data consumes too much energy, ML could perform a partial computation to reduce the data to transmit and the frequency of communications.

tinyML presents endless possibilities, and the few mentioned are a small fraction of what is achievable. For those seeking to expand their knowledge and skills in this field, tinyML Foundation is the ideal community to join.

Join the tinyML community!

tinyML Foundation (www.tinyml.org) is a non-profit organization that aims to educate, inspire, and connect the worldwide tinyML community.

Supported by companies such as Arm, Edge Impulse, Google, and Qualcomm, the foundation is energizing a diverse global community of engineers, scientists, academics, and business professionals to envision a *world of ubiquitous devices powered by tinyML to create a healthier and sustainable environment*.

Through free virtual and in-person initiatives, the tinyML Foundation promotes knowledge sharing, engagement, and connection among experts and newcomers. In 2023, over 13,000 people joined the group, and there have been 47 Meetup groups in 39 countries.



With several Meetup (<https://www.meetup.com>) groups in different countries, you can join any near you for free (<https://www.meetup.com/en-AU/pro/TinyML/>) to always be up to date with new tinyML technologies and upcoming events.

After this brief introduction to tinyML, it is time to explore its ingredients in more detail. The following section will start analyzing the element that makes our devices capable of intelligent decisions.

Overview of deep learning

ML is the ingredient that makes our tiny devices capable of making intelligent decisions. These software algorithms heavily rely on the correct data to learn patterns or actions based on experience. As we commonly say, *data is everything for ML because it is what makes or breaks an application*.

This book will refer to **deep learning (DL)** as a specific class of ML that can perform complex prediction tasks directly on raw images, text, or sound. These algorithms have state-of-the-art accuracy and can be better and faster than humans in solving some data analysis problems.

A complete discussion of DL architectures and algorithms is beyond the scope of this book. However, this section will summarize some essential points relevant to understanding the following chapters.

Deep neural networks

A deep neural network consists of several stacked layers aimed at learning patterns.

Each layer contains several neurons, the fundamental computing elements for **artificial neural networks (ANNs)** inspired by the human brain.

A neuron produces a single output through a linear transformation, defined as the weighted sum of the inputs plus a constant value called **bias**, as shown in the following diagram:

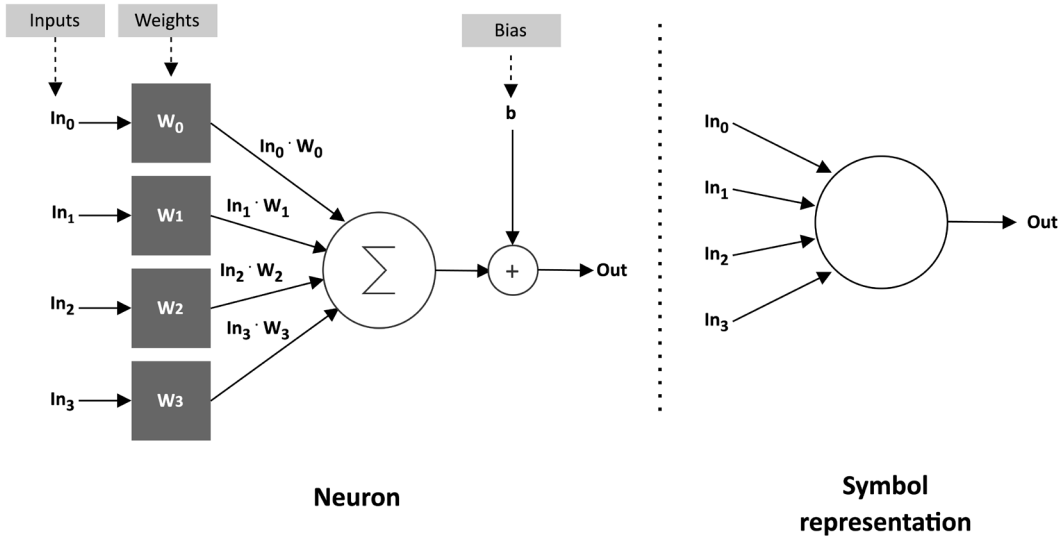


Figure 1.3: A neuron representation

The coefficients of this weighted sum are called **weights**.

Weights and bias are obtained after an iterative training process to make the neuron capable of learning complex patterns. However, neurons can only solve simple linear problems with linear transformations. Therefore, non-linear functions, called **activations**, generally follow the neuron's output to help the network learn complex patterns:

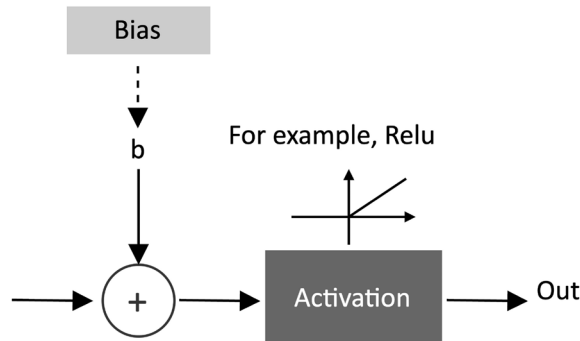


Figure 1.4: An activation function

An example of a widely adopted activation function is the **rectified linear unit (ReLU)**, which returns the maximum value between the input value and 0:

```
float relu(float input) {  
    return max(input, 0);  
}
```

Its computational simplicity makes it preferable to other non-linear functions, such as a hyperbolic tangent or logistic sigmoid, requiring more computational resources.

In the following subsection, we will see how the neurons are connected to solve complex visual recognition tasks.

Convolutional neural networks

Convolutional neural networks (CNNs) are specialized deep neural networks predominantly applied to visual recognition tasks.

We can consider CNNs as the evolution of a regularized version of the **classic fully connected neural networks** with dense layers, also known as **fully connected layers**.

As we can see in the following diagram, a characteristic of fully connected networks is connecting every neuron to all the output neurons of the previous layer:

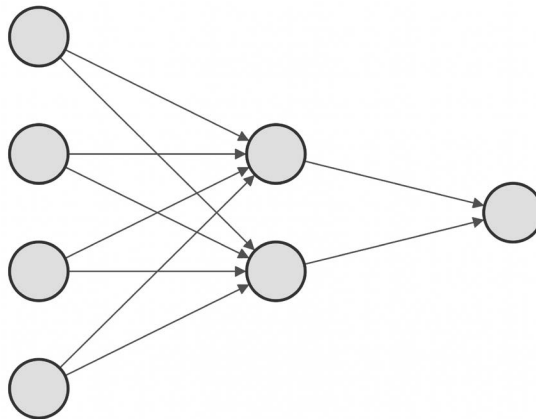


Figure 1.5: A fully connected network

Unfortunately, this method of connecting neurons does not work well for training a model for image classification.

For instance, if we considered an RGB image of size 320x240 (width x height), we would need 230,400 ($320 \times 240 \times 3$) weights for just one neuron. Since our models will undoubtedly need several layers of neurons to discern complex problems, the model will likely **overfit**, given the unmanageable number of trainable parameters. Overfitting implies that the model learns to predict the training data well but struggles to generalize data not used during the training process (**unseen data**).

In the past, data scientists adopted manual *feature engineering techniques* to extract a reduced set of good features from images. However, the approach suffered from being difficult, time-consuming, and domain-specific.

With the rise of CNNs, visual recognition tasks saw improvement thanks to **convolution layers**, which make feature extraction part of the learning problem.

Based on the assumption that we are dealing with images and inspired by biological processes in the animal visual cortex, the convolution layer borrows the widely adopted convolution operator from image processing to create a set of learnable features.

The convolution operator is performed similarly to other image processing routines: sliding a window application (filter or kernel) onto the entire input image and applying the dot product between its weights and the underlying pixels, as shown in *Figure 1.6*:

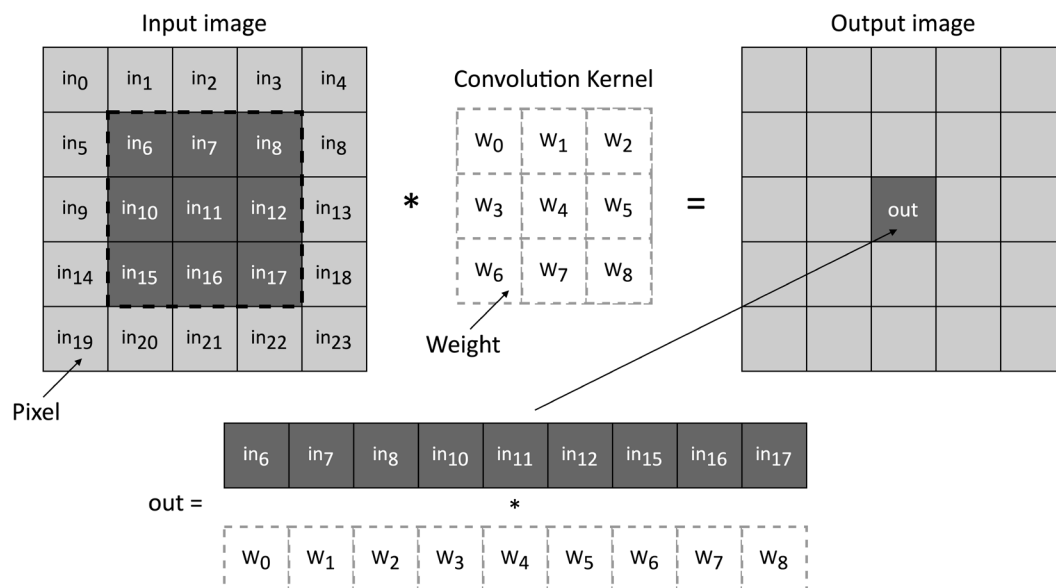


Figure 1.6: Convolution operator

This approach brings two significant benefits:

- It extracts the relevant features automatically without human intervention.
- It reduces the number of input signals per neuron considerably.

For instance, applying a 3x3 filter on the preceding RGB image would only require 27 weights ($3 \times 3 \times 3$).

Like fully connected layers, convolution layers need several kernels to learn as many features as possible. Therefore, the convolution layer's output generally produces a set of images (**feature maps**), commonly kept in a multidimensional memory object called a **tensor**, as shown in the following illustration:

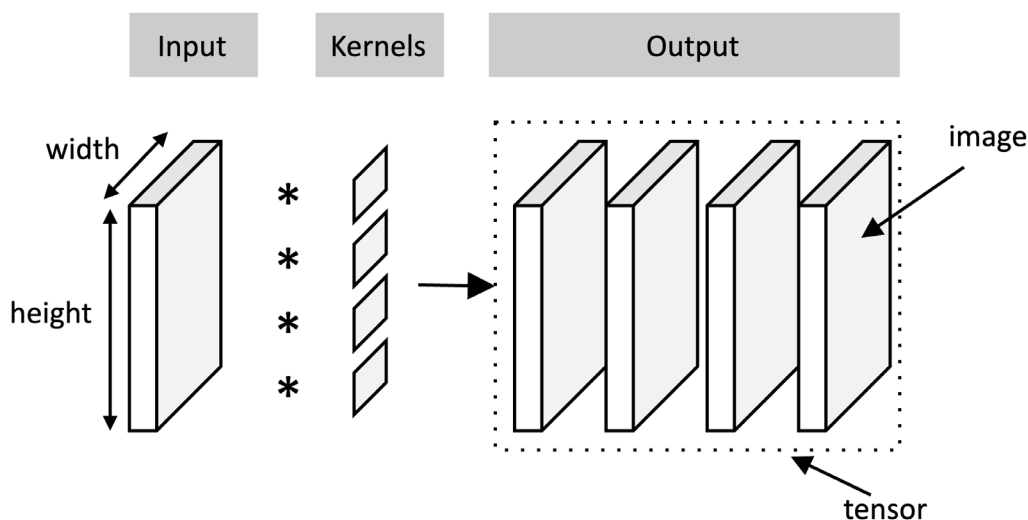


Figure 1.7: Representation of a 3D tensor

Traditional CNNs for visual recognition tasks usually include the fully connected layers at the network's end to carry out the prediction stage. Since the output of the convolution layers is a set of images, we generally adopt subsampling strategies to reduce the information propagated through the network and the risk of overfitting when feeding the fully connected layers.

Typically, there are two ways to perform subsampling:

- Skipping the convolution operator for some input pixels. As a result, the output of the convolution layer will have fewer spatial dimensions than the input ones.
- Adopting subsampling functions such as **pooling layers**.

The following figure shows a generic CNN architecture, where the pooling layer reduces the spatial dimensionality, and the fully connected layer performs the classification stage:

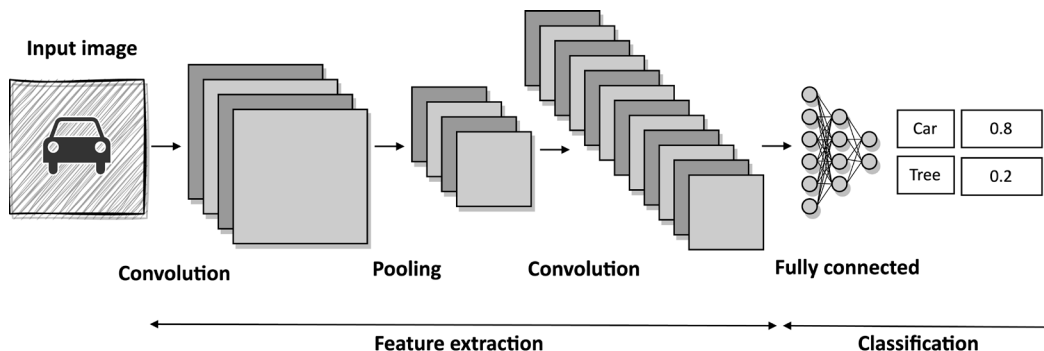


Figure 1.8: Traditional CNN with a pooling layer to reduce the spatial dimensionality

When developing DL networks for tinyML, one of the most crucial factors is the model's size, defined as the number of trainable weights. Due to the limited physical memory of our platforms, the model needs to be compact to fit the target device. However, memory constraints are not the only challenge we may face. For instance, while trained models often use floating-point precision arithmetic operations, the CPUs on our platforms may lack hardware acceleration.

Thus, to overcome these limitations, quantization becomes an indispensable technique.

Model quantization

Quantization is the process of performing neural network computations in lower bit precision. The widely adopted technique for microcontrollers applies the quantization post-training and converts the 32-bit floating-point weights to 8-bit integer values. This technique brings a 4x model size reduction and a significant latency improvement with little or no accuracy drop.

Other techniques like **pruning** (setting weights to zero) or **clustering** (grouping weights into clusters) can help reduce the model size. However, in this book, we will limit the scope to the quantization technique because it is sufficient to showcase the model deployment on microcontrollers.



If you are interested in learning more about pruning and clustering, you can refer to the following practical blog post, which shows the benefit of these two techniques on the model size: <https://community.arm.com/arm-community-blogs/b/ai-and-ml-blog/posts/pruning-clustering-arm-ethos-u-npu>.

As we know, ML is the component that allows smartness into our application. Nevertheless, to ensure the longevity of battery-powered applications, it is essential to use low-power devices. So far, we have mentioned power and energy in general terms, but let's see what they mean practically in the following section.

Learning the difference between power and energy

Power matters in tinyML, and its target is in the **milliwatt (mW)** range or below, which means thousands of times more efficient than a traditional desktop machine.

Although there are cases where we might consider using **energy** harvesting solutions, such as solar panels, those could not always be possible because of cost and physical dimensions.

However, what do we mean by power and energy? Let's discover these terms by giving a basic overview of the fundamental physical quantities governing electronic circuits. This knowledge will be crucial for building electronic circuits with microcontrollers in the following chapters.

Voltage versus current

Current is what makes an electronic circuit work, which is the *flow of electric charges across surface A of a conductor in a given time*, as described in the following diagram:

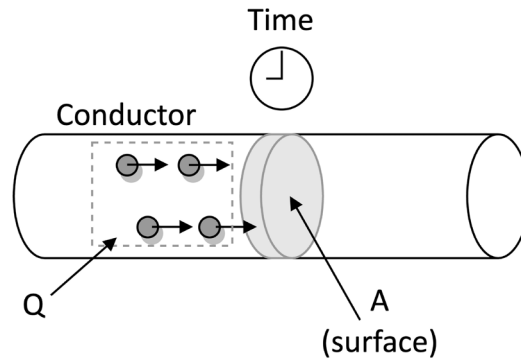


Figure 1.9: Current is a flow of electric charges across surface A at a given time

The current is defined as follows:

$$I = \frac{Q}{t}$$

Here, we have the following:

- I : Current, measured in **amperes (A)**
- Q : The electric charges across surface A in a given time, measured in **coulombs (C)**
- t : Time, measured in **seconds (s)**

The current flows in a circuit under the following conditions:

- We have a *conductive material* (for example, copper wire) to allow the electric charge to flow.
- We have a *closed circuit*, so a circuit without interruption provides a continuous path to the current flow.
- We have a *source of energy*, which is a *potential difference source* called **voltage**.

The voltage is measured with **volts (V)** and *produces an electric field to allow the electric charge to flow in the circuit*. Both the USB port and battery are potential difference sources. The symbolic representation of a power source is given in the following figure:

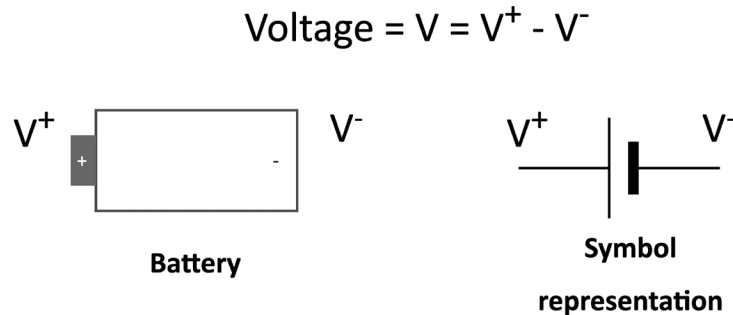


Figure 1.10: Battery symbol representation



To avoid constantly referring to V^+ and V^- , we will define the battery's negative terminal as a reference by convention, assigning it **0 V (GND)**.

Ohm's law relates voltage and current, which says through the following formula that *the current through a conductor is proportional to the voltage across a resistor*:

$$I = \frac{V}{R}$$

A **resistor** is an *electrical component used to reduce the current flow*. This component, whose symbolic representation is reported in the following figure, has a **resistance** measured with **Ohm (Ω)** and identified with the letter R:



Figure 1.11: Resistor symbol representation

Resistors are essential components for any electronic circuit, and for those used in our projects, their value is reported through colored bands on the elements. Standard resistors have four, five, or six bands. The color on the bands denotes the resistance value, as illustrated in the following example via the different shades:

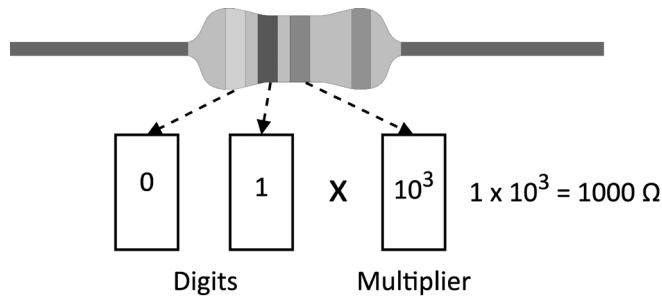


Figure 1.12: Example of a four-band resistor



To easily decode the color bands, we recommend using the online tool at **Digi-Key** (<https://www.digikey.com/en/resources/conversion-calculators/conversion-calculator-resistor-color-code>).

With an understanding of the main physical quantities governing electronic circuits, we are now prepared to talk about the difference between power and energy.

Power versus energy

Sometimes, we interchange the words power and energy because we believe they are the same. However, although they are related, they represent distinct physical quantities. **Energy** is the *capacity for doing work* (for example, using force to move an object), while **power** is the *energy consumption rate*.

In practical terms, power indicates *how fast we drain the battery*, so high power implies a faster discharge.

Power and energy are related to voltage and current through the following formulas:

$$P = V \cdot I$$

$$E = P \cdot T$$

The following table presents the physical quantities reported in the power and energy formulas:

Physical quantity	Unit	Meaning
P	Watt (W)	Power
E	Joule (J)	Energy
V	Volts (V)	Voltage supply
I	Ampere (A)	Current consumption
T	Seconds (s)	Operating time

Figure 1.13: Table reporting the physical quantities in the power and energy formulas

On microcontrollers, the voltage supply is in the order of a few volts (for example, 3.3 V), while the current consumption is in the range of **microampere** (μA) or **milliampere** (**mA**). For this reason, we commonly refer to **microwatt** (μW) or **milliwatt** (**mW**) for power and **microjoule** (μJ) or **millijoule** (**mJ**) for energy.

Now, consider the following problem to familiarize yourself with the presented concepts.

Suppose you have a processing task, and you have the option to run it on two different processors with the following power consumptions in the active state:

Processing unit	Power consumption
PU1	12
PU2	3

Figure 1.14: Table reporting two processing units with different power consumptions

What processor would you use to run the task?

Although PU1 has higher (4x) power consumption than PU2, this does not imply that PU1 is less energy efficient. On the contrary, PU1 could be more computationally performant than PU2 (for example, 8x), making it the best choice from an energy perspective, as demonstrated by the following calculations:

$$\begin{aligned} E_{PU1} &= 12 \cdot T_1 \\ E_{PU2} &= 3 \cdot T_2 = 3 \cdot 8 \cdot T_1 = 24 \cdot T_1 \end{aligned}$$

Based on the preceding example, we can conclude that PU1 is our better choice because it needs less energy from the battery under the same workload.



Commonly, we adopt **OPS per Watt** (arithmetic operations performed per Watt) to bind the power consumption to the computational resources of our processors.

In terms of power and energy concepts, that is all we need to know about it. Therefore, the only remaining aspect to discuss concerns the devices used for our tinyML projects: the microcontrollers.

Programming microcontrollers

A **microcontroller**, often shortened to **MCU**, is a full-fledged computer because it consists of a processor (which can also be multicore nowadays), a memory system, and some peripherals. Unlike a standard computer, a microcontroller fits entirely on an integrated chip, is incredibly low-power, and is inexpensive.

We often confuse microcontrollers with microprocessors, but they refer to different devices. In contrast to a microcontroller, a **microprocessor** integrates only the processor on a chip, requiring external connections to a memory system and other components to form a fully operating computer.

The following figure summarizes the main differences between a microprocessor and a microcontroller:

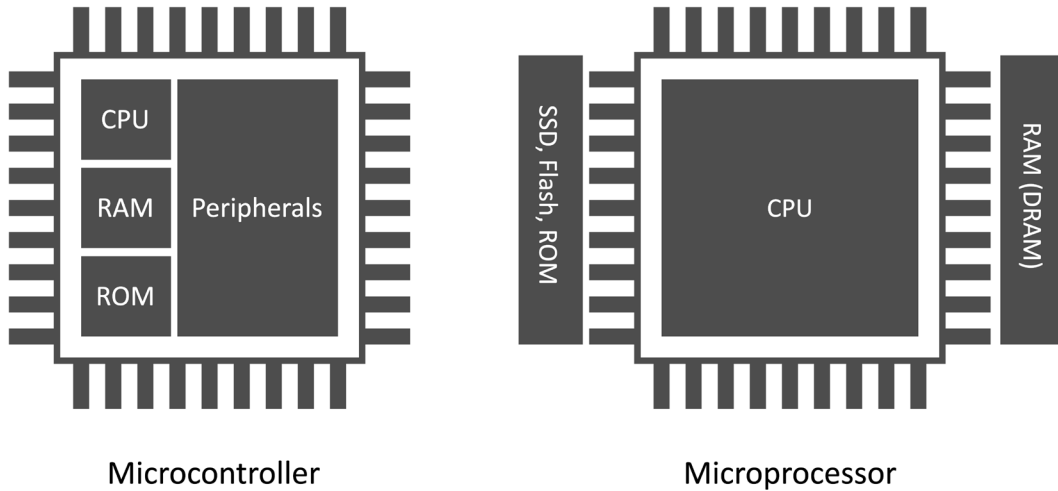


Figure 1.15: Microprocessor versus microcontroller

As for all processing units, the target application influences their architectural design choice.

For example, a microprocessor tackles scenarios where the tasks are usually as follows:

- Dynamic, which means they can change with user interactions or time
- General-purpose
- Compute-intensive

A microcontroller addresses completely different scenarios, as the applications can:

- Be single-purpose and repetitive
- Have time frame constraints
- Be battery-powered
- Need to fit in a small physical space
- Be cost-effective

Tasks are generally single-purpose and repetitive. Therefore, the microcontroller does not require strict re-programmability. Typically, microcontroller *applications are less computationally intensive* than microprocessor ones and do not have frequent interactions with the user. However, they can interact with the environment or other devices. As an example, consider the thermostat.

The device only requires monitoring the temperature regularly and communicating with the heating system.

Sometimes, tasks *must be executed within a specific time frame*. This requirement is characteristic of **real-time applications (RTAs)**, where the violation of the time constraint may affect the quality of service (**soft real time**) or be hazardous (**hard real time**). A car's **anti-lock braking system (ABS)** is an example of a hard RTA because the electronic system must respond within a time frame to prevent the wheels from locking when applying brake pedal pressure.

RTA applications require a **latency-predictable device**, so all hardware components (CPU, memory, interrupt handler, and so on) must respond in a precise number of clock cycles.



Hardware vendors commonly report latency in the datasheet, expressed in clock cycles.

The time constraint poses some architectural design adaptations and limitations for a general-purpose microprocessor. For instance, the **memory management unit (MMU)**, used to translate virtual memory addresses, is generally not integrated into CPUs for microcontrollers.

Microcontroller applications can be *battery-powered*, as the device has been designed to be low-power. As per the time frame constraints, power consumption also poses some architectural design differences from a microprocessor. Without going deeper into the hardware details, *all the off-chip components generally reduce power efficiency as a rule of thumb*. That is the main reason microcontrollers typically integrate memories within a chip.



Microcontrollers typically have lower clock frequency than microprocessors to consume less energy.

Microcontrollers are also an ideal choice for building products that need a *compact physical footprint* and *cost-effectiveness*. Since these devices are computers within a chip, the package size is typically a few square millimeters and is economically more advantageous than microprocessors.

In the following table, we have summarized what we have just discussed for easy future reference:

Feature	Microprocessor	Microcontroller
Application	General-purpose	Single-purpose
CPU arithmetic	It can perform heavy mathematical calculations in floating-point or double precision	Mainly integer arithmetic
RAM	A few GB	A few hundred KB
ROM (or hard-drive)	GB or TB	KB or MB
Clock frequency	GHz	MHz
Power consumption	W	mW or below
Operating System (OS)	Required	Not strictly required
Cost	From ten to hundreds of dollars	From a few cents (low-end) to a few dollars (high-end)

Figure 1.16: Table comparing a microprocessor with a microcontroller

In the next section, we will go deeper into microcontrollers’ architectural aspects by analyzing the memory architecture and internal peripherals crucial for ML model deployment.

Memory architecture

Microcontrollers are CPU-based embedded systems, meaning the *CPU is responsible for interacting with all its subcomponents*.

All CPUs require at least one memory to read the instructions and store/read variables during the program’s execution. In the microcontroller context, we typically dedicate two separate memories for the instructions and data: **program** and **data memory**.

Program memory is non-volatile **read-only memory (ROM)** reserved for the program to execute. Although its primary goal is to contain the program, it can also store constant data. Thus, program memory is similar to our everyday computers’ hard drives.

Data memory is **volatile memory** reserved to store/read temporary data. Therefore, it operates similarly to RAM in a personal computer, as its contents are lost when switching off the system.

Given the different program and data memory requirements, we usually employ other semiconductor technologies. In particular, we can find flash technologies for the program memory and **static random-access memory (SRAM)** for the data memory.

Flash memories are non-volatile and offer low power consumption but are generally slower than SRAM. However, given the cost advantage over SRAM, we can find larger program memory than data memory.

Now that you know the difference between program and data memory, *where would you store the weights for a deep neural network model?*

The answer to this question depends on whether the model has constant weights. If the weights are constant during inference, it is more efficient to store them in program memory for the following reasons:

- Program memory has more capacity than SRAM.
- It reduces memory pressure on the SRAM, since other functions require storing variables or chunks of memory at runtime.

We want to remind you that microcontrollers have limited memory resources, so a decision like this can significantly reduce SRAM memory usage.

Microcontrollers offer extra on-chip features to expand their capabilities and make these tiny computers different from each other. These features are the **peripherals**, which are discussed in the upcoming subsection.

Peripherals

Peripherals are essential in microcontrollers to interface with sensors or other external components.

Each peripheral has a dedicated functionality and is assigned to a metal leg (**pin**) of the integrated circuit.



You can refer to the *peripheral pin assignment* section in the microcontroller datasheet to find out each pin's functionalities.

Hardware vendors typically number the pins anti-clockwise, starting from the top-left corner of the chip, marked with a dot for easy reference, as shown in *Figure 1.17*:

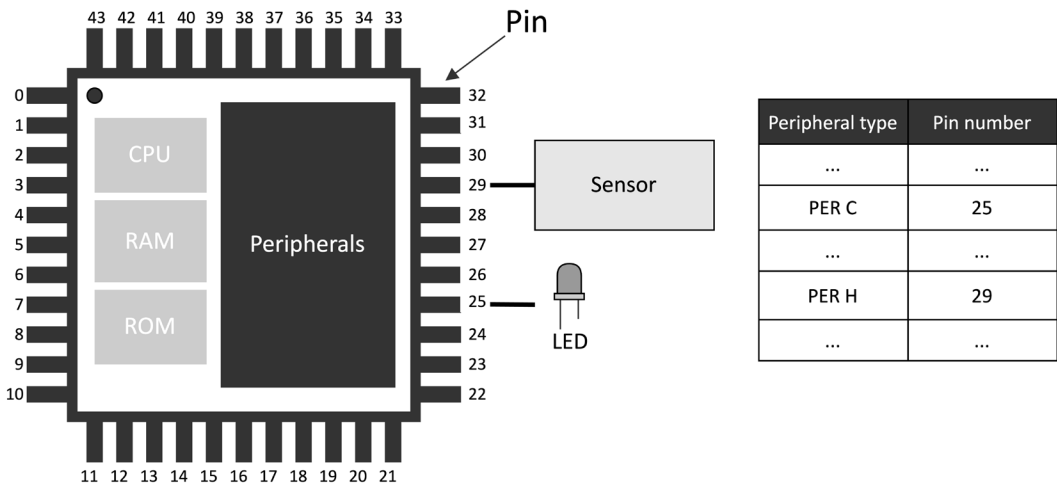


Figure 1.17: Viewed from the top, pins are numbered anti-clockwise, starting from the top-left corner, marked with a dot

Peripherals can be of various types, and the following subsection will provide a brief overview of those commonly integrated into microcontrollers.

General-purpose input/output (GPIO or IO)

GPIOs do not have a predefined and fixed purpose. Their primary function is to provide or read **binary signals** that, by nature, can only live in two states: **HIGH (1)** or **LOW (0)**. The following figure shows an example of a binary signal:

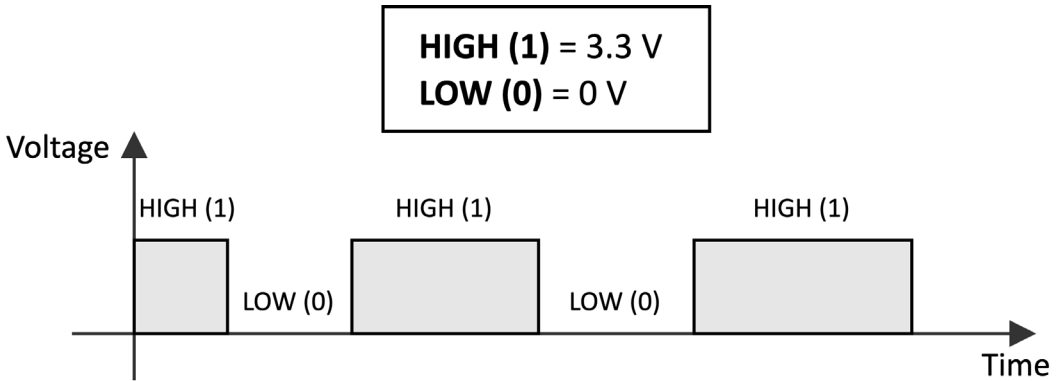


Figure 1.18: Binary signal

Typical GPIO usages are as follows:

- Turning on and off an LED
- Detecting whether a button is pressed
- Implementing complex digital interfaces/protocols such as VGA

GPIO peripherals are versatile and generally available in all microcontrollers. We will use this peripheral often, such as turning on and off LEDs or detecting whether a button has been pressed.

Analog/digital converters

When developing tinyML applications, we will likely deal with time-varying physical quantities, such as images, audio, and temperature.

Whatever these quantities are, the **sensor** transforms them into a *continuous electrical signal* interpretable by the microcontrollers. This electrical signal, which can be either a voltage or current, is called an **analog signal**.

The microcontroller, in turn, needs to convert the analog signal into a digital format so that the CPU can process the data.

Analog/digital converters act as *translators* between analog and digital worlds. Thus, we have the **analog-to-digital converter (ADC)** that converts the electrical signal into a digital format, and the **digital-to-analog converter (DAC)**, which performs the opposite functionality.

In this book, we will use this peripheral to transform the analog signal the microphone generates into a digital format.

Serial communication

Communication peripherals integrate standard communication protocols to control external components. Typical serial communication peripherals available in microcontrollers are **I2C**, **SPI**, **UART** (commonly called **serial**), and **USB**.

The serial peripheral will be used extensively in our projects to transmit messages from the microcontroller to our computer (we'll refer to this communication as over the serial throughout this book). For example, we will use this peripheral to debug our applications and generate media files.

Timers

In contrast to all the peripherals we just described, **timers** do not interface with external components, since they are used to trigger or synchronize events. For example, a timer can be set up to acquire data from a sensor at a specific time interval.

Having covered the topic of peripherals, we have completed our overview of the tinyML ingredients. With a grasp of the relevant terminology and fundamental concepts about ML, power/energy consumption, and microcontrollers, we can now introduce the development platforms used in this book.

Introduction to the development platforms

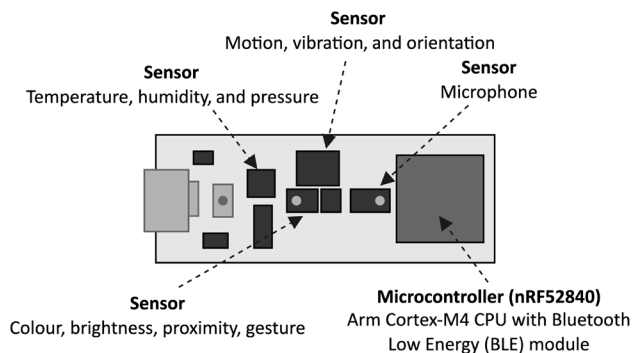
The development platforms used in this book are microcontroller boards. A **microcontroller board** is a **printed circuit board (PCB)** that combines a microcontroller with the necessary electronic circuit to make it ready for use. In some cases, these platforms could also include additional devices, such as sensors or additional external memory, to target specific end applications.

The **Arduino Nano 33 BLE Sense** (Arduino Nano for short), **Raspberry Pi Pico**, and the **SparkFun RedBoard Artemis Nano** (SparkFun Artemis Nano for short) are the microcontroller boards used in this book.

As we will see in more detail in the upcoming subsections, the platforms have an incredibly small form factor, a USB port for power/programming, and an Arm-based microcontroller. At the same time, they also have unique features that make them ideal for targeting different development scenarios.

Arduino Nano 33 BLE Sense

The Arduino Nano, designed by Arduino (<https://www.arduino.cc>), is a versatile platform suitable for various tinyML applications. It integrates the nRF52840 microcontroller, powered by an Arm Cortex-M4 CPU that runs at 64 MHz, as well as 1 MB of program memory and 256 KB of data memory, along with various sensors and a Bluetooth radio:



Arduino Nano 33 BLE Sense

CPU: Arm Cortex-M4 @ 64MHz
Program memory: 1MB
Data memory: 256KB
Board size: 45x18mm
Cost: 31.10\$

Figure 1.19: Arduino Nano board

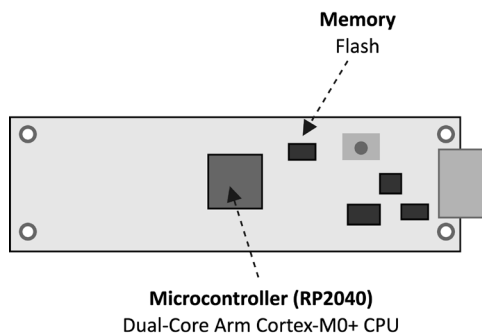
When developing on the Arduino Nano, we only need to add a few additional external components, as most are already on-board.



The Arduino Nano 33 BLE Sense underwent an upgrade to the Rev2 version in 2023. This updated version retains the same form factor and processor as the Rev1 but includes enhanced sensors to cover a broader range of applications. The projects featured in this book are compatible with both the Rev1 and Rev2 versions.

Raspberry Pi Pico

Raspberry Pi Pico, designed by **Raspberry Pi** (<https://www.raspberrypi.org>), does not provide sensors and the Bluetooth module on-board. Still, it has the **RP2040** microcontroller powered by a dual-core Arm Cortex-M0+ processor, running at 133 MHz with 264 KB of SRAM. The device boasts an external flash memory of 2 MB for the program, making it an excellent choice for tinyML applications that require speed and memory space:



Raspberry Pi Pico

CPU: Dual core Arm Cortex-M0+ @ 133MHz
Program memory: 2MB (external)
Data memory: 264KB
Board size: 51.3x21mm
Cost: 4\$

Figure 1.20: Raspberry Pi Pico board

In this book, this board will be ideal for learning how to interface with external sensors and build electronic circuits.

SparkFun RedBoard Artemis Nano

The **SparkFun RedBoard Artemis Nano**, designed by **SparkFun Electronics** (<https://www.sparkfun.com/>), is a platform that integrates the **Apollo3** microcontroller, powered by an Arm Cortex-M4F processor running at 48 MHz with 1 MB of program memory and 384 KB of data memory.

The platform also boosts a digital microphone, making it ideal for those interested in developing always-on voice command applications:

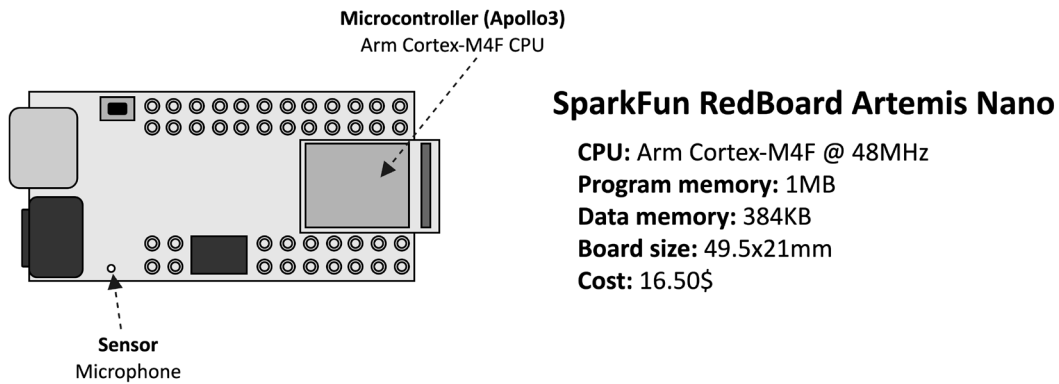


Figure 1.21: SparkFun RedBoard Artemis Nano

This platform is optional but recommended to grasp the concepts presented in the recipes for the Arduino Nano and Raspberry Pi Pico, using an alternative device.

This book will not include a comprehensive discussion about projects for the SparkFun RedBoard Artemis Nano. However, when you come across the *There's more...with the SparkFun Artemis Nano!* section at the end of a recipe, you can find the instructions to replicate it on this device.



Although the book will not discuss projects for the SparkFun RedBoard Artemis Nano, the source code for this platform will be accessible on GitHub.

Setting up the software development environment

To develop tinyML applications, we require different software tools and frameworks to cover both ML development and embedded programming.

In the following subsection, we will start by introducing the Arduino development environment used to write and upload programs to the Arduino Nano, Raspberry Pi Pico, and the SparkFun RedBoard Artemis Nano.

Getting ready with Arduino IDE

Arduino Integrated Development Environment (Arduino IDE) is a software application developed by Arduino (<https://www.arduino.cc/en/software>) to write and upload programs to Arduino compatible boards.



The Arduino Nano, Raspberry Pi Pico, and SparkFun RedBoard Artemis Nano are Arduino compatible boards.

Programs are written in C++ and are commonly called **sketches** by Arduino programmers.

Arduino IDE makes software development accessible and straightforward to developers with no background in microcontroller programming. In fact, the tool abstracts all the complexities we might have when dealing with these platforms, such as cross-compilation and device programming.



To download, install, and set up the Arduino IDE on your computer, you can follow the instructions provided at the following link: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Docs/setup_local_arduino_ide.md.

In addition to the standalone version, Arduino offers a browser-based IDE called the **Arduino Web Editor** (<https://create.arduino.cc/editor>). The Arduino Web Editor enables even more streamlined programmability, as programs can be written, compiled, and uploaded directly from the web browser to microcontrollers.

To install the Arduino Web Editor, you can follow the guide available on the Arduino website: <https://docs.arduino.cc/learn/starting-guide/the-arduino-web-editor>.



The free version of the Arduino Web Editor has a daily compilation time limit of 200 seconds. Therefore, users may want to upgrade to a paid plan or use the free local Arduino IDE to avoid the compilation time constraint and have unlimited compilation time.

The Arduino projects presented in this book for the Arduino Nano and Raspberry Pi Pico are compatible with both IDEs, although the screenshots exclusively showcase the cloud-based Arduino Web Editor. However, the SparkFun RedBoard Artemis Nano projects can only be developed using the local Arduino IDE.



To install the SparkFun RedBoard Artemis Nano board in the Arduino IDE, you must follow the instructions provided at the following link: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Docs/setup_sparkfun_artemis_nano.md.

From now on, we will use the term Arduino IDE interchangeably for both the Arduino Web Editor and the local Arduino IDE. However, when mentioning the SparkFun RedBoard Artemis Nano, the Arduino IDE will specifically denote the local version.

Having introduced the development environment for microcontroller programming, let's now introduce the framework and software environment to train ML models, which are TensorFlow and Google Colaboratory.

Getting ready with TensorFlow

TensorFlow (<https://www.tensorflow.org>) is an end-to-end free and open-source software platform developed by Google for ML. We will use this software to build and train our ML models, using Python in Google Colaboratory.

Colaboratory (<https://colab.research.google.com/notebooks>) – **Colab** for short– is a free Python development environment that runs in the browser using Google Cloud. It is like a Jupyter notebook but has some essential differences, such as the following:

- It does not need setting up.
- It is cloud-based and hosted by Google.
- There are numerous Python libraries pre-installed (including TensorFlow).
- It is integrated with Google Drive.
- It offers free access to GPU and TPU shared resources.
- It is easy to share (also on GitHub).

Therefore, TensorFlow does not require setting up because Colab comes with it.

In Colab, we recommend enabling the GPU acceleration on the **Runtime** tab to speed up the computation on TensorFlow. To do so, navigate to **Runtime** | **Change runtime type**, and select **GPU** from the **Hardware accelerator** drop-down list, as shown in *Figure 1.22*:

Notebook settings

Hardware accelerator

GPU  

To get the most out of Colab Pro, avoid using a GPU unless you need one. [Learn more](#)

Figure 1.22: Hardware accelerator drop-down list

Since the GPU acceleration is a shared resource among other users, there is limited access to the free version of Colab.



You could subscribe to Colab Pro (<https://colab.research.google.com/>) to get priority access to the fastest GPUs.

TensorFlow is not the only software from Google that we will use. In fact, once we have produced the ML model, we will need to run it on the microcontroller. For this, Google developed TensorFlow Lite for Microcontrollers.

TensorFlow Lite for Microcontrollers (<https://www.tensorflow.org/lite/microcontrollers>) – **tf-lite-micro** for short – is the crucial software library to unlock ML applications on low-power microcontrollers. The project is part of TensorFlow and allows you to run DL models on devices with a few KB of memory. Written in C/C++, the library does not require an operating system and dynamic memory allocation.

To build a tf-lite-micro-based application into any Arduino project, you first need to create the **Arduino TensorFlow Lite library** (<https://github.com/tensorflow/tf-lite-micro-arduino-examples>) and then import it into the Arduino IDE.

For your convenience, we have already produced this library, which is compatible with the Arduino Nano, Raspberry Pi Pico, and SparkFun RedBoard Artemis Nano and is available at the following link: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/ArduinoLibs/Arduino_TensorFlowLite.zip.

At the moment, you do not need to import this library. When it is time to deploy the ML models on microcontrollers, we will guide you through the precise steps to import the library into the Arduino IDE.



For those interested in the process of creating the Arduino TensorFlow Lite library, we have outlined the steps on GitHub, which can be found at the following link: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Docs/build_arduino_tflitemicro_lib.md.

In this book, TensorFlow won't be our only avenue to design and train ML models. Another framework will accompany us in preparing ML models for microcontrollers. This framework is Edge Impulse.

Getting ready with Edge Impulse

Edge Impulse (<https://www.edgeimpulse.com>) is an all-in-one software platform for ML development from data acquisition to model deployment. It is free for developers, and in a few minutes, we can have an ML model up and running on our microcontrollers. This platform features a wide range of integrated tools for the following:

- Data acquisition from sensor data
- Data labeling
- Applying digital signal processing routines on the input data
- Designing, training, and testing ML models via a user-friendly interface
- Deploying ML models on microcontrollers
- AutoML

Developers just need to sign up on the Edge Impulse website to access all these features directly within the **user interface (UI)**.

We are approaching the end of this first chapter. However, before we wrap up, we want to ensure we can successfully run a basic sketch on our microcontrollers. Therefore, in the upcoming section, we will build a simple Arduino pre-built application, marking the beginning of our journey into tinyML.

Deploying a sketch on microcontrollers

Following the introductory section, we will delve into our first recipe to familiarize ourselves with the Arduino IDE and better understand how to compile and upload a sketch on an Arduino platform. To accomplish this objective, we will use a pre-built Arduino sketch to blink the LED on our microcontroller boards.

Getting ready

An Arduino sketch consists of two functions, `setup()` and `loop()`, as shown in the following code block:

```
void setup() {  
}  
  
void loop() {  
}
```

The `setup()` function is the first function executed by the program when we press the reset button or power up the board. This function is executed only once and is generally responsible for initializing variables and peripherals.

After the `setup()` function, the program executes the `loop()` one, which runs iteratively and forever, as illustrated in the following diagram:

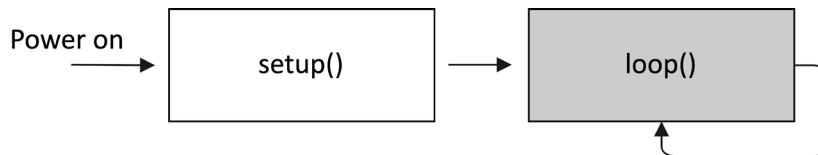


Figure 1.23: The `setup()` function runs once

These two functions are required in all Arduino programs.

How to do it...

Open the Arduino IDE, and follow the steps to make the on-board LED of our microcontroller boards blink:

Step 1:

Connect either the Arduino Nano or Raspberry Pi Pico to a laptop/PC through the micro-USB data cable. Next, check that the Arduino IDE reports the board's name and serial port in the device drop-down menu:



Figure 1.24: The device drop-down menu reporting the board's name and serial port

If you have connected the Arduino Nano, the device drop-down menu in the Arduino IDE should report **Arduino Nano 33 BLE** as the board's name, as shown in *Figure 1.24*.



Instead, if you connect the Raspberry Pi Pico, the Arduino IDE should report **Raspberry Pi Pico** as the board's name.

Near the board's name, you can also find the serial port. The serial port, which in *Figure 1.24* is **/dev/ttyACM0**, depends on the **operating system (OS)** and the device driver. This serial port will be our bridge for communication between the microcontroller and the computer.

Step 2:

Open the prebuilt **Blink** example by clicking on **Examples** from the left-hand side menu, **BUILT IN** from the new menu, and then **Blink**, as shown in the following screenshot:

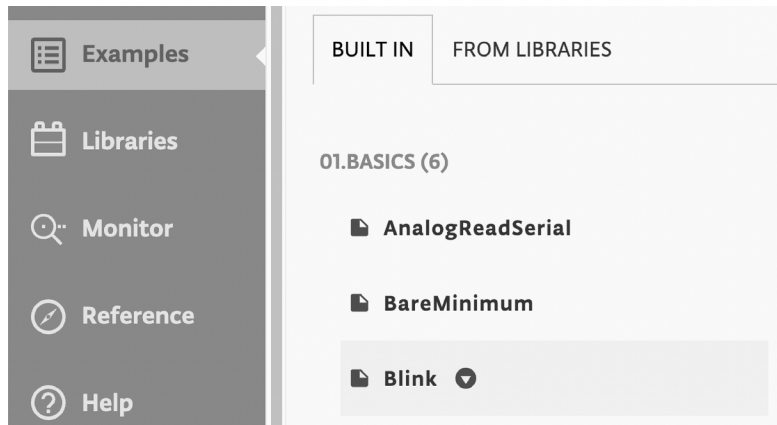


Figure 1.25: Built-in LED blink example

Once you have clicked on the **Blink** sketch, the code will be visible in the editor area.

Step 3:

Click on the arrow on the left of the board dropdown to compile and upload the program to the target device, as shown in *Figure 1.26*:



Figure 1.26: The arrow on the left of the board dropdown will compile and flash the program on the target device



In embedded programming, we generally use the term **flashing** when referring to the uploading of the program to the microcontroller.

The console output should return **Done** at the bottom of the page, and the on-board LED should start blinking, which means the sketch has been successfully compiled and uploaded to the microcontroller!

There's more...with the SparkFun Artemis Nano!

The LED blinking sketch we just uploaded on the Arduino Nano and Raspberry Pi Pico is also available for the SparkFun Artemis Nano microcontroller.

In the local Arduino IDE, the Blink example is in **File -> Examples -> 01.Basics -> Blink**:

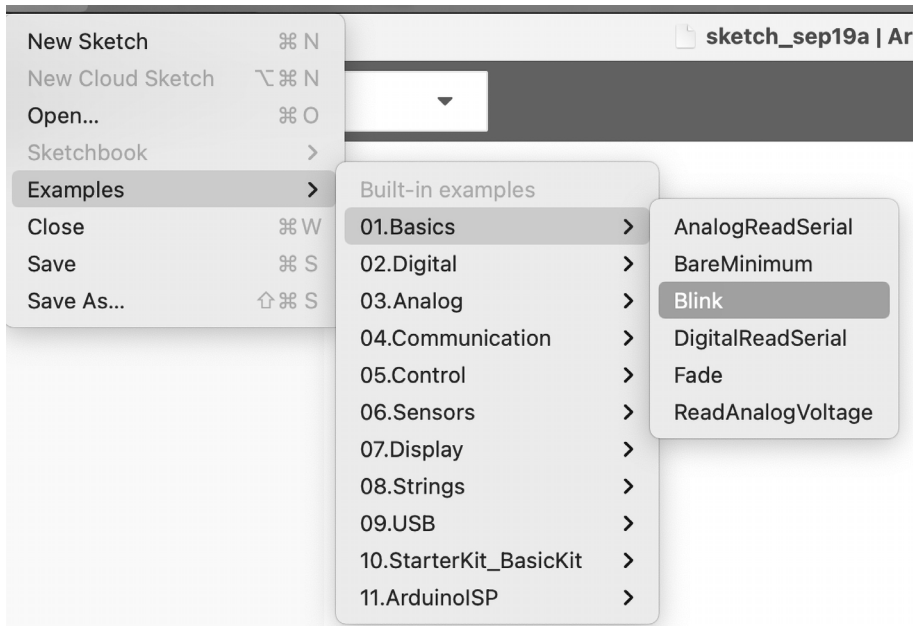


Figure 1.27: Built-in LED blink example in the local Arduino IDE

Once you click the **Blink** example, a new window with the sketch will be displayed. Before compiling the program, connect the SparkFun Artemis Nano to a laptop/PC through the USB-C data cable and make sure the device drop-down menu shows **RedBoard Artemis Nano** as the board's name:

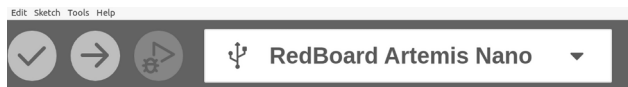


Figure 1.28: The device drop-down menu reporting the SparkFun Artemis Nano board

Then, click on the arrow on the left of the board dropdown to compile and upload the program to the target device. After a few seconds, the console output should return **Upload complete**, and the on-board LED of the SparkFun Artemis Nano should start blinking!

Summary

In this opening chapter, we have presented the ingredients to build low-power ML applications on microcontrollers. Initially, we uncovered the factors that make tinyML particularly appealing (cost, energy, and privacy) and motivated our choice to use microcontrollers as target devices.

We delved into the core components of this technology, giving a quick recap of ML and providing an overview of the essential features of microcontrollers necessary for the following chapters. After introducing microcontrollers and their unique features, we presented the leading software tools and frameworks used in this book to bring ML to microcontrollers: the Arduino IDE, TensorFlow, and Edge Impulse.

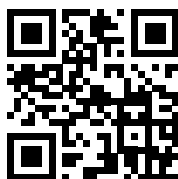
Finally, we built a pre-built sketch in the Arduino IDE to blink the on-board LED on the Arduino Nano, Raspberry Pi Pico, and SparkFun Artemis Nano.

In the following chapter, we will start our practical tinyML journey by exploring how to craft microcontroller applications from the very basics.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



2

Unleashing Your Creativity with Microcontrollers

Bringing **machine learning (ML)** to life on microcontrollers is a thrilling adventure because our creations can go beyond our computers' boundaries and make an impact in the real world. However, before diving into this fascinating world, let's take a moment to explore how to craft basic applications on microcontrollers to get up to speed with the principles of embedded programming.

In this chapter, we will start our exploration by handling data transmission over the serial communication protocol, equipping ourselves with a foundation for basic code debugging. The transmitted data will be captured in a log file and uploaded to our cloud storage in **Google Drive**.

Afterward, we will delve into programming the **GPIO** peripheral using the **Arm Mbed API** and use a solderless **breadboard** to connect external components, such as **LEDs** and **push-buttons**.

The aim of this chapter is to delve into the basic principles of microcontroller programming concepts that are essential for the projects developed in the rest of this book.

In this chapter, we're going to cover the following recipes:

- Transmitting data over serial communication
- Reading serial data and uploading files to Google Drive with Python
- Implementing an LED status indicator on the breadboard
- Controlling an external LED with GPIO
- Turning an LED on and off with a push-button
- Using interrupts to read the push-button state

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- An Arduino Nano 33 BLE Sense
- A Raspberry Pi Pico
- A SparkFun RedBoard Artemis Nano (optional)
- A micro-USB data cable
- A USB-C data cable (optional)
- 1 x half-size solderless breadboard (30 rows and 10 columns)
- 1 x red LED
- 1 x 220 Ω resistor
- 1 x push-button
- 5 x jumper wires
- Laptop/PC with either Linux, macOS, or Windows
- Google Drive account



The source code and additional material are available in the Chapter02 folder of the GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Chapter02.

Transmitting data over serial communication

Code debugging is a fundamental process of software development to identify errors in code.

This recipe will demonstrate how to conduct print debugging on the Arduino Nano and Raspberry Pi Pico by transmitting the following strings to the serial terminal:

- Initialization completed: once the serial port on the microcontroller has finished initializing
- Executed: after every 2 seconds of program execution

Getting ready

All programs are prone to bugs, and print debugging is a basic process that displays statements on the output terminal, providing valuable insight into the program's execution, as shown in the following example:

```
int func (int func_type, int a) {
    int ret_val = 0;
    switch(func_type){
        case 0:
            printf("FUNC0\n");
            ret_val = func0(a)
            break;
        default:
            printf("FUNC1\n");
            ret_val = func1(a);
    }
    return ret_val;
}
```

In the preceding code snippet, the `printf()` function is utilized to indicate which case statement the program enters during its execution.

The Arduino programming language offers a similar function to `printf()`, the `Serial.print()` function.

The `Serial.print()` function can send characters, numbers, or even binary data from the microcontroller board to our computer through the serial port, commonly called **UART** or **USART**. You can refer to <https://www.arduino.cc/reference/en/language/functions/communication/serial/print/> for the complete list of input arguments.

However, in contrast to the standard C library `printf()` function, the `Serial.print()` routine requires initialization before transmitting any data, generally done in the Arduino's `setup()` function through the `Serial.begin()` function. The `Serial.begin()` routine only requires the **baud rate** as an input argument, which is the data transmission rate in **bits per second (bps)**. Therefore, the higher the baud rate, the quicker the data transfer.

How to do it...

Open the Arduino IDE and create a new empty project by clicking on **Sketchbook** from the leftmost menu (**EDITOR**) and then click on **NEW SKETCH**, as shown in the following figure:

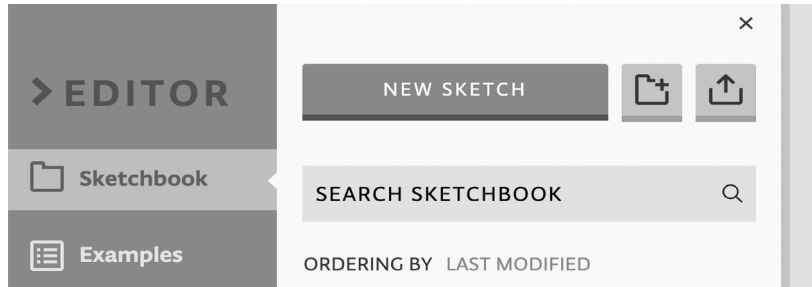


Figure 2.1: Click on the NEW SKETCH button to create a new project

All sketches require at least one file containing the `setup()` and `loop()` functions. As a result, the following steps will demonstrate how to write these functions to accomplish the task of this recipe.

Step 1:

In the `setup()` function, initialize the Serial peripheral with a baud rate of 9600 bps and wait until it is open:

```
void setup() {  
  Serial.begin(9600);  
  while(!Serial);  
}
```

In the preceding code, `while(!Serial)` is used to wait until the Serial peripheral is ready to transmit or receive data.

Step 2:

In the `setup()` function, transmit the Initialization completed string after the `Serial.begin()` function:

```
Serial.print("Initialization completed\n");
```

The `Serial.print("Initialization completed\n")` function is used to transmit the Initialization completed string over the serial communication protocol.

Step 3:

In the `loop()` function, transmit the Executed string every 2 seconds:

```
void loop() {  
  delay(2000);  
  Serial.print("Executed\n");  
}
```

Since the `loop()` function is called iteratively, we use Arduino's `delay()` function to pause the program execution for 2 seconds. The `delay()` routine accepts the amount of time in milliseconds (1 s = 1000 ms) as an input argument.

The program we have just written works with any Arduino compatible platform, such as Arduino Nano and Raspberry Pi Pico. The Arduino IDE, in fact, will correctly compile the code accordingly with the selected platform in the device drop-down menu.

Step 4:

Make sure that either the Arduino Nano or Raspberry Pi Pico is plugged into your computer through the micro-USB data cable.

If the device is recognized, we can open the serial monitor by clicking the **Monitor** button from the **EDITOR** menu. From the serial monitor, we will see any data transmitted by the microcontroller through the UART peripheral. However, before communication starts, ensure the serial monitor uses the same baud rate as the microcontroller peripheral. In our case, the baud rate is **9600**, as shown in the following figure:

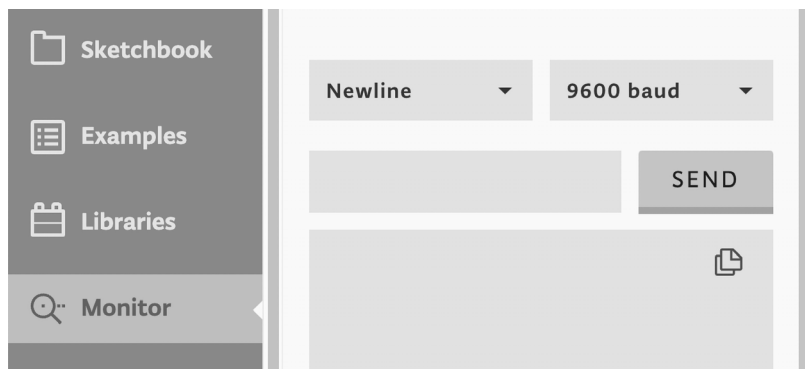


Figure 2.2: The serial monitor must use the same baud rate as the UART peripheral

With the serial monitor open, we can click the arrow near the device drop-down menu to compile and upload the program to the target platform.

Once the sketch has been uploaded, the serial monitor will receive the **Initialization completed** and **Executed** messages, as shown in the following screenshot:

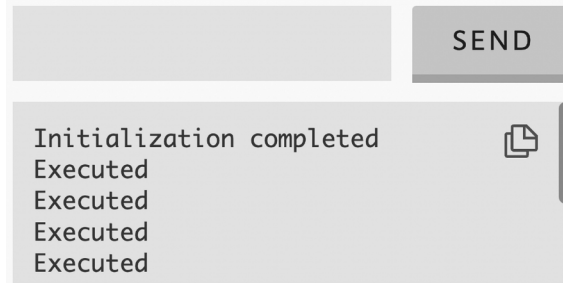


Figure 2.3: Expected output on the serial monitor

As we can see from the serial monitor output, **Initialization completed** is printed once because the `setup()` function is called when starting the program.

There's more...

In this recipe, we learned how to transmit messages over serial communication.

Although print debugging with the serial peripheral is a straightforward method for troubleshooting, its drawbacks become more evident as software complexity increases. For example, some of the main disadvantages are:

- Needing to re-compile and upload the sketch on the board every time we add or move the `Serial.print()` function
- The `Serial.print()` function increases the program memory footprint
- We could potentially report incorrect information, such as displaying a signed integer variable that is unsigned

We will not cover more advanced debugging methods in this book, but we recommend looking at **serial wire debug (SWD)** debuggers (<https://developer.arm.com/architectures/cpu-architecture/debug-visibility-and-trace/coresight-architecture/serial-wire-debug>) to make this process less painful. SWD is an Arm debug protocol for almost all Arm Cortex processors that you can use to flash the microcontroller, step through the code, add breakpoints, and so on with only two wires.

The upcoming recipe will reveal how to read the transmitted messages through serial communication using Python and demonstrate how to generate a log file that can be uploaded to Google Drive.

Reading serial data and uploading files to Google Drive with Python

When developing tinyML projects, our microcontrollers can use serial communication to transfer data of any type to our computer.

In this recipe, we will showcase how to develop a local Python script to retrieve the transmitted data from the PC's serial port. The program will record one minute of data transmission to a file, which will be uploaded to Google Drive.

Getting ready

Throughout the book, the microcontroller will use serial communication for various scopes, such as:

- Tracking events when the program runs
- Debugging sensor functionalities
- Gathering relevant data for building the dataset used to train and test an ML model

The last point will likely be the most enjoyable. For instance, you will use the microphone to record your vocals or musical recordings and a camera to snap pictures. However, unlike our standard computers or laptops, which have an **operating system (OS)** and pre-installed drivers, microcontrollers do not have an OS to drive these sensors effortlessly. As a result, a thorough knowledge of the microcontroller features, sensor specifications, and communication protocols will be necessary to read data from sensors. When we interface directly with sensors, we generally deal with **raw data**, such as audio samples or image pixels. Since microcontrollers lack a sophisticated OS capable of file creation, we must find an alternative approach to generate files to be processed by our multimedia software programs or used for training an ML model. One method involves transmitting the raw data over the serial communication protocol and using a local Python script to grab the data and generate the appropriate file.

The Python library employed for reading data from the serial communication protocol is **pySerial**, which is discussed in the subsequent subsection.

Reading data from the serial port with pySerial

Parsing data from the serial port is straightforward with the pySerial library. As we will demonstrate in the following *How to do it...* section, we will only need a few lines of code to grab the serial data.

The library can be installed through the pip Python package manager as follows:

```
$ pip install pyserial
```

To start using this library, we will only need the two standard pieces of information required for serial communication, namely:

- Port name
- Baud rate

As stated at the beginning of this section, serial communication can be employed to retrieve sensor data gathered by the microcontroller, which can then be included in the dataset used to train and evaluate an ML model.

Throughout this book, the ML model will be designed and trained in the cloud using the free **Google Colab** Jupyter Notebook platform. However, this cloud-based service cannot access the local serial port. Therefore, we have two options to upload the files in Google Colab:

- Uploading the files manually using the Google Colab web interface
- Uploading the files into Google Drive using a Python local script

As you can imagine, manually uploading files to Google Colab can sometimes be tedious and inconvenient. Therefore, the following subsection will give you some insights into how you can upload your local files to Google Drive using the **Google Drive API** with Python.

Enabling the Google Drive API

The Python library used to upload files to Google Drive is **PyDrive**. PyDrive can be installed by using the following pip Python package manager command:

```
$ pip install pydrive
```

However, we cannot directly access Google Drive from Python without telling the Google Drive API (<https://developers.google.com/drive/api/guides/about-sdk>) who needs to access the Google Drive storage. The following diagram shows the relation between your Python application, PyDrive, the Google Drive API, and your Google Drive storage:

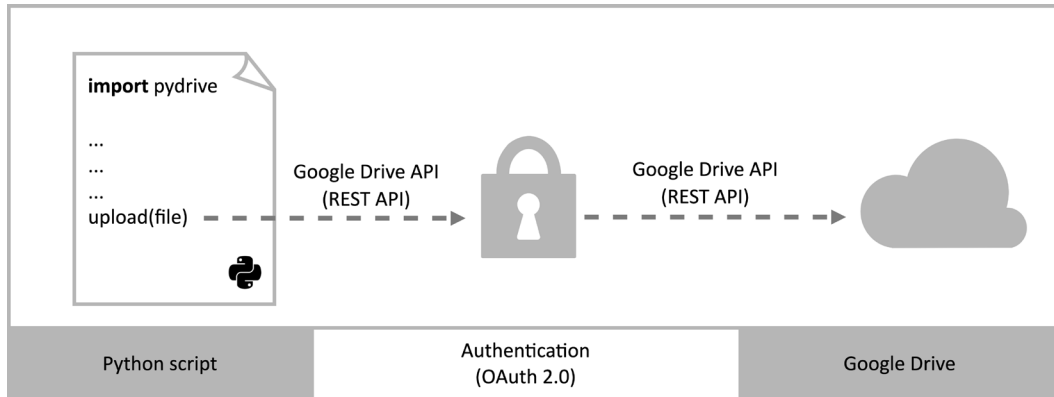


Figure 2.4: Relationship between PyDrive, the Google Drive API, and Google Drive storage

As you can see from the preceding image, the Google Drive API is a **REST API** that requires user authentication through an authentication protocol (**OAuth 2.0**) before accessing any files in Google Drive.

The Google Drive API needs to be enabled in **Google Cloud** (<https://console.cloud.google.com/>), and the following steps, which include screenshots and comments, will show you how to do it.

Step 1:

Click on the **Select a project** drop-down menu near the Google Cloud logo, and then click on the **NEW PROJECT** button to create a new Google Cloud project:

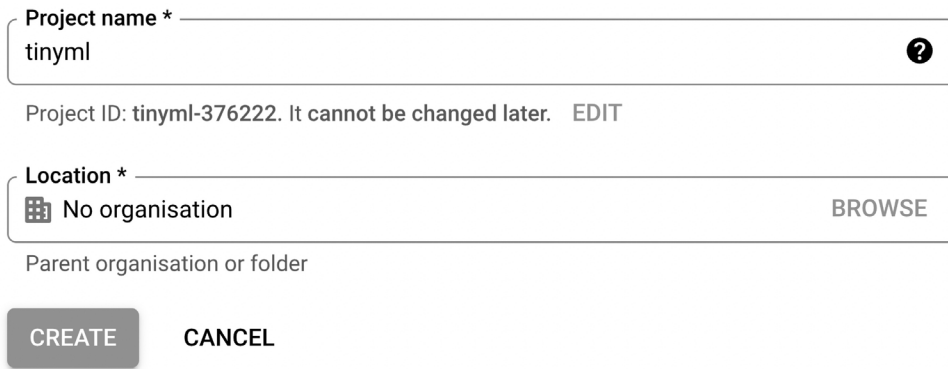


Figure 2.5: Create a new project in the APIs and services section

The Google Cloud project forms the basis for using the Google Drive API.

Step 2:

Assign a name to the project, for instance, `tinym1`:



Project name * ?

Project ID: tinym1-376222. It cannot be changed later. [EDIT](#)

Location * [BROWSE](#)

Parent organisation or folder

[CREATE](#) [CANCEL](#)

Figure 2.6: Assign the name to the project

Click on the **CREATE** button to create the project and select the `tinym1` project.

Step 3:

Activate the Google Drive API for the project we created in the previous step. To do so, click on the **ENABLE APIS AND SERVICES** button in the **APIs and services** section:

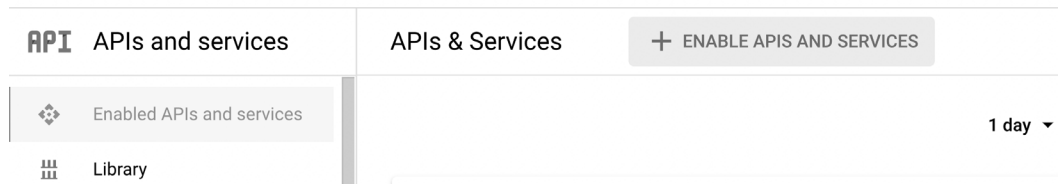


Figure 2.7: Enter the section to enable the Google Drive API

On the new page, search for **Google Drive API** through the **Search for APIs and services** field:

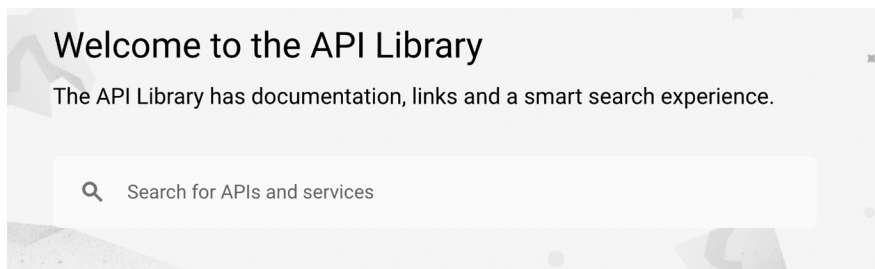


Figure 2.8: Search for the Google Drive API

Then, click on the **Google Drive API** icon:



Google Drive API

Google Enterprise API ?

With the Google Drive API, you can access resources from Google Drive to create files,

Figure 2.9: Click on the Google Drive API icon

On the new page, click on the **Enable** button and then click on **CREATE CREDENTIALS**. Now, select **User data** as the data type that we will be accessing in Google Drive:

What data will you be accessing? *

Different credentials are required to authorise access depending on the type of data that you request. [Learn more](#)

☒ User data ?

Data belonging to a Google user, like their email address or age. User consent required. This will create an OAuth client.

Figure 2.10: Select User data as the data type that we will be accessing in Google Drive

Step 4:

Go to the bottom of the page and click on the **Done** button.

You should be in the **Scope** section. In this section, you can leave all the fields empty and click on the **SAVE AND CONTINUE** button directly:

🔒 Your restricted scopes

Restricted scopes are scopes that request access to highly sensitive user data.

API ↑	Scope	User-facing description
No rows to display		

SAVE AND CONTINUE

CANCEL

Figure 2.11: The Scope section

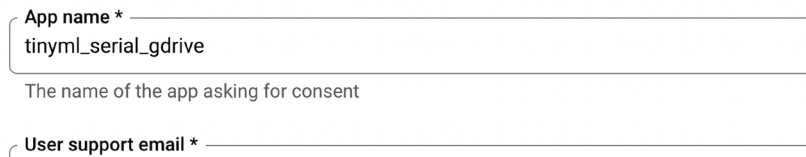
You should now be in the **App information** section.

Step 5:

In the **App information** section, you can just fill in all the mandatory fields marked with the asterisk (*), such as **App name**, which is the application's name, asking to access Google Drive. In our case, we have called the application `tinym1_serial_gdrive`:

App information

This shows in the consent screen, and helps end users know who you are and contact you



App name *
tinym1_serial_gdrive

The name of the app asking for consent

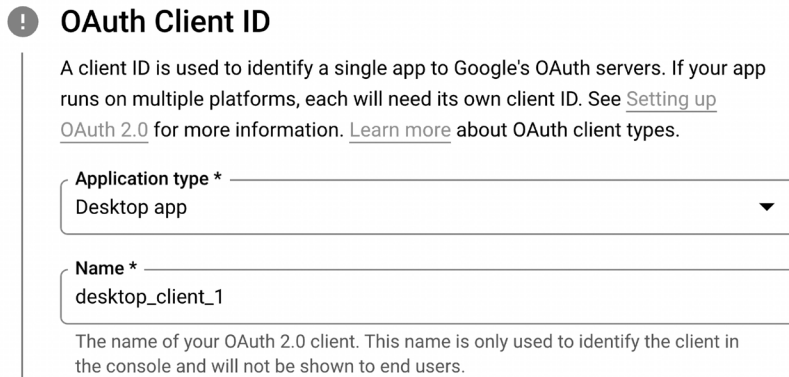
User support email *

Figure 2.12: Provide the app information

Then, you can click the **SAVE AND CONTINUE** button at the bottom of the page.

Step 6:

You should be in the **OAuth Client ID** section. Here, select **Desktop app** in the **Application type** drop-down menu:



! OAuth Client ID

A client ID is used to identify a single app to Google's OAuth servers. If your app runs on multiple platforms, each will need its own client ID. See [Setting up OAuth 2.0](#) for more information. [Learn more](#) about OAuth client types.

Application type *
Desktop app ▼

Name *
desktop_client_1

The name of your OAuth 2.0 client. This name is only used to identify the client in the console and will not be shown to end users.

Figure 2.13: Create the OAuth client

You can keep the default name in the **Name** field. In our example, it is **desktop_client_1**.

Step 7:

Click on the **Create** button and wait for the confirmation for **OAuth client created**:

OAuth client created

The client ID and secret can always be accessed from Credentials in APIs & Services

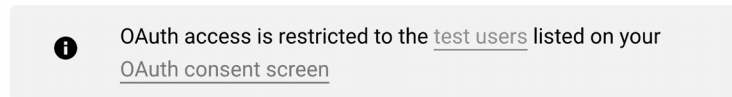


Figure 2.14: Message confirmation about the creation of the OAuth client

At the bottom of the page, click on the **DOWNLOAD JSON** button to download the credentials as a JSON file:

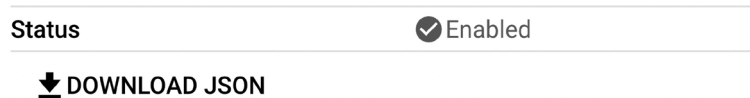


Figure 2.15: Download the credentials as a JSON file

The PyDrive library requires this file to access Google Drive, which must be located in the same directory as your Python script and renamed `client_secrets.json`.

Now, you can click on the **DONE** button.

Step 8:

On the previous **OAuth client created** pop-up page, you would have noticed the following warning: **OAuth access is restricted to the test users listed on your OAuth consent screen**. This message wants to remind us that the status of our application (for example, `tinym1_serial_gdrive`) is **Testing**, which means that only a limited number of users can access the application.

You can verify the status of your application by clicking on the **OAuth consent screen** option in the left-hand side menu of **APIs and services**:

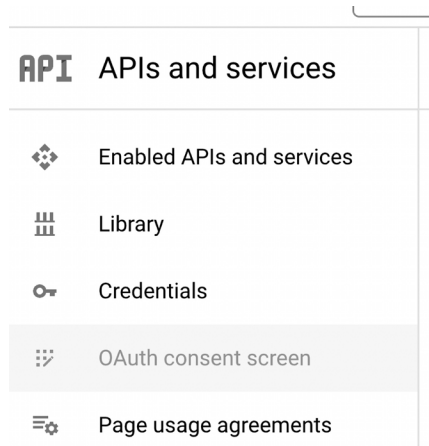



Figure 2.16: Enter the OAuth consent screen section

In the right-hand side frame, you should see the application name (for example, **tinymml_serial_gdrive**) and **Publishing status** reporting **Testing**:

tinymml_serial_gdrive  EDIT APP

Publishing status

Testing

Figure 2.17: The publishing status is Testing

To add yourself as a test user, click the **ADD USERS** button at the bottom of the right-hand frame. In the new window, you only need to provide the email address of your Google account.

The following steps will show you how to read data from the serial port and upload the log file to Google Drive.

How to do it...

In your local **Python Virtual Environment (virtualenv)**, create a new working directory (for example, `gdrive`). Enter the working directory and copy the `client_secrets.json` file into it.

In the same folder, create a new Python script called `serial_to_gdrive.py` and follow the following steps to implement a Python script to read data from the serial port and upload the log file to Google Drive every minute.

Step 1:

Import the serial Python module and initialize pySerial with the port and baud rate used by the microcontroller:

```
import serial

ser = serial.Serial()
ser.port = '/dev/ttyACM0'
ser.baudrate = 9600
```

The baud rate is 9600, as we set in the previous recipe, *Transmitting data over serial communication*.

To determine the port name, the simplest method is to check the device drop-down menu in the Arduino IDE:



Figure 2.18: You can find the port name in the device drop-down menu in the Arduino IDE

In the preceding screenshot, the serial port name is `/dev/ttyACM0`.

Step 2:

Open the serial port and discard the content in the input buffer:

```
ser.open()
ser.reset_input_buffer()
```

The `reset_input_buffer()` function is responsible for discarding any data available in the input buffer received so far.

Step 3:

Create a utility function to return a line from the serial port as a string:

```
def serial_readline(obj):
```

```
data = obj.readline()
return data.decode("utf-8")
```

The string sent by the microcontroller over the serial is encoded using the **UTF-8** format. Therefore, we use the `.decode("utf-8")` function to convert the UTF-8 encoded bytes back to a string. The input argument `obj` is the object created at *Step 1* using the `serial.Serial()` command.

Step 4:

Record one minute of serial data transmission in a string:

```
import time
text = ""

start_s = time.time()
elapsed_s = 0
while(elapsed_s < 1):
    text += serial_readline(ser)
    end_s = time.time()
    elapsed_s = end_s - start_s
```

The serial transmission is acquired line by line using the `serial_readline()` function and then stored as a string in the `txt` variable. To record the serial data for one minute, we use the **time** library. In particular, we use the `time()` method to:

- Save the timestamp in seconds in the `start_s` variable before entering the `while()` loop
- Save the timestamp in seconds at the end of the `while()` statement in the `end_s` variable
- Save the difference between the `end_s` and `start_s` variables in the `elapsed_s` variable

We exit the `while()` loop when the `elapsed_s` variable is greater than one second.

Step 5:

Write the `txt` string to a text file:

```
filename = "test.log"
text_file = open(filename, "w")
text_file.write(text)
text_file.close()
```

To write the string to a text file, we open the `test.log` file using the `open()` function. Afterward, we use the `write()` routine to store the string in the file.

Step 6:

Import the necessary Python modules to use the PyDrive library:

```
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
```

Then, grant permission for your account to use the Google Drive API:

```
gauth = GoogleAuth()
gauth.LocalWebserverAuth()
```

The preceding code will open a window in the web browser to log in to your Google account. Once logged in, you should see the **Authentication successful** message printed on the web page.

Step 7:

Use the `gauth` **GoogleAuth** object to create a local instance of Google Drive:

```
drive = GoogleDrive(gauth)
```

In the preceding code, the `GoogleDrive()` function returns the local instance of Google Drive in the `drive` object.

Step 8:

Open Google Drive and find a suitable directory to store the log file. Then take note of the **Google Drive file ID**, which is the last part of the **Uniform Resource Locator (URL)** of the file or directory when viewed in the web browser, as shown in the following screenshot by the sequence of sharp (#) characters:



Figure 2.19: The Google Drive file ID is the last part of the URL of the file viewed in the web browser

The Google Drive file ID is a unique *alphanumeric* identifier assigned to each file or directory in Google Drive and is required by the PyDrive library to interact with directories and files.

Step 9:

Upload the `test.log` file into the Google Drive directory:

```
gdrive_id = "xxxxxxxxxxxxx"

gfile = drive.CreateFile(
    {'parents': [{'id': gdrive_id}]})

gfile.SetContentFile(filename)
gfile.Upload()
```

In the previous code, we created an instance of **GoogleDriveFile** with the `CreateFile()` method. This step is required to specify the destination directory through its **Google Drive file ID** (the `gdrive_id` variable).



Remember to replace the sequence of x characters in the `gdrive_id` variable with the Google Drive file ID to upload the file to Google Drive successfully.

Once we have defined the destination directory, we can simply provide the name of the file we want to upload with the `SetContentFile()` function and call the `Upload()` method to copy the file to the Google Drive directory.

Step 10:

Ensure the microcontroller is running and you do not have the serial monitor opened in the Arduino IDE, as only one application can use the serial port. Then, run the Python script:

```
$ python serial_to_gdrive.py
```

If the Python script can communicate with the microcontroller, the `log test.log` file will be uploaded to Google Drive and updated every minute.

There's more...

In this recipe, we learned how to use the PyDrive library to upload data captured with the microcontroller to the cloud automatically.

PyDrive is not restricted to uploading files to Google Drive only. In reality, PyDrive can carry out the usual tasks that are done in the web browser, such as:

- Creating files
- Downloading files
- Searching for files
- Delegating files

For more information, refer to the official PyDrive documentation at <https://pythonhosted.org/PyDrive/>.

For example, you may consider extending the Python script to create a directory in Google Drive to automate uploading files in the cloud fully.

Serial communication is undoubtedly an easy way to get information during the program execution. However, its relatively slow data transfer speed could make it unsuitable for some applications.

The upcoming recipe will show an alternative approach that can only display simple information to the user but does not suffer from this slowness. This approach is based on the light of LEDs.

Implementing an LED status indicator on the breadboard

Microcontrollers enable us to interact with the world around us by using sensors and performing physical actions, such as turning an LED on and off or moving an actuator.

In this recipe, we will learn how to connect external components with the microcontroller by building the following electronic circuit on the breadboard:

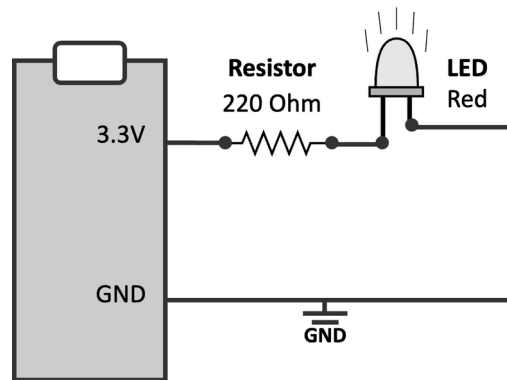


Figure 2.20: The LED power status indicator circuit

The electronic circuit illustrated in *Figure 2.20* uses the red LED to indicate whether the microcontroller is connected to the power source.

Getting ready

Connecting external components to the microcontroller means physically joining two or more metal connectors. Although we could solder these connectors, it is not usual for prototyping because it is not quick and straightforward.

Therefore, this recipe presents a solderless alternative to connect our components effortlessly.

Making contact directly with the microcontroller's pins can be extremely hard for the tiny space between each pin. For example, considering the RP2040 microcontroller, the pin space is roughly 0.5 mm since the chip size is 7x7 mm. Therefore, connecting any of our components safely would be impossible since most terminals have a wire diameter of ~1 mm.

For this reason, our platforms provide alternative points of contact with wider spacing on the board. These contact points on the Arduino Nano and Raspberry Pi Pico are the two rows of pre-drilled holes located at the platform's edge.

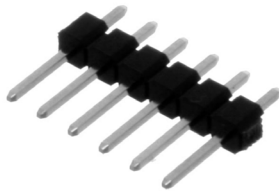
The simplest way to know the correspondence between these contacts and the microcontroller pins is to refer to the datasheet of the microcontroller boards. Hardware vendors usually provide the pinout diagram to note the pins' arrangement and functionality.

For example, the following list reports the links to the Arduino Nano, Raspberry Pi Pico, and SparkFun Artemis Nano pinout diagrams:

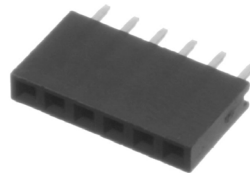
- **Arduino Nano:** https://content.arduino.cc/assets/Pinout-NANOsense_latest.pdf
- **Raspberry Pi Pico:** <https://datasheets.raspberrypi.org/pico/Pico-R3-A4-Pinout.pdf>
- **SparkFun Artemis Nano:** <https://cdn.sparkfun.com/assets/5/5/1/6/3/RedBoard-Artemis-Nano.pdf>

On top of these pre-drilled holes, which often come with a 2.54 mm spacing, we can solder a header to insert and connect the electronic components easily.

The header can be either a male (**pin header**) or a female connector (**socket header**), as shown in *Figure 2.21*:



Male header (pin header)



Female header (socket header)

Figure 2.21: Male header and female header (image from https://en.wikipedia.org/wiki/Pin_header)



If you are unfamiliar with soldering or prefer a ready-to-use solution, we recommend buying devices with pre-soldered male headers.

As we have seen, the boards provide a way to connect the external components with the microcontroller. However, how can we attach other electrical elements to build a complete electronic circuit?

Prototyping on a breadboard

The breadboard is a solderless prototyping platform that allows us to build circuits by inserting the pins of electronic components into a grid of metal holes arranged in a rectangular shape:

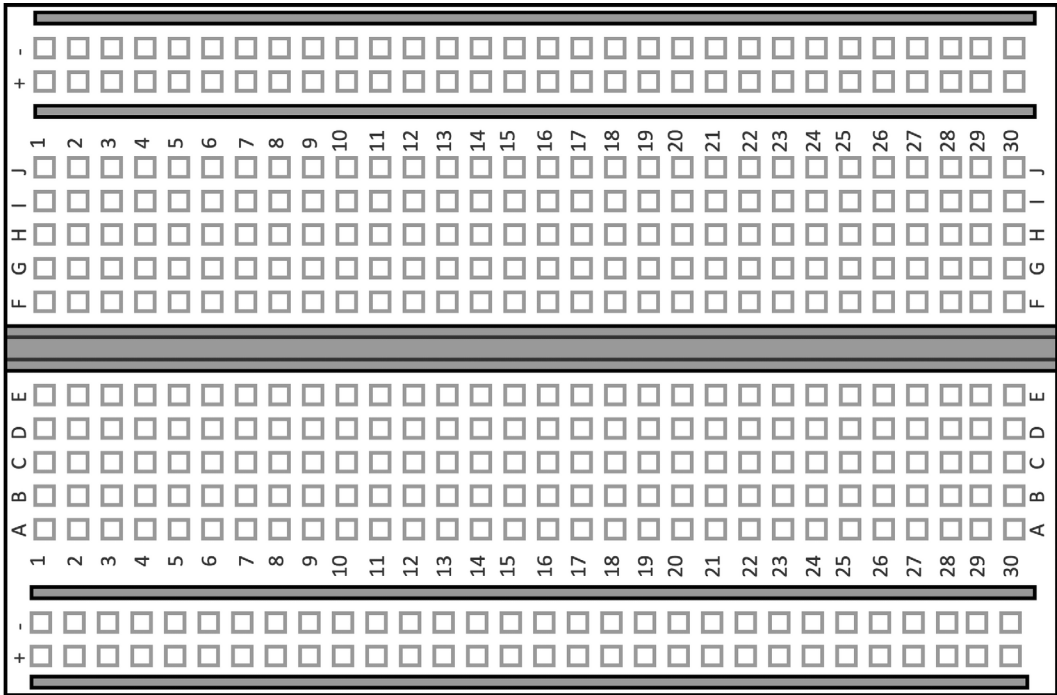


Figure 2.22: Solderless breadboard

Breadboards provide two connecting areas for our components: **bus rails** and **terminal strips**.

Bus rails are usually located on both sides of the breadboard and consist of two columns of holes identified with the symbols + and –, as shown in the following diagram:

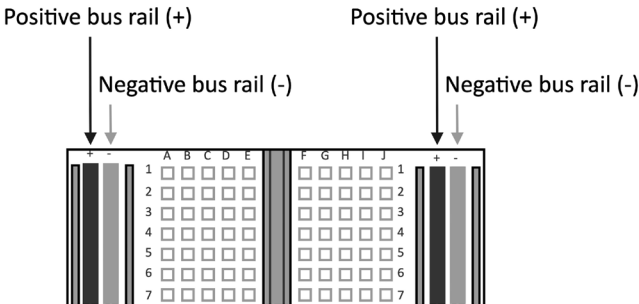


Figure 2.23: Bus rails labeled with + and - on both sides of the breadboard

All the holes of the same column are internally connected. Therefore, we will have the same voltage through all its columns when applying a voltage to whatever hole.

Since bus rails are beneficial for having reference voltages for our circuits, we should never apply different voltages on the same bus column.

Terminal strips are located in the central area of the breadboard and join only the holes of the same row so that the following occurs:

- Holes on the same row have the same voltage
- Holes on the same column might have a different voltage

However, since we typically have a notch running parallel in the middle of the breadboard, we have two different terminal strips per row, as shown in *Figure 2.24*:

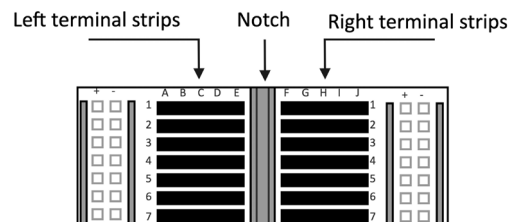


Figure 2.24: Terminal strips are located in the central area of the breadboard

On the breadboard, we can insert multiple devices. For example, in the upcoming *How to do it...* section, you will discover how to insert the Arduino Nano and Raspberry Pi Pico between the left and right terminal strips and use **jumper wires** to connect these devices to other electronic components.



The number of rows and columns in the terminal area defines the size of a breadboard. This book always refers to a half-sized breadboard with 30 rows and 10 columns.

How to do it...

To prevent accidental damage to the electronic components, disconnect the micro-USB cable from the microcontrollers.

After disconnecting the board from the power source, proceed with the following steps to build the circuit on the breadboard.

Step 1:

Put the microcontroller board on the breadboard:

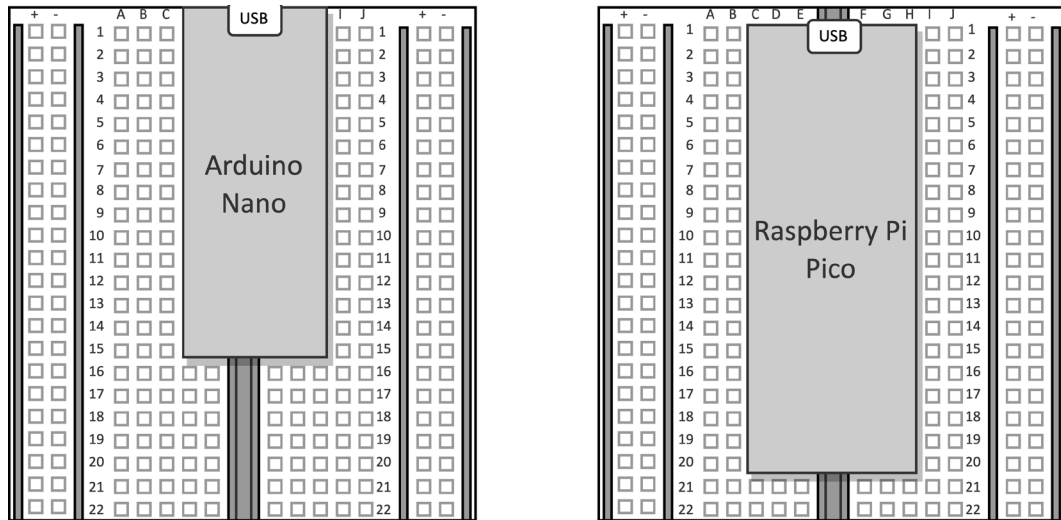


Figure 2.25: Vertically mount the microcontroller board between the left and right terminal strips

As the notch is oriented parallel, inserting the devices in this manner is safe since the left and right pin headers will make contact with two different terminal strips.

Step 2:

Use two jumper wires to connect the **3.3 V** and **GND** pins of the microcontroller board with the **+** and **-** bus rails:

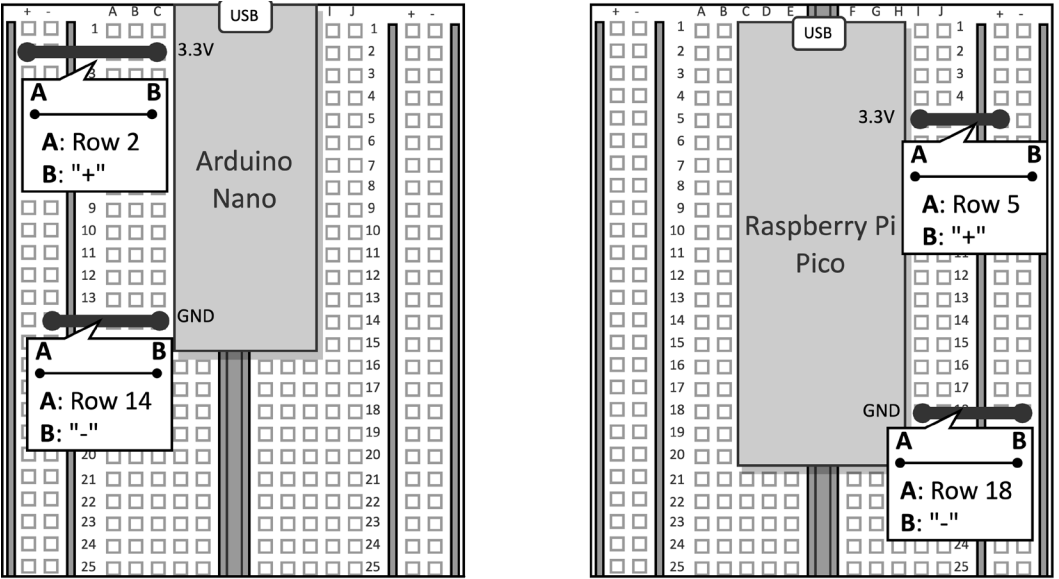


Figure 2.26: Use the jumper wires to connect the 3.3 V and GND to the + and - bus rails

Note that the bus rails' holes will carry 3.3V and GND only when the microcontroller is connected to power.

Step 3:

Place the LED pins into two terminal strips:

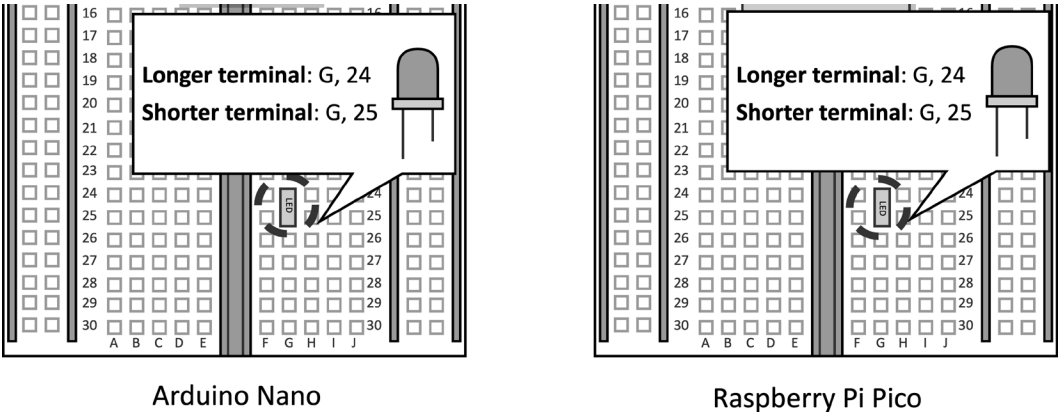


Figure 2.27: Insert the LED on the breadboard

In the preceding figure, we have the longer LED terminal in (G 24) and the shorter one in (G, 25). Do not invert the longer and shorter terminals because the LED won't turn on.

Step 4:

Place a 220 Ω resistor in series with the LED:

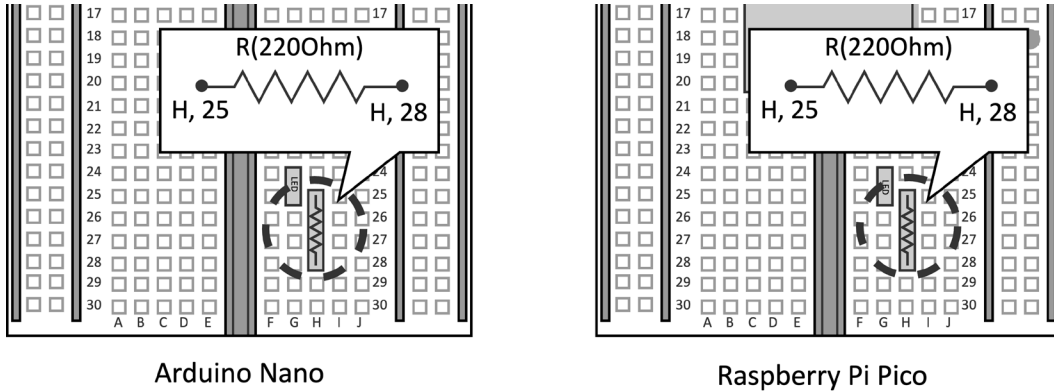


Figure 2.28: Place the resistor in series with the LED

The color bands of the resistor can be determined through the **DigiKey web tool** (<https://www.digikey.com/en/resources/conversion-calculators/conversion-calculator-resistor-color-code>). For example, a 220 Ω resistor with five or six bands is encoded with the following colors:

- First band: red (2)
- Second band: red (2)
- Third band: black (0)
- Fourth band: black (1)

As shown in the electronic circuit presented at the beginning of this recipe, one terminal of the resistor makes contact with the shorter LED pin. We can achieve this by inserting any of the two terminals of the resistor in (H, 25). The remaining terminal can be inserted in whichever unused terminal strip. In our case, we insert this terminal in (H, 28).

Step 5:

Complete the circuit by connecting the + bus rail (3.3 V) to the longer LED pin and the - bus rail (GND) to the resistor terminal:

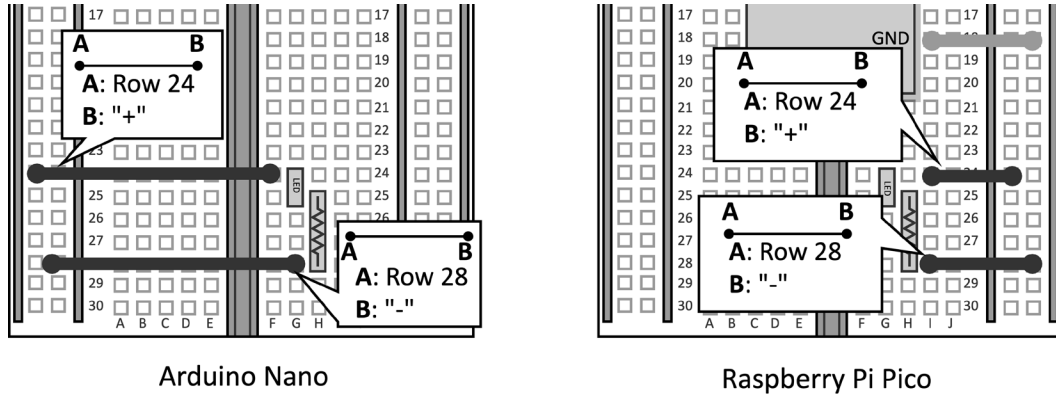


Figure 2.29: Close the circuit by connecting 3.3 V and GND

The previous figure shows how to connect the two remaining jumper wires to close the circuit. One jumper wire links the + bus rail with the longer LED terminal (F, 24), while the other wire connects the - bus rail with the resistor (G, 28).

Now, the LED should emit light whenever we plug the microcontroller into the power with the micro-USB cable.

There's more...with the SparkFun Artemis Nano!

In this recipe, you learned the basics of building an electronic circuit on the breadboard. Therefore, what do you think about repeating the same recipe with the SparkFun Artemis Nano microcontroller board? To accomplish this task, you must identify the 3.3V and GND pins on the board. The following *Hookup Guide for the SparkFun RedBoard Artemis Nano* provides additional information that you may find helpful as you work with this powerful board: <https://learn.sparkfun.com/tutorials/hookup-guide-for-the-sparkfun-redboard-artemis-nano>.

The solution for this exercise is available in the Chapter02 folder in the GitHub repository.

Now that we know the basics for building electronic circuits, it becomes imperative to understand how LEDs work to employ this device properly.

The upcoming recipe will delve into the details of SparkFun Artemis Nano functionalities and programmability through the GPIO peripheral.

Controlling an external LED with the GPIO

Nowadays, LEDs are everywhere, particularly in our houses, because they use less energy than older lights for the same luminous intensity. However, the LEDs considered for our experiments are not light bulbs but through-hole LEDs for rapid prototyping on the breadboard.

In this recipe, we will discover how to build a basic circuit with an external LED and program the GPIO peripheral to control its light.

Getting ready

To implement this recipe, we need to know how the LED can emit light and how to program the microcontroller GPIO peripheral to turn the light on and off.

Let's start by explaining what an LED is and how it works.

Understanding the LED functionality

LED stands for **Light Emitting Diode** and is a semiconductor component that emits light when a current flows through it.

In this book, we will use **through-hole LEDs**, which are made of the following:

- A **head** of transparent material from where the light comes. The head can be of different diameters but typically comes in 3 mm, 5 mm, and 10 mm sizes.
- Two legs (**leads**) of different lengths to identify the positive (**anode**) from the negative (**cathode**) terminal. The anode is the longer lead.

The following diagram shows the basic structure of a through-hole LED and its symbolic representation in an electronic circuit:

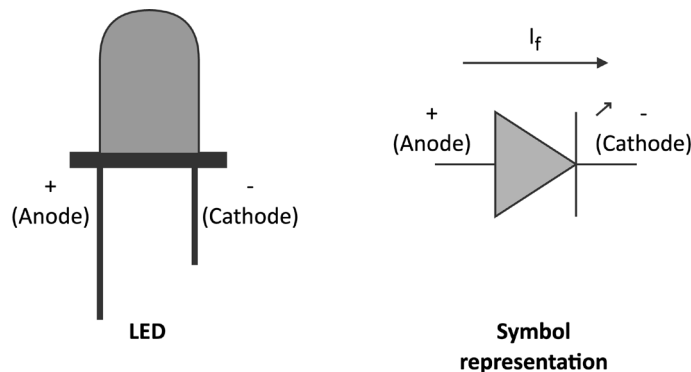


Figure 2.30: LED with symbolic representation

As mentioned, the LED emits light when the current flows through it. However, in contrast to the resistors, the current flows only in one direction, specifically from the anode to the cathode. This current is commonly called **forward current** (I_f).

The brightness of the LED is proportional to I_f , so the higher it is, the brighter it will appear.

The LED has a maximum operating current that we must not exceed to avoid breaking it instantly. For standard through-hole 5 mm LEDs, the maximum current is typically 20 mA, so values between 4 mA and 15 mA should be enough to see the LED emitting the light.

To allow the current to flow, we need to apply a specific voltage to the terminals' LED, called **forward voltage** (V_f). We define V_f as:

$$V_f = V_{anode} - V_{cathode}$$

Typical V_f range for some LED colors are reported in the following table:

LED color	Forward voltage (V)
Red	1.8 – 2.1
Orange	1.9 - 2.2
Yellow	1.9 – 2.2
Green	2 – 3.1
Blue	3 – 3.7
White	3 – 3.4

Figure 2.31: Typical LED forward voltage

From the preceding table, we can observe the following about the forward voltage range:

- It depends on the color.
- In most cases, it is narrow and less than the typical 3.3 V required to power a microcontroller.

From these observations, three questions come to mind:

- How can we apply the forward voltage on the LED terminals since we typically only have 3.3 V from the microcontroller?
- What happens if we apply a voltage lower than the minimum V_f ?
- What happens if we apply a voltage higher than the maximum V_f ?

The answers rely on the following physical relationship between the voltage and current of the LED:

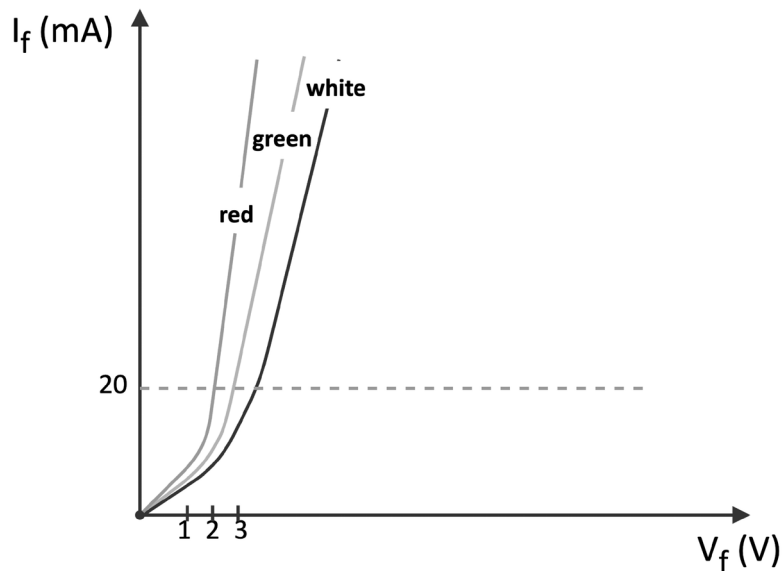


Figure 2.32: Voltage-current (V_I) characteristic of LED

From the previous chart, where the x and y axes report the voltage and current, we can deduce the following:

- If we applied a voltage much lower than V_f to the LED, the LED would not turn on because the current passing through it would be insufficient.
- If we applied a voltage much higher than V_f to the LED, the LED would be damaged because the current would exceed the 20 mA limit.

Therefore, keeping V_f at the required operating voltage guarantees proper functioning and prevents potential damage.

The solution is simple and only requires a resistor in series with the LED, as shown in the following figure:

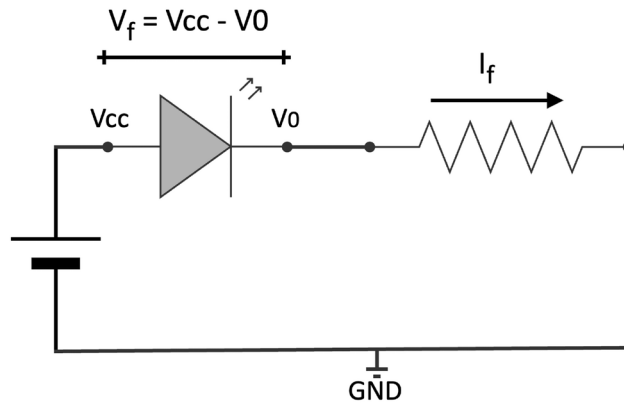


Figure 2.33: The resistor in series with the LED limits the current

By now, it should be evident why we added the resistor to the circuit of the previous recipe. Since the LED has a fixed voltage drop when it emits the light (V_f), the resistor limits the current at the desired value, such as 4 mA–15 mA. Therefore, having the LED current in the acceptable range means that V_f does not fall out of the expected operating range.

The following equation allows you to calculate the resistor's value to add in series to the LED:

$$R = \frac{V_{cc} - V_f}{I_f}$$

Where:

- V_{cc} is the microcontroller's positive voltage (for example, 3.3V)
- V_f is the forward voltage
- I_f is the forward current
- R is the resistance

The forward voltage/current and LED brightness information are generally available on the LED datasheet.

Now, it is time to see how we can turn the light on and off with the GPIO peripheral.

Introducing the GPIO peripheral

The most common and versatile peripheral on the microcontroller is **General-Purpose Input-Output (GPIO)**. The GPIO peripheral facilitates the transmission (output) and reception (input) of digital signals (*1* or *0*) through the external pins, commonly called either *GPIO*, *IO*, or *GP*.

A microcontroller can integrate multiple GPIO peripherals, each dedicated to controlling a specific pin of the integrated chip.

GPIO has similar behavior to `std::cout` and `std::cin` of the C++ `iostream` library. However, unlike `std::cout` and `std::cin`, it writes and reads voltages rather than characters.

The commonly applied voltages for the logical *1* and *0* levels are as follows:

Logical level	Voltage (V)
1 or HIGH	Vcc, microcontroller supply voltage (e.g., 3.3 V, 5V)
0 or LOW	GND

Figure 2.34: Relation between logical levels and voltages

Since GPIO can supply the voltages programmatically, we can use this peripheral to drive an LED.

There are two ways to connect the LED with the GPIO pin, and the direction of the current makes them different. The first way is current sourcing, where the current flows out of the microcontroller board. To do so, we need to do the following:

1. Connect the LED anode to the GPIO pin.
2. Connect the LED cathode to the resistor in the series.
3. Connect the remaining resistor terminal to GND.

The following circuit shows how to drive an LED with a current sourcing circuit:

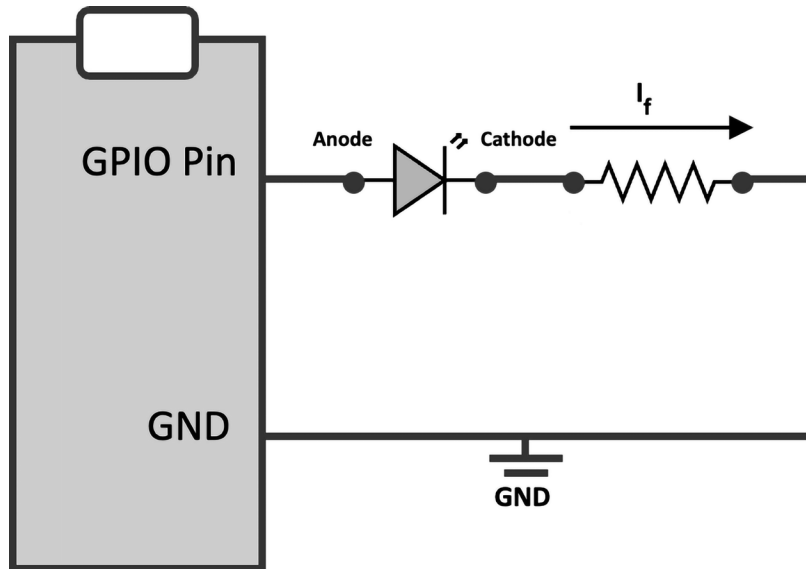


Figure 2.35: Current sourcing – the current goes out of the microcontroller board

The preceding circuit shows that the GPIO pin should supply the logical level 1 to turn on the LED.

The second and opposite way is current sinking, where the current flows into the microcontroller board. In this case, we need to do the following:

1. Connect the LED cathode to the GPIO pin.
2. Connect the LED anode to the resistor in series.
3. Connect the remaining resistor terminal to 3.3 V.

As we can observe from the following circuit, the GPIO pin should supply the logical level 0 to turn on the LED:

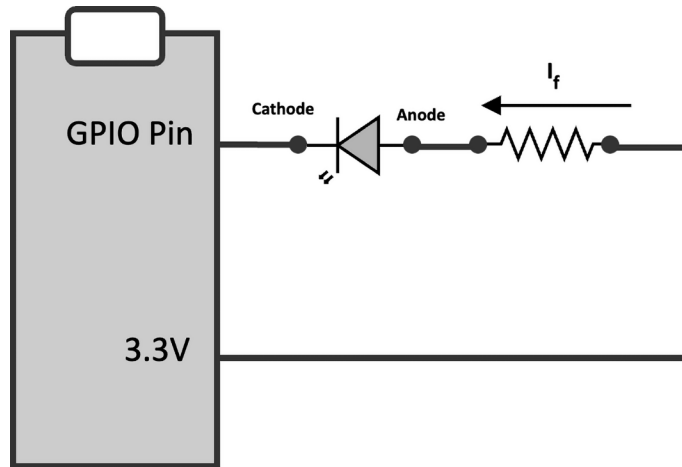


Figure 2.36: Current sinking – the current goes into the microcontroller board

Whatever solution we adopt, it is crucial to remember that pins have limits on the maximum current, which can differ depending on its direction. For example, the Arduino Nano has a maximum output current of 15 mA and a maximum input current of 5 mA. Therefore, when designing a circuit to drive an LED, we must always consider these limits to avoid damaging the device.

Before we can move on to the *How to do it...* section, there is one more piece of information we need to have: how to program the GPIO peripheral in software.

Using the `mbed::DigitalOut` function

In this recipe, we will use the `mbed::DigitalOut` object (<https://os.mbed.com/docs/mbed-os/v6.15/apis/digitalout.html>) from **Mbed OS** to “write” the **LOW (0)** or **HIGH (1)** digital signals to the GPIO pin.



Mbed OS (<https://os.mbed.com/>) is a **real-time operating system (RTOS)** specifically for **Arm Cortex-M** processors, which offers functionalities typical of a canonical OS and drivers to control microcontroller peripherals. The benefit of using Mbed OS is that it allows writing portable code for a wide range of Arm-based microcontrollers, such as Arduino Nano, Raspberry Pi Pico, SparkFun Nano, and many others.

When using the `mbed::DigitalOut()` function, you must provide the **pin name** (`PinName`) corresponding to the pin you want to program. The `PinName` is a number mapped to your microcontroller's actual pin name, reported in your board's pinout diagram.

For example, on the Arduino Nano, you might use the `PinName` value `P0_23` to refer to the **P0_23** pin reported in the pinout diagram. On the other hand, on the Raspberry Pi Pico, you might use the `PinName` value `p22` to refer to the pin reported as **GP22**. In other words, the specific `PinName` values used for a particular pin will depend on the naming convention used by the board manufacturer.

In the case of the Arduino Nano and Raspberry Pi Pico, the `PinName` C constants are provided within the `PinNames.h` header file of `ArduinoCore-mbed` (<https://github.com/arduino/ArduinoCore-mbed/tree/3.5.1/>). Each device has its own `PinNames.h` header file that you can find within the `targets/` directory in `ArduinoCore-mbed`: <https://github.com/arduino/ArduinoCore-mbed/tree/3.5.1/cores/arduino/mbed/targets>.

How to do it...

In this part, we will build the electronic circuit and program the GPIO peripheral in output mode. The upcoming subsections will take you through the necessary steps to accomplish these tasks.

Connecting the LED to the GPIO pin

Disconnect the microcontroller boards from the power and keep the LED and resistor on the breadboard. However, unplug all the jumper wires except the one connected to the - bus rail (**GND**).

The following diagram shows what you should have on the breadboard:

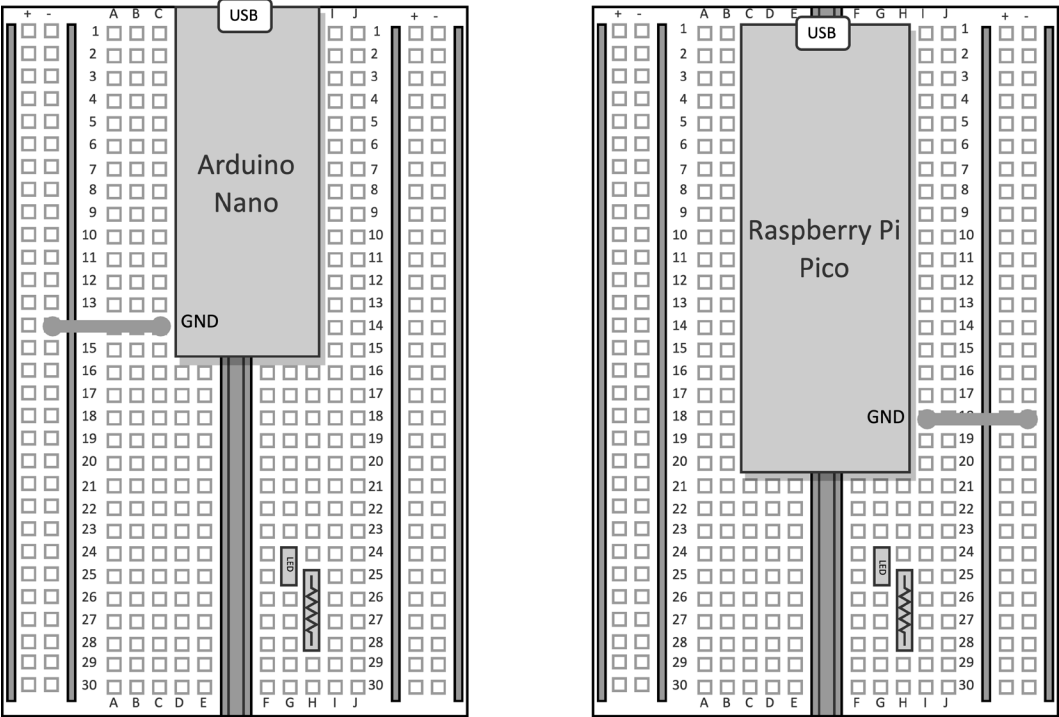


Figure 2.37: Starting components on the breadboard

Since the LED cathode is connected to the terminal resistor, the LED will be driven by a current-sourcing circuit.

The following steps show how to turn on the LED light through the GPIO peripheral.

Step 1:

Choose the GPIO pin to drive the LED. The following table reports our choice:

Platform	GPIO Pin
Arduino Nano	P0.23
Raspberry Pi Pico	GP22

Figure 2.38: The selected GPIO pins for driving the LED

Then, connect the LED anode to the GPIO pin with a jumper wire:

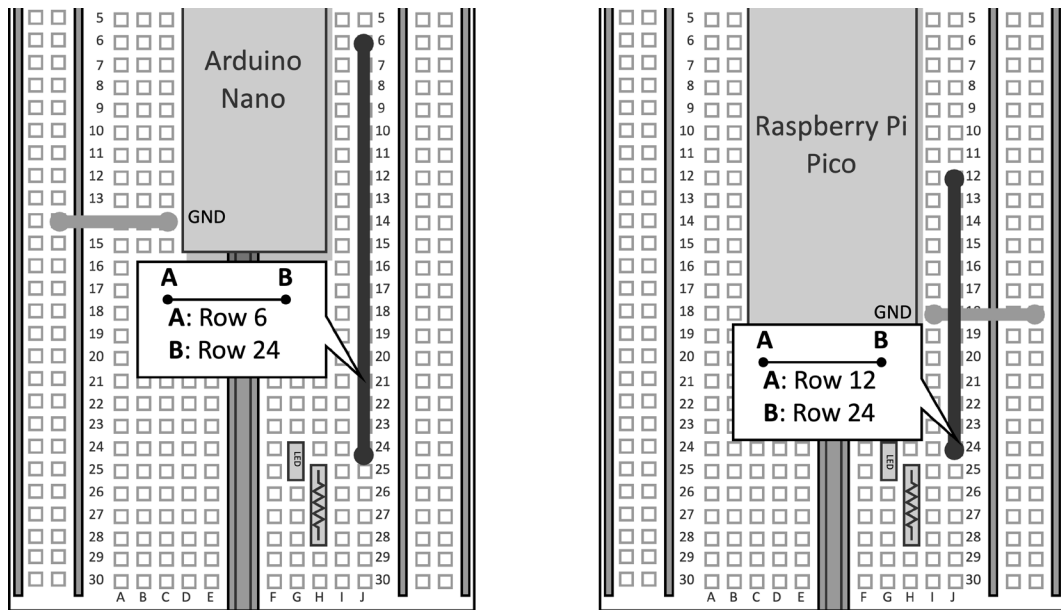


Figure 2.39: Connect the LED anode to the GPIO pin

On the Arduino Nano, we use a jumper wire to connect (J, 6) with (J, 24). On the Raspberry Pi Pico, we use a jumper wire to connect (J, 12) with (J, 24).

Step 2:

Connect the terminal resistor to GND:

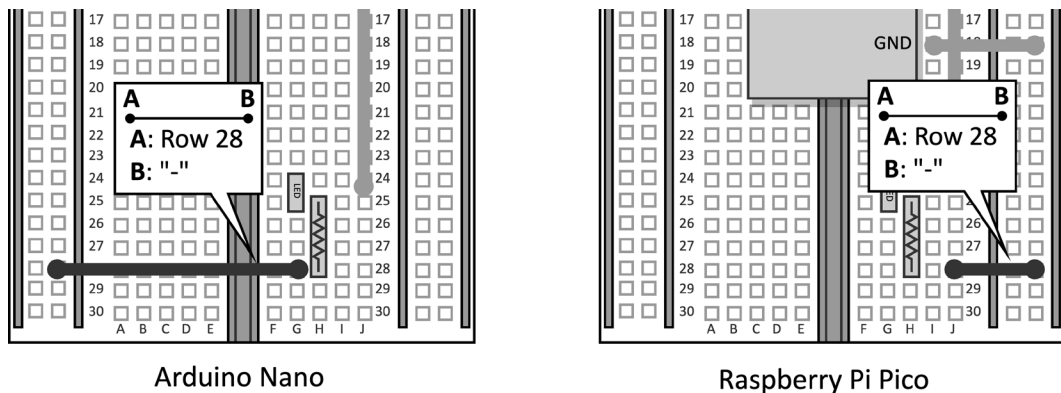


Figure 2.40: Connect the resistor to GND

As the previous image shows, we link (J, 28) with the - bus rail on the Arduino Nano and Raspberry Pi Pico.

The 220 Ω resistor imposes an LED current of ~5 mA, which is below the maximum 20 mA LED current and below the maximum output GPIO current, as reported in the following table:

Platform	Max GPIO current (sourcing) – mA
Arduino Nano	12
Raspberry Pi Pico	10

Figure 2.41: Max GPIO current (sourcing) on the Arduino Nano and Raspberry Pi Pico

Since the electronic circuit is completed, we can shift our focus to the software part.

Programming the GPIO peripheral in output mode

Open the Arduino IDE and create a new sketch. Then, follow the next steps to program the GPIO and turn the LED on.

Step 1:

Include the `mbed.h` header file to access the Mbed OS functionalities:

```
#include "mbed.h"
```

Step 2:

Declare and initialize the `PinName` variable to be passed to the `mbed::DigitalOut()` function:

For the Arduino Nano, we have the following:

```
const PinName gpio_pin_out = P0_23;
```

- Instead, for the Raspberry Pi Pico, we have the following:

```
const PinName gpio_pin_out = p22;
```

Then, initialize a global `mbed::DigitalOut` object:

```
mbed::DigitalOut led(gpio_pin_out);
```

The `gpio_pin_out` variable stores the `PinName` mnemonic that corresponds to the physical pin connected to the LED.

Step 3:

Set the `led` variable to 1 to turn on the LED in the `loop()` function:

```
void loop() {  
    led = 1;  
}
```

Compile the sketch and upload the program to the microcontroller. The LED should now emit the light.

There's more...with the SparkFun Artemis Nano!

In this recipe, we learned how to use the GPIO peripheral and control LEDs. To determine whether we have fully grasped the principles behind this project, it might be worth attempting to replicate it on the SparkFun Artemis Nano microcontroller. To do this, we will first need to construct the electronic circuit and then write the code to program the GPIO pin to control the LED.

One challenge you may encounter when working on this exercise is figuring out the correct `PinName` value for the `mbed::DigitalOut` function. To help you with this, you can refer to the `PinNames.h` header file for the SparkFun Artemis Nano, available at the following link: https://github.com/sparkfun/mbed-os-ambiq-apollo3/blob/22c8b6b1ee181d5efcc042d475f5c026545c4f84/targets/TARGET_Ambiq_Micro/TARGET_Apollo3/TARGET_SFE_ARTEMIS_NANO/PinNames.h.

The solution for this exercise using the D13 pin is available in the `Chapter02` folder in the GitHub repository.

If LEDs can visually output information, buttons are certainly the devices that immediately come to mind for allowing interactive input.

The upcoming recipe will delve into the details of push-buttons and the use of the GPIO peripheral to read their state.

Turning an LED on and off with a push-button

In contrast to a PC, where the keyboard, mouse, or touchscreen facilitates human interactions with software applications, the physical button represents the most common way to interact with a microcontroller.

In this recipe, we will learn how to integrate a push-button into the electronic circuit built in the previous recipe. Then, we will employ the GPIO peripheral to detect whether the button is *pushed* or *released* and use this information to control the LED light.

Getting ready

Before diving into the practical part of this recipe, let's start by introducing the operating principles of the push-button.

The operating principles of the push-button

From an electronics point of view, a **push-button** is a device that makes (a.k.a. shorts) or breaks (a.k.a. opens) the connection between two wires. When we press the button, we connect the wires through a mechanical system, allowing the current to flow. However, unlike a standard light switch, the push-button does not keep the wires connected when we stop pressing. Hence, when we don't apply pressure to the button, the wires disconnect, and the current stops flowing.

Thus, the push-button is a type of button that can live in two states: *pushed* or *released*.

Despite having four metal legs, this device is considered a two-terminal device because the contacts on the opposite side (1, 4 and 2, 3) are connected internally, as shown in the following figure:

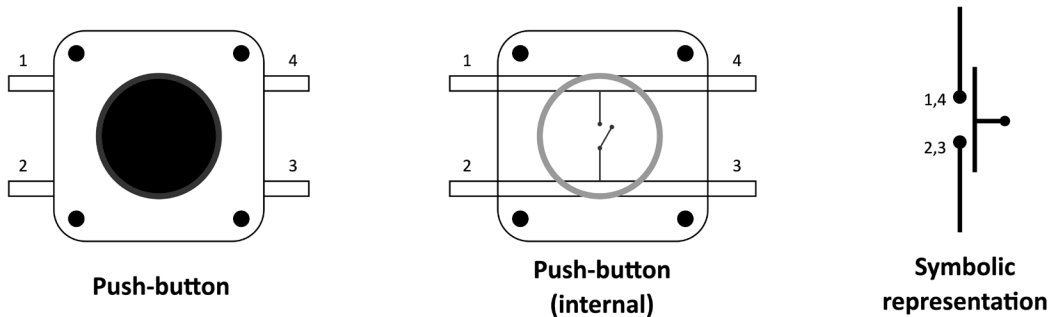


Figure 2.42: Push-button representation

When building a circuit with this component, the legs on the same side (1, 2 or 4, 3 in the preceding figure) are responsible for connecting two points. These two points will have the same voltage when the push-button is pressed. Therefore, we can use the GPIO peripheral to read the applied voltage and determine whether the button is pressed.

In the following diagram, the voltage on the GPIO pin is GND when we press the button. However, what is the voltage when the button is released?

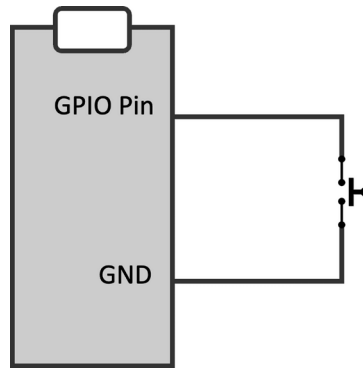


Figure 2.43: What is the voltage on the GPIO pin when we release the push-button?

Although the pin could only assume two logical levels, this could not be true in some input mode circumstances. A third logical level called **floating** (or **high impedance**) could occur if we do not take circuit precautions. When the floating state occurs, the pin's logical level is undefined because the voltage fluctuates between 3.3 V and GND. Since the voltage is not constant, we cannot know whether the push-button is pressed. To prevent this problem, we must include a resistor in our circuit to always have a well-defined logical level under all conditions.

Depending on what logical level we want in the *pushed* state, the resistor can be as follows:

- **Pull-up:** The resistor connects the GPIO pin to the 3.3 V. Thus, the GPIO pin reads *LOW* (0) in the *pushed* state and *HIGH* (1) in the released state.
- **Pull-down:** The resistor connects the GPIO pin to GND in contrast to the pull-up configuration. Thus, the GPIO pin reads the logical level *HIGH* (1) in the pushed state and *LOW* (0) in the released state.

Figure 2.44 shows the difference between the pull-up and pull-down configurations:

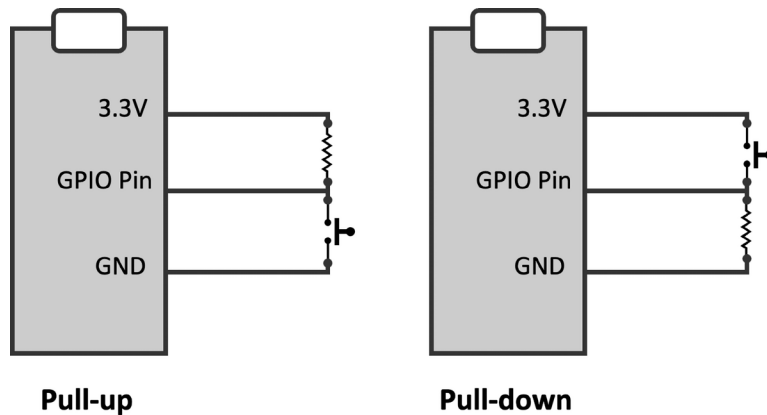


Figure 2.44: Pull-up versus pull-down configurations

Typically, a 10 K resistor should be okay for both cases. However, most microcontrollers offer an internal and programmable pull-up resistor, so the external one is often unnecessary. As we will see in the *How to do it...* section, we will be able to read the digital signal and use the microcontroller's pull-up resistor through the `mbed::DigitalIn()` (<https://os.mbed.com/docs/mbed-os/v6.15/apis/digitalin.html>) function. Similar to the `mbed::DigitalOut()` function, it must be initialized with the GPIO pin (`PinName`) connected to the push-button before being used.

How to do it...

Keep all the components on the breadboard from the previous recipe. The upcoming subsections will take you through the steps to add the push-button in the electronic circuit and program the GPIO peripheral in input mode.

Connecting the push-button to the GPIO pin

The following steps will show how to connect the push-button to the microcontrollers.

Step 1:

Select the GPIO pin to read the state of the push-button. Our selection is presented in the following table:

Platform	GPIO Pin INPUT
Arduino Nano	P0.30
Raspberry Pi Pico	GP10

Figure 2.45: GPIO pin used to read the push-button state

Then, insert the push-button between the left and right terminal strips of the breadboard:

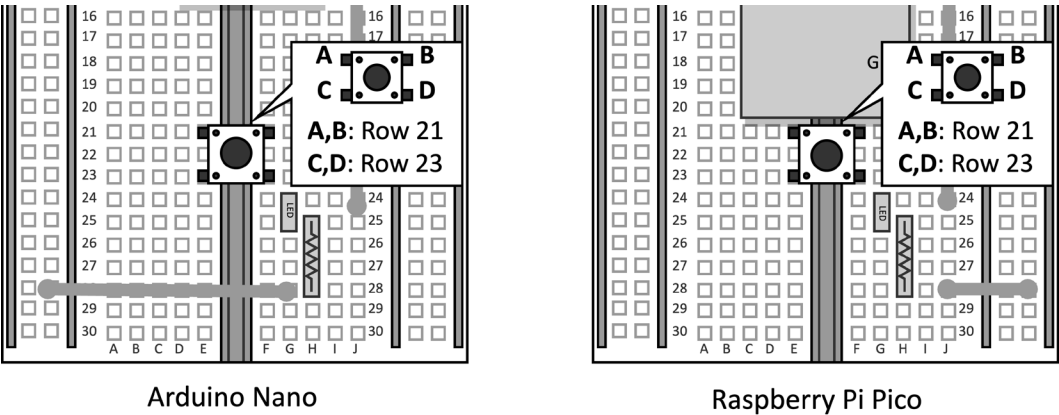


Figure 2.46: The push-button is mounted between the terminal strips 21 and 23

As you can observe from the preceding figure, we use terminal strips that are not employed by any other devices.

Step 2:

Connect the push-button to the GPIO pin and GND:

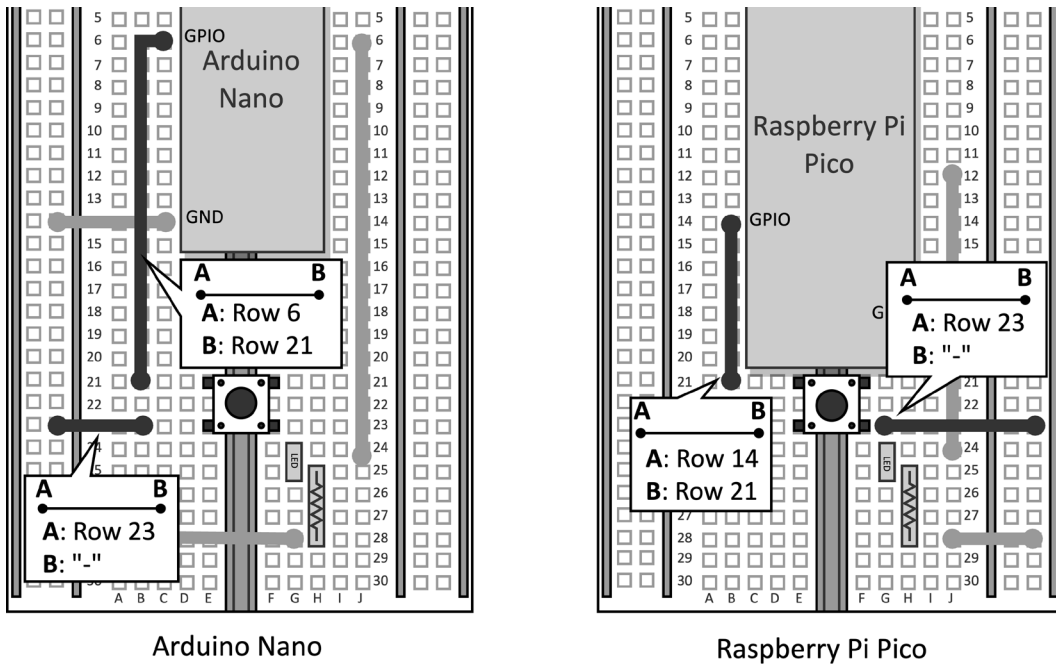


Figure 2.47: The push-button is only connected to the GPIO pin and GND

The electronic circuit has been fully assembled. Nonetheless, upon examining the preceding figure, you might have noticed that neither pull-up nor pull-down resistors have been added. It was a deliberate choice, as we intend to program the GPIO to use the microcontroller pull-up resistor and thus prevent any occurrences of the floating state.

Programming the GPIO peripheral in input mode

At this point, we can open the Arduino IDE and extend the sketch prepared in the previous recipe by following the next steps.

Step 1:

Declare and initialize the `PinName` variable to be passed to the `mbd::DigitalIn()` function:

- For the Arduino Nano, we have the following:

```
const PinName gpio_pin_in = P0_30;
```

- Instead, for the Raspberry Pi Pico, we have the following:

```
const PinName gpio_pin_in = p10;
```

Then, initialize a global `mbed::DigitalIn` object:

```
mbed::DigitalIn button(gpio_pin_in);
```

The `gpio_pin_in` variable stores the `PinName` mnemonic that corresponds to the physical pin connected to the push-button.

Step 2:

Set the button mode to `PullUp` in the `setup()` function:

```
void setup() {  
    button.mode(PullUp);  
}
```

The preceding code enables the microcontroller's internal pull-up resistor.

Step 3:

Turn the LED on when the push-button is *LOW* (0) in the `loop()` function:

```
void loop() {  
    led = !button;  
}
```

As the previous code snippet shows, we set the `led` object to the opposite value returned by the button to light up the LED when the push-button is pressed.

Compile the sketch and upload the program to the microcontroller. Now, the LED should emit light only when you press the push-button.



When the push-button is pressed, the mechanical nature of the component may cause it to generate spurious logical-level transitions. This issue is referred to as **button bouncing**, as the logical value of the push-button will briefly bounce between *HIGH* and *LOW*. To prevent the generation of multiple transitions, you may want to implement a **debouncing algorithm**, such as the one available at the following link: <https://os.mbed.com/teams/TVZ-Mechatronics-Team/wiki/Timers-interrupts-and-tasks>.

There's more...with the SparkFun Artemis Nano!

In this recipe, you learned how to read the push-button state using the `mbed::DigitalIn()` Mbed object.

The same recipe can be implemented effortlessly on the SparkFun Artemis Nano microcontroller board. A solution that employs the D7 pin is available in the Chapter02 folder in the GitHub repository.

The `mbed::DigitalIn()` object is not the sole approach for determining the state of a push-button.

In the upcoming recipe, we will explore a more efficient method for reading the push-button state that can also help minimize power consumption.

Using interrupts to read the push-button state

The previous recipe showed how to read digital signals with the GPIO peripheral. However, the proposed solution is inefficient because the CPU wastes clock cycles waiting for the button to be pressed while it could perform other tasks in the meantime. Furthermore, this could be a situation where we would keep the CPU in low-power mode when there are no other tasks to run.

Therefore, this recipe will show you how to change the sketch developed in the previous recipe to read the push-button state efficiently using **interrupts**.

Getting ready

Let's prepare this recipe by learning what an interrupt is and which Mbed OS API we can use to read the push-button efficiently.

Working with interrupts using the Mbed OS API

An **interrupt** is a signal that temporarily pauses the main program to address an event through a dedicated function known as an **interrupt handler** or **interrupt service routine (ISR)**. When the ISR ends the execution, the processor resumes the main program from the point at which it was interrupted, as shown in the following diagram:

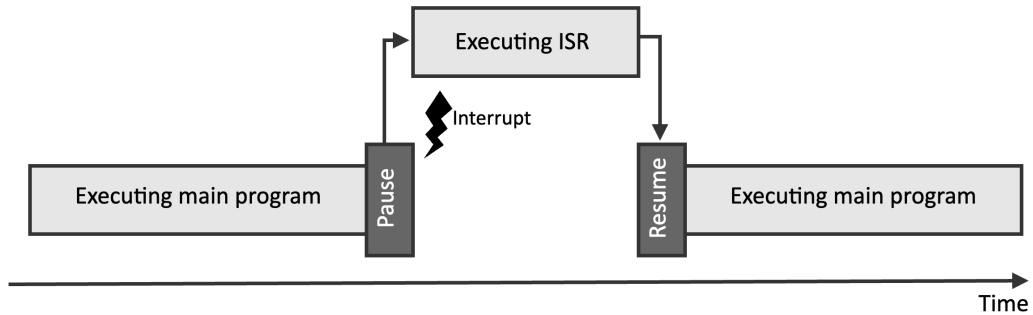


Figure 2.48: Interrupt pauses the main program temporarily

An interrupt is a very efficient mechanism to save energy because we can put the CPU in a sleep state and wait for an event before starting the computation.

A microcontroller typically has multiple interrupt sources, and we can write a dedicated ISR for each one. Despite being a function, certain constraints limit the implementation of an ISR, such as:

- It does not have input arguments.
- It does not return a value. Therefore, we need to use global values to report status changes.
- It must be short to not steal too much time from the main program. We want to remind you that *the ISR is not a thread* since the processor can only resume the computation when the ISR finishes.

When using GPIO peripherals to read digital signals, the `mbed::InterruptIn` object (<https://os.mbed.com/docs/mbed-os/v6.15/apis/interruptin.html>) can be employed to trigger an event whenever the logical level on the pin changes, as shown in Figure 2.49:

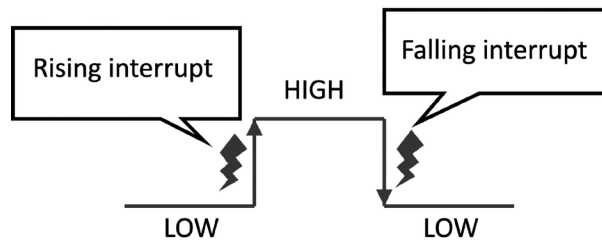


Figure 2.49: Rising interrupt versus falling interrupt

As we can observe from the preceding diagram, the `mbed::InterruptIn` object can trigger interrupts when the logical level on the pin goes from *LOW* to *HIGH* (*rising interrupts*) or *HIGH* to *LOW* (*falling interrupt*). The initialization of this routine is the same as the `mbed::DigitalIn()` one.

How to do it...

Open the sketch of the previous recipe and follow the next steps to control the LED light with the GPIO interrupt.

Step 1:

Define and initialize the `mbed::InterruptIn` object with the `PinName` of the GPIO pin connected to the push-button:

```
mbed::InterruptIn button(gpio_pin_in);
```

You can remove the `mbed::DigitalIn` object from the sketch, as it is not required anymore.

Step 2:

Write the ISR for handling the interrupt request on the rising edge (*LOW* to *HIGH*) of the input signal:

```
void rise_ISR() {  
    led = 0;  
}
```

The `rise_ISR()` ISR routine will turn the LED off (`led = 0`) when the digital signal on the pin changes from *LOW* to *HIGH*.

Step 3:

Write the ISR for handling the interrupt request on the falling edge (*HIGH* to *LOW*) of the input signal:

```
void fall_ISR() {  
    led = 1;  
}
```

The `fall_ISR()` ISR routine will turn the LED on (`led = 1`) when the digital signal on the pin changes from *HIGH* to *LOW*.

Step 4:

Initialize the button variable in the `setup()` function:

```
void setup() {  
    button.mode(PullUp);  
    button.rise(&rise_ISR);  
    button.fall(&fall_ISR);  
}
```

In the previous code, we configured the `mbed::InterruptIn` object by doing the following:

- Enabling the internal pull-up resistor (`button.mode(PullUp)`)
- Attaching the ISR function to run when the rising interrupt occurs (`button.rise(&rise_ISR)`)
- Attaching the ISR function to run when the falling interrupt occurs (`button.fall(&fall_ISR)`)

Now, replace the code in the `loop()` function with `delay(4000)`:

```
void loop() {  
    delay(4000);  
}
```

In the previous code snippet, we use the `delay()` function to pause the program for four seconds. In theory, we could leave the `loop()` function empty. However, we recommend using the `delay()` routine when there are no tasks to be performed, as it can put the system in low-power mode, thus saving energy.

Compile the sketch and upload the program to the microcontroller. You should still be able to control the LED light with the push-button.

There's more...

In this recipe, we learned how to read the GPIO input signal using the efficient mechanism of interrupts. If we use the Mbed OS API, we only need to replace the `mbed::DigitalIn()` function with the `mbed::InterruptIn()` one, write the ISRs, and that's it. This type of interrupt is just one example of the many interrupt sources accessible within a microcontroller. For instance, another extensively used interrupt originates from the **Timer** peripheral, which allows tasks to be executed at specified intervals.

Summary

The recipes presented in this chapter covered the basic principles of microcontroller programming, a prerequisite for the projects developed in the rest of this book.

In the first part, we learned how to use the microcontroller to transmit data serially to the computer for generating files to upload to Google Drive.

Then, our focus shifted to the principles of controlling the LED light through the GPIO peripheral. These recipes taught us how to build electronic circuits on the breadboard, determine the appropriate resistor based on the LED emitting light color, and program the GPIO peripheral to output digital signals.

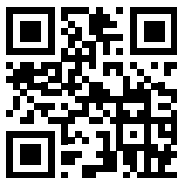
Finally, we discovered how to attach a push-button to the microcontroller and program the GPIO peripheral to read its state. Upon completing this chapter, you should be well prepared to delve into developing your first tinyML project.

In the next chapter, we will implement a basic weather station to predict the occurrence of snow-fall using the temperature and humidity sensor.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



3

Building a Weather Station with TensorFlow Lite for Microcontrollers

Nowadays, we can quickly obtain information about the weather thanks to the convenience of smartphones, laptops, and tablets connected to the internet. However, have you ever considered how you would forecast the weather in remote regions without internet access?

This chapter will teach us how to implement a simple weather station with **machine learning (ML)** to predict the occurrence of snowfall based on the temperature and humidity of the last three hours.

In this chapter, we will focus on dataset preparation and show how to acquire historical weather data from **WorldWeatherOnline**. After preparing the dataset, we will show how to train a **neural network** with **TensorFlow** and quantize the model to 8-bit with **TensorFlow Lite**. In the last part, we will deploy the model on the Arduino Nano and Raspberry Pi Pico with **TensorFlow Lite for Microcontrollers** to build an application capable of predicting the occurrence of snowfall from the temperature and humidity data recorded over the last three hours.

The goal of this chapter is to guide you through all the development stages to create an ML model trained with TensorFlow Lite on Microcontrollers and learn how to acquire temperature and humidity sensor data.

In this chapter, we're going to implement the following recipes:

- Importing weather data from WorldWeatherOnline
- Preparing the dataset
- Training the model with TensorFlow
- Evaluating the model's effectiveness
- Quantizing the model with the TensorFlow Lite converter
- Reading temperature and humidity data with the Arduino Nano
- Reading temperature and humidity data with the DHT22 sensor and the Raspberry Pi Pico
- Preparing the input features for the model inference
- On-device inference with TensorFlow Lite for Microcontrollers

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- An Arduino Nano 33 BLE Sense
- A Raspberry Pi Pico
- A SparkFun RedBoard Artemis Nano (optional)
- A micro-USB data cable
- A USB-C data cable (optional)
- 1 x half-size solderless breadboard (Raspberry Pi Pico only)
- 1 x AM2302 module with the DHT22 sensor (Raspberry Pi Pico only)
- 3 x jumper wires (Raspberry Pi Pico only)
- A laptop/PC with either Linux, macOS, or Windows



The source code and additional material are available in the Chapter03 folder in the GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Chapter03

Importing weather data from WorldWeatherOnline

The effectiveness of ML algorithms depends heavily on the data used for training. Thus, as we commonly say, *the ML model is only as good as the dataset*. The essential requirement for a good dataset is that the data must represent the problem we want to solve.

Therefore, we should consider factors such as temperature and humidity in our specific situation, as they directly impact snow formation.

In this recipe, we will show how to gather historical hourly temperature, humidity, and snowfall data to build a dataset for forecasting snow.

Getting ready

On the internet, there are various sources from which we can gather hourly weather data, but most of them are not free or have limited usage.

For this recipe, **WorldWeatherOnline** (<https://www.worldweatheronline.com/developer/>) has been our choice, which has a free trial period of 30 days and provides the following:

- A simple API through HTTP requests to acquire the data
- Historical worldwide weather data

WorldWeatherOnline provides an API called the *Past Historical Weather API* (<https://www.worldweatheronline.com/developer/premium-api-explorer.aspx>), which allows us to retrieve historical forecasted weather information dating back to January 2009.

In this recipe, we will not directly deal with its native API. Instead, we will employ the Python package `wwo-hist` (<https://github.com/ekapope/WorldWeatherOnline>) to export the data directly to a **pandas DataFrame**.

How to do it...

The first step to download historical weather data from **WorldWeatherOnline** is to register on their website at the following link: <https://www.worldweatheronline.com/developer/>. As a first-time service user, you will be eligible for a 30-day free trial.

Then, open Colab in the web browser and follow these steps:

Step 1:

Install the `wwo-hist` package:

```
!pip install wwo-hist
```

Step 2:

Import the `retrieve_hist_data` function from `wwo-hist`:

```
from wwo_hist import retrieve_hist_data
```


The `retrieve_hist_data` function is required to acquire data from WorldWeatherOnline and export the data to either pandas DataFrames or CSV files.

Step 3:

Acquire data for 10 years (01-JAN-2011 to 31-DEC-2020) with an hourly frequency from *Canazei*:

```
# Hourly frequency
frequency=1
api_key = 'YOUR API KEY'
location_list = ['canazei']

hist_df = retrieve_hist_data(api_key,
                             location_list,
                             '01-JAN-2011',
                             '31-DEC-2020',
                             frequency,
                             location_label = False,
                             export_csv = False,
                             store_df = True)
```

The `www-hist` library will export the data to the `hist_df` object, which is a list of pandas DataFrames.

In this step, the input arguments provided to the `retrieve_hist_data` function are as follows:

- **api_key:** The API key is reported in the WorldWeatherOnline subscription dashboard and should replace the `YOUR_API_KEY` string.
- **location_list:** This is the list of locations from which to acquire the weather data. Since we are building a dataset to forecast the snow, we should consider places where it snows periodically. For example, you can consider *Canazei* (<https://en.wikipedia.org/wiki/Canazei>), located in northern Italy, where snowfall can occur between December and March. We could also include other locations to make the ML model more generic.
- **Start and end dates:** The start and end dates define the time interval to gather the data. The date format is `dd-mmm-yyyy`. Since we want a large, representative dataset, we query 10 years of weather data. Therefore, the time interval is set to `01-JAN-2011 - 31-DEC-2020`.
- **frequency:** This parameter defines the hourly frequency. For example, `1` stands for every hour, `3` for every three hours, `6` for every six hours, and so on. We opt for an *hourly frequency* since we need the temperature and humidity of the last three hours to forecast snow.

- `location_label`: Since we might need to acquire data from different locations, this flag binds the acquired weather data to the place. We set this option to `False` because we only use a single location.
- `export_csv`: This flag is used to export the weather data to a CSV file. We set it to `False`, allowing us to save the dataset locally for convenient reuse whenever necessary.
- `store_df`: This flag is used to export the weather data to a pandas DataFrame. We set it to `True`.

Once the weather data is retrieved, the `www-hist` library will report the `export to canazei completed!` string on the console output.

Step 4:

Export the temperature, humidity, and snowfall output data and express them in list format:

```
t_list = hist_df[0].tempC.astype(float).to_list()
h_list = hist_df[0].humidity.astype(float).to_list()
s_list = hist_df[0].totalSnow_cm.astype(float).to_list()
```

The generated `hist_df[]` dataset includes several weather conditions for each requested date and time. For example, we can find the pressure in millibars, cloud coverage in percentage, visibility in kilometers, and, of course, the physical quantities that we're interested in:

- `tempC`: The temperature in degrees Celsius (°C)
- `humidity`: The relative air humidity in percentage (%)
- `totalSnow_cm`: The total snowfall in centimeters (cm)

In this final step, we export the hourly temperature, humidity, and snowfall in cm to three lists using the `to_list()` method.

Now, we have all we need to prepare the dataset for forecasting the snow.

There's more...

In this recipe, we learned how to acquire historical weather from **WorldWeatherOnline** to build a dataset for forecasting snow using the `woo-hist` Python module.

An alternative approach to retrieve the historical weather data is through the **Historical Weather API**, which supports **GET** (<https://www.worldweatheronline.com/weather-api/api/docs/historical-weather-api.aspx>).

Within the Colab notebook available in the Chapter03 directory of the GitHub repository, we have included the Python code to offer guidance on using this API to collect historical weather data.

After obtaining the relevant physical variables for our task, our next step is to build the dataset.

In the upcoming recipe, we will see how to build a balanced dataset for training a binary classification model to forecast snow.

Preparing the dataset

The dataset preparation is a crucial phase in any ML project because it has implications for the effectiveness of the resulting trained model.

In this recipe, we will create a dataset from the physical quantities acquired earlier by adopting two techniques to make it suitable for training an accurate model. These two techniques will allow us to obtain a balanced dataset and bring its values into the same numerical range.

Getting ready

The dataset required for our task must be prepared to train a **binary** classification model, meaning the output can only belong to two classes: *Yes, it snows* or *No, it does not snow*. Therefore, mapping the snowfall in cm into these two classes is the first thing we must do. In this project, we have considered the snow formation only when the snowfall in cm is above 5 cm.

The next step for building the dataset concerns the selection of input features to predict the snow. Considering our goal of forecasting snow, it is logical to use historical temperature and humidity data from the past few hours. Therefore, we have chosen the temperature and humidity values from the three preceding hours as input features.

Nonetheless, the dataset preparation extends beyond label preparation and input feature selection. In fact, at least two additional steps are required to ensure the dataset is suitable for training an ML model. The former is for balancing the dataset. The latter is for bringing the input features into a similar numerical range.

In the upcoming subsection, we will address the problem of balancing the dataset.

Balancing the dataset

An **unbalanced dataset** is a dataset where one of the classes has considerably more samples than the others. Training with an unbalanced dataset could produce a model with high accuracy but that is incapable of solving our actual problem. For example, consider a dataset where the class “*No, it does not snow*” has 99% of the samples.

If the network misclassified the “*Yes, it snows*” class, we would still have 99% accuracy, but the model would be ineffective.

Hence, we need a balanced dataset where each output class has approximately the same number of input samples.

Balancing a dataset can be done in various ways, such as:

- **Collecting additional input samples for the underrepresented class:** This should be our initial step to guarantee that the dataset is accurately constructed. However, obtaining more data may not always be feasible, especially when dealing with infrequent events.
- **Oversampling the minority class:** We could randomly duplicate samples from the under-represented class. However, this approach may increase the risk of overfitting the minority class.
- **Undersampling the majority class:** We could randomly delete samples from the over-represented class. Since this approach reduces the dataset’s size, we could lose valuable training information.
- **Generating synthetic samples for the minority class:** We could develop artificially manufactured samples. The most common algorithm for this is **Synthetic Minority Over-sampling Technique (SMOTE)**. SMOTE is an oversampling technique that creates new data instead of duplicating under-represented instances. Although this technique reduces the risk of overfitting caused by oversampling, the generated synthetic samples could be incorrect near the class separation border, adding undesirable noise to the dataset.

As we can see, despite the variety of techniques, there is no overall best solution to fix an unbalanced dataset. The adopting method or methods will depend on the problem to solve.

Once we have balanced the dataset, the next step crucial for the dataset preparation is feature scaling, which ensures that input features have a similar numerical range. In this project, we will employ the Z-score normalization technique, discussed in the following subsection.

Feature scaling with Z-score

Our input features, the humidity and temperature of the last three hours, exist in different numerical ranges.

For example, the humidity is between 0 and 100, while the temperature on the Celsius scale can be negative and have a smaller positive numerical range than humidity.

Generally, if the input features have different numerical ranges, the ML model may be influenced more by those with larger values, impacting its effectiveness. Therefore, the input features must be rescaled to ensure that each input feature contributes equally during training.

Z-score is a common scaling technique adopted in neural networks, and it is defined with the following mathematical formula:

$$value_{new} = \frac{value_{old} - \mu}{\sigma}$$

Let's break down this formula:

- μ : The mean value of the input features
- σ : The standard deviation of the input features

Therefore, the Z-score function can bring the input features to a similar, albeit not identical, numerical range.

How to do it...

Continue working in the Colab notebook and take the following steps to discover how to balance the dataset and rescale the input features with the Z-score function:

Step 1:

Generate the output labels (Yes and No):

```
def gen_label(snow):
    if snow > 5:
        return "Yes"
    else:
        return "No"

labels_list = []

for snow, temp in zip(s_list, t_list):
    labels_list.append(gen_label(snow))
```

Since we are only forecasting snow, only two classes are needed: Yes, it snows or No, it does not snow. At this scope, we convert totalSnow_cm to the corresponding class (Yes or No) through the gen_label() function.

The mapping function assigns Yes when totalSnow_cm exceeds 5 cm.

Step 2:

Build the dataset:

```
import pandas as pd
csv_header = ["Temp0", "Temp1", "Temp2",
              "Humi0", "Humi1", "Humi2", "Snow"]
dataset_df = pd.DataFrame(list(zip(
    t_list[:-2], t_list[1:-1], t_list[2:],
    h_list[:-2], h_list[1:-1], h_list[2:],
    labels_list[2:])), columns = csv_header)
```

If t_0 is the current time, the values stored in the dataset are as follows:

- $t_list[:-2]/h_list[:-2]$: Temperature/humidity at time $t = t_0 - 2$
- $t_list[1:-1]/h_list[1:-1]$: Temperature/humidity at time $t = t_0 - 1$
- $t_list[2:]/h_list[2:]$: Temperature/humidity at time $t = t_0$
- $labels_list[2:]$: Label reporting whether it will snow at time $t = t_0$

Therefore, we just need a zip operation and a few index calculations to build the dataset.

Step 3:

Balance the dataset by undersampling the majority class:

```
df0 = dataset_df[dataset_df['Snow'] == "No"]
df1 = dataset_df[dataset_df['Snow'] == "Yes"]

if len(df1.index) < len(df0.index):
    df0_sub = df0.sample(len(df1.index))
    dataset_df = pd.concat([df0_sub, df1])
else:
    df1_sub = df1.sample(len(df0.index))
    dataset_df = pd.concat([df1_sub, df0])
```

The initial dataset is imbalanced due to the nature of snowfall in the chosen location, which generally occurs during the winter season from December to March.

As demonstrated in the subsequent bar chart, the No class accounts for 98.6% of all instances:

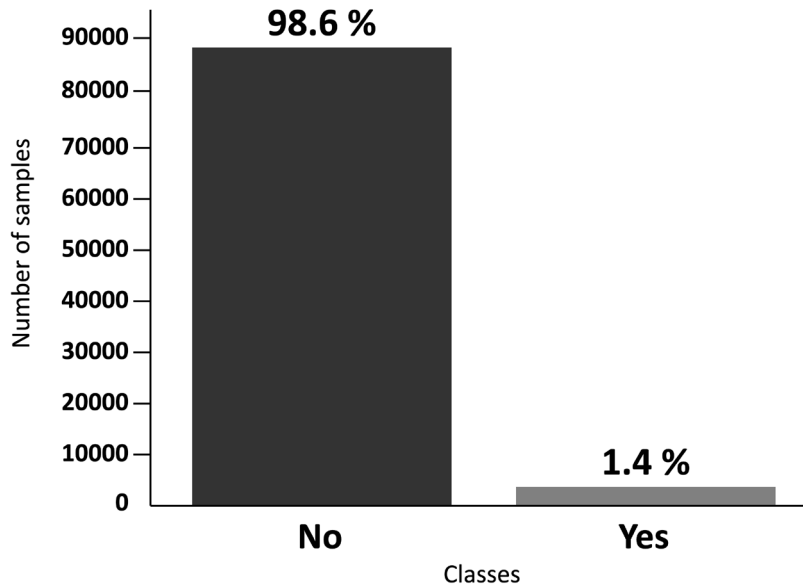


Figure 3.1: Distribution of the dataset samples

Therefore, as you can see from the bar chart, we need to employ one of the methods outlined in the previous *Getting ready* section to achieve a balanced dataset.

Since the minority class already contains enough samples for training the model (1,200), we can randomly undersample the majority class so the two categories have the same number of observations.

Step 4:

Scale the input features with the Z-score independently. To do so, extract all the temperature and humidity values:

```
t_list = dataset_df['Temp0'].tolist()
h_list = dataset_df['Humi0'].tolist()
t_list = t_list + dataset_df['Temp2'].tail(2).tolist()
h_list = h_list + dataset_df['Humi2'].tail(2).tolist()
```

Next, calculate the mean and standard deviation for the temperature and humidity input features and display them on the output console:

```
import numpy as np
from numpy import mean
from numpy import std

t_avg = mean(t_list)
h_avg = mean(h_list)
t_std = std(t_list)
h_std = std(h_list)
print("COPY ME!")
print("Temperature - [MEAN, STD] ", round(t_avg, 5), round(t_std, 5))
print("Humidity - [MEAN, STD]     ", round(h_avg, 5), round(h_std, 5))
```

The preceding code will print the mean and standard deviation for the temperature and humidity input features on the output console.



Copy the mean and standard deviation values because they will be required when deploying the application on the microcontroller.

Finally, scale the input features with the Z-score:

```
def scaling(val, avg, std):
    return (val - avg) / (std)

dataset_df['Temp0'] = dataset_df['Temp0'].apply(lambda x: scaling(x, t_avg, t_std))
dataset_df['Temp1'] = dataset_df['Temp1'].apply(lambda x: scaling(x, t_avg, t_std))
dataset_df['Temp2'] = dataset_df['Temp2'].apply(lambda x: scaling(x, t_avg, t_std))
dataset_df['Humi0'] = dataset_df['Humi0'].apply(lambda x: scaling(x, h_avg, h_std))
dataset_df['Humi1'] = dataset_df['Humi1'].apply(lambda x: scaling(x, h_avg, h_std))
dataset_df['Humi2'] = dataset_df['Humi2'].apply(lambda x: scaling(x, h_avg, h_std))
```


Now, the dataset is ready for training our model!

There's more...

In this recipe, we learned how to build a balanced dataset for training a model to forecast snow from temperature and humidity data.

Suppose you are interested in seeing how normalization affects the range of input features. In that case, you can display the distribution of values before and after applying Z-score normalization using the code that follows:

```
import seaborn as sns
import matplotlib.pyplot as plt

t_norm_list = dataset_df['Temp0'].tolist()
h_norm_list = dataset_df['Humi0'].tolist()

fig, ax=plt.subplots(1,2)
ax[0].set_title("Raw temperature")
ax[1].set_title("Raw humidity")
sns.histplot(t_list, ax=ax[0], kde=True)
sns.histplot(h_list, ax=ax[1], kde=True)

fig, ax=plt.subplots(1,2)
sns.histplot(t_norm_list, ax=ax[0], kde=True)
ax[0].set_title("Scaled temperature")
ax[1].set_title("Scaled humidity")
sns.histplot(h_norm_list, ax=ax[1], kde=True)
```

The preceding code should produce the following charts that compare the raw and scaled input feature distributions:

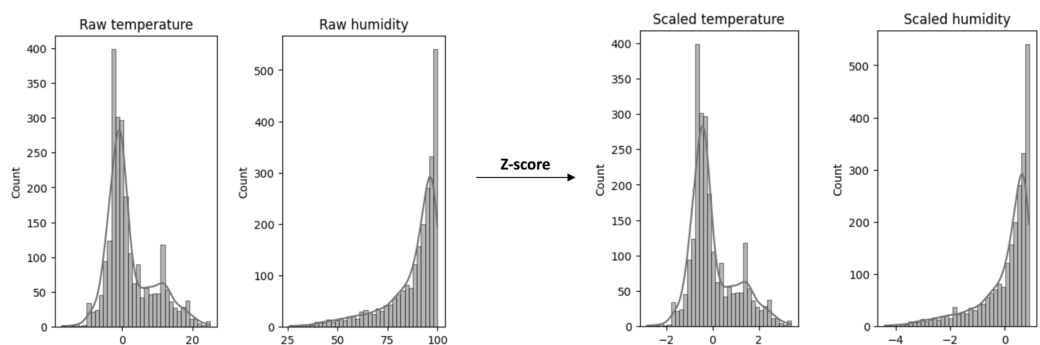


Figure 3.2: Raw (left charts) and scaled (right charts) input feature distributions

As you can observe from the charts, Z-score normalization results in a similar value range (x axis) for both features, centered around 0.

Now that we have the dataset, we are ready to train the model.

In the upcoming recipe, we will design and train a simple classifier to forecast the snow using TensorFlow.

Training the model with TensorFlow

The model designed for forecasting the snow is a simple binary classifier based on a neural network, and it is illustrated in the following diagram:

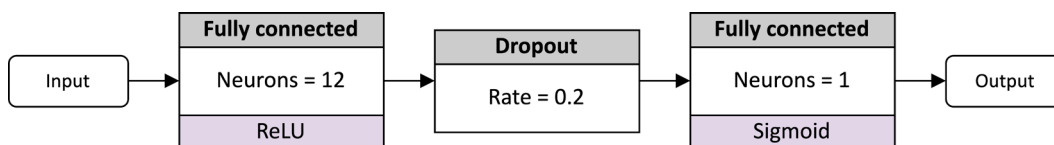


Figure 3.3: Neural network model for forecasting snow

The network consists of the following layers:

1. 1 x **fully connected** layer with 12 neurons followed by a ReLU activation function
2. 1 x **dropout** layer with a 20% rate (0.2) to prevent **overfitting**
3. 1 x **fully connected** layer with one output neuron followed by a sigmoid activation function

In this recipe, we will train the preceding model with TensorFlow.

Getting ready

As you will remember from physics class, the likelihood of snow formation is high when the temperature falls below 0°C (freezing condition) and there is sufficient humidity.

Therefore, given this relationship, we can infer that a basic **feedforward** neural network, like the one illustrated in *Figure 3.3*, should be enough to solve our problem. This model takes six input features (the temperature and humidity for each of the past three hours) and returns the probability of snow formation. As the sigmoid function produces the output, the result ranges between 0 and 1, and it is classified as *No* when it is below 0.5 and as *Yes* otherwise.

Typically, training a neural network involves the following four sequential steps:

1. Encoding the output labels
2. Splitting the dataset into training, test, and validation datasets
3. Creating the model
4. Training the model

In this recipe, we will use TensorFlow and scikit-learn to implement them.

Scikit-learn (<https://scikit-learn.org/>) is a high-level Python library for implementing generic ML algorithms, such as SVM, random forest, and logistic regression. Therefore, it is not a DNN-specific framework but a software library for a wide range of ML algorithms.

How to do it...

Continue working in Colab and take the following steps to train the model presented in *Figure 3.3* with TensorFlow:

Step 1:

Extract the input features (x) and output labels (y) from the dataset (dataset_df):

```
f_names = dataset_df.columns.values[0:6]
l_name  = dataset_df.columns.values[6:7]
x = dataset_df[f_names]
y = dataset_df[l_name]
```

Step 2:

Encode the labels to numerical values:

```
from sklearn.preprocessing import LabelEncoder

labelencoder = LabelEncoder()
labelencoder.fit(y.Snow)
y_encoded = labelencoder.transform(y.Snow)
```

This step converts the output labels (Yes and No) into numerical values since neural networks can only deal with numbers. For this scope, we use scikit-learn to transform the target labels into integer values (0 and 1), which requires calling the following three functions:

- `LabelEncoder()` to initialize the `LabelEncoder` module
- `fit()` to identify the target integer values by parsing the output labels
- `transform()` to translate the output labels into numerical values

After calling the `transform()` function, the encoded labels are available in the `y_encoded` variable.

Step 3:

Split the dataset into train (80%), validation (10%), and test (10%) datasets:

```
from sklearn.model_selection import train_test_split

# Split 1 (80% vs 20%)
x_train, x_validate_test, y_train, y_validate_test = train_test_split(x,
y_encoded, test_size=0.20, random_state = 1)

# Split 2 (50% vs 50%)
x_test, x_validate, y_test, y_validate = train_test_split(x_validate_test,
y_validate_test, test_size=0.50, random_state = 3)
```

The following diagram shows how we split the train, validation, and test datasets:

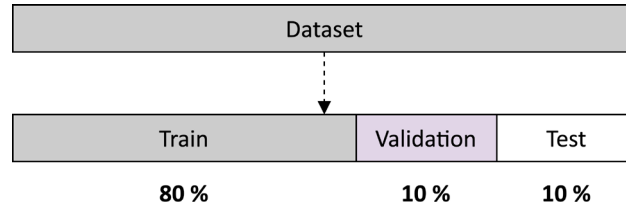


Figure 3.4: Dataset split

These three datasets are as follows:

- **Training dataset:** This dataset contains the samples to train the model. The weights and biases are learned with this data.
- **Validation dataset:** This dataset contains the samples to evaluate the model's accuracy on unseen data. The dataset is used during training to indicate how well the model generalizes because it includes instances not included in the training dataset. However, since this dataset is still used during training, we could indirectly influence the output model by fine-tuning some training hyperparameters.
- **Test dataset:** This dataset contains the samples for testing the model after training. Since the test dataset is not employed during training, it evaluates the final model without bias.

The dataset splitting is done with the `train_test_split()` function from `scikit-learn`, which splits the dataset into training and test datasets. The split proportion is defined with the `test_size` input argument, representing the input dataset's percentage to include in the test split. The `train_test_split()` function is called twice to generate the three datasets. The first split generates the 80% training dataset by providing `test_size=0.20`. The second split produces the validation and test datasets by halving the 20% dataset from the first split.

Step 4:

Create the model with the Keras API:

```
import tensorflow as tf
from tensorflow.keras import layers

model = tf.keras.Sequential()
model.add(layers.Dense(12, activation='relu',
                      input_shape=(len(f_names),)))
```

```
model.add(layers.Dropout(0.2))
model.add(layers.Dense(1, activation='sigmoid'))
model.summary()
```

The preceding code generates the following output:

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 12)	84
dropout (Dropout)	(None, 12)	0
dense_1 (Dense)	(None, 1)	13
Total params: 97		
Trainable params: 97		
Non-trainable params: 0		

Figure 3.5: Model summary returned by `model.summary()`

The summary reports valuable information about the neural network model's architecture, such as the layer types, output shapes, and required trainable weights.



In tinyML, monitoring the number of weights is crucial because it relates to the program's memory utilization.

Looking at the model's architecture, it should be evident that it is relatively small, as it only has 97 trainable parameters.

Step 5:

Compile the model:

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

In this step, we initialize the training parameters as follows:

- **loss**: This is the loss function to minimize during training. The loss indicates how far the predicted output is from the expected result, so the lower the loss, the better the model. **Cross-entropy** is the standard loss function for classification problems because it produces faster training with better model generalization. For a binary classifier, we should use `binary_crossentropy`.
- **optimizer**: This is the algorithm used to update the network weights during training. The optimizer mainly affects the training time. In our example, we use the widely adopted **Adam** optimizer.
- **metrics**: This is the performance metric used to evaluate how well the model predicts the output classes. We use **accuracy**, defined as the *ratio between the number of correct predictions and the total number of tests*, as reported by the following equation:

$$\text{accuracy} = \frac{\text{number of correct predictions}}{\text{total number of tests}}$$

Once we have initialized the training parameters, we are ready to train the model.

Step 6:

Train the model:

```
NUM_EPOCHS=20
BATCH_SIZE=64
history = model.fit(x_train, y_train,
                    epochs=NUM_EPOCHS,
                    batch_size=BATCH_SIZE,
                    validation_data=(x_validate,
                                    y_validate))
```

During training, TensorFlow reports the loss and accuracy after each epoch on both the train and validation datasets, as shown in *Figure 3.6*:

```
loss: 0.3020 - accuracy: 0.8655 - val_loss: 0.2969 - val_accuracy: 0.8756
loss: 0.3024 - accuracy: 0.8659 - val_loss: 0.2961 - val_accuracy: 0.8743
```

Figure 3.6: Accuracy and loss are reported on both the train and validation datasets

The `accuracy` and `loss` values are the accuracy and loss on the train data, while `val_accuracy` and `val_loss` are the accuracy and loss on the validation data.

To prevent overfitting and to see how the model behaves on unseen data, it is advisable to rely on the accuracy and loss of the validation data.

Step 7:

Analyze the accuracy and loss after each training epoch:

```
loss_train = history.history['loss']
loss_val   = history.history['val_loss']
acc_train  = history.history['accuracy']
acc_val    = history.history['val_accuracy']
epochs     = range(1, NUM_EPOCHS + 1)

def plot_train_val_history(x, y_train, y_val, type_txt):
    plt.figure(figsize = (10,7))
    plt.plot(x, y_train, 'g', label='Training'+type_txt)
    plt.plot(x, y_val, 'b', label='Validation'+type_txt)
    plt.title('Training and Validation'+type_txt)
    plt.xlabel('Epochs')
    plt.ylabel(type_txt)
    plt.legend()
    plt.show()

plot_train_val_history(epochs,
                       loss_train, loss_val,
                       "Loss")
plot_train_val_history(epochs,
                       acc_train, acc_val,
                       "Accuracy")
```


The preceding code will display two plots similar to those shown in the following figure:

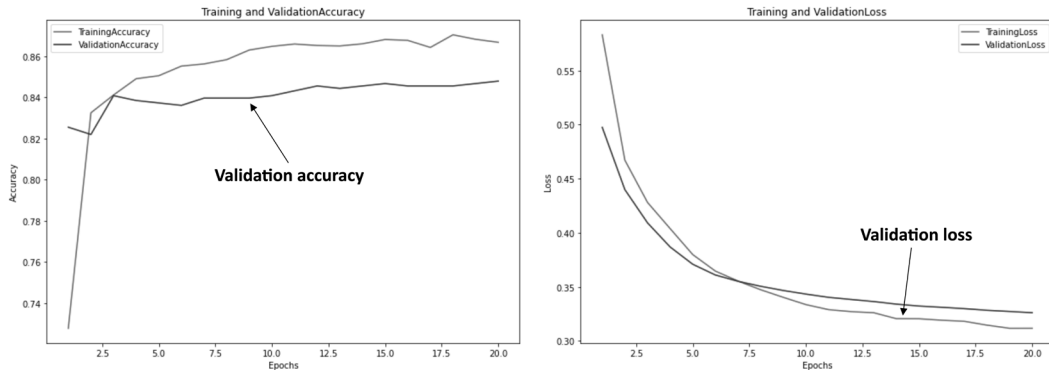


Figure 3.7: Plot of the accuracy (left chart) and loss (right chart) over training epochs

From the plots of the accuracy and loss, we can see the trend of the model's performance. The trend tells us whether we should train less to avoid overfitting or more to prevent underfitting. The validation accuracy and loss are at their best around 10 epochs in our case. Therefore, we should consider terminating the training earlier to prevent overfitting. To do so, you can either re-train the network for 10 epochs or use the `EarlyStopping()` Keras function to stop training when a monitored performance metric has stopped improving. You can discover more about the `EarlyStopping()` function at the following link: https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping.

Step 8:

Save the entire TensorFlow model as a SavedModel:

```
model.save("snow_forecast")
```

The preceding command creates the `snow_forecast` folder, which contains the TensorFlow model as a protobuf binary (with the `.pb` file extension).

There's more...

In this recipe, we learned how to train a simple feedforward neural network to predict the snow from the temperature and humidity of the last three hours.

While the model we trained is small in terms of its trainable parameters and number of layers, it is not necessarily the smallest possible model we can design. For this reason, we encourage you to experiment with changing the number of layers to see if you can achieve similar or better accuracy.

The model's accuracy seems promising, but it is crucial to note that accuracy alone often falls short in assessing the model's effectiveness.

Therefore, in the upcoming recipe, we will assess the model's performance with the confusion matrix and evaluate the recall, precision, and F1-score metrics.

Evaluating the model's effectiveness

Accuracy and loss are not enough to judge the model's effectiveness. In general, *accuracy is a good performance indicator if the dataset is balanced*, but it does not tell us the strengths and weaknesses of our model. For instance, what classes do we recognize with high confidence? What frequent mistakes does the model make?

This recipe will judge the model's effectiveness by visualizing the confusion matrix and evaluating the **recall**, **precision**, and **F1-score** performance metrics.

Getting ready

To complete this recipe, we must familiarize ourselves with the confusion matrix and the alternative performance metrics crucial for evaluating the model's effectiveness. Let's start by learning the confusion matrix in the following subsection.

Evaluating the performance with the confusion matrix

A confusion matrix is an $N \times N$ matrix reporting the number of correct and incorrect predictions on the test dataset, where N is the number of output classes.

For our binary classification model, where there are two output categories, we have a 2x2 matrix like the one in *Figure 3.8*:

Actual	Positive ("Yes")	<div>TP (True Positive)</div>	<div>FP (False Positive)</div>
	Negative ("No")	<div>FN (False Negative)</div>	<div>TN (True Negative)</div>
		Positive ("Yes")	Negative ("No")
		Predicted	

Figure 3.8: A confusion matrix

The four values reported in the previous confusion matrix are as follows:

- **True positive (TP)**: The number of predicted positive results that are actually positive
- **True negative (TN)**: The number of predicted negative results that are actually negative
- **False positive (FP)**: The number of predicted positive results that are actually negative
- **False negative (FN)**: The number of predicted negative results that are actually positive

Ideally, we would like to have 100% accuracy, defined as the ratio of correctly predicted instances (both positive and negative) to the total number of instances in the dataset:

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

The preceding formula implies that the confusion matrix's gray cells (FN and FP) should be 0 to obtain 100% accuracy.

However, although accuracy is a valuable metric, it does not provide a complete picture of model performance. Therefore, the following subsections will present alternative performance metrics for assessing the model's effectiveness.

Evaluating recall, precision, and F-score

The first performance metric we want to present is **recall**, which quantifies *how many of all positive (“Yes”) samples we predicted correctly*:

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

As a result, recall should be as high as possible.

However, this metric does not consider the misclassification of negative samples. Hence, the model could be excellent at classifying positive samples but incapable of classifying negative ones.

For this reason, there is an alternative performance indicator that considers FPs. It is **precision**, which quantifies *how many predicted positive classes (“Yes”) were actually positive*:

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

Therefore, as with recall, precision should be as high as possible.

If we are interested in evaluating both recall and precision simultaneously, the **F-score** metric is what we need. In fact, this metric combines recall and precision with a single formula as follows:

$$f_{score} = \frac{2 \cdot recall \cdot precision}{recall + precision}$$

The higher the F-score, the better the model’s effectiveness.

How to do it...

Continue working in Colab, and follow the steps to visualize the confusion matrix and calculate the recall, precision, and F-score metrics:

Step 1:

Use the trained model to predict the output classes of the test dataset:

```
y_test_pred = model.predict(x_test)
y_test_pred = (y_test_pred > 0.5).astype("int32")
```

The line `y_test_pred = (y_test_pred > 0.5).astype("int32")` binarizes the predicted values using a threshold of 0.5. If a predicted value is greater than 0.5, it is converted to 1.

Otherwise, it is converted to 0.

Step 2:

Compute the confusion matrix with scikit-learn:

```
import sklearn

cm = sklearn.metrics.confusion_matrix(y_test,
                                       y_test_pred)
```

The confusion matrix is obtained with the `confusion_matrix()` function from the scikit-learn library, which takes two arguments: the true labels of the test dataset (`y_test`) and the predicted labels (`y_test_pred`). The `cm` variable stores the confusion matrix.

Step 3:

Display the confusion matrix in a heatmap:

```
index_names = ["Actual No Snow", "Actual Snow"]
column_names = ["Predicted No Snow", "Predicted Snow"]

df_cm = pd.DataFrame(cm, index = index_names,
                     columns = column_names)

plt.figure(dpi=150)
sns.heatmap(df_cm, annot=True, fmt='d', cmap="Blues")
```

The previous code should produce a heatmap similar to the following one:

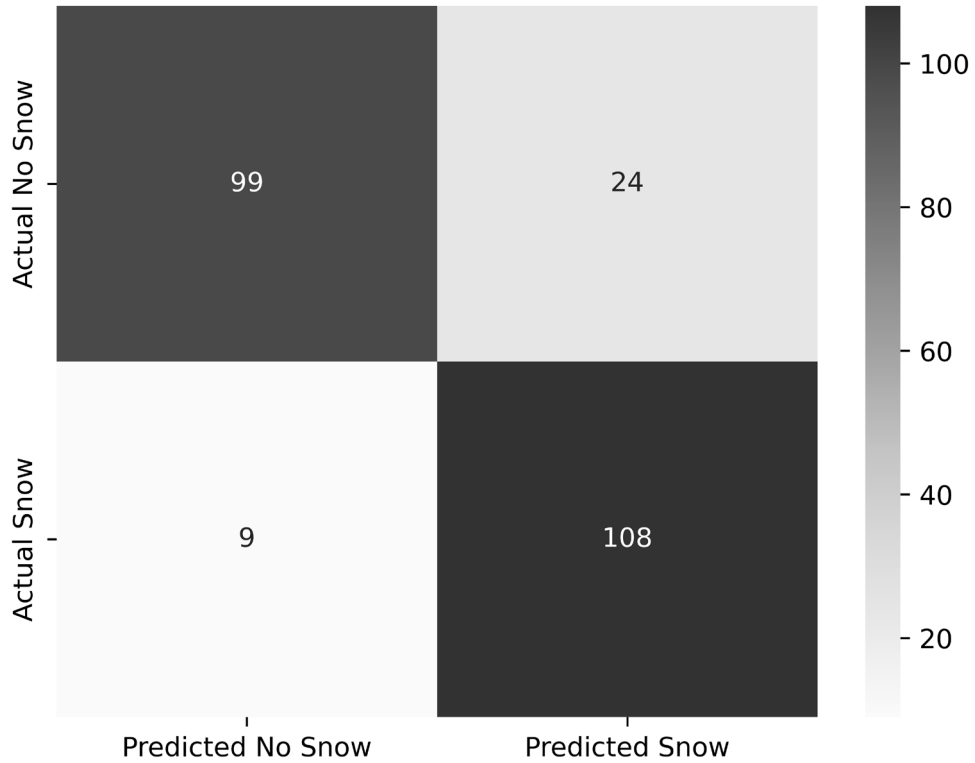


Figure 3.9: Confusion matrix obtained with the test dataset

The confusion matrix shows that the samples are mainly distributed in the leading diagonal, and there are more FPs than FNs. Therefore, although the network is suitable for detecting snow, we should expect some false detections.

Step 4:

Calculate the recall, precision, and F-score performance metrics:

```
TN = cm[0][0]
TP = cm[1][1]
FN = cm[1][0]
FP = cm[0][1]

accur = (TP + TN) / (TP + TN + FN + FP)
precis = TP / (TP + FP)
recall = TP / (TP + FN)
f_score = (2 * recall * precis) / (recall + precis)

print("Accuracy: ", round(accur, 3))
print("Recall:    ", round(recall, 3))
print("Precision: ", round(precis, 3))
print("F-score:   ", round(f_score, 3))
```

The preceding code prints the performance metrics on the output console, resulting in an output similar to what is shown in the following screenshot:

```
Accuracy:  0.862
Recall:    0.923
Precision: 0.818
F-score:   0.867
```

Figure 3.10: Precision, recall, and F-score results

Based on the results reported in the preceding screenshot, which might slightly differ from yours, we can observe that the model has a high recall value of 0.923, indicating that it can accurately predict snowfall. However, the precision value of 0.818 is comparatively lower, meaning the model may produce some false alarms.

The F-score value of 0.867 demonstrates a balance between recall and precision metrics, meaning the model can accurately predict snow instances using the given input features.

There's more...

In this recipe, we learned how to assess the model's effectiveness by visualizing the confusion matrix and evaluating the recall, precision, and F-score metrics.

However, scikit-learn is not the only way to compute the confusion matrix. In fact, TensorFlow also provides a tool to calculate the confusion matrix as well. To delve deeper into this topic, we recommend referring to the TensorFlow documentation at the following link: https://www.tensorflow.org/versions/r2.13/api_docs/python/tf/math/confusion_matrix.

After evaluating the model's effectiveness, the model's quantization is the only step separating us from the beginning of the model deployment on the microcontroller.

In the upcoming recipe, we will compress the trained model by quantizing it to 8-bit using the TensorFlow Lite converter.

Quantizing the model with the TensorFlow Lite converter

The TensorFlow model produced in the previous recipe is well suited for sharing or resuming training sessions. However, the model cannot be used for a microcontroller deployment because of its high memory requirements, which are mainly due to the following reasons:

- The weights are stored in floating-point format
- It keeps information that's not required for the inference

Since our target device has computational and memory constraints, it is crucial to transform the trained model into something more compact.

This recipe will teach you how to convert the trained model into a lightweight format with the help of **TensorFlow Lite** and post-training integer 8-bit quantization.

Getting ready

TensorFlow Lite and post-training integer 8-bit quantization are the main ingredients that make the trained model suitable for inference on devices with reduced memory computational capabilities.

TensorFlow Lite (<https://www.tensorflow.org/lite>) is part of the TensorFlow family and has been designed by Google with the primary objective of offering efficient inference on edge devices such as smartphones or embedded platforms. TensorFlow Lite provides a subset of all TensorFlow operators and, as a result, not all models can be directly exported to TensorFlow Lite. Nevertheless, most popular architectures are well supported.



We recommend reading the *TensorFlow Lite and TensorFlow operator compatibility* guide, available at the following link, to build a TensorFlow model that can be converted efficiently to TensorFlow Lite: https://www.tensorflow.org/lite/guide/ops_compatibility

TensorFlow Lite provides a compact model representation through the **FlatBuffers** (<https://google.github.io/flatbuffers/>) format.



The model is generally stored in a file identified with the `.tflite` extension.

In addition to providing this compact model representation, TensorFlow Lite includes two other key components:

- The **TensorFlow Lite converter**, a tool for converting and optimizing TensorFlow models to TensorFlow Lite format, enabling efficient inference on edge devices
- A library for running TensorFlow Lite models on the target device

The following figure shows the relationship between TensorFlow and TensorFlow Lite components that we have just outlined:

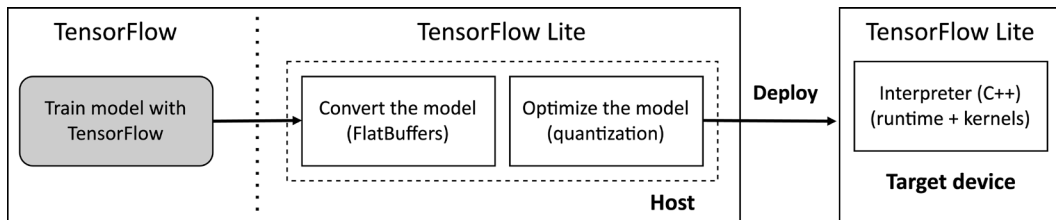


Figure 3.11: TensorFlow Lite components

The TensorFlow Lite converter is responsible for applying optimizations based on 8-bit integer quantization to reduce the model size and improve latency. The upcoming subsection will provide all the necessary details to comprehend the mechanics of quantization.

Model quantization

An indispensable technique to make the model suitable for microcontrollers is **quantization**.

Model quantization, or simply quantization, has three significant advantages:

- It reduces the model size by converting all the weights to lower bit precision.
- It reduces power consumption by reducing the memory bandwidth.
- It improves inference performance by employing integer arithmetic for all the operations.

This widely adopted technique applies the quantization post-training and converts the 32-bit floating-point weights into 8-bit integer values. To understand how quantization works, consider the following C-like functions that are used to convert between a 32-bit floating-point value and an 8-bit value:

```
int8 quantize(float x, float zero_point, float scale) {
    return (int8)(x + zero_point) / scale;
}

float dequantize(int8 x, float zero_point, float scale) {
    return ((float)x - zero_point) * scale;
}
```

In the preceding functions, the scale and zero_point variables are the **quantization parameters**. The scale parameter maps the floating point to the corresponding quantized value and vice versa. The zero_point variable, on the other hand, determines the position of the zero value in the quantized domain.

To understand why the zero_point could not be zero, consider the following distribution of floating-point inputs that need to be scaled to fit within the 8-bit range:

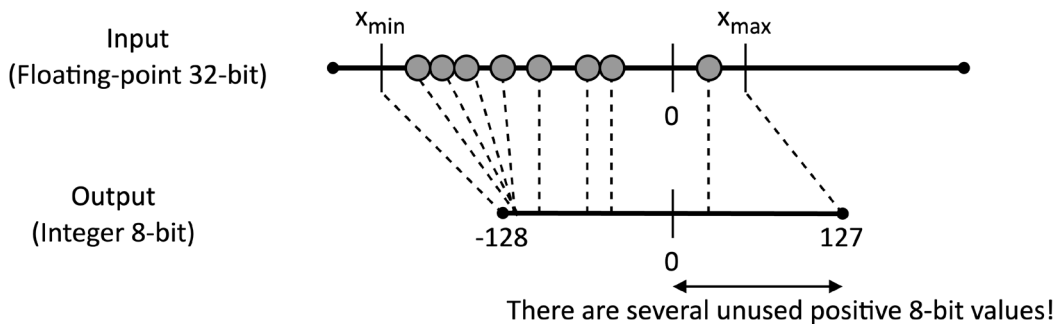


Figure 3.12: The distribution of floating-point values shifted toward the negative range

The preceding figure shows that the floating-point distribution that needs to be scaled to fit within the 8-bit range is not zero-centered; instead, it is skewed toward the negative range. Thus, if we were to scale the floating-point values to 8-bit using a `zero_point` value of 0, we might encounter the following issue:

- Multiple negative input values with the same 8-bit counterpart
- Many positive 8-bit values are unutilized

Therefore, the `zero_point` value set to 0 would be inefficient as we could dedicate a larger range to the negative values to reduce their quantization error, defined as follows:

$$\varepsilon = X_{real} - Z_{quantized}$$

When the `zero_point` value is not 0, we commonly call the quantization **asymmetric** because a different number of quantized values is allocated to the positive and negative floating-point values, as shown in the following diagram:

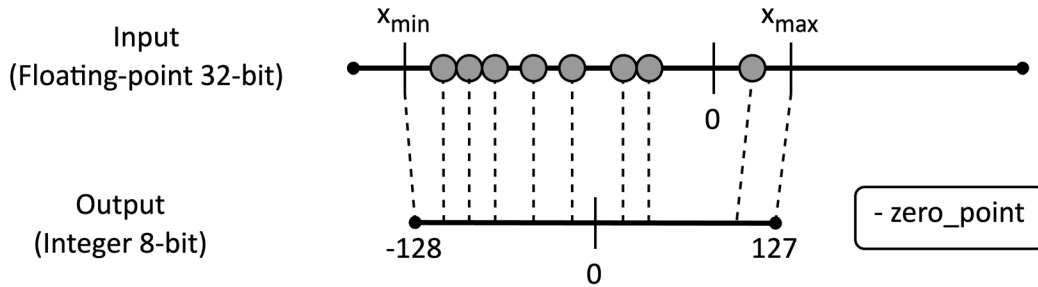


Figure 3.13: Asymmetric quantization

On the other hand, when the `zero_point` value is 0, we commonly call the quantization **symmetric** because an equal number of quantized values is allocated to the positive and negative floating-point values, as we can see in the following diagram:

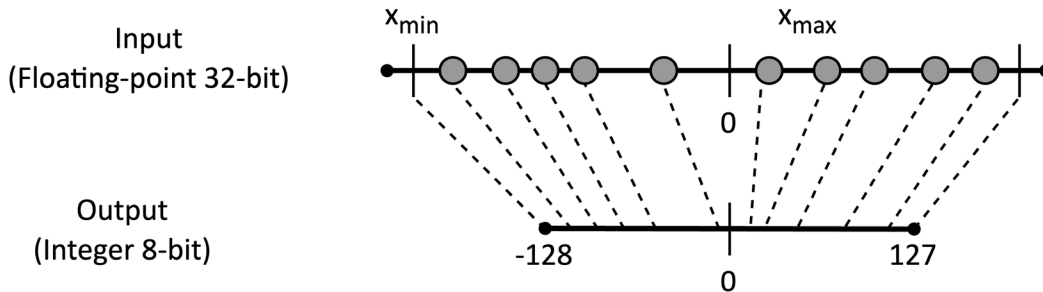


Figure 3.14: Symmetric quantization



In TensorFlow Lite models, the symmetric quantization is typically used for the model's weights, while the asymmetric one is for the input and output of each layer.

The scale and zero_point values are the only parameters required for quantization and are commonly provided in the following ways:

- **Per-tensor:** The quantization parameters are the same for all tensor elements.
- **Per-channel:** The quantization parameters are different for each feature map of the tensor.

The following diagram provides a visual representation of the per-tensor and per-channel quantization:

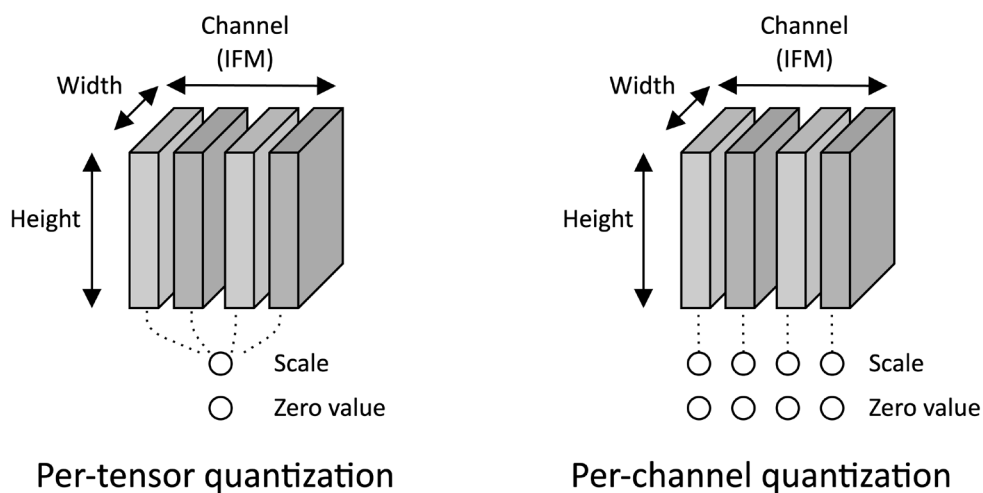


Figure 3.15: Per-tensor versus per-channel quantization

TensorFlow Lite generally adopts the per-tensor approach except for the weights and biases of the convolution and depthwise convolution layers.

How to do it...

Continue working in Colab and follow these steps to convert and quantize the TensorFlow model to TensorFlow Lite:

Step 1:

Select a few hundred samples randomly from the test dataset to calibrate the quantization:

```
def representative_data_gen():
    data = tf.data.Dataset.from_tensor_slices(x_test)
    for i_value in data.batch(1).take(100):
        i_value_f32 = tf.dtypes.cast(i_value, tf.float32)
        yield [i_value_f32]
```

This step, commonly called *generating a representative dataset*, is essential as it helps reduce the risk of an accuracy drop when the model is converted to an 8-bit format. In fact, the converter uses a set of samples to find out the distributions of the floating-point values and estimate the optimal quantization parameters. Typically, 100 samples are enough and can be taken from the test or training dataset. In our case, we used the test dataset.

Step 2:

Initialize the TensorFlow Lite converter with the TensorFlow model:

```
TF_MODEL = "snow_forecast"
converter = tf.lite.TFLiteConverter.from_saved_model(TF_MODEL)
```

Then, initialize the TensorFlow Lite converter arguments for applying the 8-bit quantization:

```
# Representative dataset
converter.representative_dataset = tf.lite.
RepresentativeDataset(representative_data_gen)

# Optimizations
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Supported ops
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_
INT8]
```

```
# Inference input/output type
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
```

The input arguments passed to the tool are as follows:

- **Representative dataset:** This is the representative dataset generated in the first step.
- **Optimizations:** This defines the optimization strategy to adopt. Currently, only DEFAULT optimization is supported, optimizing for size and latency, minimizing the accuracy drop.
- **Supported ops:** This forces the adoption of only integer 8-bit operators during the conversion. If the TensorFlow model has unsupported TensorFlow Lite operators, the conversion will not succeed.
- **Inference input/output type:** This adopts the 8-bit quantization format for the network's input and output. Therefore, we must feed the ML model with the quantized input features to run the inference correctly.

Once we have initialized the converter, we can launch the conversion.

Step 3:

Convert the TensorFlow model to TensorFlow Lite format:

```
tflite_model_quant = converter.convert()
```

Step 4:

To prepare the model for being deployed on microcontrollers, save the TensorFlow Lite model as a .tflite file and convert it to a C-byte array with xxd. So, firstly, save the TensorFlow Lite model as a file:

```
TFL_MODEL_FILE = "snow_model.tflite"
open(TFL_MODEL_FILE, "wb").write(tflite_model_quant)
```

Then, use the xxd tool to convert the TensorFlow Lite model to a constant C-byte array:

```
!apt-get update && apt-get -qq install xxd
!xxd -i "snow_model.tflite" > model.h
!sed -i 's/unsigned char/const unsigned char/g' model.h
!sed -i 's/const/alignas(8) const/g' model.h
```

The previous command outputs a C header file (the `-i` option) containing the TensorFlow model as a constant unsigned char array.



Remember the final two lines where we employ the `sed` command to make the array containing the model `alignas(8) const`. These changes guarantee that the model resides in the program memory (Flash) and that the array is aligned to the 8-byte boundary.

In the *Getting ready* section, we mentioned that the model is typically a file with a `.tflite` extension. Therefore, why do we need this extra conversion? The answer lies in the process of deploying the model on microcontrollers. *Loading a file from a disk on a microcontroller requires an additional software library in the application*. We must remember that most microcontrollers do not have OS and native filesystem support. As a result, the C-byte array format allows us to integrate the model directly into the application and overcome this limitation. The other important reason for this conversion is that *loading a file at runtime does not allow keeping the weights in program memory*. Since every byte matters and the SRAM has a limited capacity, keeping the model in program memory is generally more memory efficient when the weights are constant.

Finally, download the `model.h` file, available from the left pane in Colab.

There's more...

In this recipe, we learned how to quantize the trained model to 8-bit using the TensorFlow Lite converter.

Quantization is a necessary step to make the model compact. However, how much program memory will the model require?

The program memory usage can be quickly determined in Python by printing the size of the TensorFlow Lite model, as shown in the following code snippet:

```
size_tfl_model = len(tflite_model_quant)
print(len(tflite_model_quant), "bytes")
```

As you will see by printing the size of the TensorFlow Lite object, the model will need roughly 2 KB of program memory.

After quantizing the model to 8-bit, we can finally leave the Colab environment to start preparing the application in the Arduino IDE.

In the upcoming recipe, we will discover how to acquire temperature and humidity measurements on the Arduino Nano.

Reading temperature and humidity data with the Arduino Nano

The Arduino Nano and Raspberry Pi Pico have unique hardware features that make them ideal for tackling different development scenarios. For example, the Arduino Nano has a built-in temperature and humidity sensor. As a result, we do not need external components for our project with this board.

In this recipe, we will show how to read a single temperature and humidity sensor data using the Arduino Nano.

Getting ready

Since the temperature and humidity sensor is integrated into the board, no external components need to be connected, and the software library required to interface with the sensor is already integrated into the Arduino Web Editor.



If you use the local Arduino IDE, this library can be installed easily through the Arduino Library Manager. The instructions to install the library have been reported in the following *How to do it...* section.

As a result, to accomplish this task, we only need:

- The C header file to include in the project
- The functions to read the temperature and humidity data from the sensor

Let's start by talking about the functions used to acquire the sensor data, which are:

- `<sensor>.readTemperature()`: Function for temperature readings
- `<sensor>.readHumidity()`: Function for humidity readings

In the previous list, `<sensor>` is a placeholder representing the name of the object created by Arduino to interact with the sensor.

The object's name is determined by the type of Arduino Nano platform used, as Arduino Nano and Arduino Nano Rev2 use different temperature and humidity sensors.

Specifically, the Arduino Nano board integrates the **HTS221** (<https://www.st.com/resource/en/datasheet/HTS221.pdf>) sensor from STMicroelectronics, whereas the Arduino Nano Rev2 employs the **HS300x** (<https://www.renesas.com/us/en/document/dst/hs300x-datasheet>) from Renesas.

If you use the Arduino Nano, the `<sensor>` placeholder must be replaced with `HTS`. Instead, if you use the Arduino Nano Rev2, `<sensor>` must be replaced with `HS300x`.

It is important to note that the C header files to include in the project will differ depending on the type of Arduino Nano. Specifically, if the board is Arduino Nano, the `Arduino_HS300x.h` C header file should be included. Conversely, the `Arduino_HTS221.h` header file must be included if we use the Arduino Nano Rev2 one.

How to do it...

Open the Arduino IDE and create a new sketch.

If you use the local Arduino IDE, you must install the library for using the built-in sensor on the Arduino Nano. To install this library in the local Arduino IDE, navigate to **Sketch > Include Library > Manage Libraries...**, as shown in the following screenshot:

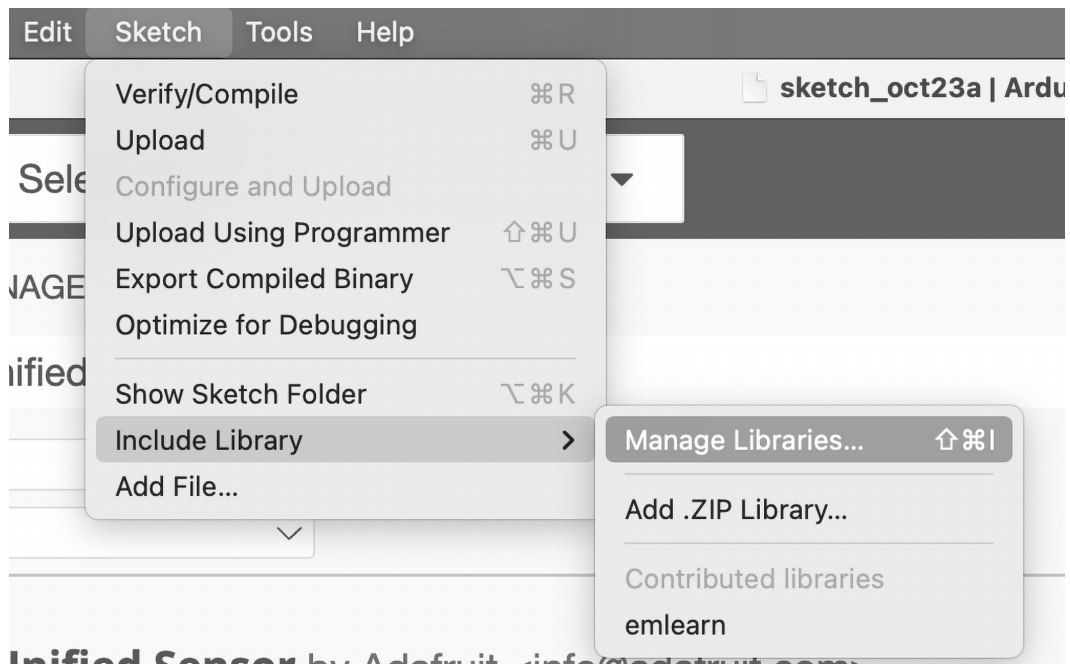


Figure 3.16: Managing libraries from the Arduino IDE

Then, find the **Arduino_HTS221** or **Arduino_HS300x**, if you are using the Arduino Nano Rev2, using the Arduino Library Manager search box, and install it.

Then, take the following steps to initialize and test the temperature and humidity sensor on the Arduino Nano:

Step 1:

Include the C header file to use the temperature and humidity sensor. If you're using the Arduino Nano, include the `Arduino_HTS221.h` C header file in the sketch:

```
#include <Arduino_HTS221.h>
#define SENSOR HTS
```

Otherwise, if you're using the Arduino Nano Rev2, include the `Arduino_HS300x.h` C header file:

```
#include <Arduino_HS300x.h>
#define SENSOR HS300x
```

In the preceding code snippet, the `SENSOR` macro is the `<sensor>` placeholder representing the name of the object created by Arduino to interact with the sensor.

Step 2:

In the `setup()` function, initialize the serial peripheral with a baud rate of 9600 and the temperature and humidity sensor:

```
Serial.begin(9600);

while (!Serial);

if (!SENSOR.begin()) {
    Serial.println("Failed sensor initialization!");
    while (1);
}
```

As you can see from the preceding code snippet, the temperature and humidity sensor is initialized by calling the `begin()` function of the `SENSOR` object.

Step 3:

In the `setup()` function, read the temperature and humidity values and transmit them over the serial connection:

```
Serial.print("Test Temperature = ");
Serial.print(SENSOR.readTemperature(), 2);
Serial.println(" °C");
Serial.print("Test Humidity = ");
Serial.print(SENSOR.readHumidity(), 2);
Serial.println(" %");
```

Now, connect the Arduino Nano to your computer. Then, compile and upload the sketch on the microcontroller. After, open the serial terminal. The device should display the temperature and humidity values measured by the sensor.

There's more...

In this recipe, we learned how to measure the temperature and humidity using the built-in sensor on the Arduino Nano.

This sensor, the first we encounter in the book, is generally known for its low power consumption. In fact, its current draw can be on the order of **micro-ampere** (μA), such as with the built-in sensor on the Arduino Nano and Arduino Nano Rev2. As a result, this sensor is an excellent sensor for applications powered by batteries.

While the Arduino Nano has a built-in temperature and humidity sensor, unfortunately, the Raspberry Pi Pico does not integrate it. Therefore, we require an external one to measure these two physical quantities.

In the upcoming recipe, we will discover what sensor we can use to measure the temperature and humidity with the Raspberry Pi Pico.

Reading temperature and humidity with the DHT22 sensor and the Raspberry Pi Pico

Unlike the Arduino Nano, the Raspberry Pi Pico requires an external sensor and an additional software library to measure the temperature and humidity.

In this recipe, we will show how to use the DHT22 sensor with a Raspberry Pico to get temperature and humidity measurements.

Getting ready

The temperature and humidity sensor module considered for the Raspberry Pi Pico is the low-cost **AM2302**.

As shown in the following diagram, the AM2302 module is a through-hole component with three pins that integrates the **DHT22** temperature and humidity sensor:

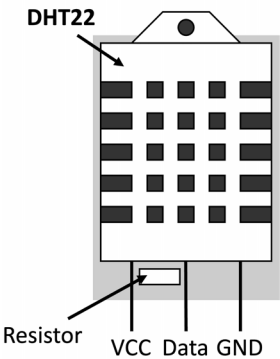



Figure 3.17: The AM2302 module with the DHT22 sensor

Figure 3.18 summarizes the key characteristics of the DHT22 sensor:

Relative humidity range	0 – 100 %
Temperature range	-40°C - 80°C
Humidity accuracy	2-5%
Temperature accuracy	± 0.5°C
Current consumption	2.5mA max when requesting data

Figure 3.18: Key characteristics of the DHT22 temperature and humidity sensor



DHT11 is another popular temperature and humidity sensor from the DHT family. However, we cannot use it in our recipe because of its reduced operating temperature range, which is between 0°C and 50°C.

In contrast to the temperature and humidity sensor on the Arduino Nano, the DHT22 requires an external library to be imported into the Arduino IDE. Adafruit has developed this library, available at the following link: <https://github.com/adafruit/DHT-sensor-library>.

The sensor operates seamlessly with the one on the Arduino Nano, as the functions used to read the temperature and humidity values have the same name and interface: `<sensor>.readTemperature()` and `<sensor>.readHumidity()`. In the case of this external sensor, `<sensor>` is the placeholder representing the name of the DHT object that we should create to interact with the sensor.

This object should be initialized with the Arduino pin number connected to the **data** pin and the type of DHT sensor being used. In our case, the sensor type is **DHT22**.

How to do it...

Create a new sketch on the Arduino IDE and follow the steps to use the DHT22 sensor with a Raspberry Pi Pico:

Step 1:

Connect the DHT22 sensor to the Raspberry Pi Pico. To do so, connect the **data** pin to the **GP10** GPIO, the **VCC** pin to 3.3V, and the **GND** pin to 0V (GND), as shown in the following figure:

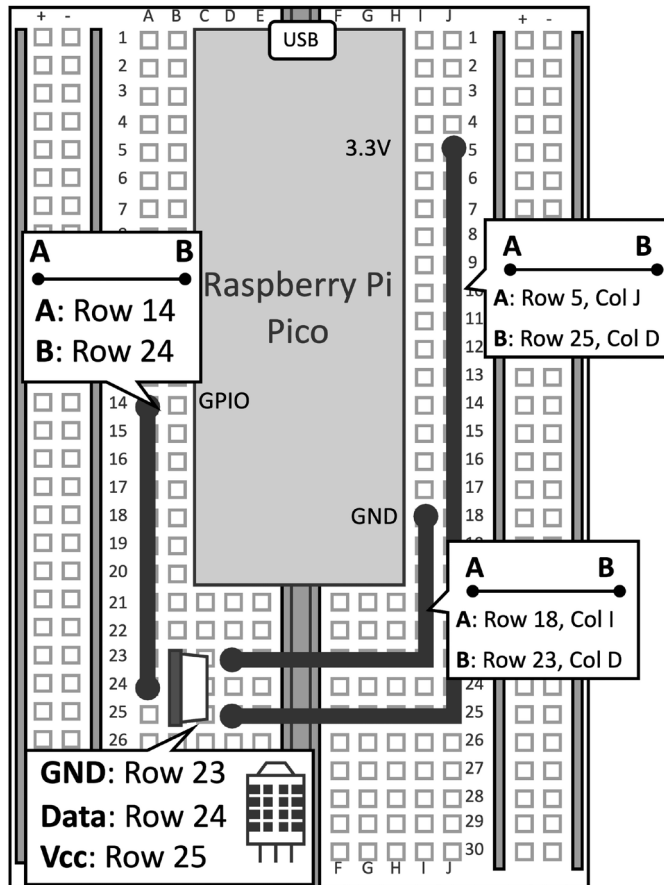


Figure 3.19: Connections between the Raspberry Pi Pico and the AM2302 sensor module

Step 2:

Download the 1.4.4 release of the Arduino **DHT sensor** library from <https://www.arduino.cc/reference/en/libraries/dht-sensor-library/>. If you are using a Mac, the library will be automatically extracted after being downloaded, so you will need to re-compress it.

Then, import the ZIP file in the Arduino IDE by clicking on the **Libraries** tab on the left pane and then **Import**, as shown in the following screenshot:

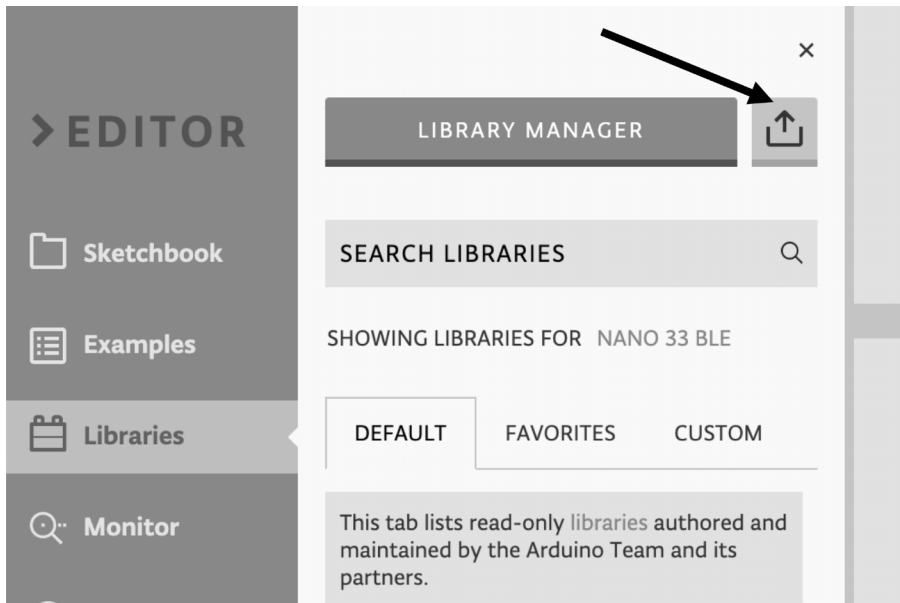


Figure 3.20: Import the DHT sensor library in the Arduino IDE

A pop-up window will inform you that the library has been successfully imported.

If you are working in the local Arduino IDE, an additional library is required to be installed. This library is **Adafruit Unified Sensor**. To install this library in the Arduino IDE, navigate to the Arduino **LIBRARY MANAGER**, as shown in *Figure 3.16*.

In **LIBRARY MANAGER**, find the Adafruit Unified Sensor library in the search box:

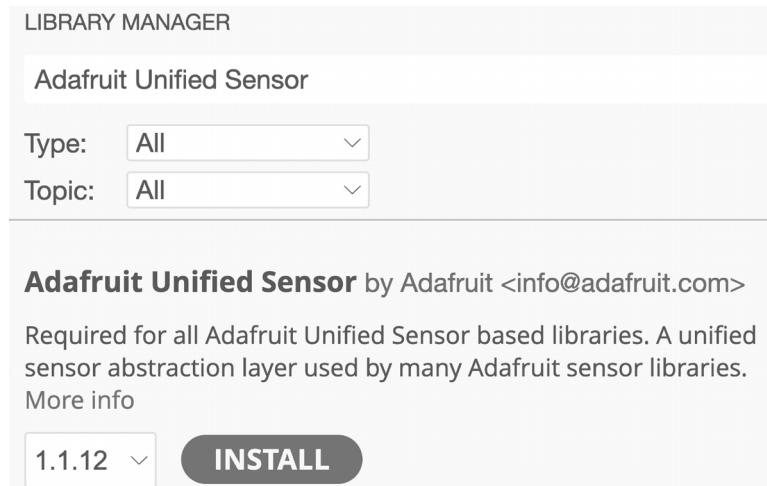


Figure 3.21: The Adafruit Unified Sensor library

Then, install the 1.1.12 Adafruit Unified Sensor library release.

Step 3:

In the sketch, include the C header file to use the DHT temperature and humidity sensor:

```
#include <DHT.h>
```

Step 4:

Define the global DHT object to interface with the DHT22 sensor:

```
const int gpio_pin_dht_pin = 10;  
DHT dht(gpio_pin_dht_pin, DHT22);  
  
#define SENSOR dht
```


As the DHT object is named `dht`, the `SENSOR` macro – which refers to the `<sensor>` placeholder – is set to `dht`.

The DHT object is initialized with the pin number used by the DHT22 data terminal (10) and the type of DHT sensor (DHT22).

Step 5:

In the `setup()` function, initialize the serial peripheral with a baud rate of 9600 and the DHT22 sensor:

```
Serial.begin(9600);  
while(!Serial);  
SENSOR.begin();  
delay(2000);
```

Unlike the `begin()` function used for the temperature and humidity sensor on the Arduino Nano, the `begin()` function, in this case, does not return any value (void).



The DHT22 can only return new data after two seconds. For this reason, we use `delay(2000)` to wait for the peripheral to be ready.

Step 6:

In the `setup()` function, read the temperature and humidity values and transmit them over the serial connection:

```
Serial.print("Test Temperature = ");  
Serial.print(SENSOR.readTemperature(), 2);  
Serial.println(" °C");  
Serial.print("Test Humidity = ");  
Serial.print(SENSOR.readHumidity(), 2);  
Serial.println(" %");
```

Now, connect the Raspberry Pi Pico to your computer. Then, compile and upload the sketch on the microcontroller. After, open the serial terminal. The device should display the temperature and humidity values measured by the sensor.

There's more...with the SparkFun Artemis Nano!

In the recipe, we learned how to connect the DHT22 sensor to the Raspberry Pi Pico, enabling us to retrieve temperature and humidity measurements.

After gaining the principles for connecting this sensor to the microcontroller, you might consider replicating this recipe with the SparkFun Artemis Nano using pin number 8 for the data signal.

In contrast to the Raspberry Pi Pico, at the time of writing, developing this recipe on the SparkFun Artemis Nano requires an additional step. This step is needed to address the issue of disabling/enabling interrupts within the DHT sensor library, which, otherwise, hinders temperature and humidity readings from the sensor.

Therefore, the required steps to replicate this recipe on the SparkFun Artemis Nano are as follows:

Step 1:

Create a new Arduino project.

Step 2:

Install the 1.4.4 release of the **DHT sensor library** from the downloaded ZIP file.

Step 3:

Install the 1.1.12 release of the **Adafruit Unified Sensor** library using the Arduino Library manager.

Step 4:

Enter the directory containing the Arduino libraries installed in the Arduino IDE. On Linux, this folder is generally in `~/Arduino/libraries`. On other **operating systems (OSs)**, you can refer to the official Arduino documentation available at the following link: <https://support.arduino.cc/hc/en-us/articles/4415103213714-Find-sketches-libraries-board-cores-and-other-files-on-your-computer>.

Inside the folder, enter the DHT sensor library directory (`DHT_sensor_library/`), and open the `DHT.h` file with a text editor. Then, replace `noInterrupts()` with `__disable_irq()` and `interrupts()` with `__enable_irq()`. After, save and close the file.



The DHT sensor library with the fix for the SparkFun Artemis Nano has been uploaded on GitHub and is available at the following link: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/ArduinoLibs/Arduino_DHT_sensor_library-1.4.4_for_sparkfun_artemis_nano.zip

Step 5:

In the Arduino IDE, implement the sketch like we did for the Raspberry Pi Pico. Use pin number 8 for the data signal of the DHT22 sensor.

Step 6:

On the breadboard, connect the DHT22 sensor to the SparkFun Artemis Nano.

Step 7:

Connect the SparkFun Artemis Nano to your computer. Then, compile and upload the sketch on the microcontroller. After, open the serial terminal. The device should display the temperature and humidity values measured by the sensor.

The solution for this exercise is available in the Chapter03 folder on the GitHub repository.

This recipe, as well as the one before it, enabled our microcontrollers to measure the temperature and humidity. However, the raw data acquired with the sensor are not the actual inputs for the model. In fact, the model's input consists of the temperature and humidity recorded over the last three hours, scaled with the Z-score function, and quantized to 8-bit.

Therefore, in the upcoming recipe, we will build an Arduino sketch to prepare the model's input from the acquired temperature and humidity readings.

Preparing the input features for the model inference

As we know, the model's input features are the scaled and quantized temperature and humidity of the last three hours.

In this recipe, we will see how to prepare this data on the microcontroller. In particular, this recipe will teach us how to acquire, scale, and quantize the sensor measurements and keep them in temporal order using a circular buffer.

Getting ready

Our application will acquire the temperature and humidity every hour to get the necessary input features for the model. However, *how can we keep the last three measurements in temporal order to feed the network the correct input?*

In this recipe, we will use a **circular buffer**, a fixed-sized data structure that implements a **First-In-First-Out (FIFO)** buffer.

This data structure is well suited to buffering data streams and can be implemented with an array and a pointer that indicates the location in memory where the element should be stored. The following diagram shows how a circular buffer with three elements works:

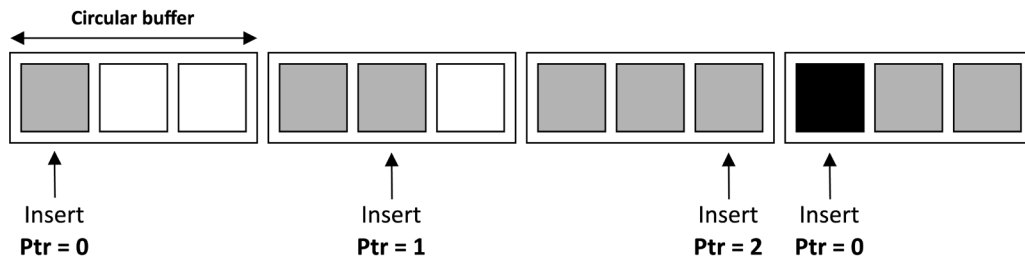


Figure 3.22: Circular buffer with three elements

As you can see from the preceding diagram, this data structure simulates a ring since the pointer (**Ptr**) is incremented after each data insertion and wraps around when it reaches the end.

How to do it...

The instructions in this section apply to the Arduino Nano and the Raspberry Pi Pico. Therefore, keep working on the sketch either created for Arduino Nano or Raspberry Pi Pico and follow the steps to prepare the input features:

Step 1:

Define two global `int8_t` arrays of size three and an integer variable to implement the circular buffer data structure:

```
constexpr int num_hours = 3;
int8_t t_vals [num_hours] = {0};
int8_t h_vals [num_hours] = {0};
int cur_idx = 0;
```

The `t_vals` and `h_vals` arrays will keep the scaled and quantized temperature and humidity measurements in temporal order. The `cur_idx` variable keeps track of the current index of the circular buffer.

Step 2:

Define the `scale` (`float`) and `zero_point` (`int32_t`) quantization parameters:

```
float tflu_i_scale = 1.0f;
int32_t tflu_i_zero_point = 0;
```

The following recipe will extract these quantization parameters from the TensorFlow Lite model.



Please note that `scale` (`tflu_i_scale`) is a floating-point number, while `zero_point` (`tflu_i_zero_point`) is a 32-bit integer. As we will see in the following recipe, these are the data types used by TensorFlow Lite for the quantization parameters.

Step 3:

Take the average of three temperature and humidity samples captured every three seconds in the `loop()` function:

```
constexpr int num_reads = 3;
void loop() {
    float t = 0.0f;
    float h = 0.0f;
    for(int i = 0; i < num_reads; ++i) {
        t += SENSOR.readTemperature();
        h += SENSOR.readHumidity();
        delay(3000);
    }
    t /= (float)num_reads;
    h /= (float)num_reads;
```

The preceding code captures `num_reads` temperature and humidity data samples for a more reliable measurement. After each reading, a delay of 3000 milliseconds (three seconds) is introduced to wait for the sensor to be ready for the next one.



Remember that the DHT22 sensor requires a minimum of two seconds to provide new data. Thus, the code waits three seconds before each reading to ensure the sensor is ready to provide accurate readings.

Step 4:

Scale the temperature and humidity data with the Z-score:

```
constexpr float t_mean = 2.08993f;
constexpr float h_mean = 87.22773f;
constexpr float t_std  = 6.82158f;
constexpr float h_std  = 14.21543f;
t = (t - t_mean) / t_std;
h = (h - h_mean) / h_std;
```

The Z-score function requires the mean and standard deviation of the input feature we calculated in this chapter's second recipe.

Step 5:

Quantize the input features:

```
t = (t / tflu_i_scale);
t += (float)tflu_i_zero_point;
h = (h / tflu_i_scale);
h += (float)tflu_i_zero_point;
```

The samples are quantized using the `tflu_i_scale` and `tflu_i_zero_point` input quantization parameters. Remember that the model's input uses the per-tensor quantization schema, so all input features must be quantized with the same scale and zero point.

Step 6:

Store the temperature and humidity sensor in the circular array:

```
t_vals[cur_idx] = t;
h_vals[cur_idx] = h;
```

```
cur_idx = (cur_idx + 1) % num_hours;

delay(2000);
```

The pointer of the circular buffer (`cur_idx`) is updated after each data insertion with the following formula:

$$idx_{new} = (idx_{old} + 1) \% array_{size}$$

In the preceding formula, $array_{size}$ is the size of the circular buffer, while idx_{old} and idx_{new} are the pointer's values before and after the data insertion.

At the end of the code, we have a delay of two seconds, but it should be one hour in the actual application. The pause of two seconds is used to avoid waiting too long in our experiments.

There's more...with the SparkFun Artemis Nano

In this recipe, we learned how to develop an Arduino sketch to acquire, scale, and quantize the sensor measurements and keep them in temporal order using a circular buffer.

If you have successfully acquired temperature and humidity readings with the SparkFun Artemis Nano, replicating this recipe on this platform will be effortless, as the existing sketch does not require any changes.

The solution for this exercise is available in the `Chapter03` folder on the GitHub repository.

Now that we know how to prepare the model's input, deploying the TensorFlow Lite model on the microcontrollers is the only remaining step.

In the upcoming final recipe, we will show for the first time how to use TensorFlow Lite for Microcontrollers to run the model inference on the Arduino Nano and Raspberry Pi Pico.

On-device inference with TensorFlow Lite for Microcontrollers

Here we are, ready to dive into our first ML application on microcontrollers.

This recipe will guide us through deploying the trained model using **TensorFlow Lite for Microcontrollers** (`tflite-micro`) on the Arduino Nano and Raspberry Pi Pico.

Getting ready

Tflite-micro is a component of TensorFlow Lite designed explicitly by Google and the open-source community to run ML models on microcontrollers and other devices with only a few kilobytes of memory.

Theoretically, nothing prevents you from using tflite-micro to run ML models on your laptop. However, it may not perform well since tflite-micro is optimized for low-resource devices such as microcontrollers.

Running a model with TensorFlow Lite or tflite-micro typically consists of the following:

1. **Loading the model:** We load the weights and network architecture stored in the TensorFlow Lite model.
2. **Preparing the input data:** We convert the input data acquired from the sensor to the expected format required by the model.
3. **Running the model:** We run the model using the **TensorFlow Lite Interpreter**.

Tflite-micro integrates software libraries to get the best performance and memory usage on various microcontrollers.

For example, tflite-micro integrates **CMSIS NN** (<https://www.keil.com/pack/doc/CMSIS/NN/html/index.html>), a free, open-source software library developed by Arm that provides optimized **deep neural network (DNN)** operators for Arm Cortex-M processors. These optimizations are relevant to critical DNN primitives such as convolution, depthwise convolution, and the fully connected layer and are compatible with the Arm processors in Arduino Nano, Raspberry Pi Pico, and SparkFun Artemis Nano.

How to do it...

The instructions presented in this section apply to both the Arduino Nano and the Raspberry Pi Pico. Therefore, keep working on the sketch either created for the Arduino Nano or Raspberry Pi Pico and follow the steps to use tflite-micro to run the snow forecast model on these boards:

Step 1:

Download the Arduino TensorFlow Lite library from the TinyML-Cookbook_2E GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/ArduinoLibs/Arduino_TensorFlowLite.zip

After downloading the ZIP file, import it into the Arduino IDE by clicking on the **Libraries** tab on the left pane and **Import**.

Step 2:

Import the `model.h` file into the Arduino project. (Note that we originally generated and downloaded this file from Google Colab, in the *Quantizing the model with the TensorFlow Lite converter* recipe.) As shown in the following screenshot, click on the tab button with the upside-down triangle and click on **Import File into Sketch**:

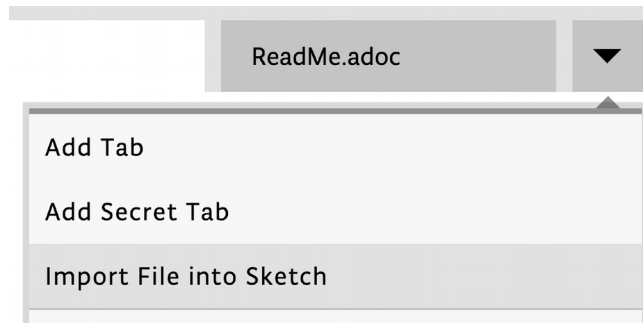


Figure 3.23: Importing the `model.h` file into the Arduino project

Once the file has been imported, include the `model.h` file in the Arduino sketch:

```
#include "model.h"
```

Step 3:

Include the necessary header files for `tf-lite-micro`:

```
#include <TensorFlowLite.h>
#include <tensorflow/lite/micro/all_ops_resolver.h>
#include <tensorflow/lite/micro/micro_interpreter.h>
#include <tensorflow/lite/micro/micro_log.h>
#include <tensorflow/lite/micro/system_setup.h>
#include <tensorflow/lite/schema/schema_generated.h>
```

The header files reported in the preceding code snippet should be included in all applications based on `tf-lite-micro`. The primary header files from the list include the following:

- `all_ops_resolver.h`: To load all the DNN operators required for running the ML model
- `micro_interpreter.h`: To load and execute the ML model

- `schema_generated.h`: For the schema of the TensorFlow Lite **FlatBuffer** format



For more information about the header files, we recommend reading the *Get started with microcontrollers* guide, available at the following link: https://www.tensorflow.org/lite/microcontrollers/get_started_low_level

Step 4:

Declare global variables for `tf-lite-micro` that are necessary for loading/running the model and pointing to the input and output tensor of the model:

```
const tf::Model* tflu_model          = nullptr;
tf::MicroInterpreter* tflu_interpreter = nullptr;
TfLiteTensor* tflu_i_tensor          = nullptr;
TfLiteTensor* tflu_o_tensor          = nullptr;
```

The global variables declared in this step are as follows:

- `tflu_model`: This is the pointer to the TensorFlow Lite model to be loaded.
- `tflu_interpreter`: This is the pointer to the TensorFlow Lite Interpreter, which will run the model.
- `tflu_i_tensor`: This is the pointer to the input tensor of the model.
- `tflu_o_tensor`: This is the pointer to the output tensor of the model, where the inference results will be stored.

Step 5:

Declare global variables for the quantization parameters of the output tensor:

```
float tflu_o_scale = 1.0f;
int32_t tflu_o_zero_point = 0;
```

Step 6:

Declare a global buffer (`tensor_arena`) to store the tensors used during the model execution:

```
constexpr int t_sz = 4096;
uint8_t tensor_arena[t_sz] __attribute__((aligned(16)));
```

The user must allocate all the tensors the model consumes, such as the input, output, and intermediate ones. The reason is that the TensorFlow Lite Interpreter does not allocate any memory at runtime. As we can guess, the minimum tensor arena's size can only be determined by knowing the tensor shapes required by the model and the dependencies in the graph. In this recipe, the tensor arena is allocated with 4,096 `int8_t` elements, which is more than enough for our needs. In future chapters of this book, we will demonstrate how to estimate the tensor arena to minimize memory usage.



The `aligned(16)` attribute ensures that the memory is aligned on a 16-byte boundary, which the Arm Cortex-M CPUs require for efficient memory access.

Step 7:

Declare a global `tflite::AllOpsResolver` object to register all the DNN operations supported by `tflite-micro`:

```
tflite::AllOpsResolver tflu_ops_resolver;
```

The `tflite-micro` Interpreter will use the preceding object to find the function pointers for each DNN operator.

Step 8:

In the `setup()` function and after the initialization of the sensor, load the TensorFlow Lite model stored in the `model.h` header file:

```
tflu_model = tflite::GetModel(snow_model_tflite);
```

Step 9:

In the `setup()` function, initialize the `tflite-micro` Interpreter:

```
static tflite::MicroInterpreter static_interpreter(
    tflu_model,
    tflu_ops_resolver,
    tensor_arena,
    t_sz);

tflu_interpreter = &static_interpreter;
```

TensorFlow Lite and tflite-micro run the model using the Interpreter, which needs to be initialized with the following arguments:

- `tflu_model`: The model to run
- `tflu_ops_resolver`: The list of DNN operations that can be used
- `tensor_arena`: The tensor arena
- `t_sz`: The tensor arena size

Step 10:

In the `setup()` function, allocate the memory required for the model and get the memory pointer of the input and output tensors:

```
tflu_interpreter->AllocateTensors();  
tflu_i_tensor = tflu_interpreter->input(0);  
tflu_o_tensor = tflu_interpreter->output(0);
```

Step 11:

In the `setup()` function, get the quantization parameters for the input and output tensors:

```
const auto* i_quant = reinterpret_  
cast<TfLiteAffineQuantization*>(tflu_i_tensor->quantization.params);  
const auto* o_quant = reinterpret_  
cast<TfLiteAffineQuantization*>(tflu_o_tensor->quantization.params);  
  
tflu_i_scale      = i_quant->scale->data[0];  
tflu_i_zero_point = i_quant->zero_point->data[0];  
tflu_o_scale      = o_quant->scale->data[0];  
tflu_o_zero_point = o_quant->zero_point->data[0];
```

The quantization parameters are returned in the `TfLiteAffineQuantization` object, containing two arrays for the `scale` and `zero_point` parameters. Since input and output tensors adopt a per-tensor quantization, each array stores a single value.

Step 12:

In the `loop()` function and just before the `delay(2000)`, initialize the input tensor with the quantized input features:

```
// Prepare input features
```

```

int32_t idx0 = cur_idx;
int32_t idx1 = (cur_idx - 1 + num_hours) % num_hours;
int32_t idx2 = (cur_idx - 2 + num_hours) % num_hours;
tflu_i_tensor->data.int8[0] = t_vals[idx2];
tflu_i_tensor->data.int8[1] = t_vals[idx1];
tflu_i_tensor->data.int8[2] = t_vals[idx0];
tflu_i_tensor->data.int8[3] = h_vals[idx2];
tflu_i_tensor->data.int8[4] = h_vals[idx1];
tflu_i_tensor->data.int8[5] = h_vals[idx0];

```

Since we need the last three samples, we use the following formula to read the correct elements from the circular buffer:

$$idx_p = (idx_0 - P + array_{size}) \% array_{size}$$

If we denote idx_0 as the pointer of the circular array at time $t = 0$, then idx_p represents the pointer within the circular buffer at time $t = -P$.

Step 13:

In the `loop()` function, run the model inference by calling the `Invoke()` method from the TensorFlow Lite Interpreter:

```

tflu_interpreter->Invoke();

```

Step 14:

In the `loop()` function, dequantize the output tensor and check whether it will snow:

```

float out_int8 = tflu_o_tensor->data.int8[0];
float out_f = (out_int8 - tflu_o_zero_point);
out_f *= tflu_o_scale;

if (out_f > 0.5) {
    Serial.println("Yes, it snows");
}
else {
    Serial.println("No, it does not snow");
}

```

The dequantization of the output value is done with the `tflu_o_scale` and `tflu_o_zero_point` quantization parameters retrieved in the `setup()` function.

Once we have the floating-point representation, the output is *No* when it is below 0.5, otherwise *Yes*.

Now, compile and upload the program on the microcontroller board. The Serial Monitor in the Arduino IDE will report *Yes, it snows* or *No, it does not snow*, depending on whether the snow is forecasted.



To check if the application can forecast snow, you can simply force the temperature to -10 and the humidity to 100. The model should return *Yes, it snows* in the serial monitor.

There's more...with the SparkFun Artemis Nano!

In this recipe, we learned how to use `tf-lite-micro` to run the model inference on the Arduino Nano and Raspberry Pi Pico.

The Arduino sketch developed in this recipe works with any Arduino-compatible platform. What about replicating this recipe with the SparkFun Artemis Nano? The only prerequisite for accomplishing this task is the installation of the Arduino TensorFlow Lite library in your local Arduino IDE, which is available at the following link: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/ArduinoLibs/Arduino_TensorFlowLite.zip.

The solution for this exercise is available in the `Chapter03` folder in the GitHub repository.

Summary

The recipes presented in this chapter demonstrated how to deploy a model trained with TensorFlow using `tf-lite-micro` on Arduino-compatible platforms, such as the Arduino Nano and Raspberry Pi Pico.

Initially, we learned how to build a dataset to forecast snow using the temperature and humidity over the last three hours. In this part, we focused on the importance of feature scaling and proposed the Z-score function to bring the input features to a similar numerical range.

Afterward, we delved into the model training phase. Here, we trained the network with TensorFlow and learned how to make the model compact with TensorFlow Lite using 8-bit quantization.

Then, we discovered how to acquire temperature and humidity measurements with the Arduino Nano and Raspberry Pi Pico. In particular, we learned how to use the built-in sensor on the Arduino Nano and connect the DHT22 sensor to the Raspberry Pi Pico.

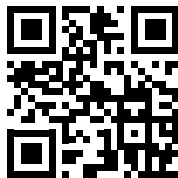
Finally, we deployed the trained model on the microcontrollers with the help of `tf.lite-micro`.

In this chapter, we implemented our first end-to-end tinyML project using TensorFlow. With the next project, we will continue our learning journey by building a keyword-spotting application on the Arduino Nano with the help of another ML framework: Edge Impulse.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



4

Using Edge Impulse and the Arduino Nano to Control LEDs with Voice Commands

Have you ever wondered how smart assistants can enable a hands-free experience with your devices? The answer lies in **keyword spotting (KWS)** technology, which recognizes the popular wake word phrases of *OK Google*, *Alexa*, *Hey Siri*, or *Cortana*. By identifying these phrases, the smart assistant wakes up and listens to your commands. Since KWS uses real-time speech recognition models, it must be on-device, always on, and running on a low-power system to be effective.

In this chapter, we will demonstrate the usage of KWS through **Edge Impulse** by building an application to voice control a **light-emitting diode (LED)** that emits a colored light (red, green, or blue) a certain amount of times (one, two, or three blinks) on the Arduino Nano.

This chapter will begin with dataset preparation, showing you how to acquire audio data with a mobile phone and the built-in microphone on the Arduino Nano. Then, we will design a model for speech recognition based on the popular **Mel-filterbank energy (MFE)** features. In these recipes, we will show you how to extract these features from audio samples, train a **machine learning (ML)** model, and fine-tune model's performance with the Edge Impulse **EON Tuner**. At the end of the chapter, we will concentrate on deploying the KWS application on the Arduino Nano.

This chapter is intended to show how to develop an **end-to-end (E2E)** KWS application with Edge Impulse and get familiar with audio data acquisition.

In this chapter, we're going to implement the following recipes:

- Acquiring audio data with a smartphone
- Acquiring audio data with the Arduino Nano
- Extracting MFE features from audio samples
- Designing and training a **convolutional neural network (CNN)**
- Tuning model performance with the EON Tuner
- Live classifications with a smartphone
- Keyword spotting on the Arduino Nano

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- An Arduino Nano 33 BLE Sense
- A smartphone (an Android phone or Apple iPhone)
- A micro-USB data cable
- A laptop/PC with either Linux, macOS, or Windows
- A Google Drive account
- An Edge Impulse account



The source code and additional material are available in the Chapter04 folder on the GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Chapter04.

Acquiring audio data with a smartphone

As with all ML problems, data acquisition is the first step to take, and Edge Impulse offers several ways to do this directly from the web browser.

In this first recipe, we will learn how to acquire audio samples using a mobile phone.

Getting ready

Edge Impulse offers a straightforward and efficient method for data acquisition using smartphones through internet connectivity. This approach is so simple and intuitive that even people with no technical background will find it easy to use.

The only factor to consider before preparing the dataset is related to the number of audio recordings to take for training the model, outlined in the upcoming subsection.

Collecting audio samples for KWS

The number of samples depends entirely on the nature of the problem—therefore, no one approach fits all. In our scenario, 25 recordings for each class, each corresponding to the utterance to recognize (*red*, *green*, *blue*, *one*, *two*, and *three*), could be sufficient to get a basic model. However, 100 or more is generally recommended to get better results. We want to give you complete freedom in this choice. However, remember to get an equal number of samples for each class to obtain a balanced dataset.

Whichever dataset size you choose, try including different variations in the instances of speech, such as accents, inflations, pitch, pronunciations, and tone. These variations will make the model capable of identifying words from different speakers. Typically, recording audio from persons of different ages and genders should cover all these cases.

Although the primary goal is to recognize six words (*red*, *green*, *blue*, *one*, *two*, and *three*), it is crucial to account for situations when none of the utterances of interest are present within the speech. Therefore, the ML model should include an additional class, called *unknown*, to encompass such cases.

How to do it...

Open the web browser and go to Edge Impulse (<https://studio.edgeimpulse.com/>). Create a new project called **voice_controlling_led** and follow the next steps to acquire audio samples with the microphone of your mobile phone:

Step 1:

In the Edge Impulse **Dashboard** section, click on the **Collect new data** option:

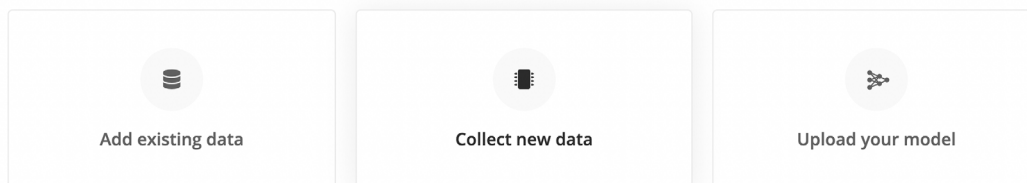


Figure 4.1: Click on the Collect new data button to start acquiring data

Edge Impulse will open a pop-up window asking how you want to collect the data. Select the **Scan QR code to connect to your phone** option to start recording audio clips with the microphone of your phone:

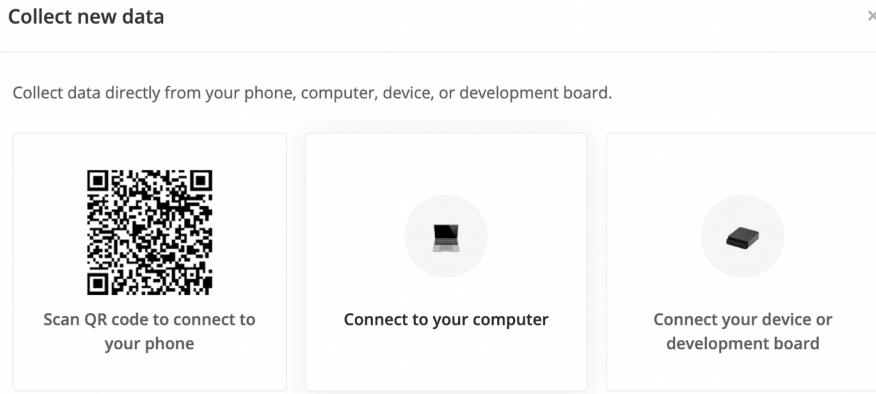


Figure 4.2: Click on the Scan QR code to connect... button to start acquiring audio samples from your computer

Scan the **quick response (QR)** code with your smartphone to pair the device with Edge Impulse. A pop-up window on your phone will confirm that the device is connected, as shown in the following screenshot:

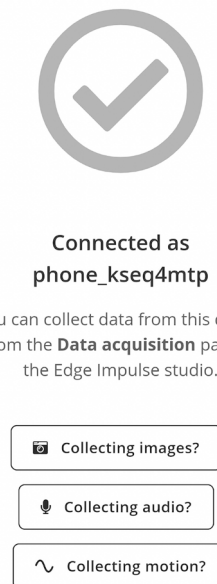


Figure 4.3: Edge Impulse message confirmation on your phone

On your mobile phone, click on **Collecting audio?** and give permission to use the microphone.



Since having a laptop and smartphone on the same network is not required, we could acquire audio samples from anywhere. As you might guess, this approach is well suited to recording sounds from different environments since it only requires a phone with internet connectivity.

Step 2:

Record 25 or more utterances for each class (*red, green, blue, one, two, and three*). Before clicking on the **Start recording** button, set **Category** to **Training** and enter one of the following labels in the **Label** field, depending on the spoken word:

Class	Red	Green	Blue	One	Two	Three
Label	00_red	01_green	02_blue	03_one	04_two	05_three

Figure 4.4: Labels to assign to each class

Our proposed label names (*00_red, 01_green, 02_blue, 03_one, 04_two, and 05_three*) are helpful for quickly identifying whether the class corresponds to a color or a number. In fact, as we will see later when training the model with Edge Impulse, *the label encoding assigns integer values to each category based on alphabetical ordering*. Therefore, by adopting these specific label names, we can make it easier to interpret the output class as a color (a label index lower than 3) or number (a label index greater than or equal to 3).

To reduce the number of files uploaded to Edge Impulse, we recommend recording multiple repetitions of the same utterance in a single recording. For instance, you could record a 20-second audio clip where you repeat the same word 10 times, separated by 1-second pauses.

The recordings will be accessible in the **Data Acquisition** section of Edge Impulse. By selecting the audio file, you can view the corresponding audio waveform:

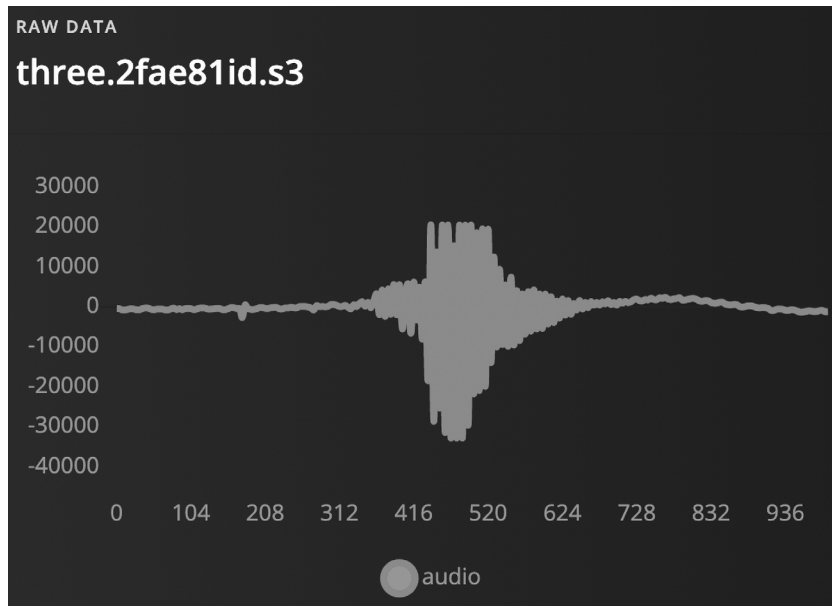


Figure 4.5: Example of an audio waveform

The **raw audio waveform** is the signal recorded by the microphone and graphically describes the sound-pressure variation over time. The vertical axis reports the signal's amplitude, while the horizontal axis reports the time. The higher waveform amplitude implies louder audio as perceived by the human ear.

Step 3:

Split the recordings containing repetitions of the utterance in individual samples by clicking the **:** button near the filename. Then, click on the **Split sample** option, as shown in the following screenshot:

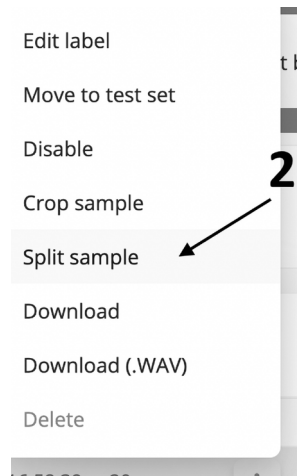


Figure 4.6: The Split sample option

Edge Impulse will automatically detect spoken words, as you can observe from the following screenshot:

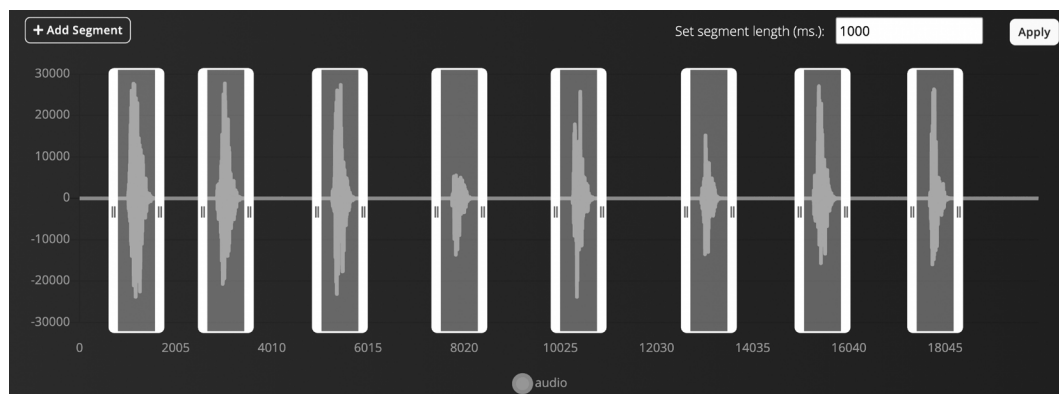


Figure 4.7: Audio waveform with the distinct samples found by Edge Impulse

In the new window, set the segment length to 1000 milliseconds (ms) (1 s), and ensure all the samples are centered within the cutting window. Then, click on the **Split** button to get the individual samples.

Step 4:

Download the keyword dataset from Edge Impulse (<https://cdn.edgeimpulse.com/datasets/keywords2.zip>) and unzip the file.

After unzipping the file, import 25 or more random samples from the **unknown/** folder into the Edge Impulse project.



Ensure you have the same number of samples for each class. Therefore, if you have recorded 25 audio clips for each output category, you should ensure that you import an equal number of 25 audio clips for the unknown one.

To do so, enter the **Data acquisition** section and click on the **Upload data** button, located in the **Dataset** area:

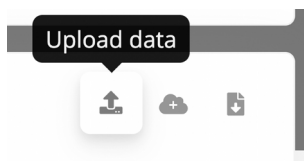


Figure 4.8: Button to upload existing training data

On the **Upload data** page, do the following:

- Select the audio files corresponding to the unknown class
- Set **Upload into category** option to **Training**
- Write **unknown** in the **Enter label** field

Click on the **Begin upload** button to import the files into the dataset.

The dataset should now have an equal number of samples for each class.

There's more

In this recipe, we learned how to use Edge Impulse to record audio with the smartphone to create a KWS dataset.

Having a diverse dataset is crucial to train a robust and accurate model. In the context of speech recognition, a dataset should include audio recordings from a wide range of speakers with different accents, ages, genders, and speech styles, as well as background noise.

A diverse dataset is essential to prevent the model from developing biases toward particular individuals and enhance its effectiveness across various users. Therefore, we recommend you ask your friends or family members to help you record additional audio samples to diversify the dataset as much as possible.



Speakers do not need to be in the same room. In fact, by using the Edge Impulse QR code, people can record audio clips anywhere in the world.

If you add new samples to the dataset, be sure to expand the unknown class to maintain a balanced dataset.

Our dataset gradually takes shape but requires additional samples before it becomes suitable for training.

In the upcoming recipe, we will augment the dataset with audio recordings obtained from the microphone of the Arduino Nano.

Acquiring audio data with the Arduino Nano

Building the dataset with recordings obtained with the mobile phone's microphone is undoubtedly good enough for many applications. However, to prevent any potential loss in accuracy during the model's deployment, we should also include audio clips recorded with the microphone used by the end application in the dataset.

Therefore, this recipe will show you how to record audio samples with the built-in microphone on the Arduino Nano through Edge Impulse.

Getting ready

In the previous recipe, you will have noticed that in the options reported in the **Collect data** section, the mobile phone was not the only one, as shown in the following screenshot:

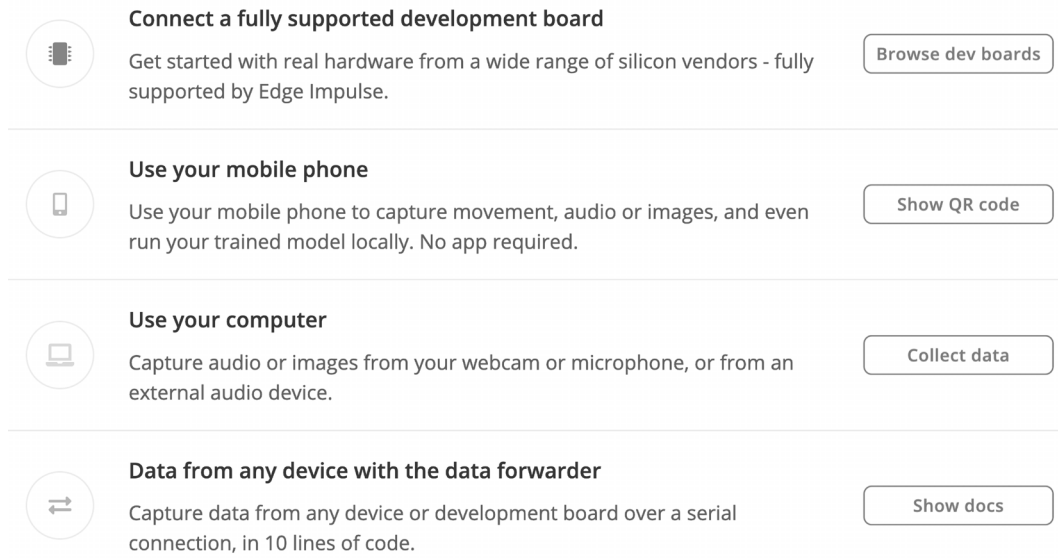


Figure 4.9: Some of the options available to collect data in Edge Impulse

For example, as reported in the previous screenshot, you can collect data using your computer's microphone or a fully supported microcontroller board.

If you click on the **Connect Fully supported development board** option, you can find the supported platforms and the instructions to acquire data using the sensors connected to the microcontroller with Edge Impulse:

Officially supported MCU targets

- Alif Ensemble E7
- Arduino Nano 33 BLE Sense
- Arduino Nicla Sense ME
- Arduino Nicla Vision
- Arduino Nicla Voice
- Arduino Portenta H7 + Vision Shield

Figure 4.10: Some of the microcontroller boards supported in Edge Impulse

Edge Impulse supports various platforms, including the Arduino Nano 33 BLE Sense board. The following section will briefly explain how Edge Impulse can collect data from these platforms.

Collecting data using a fully supported platform in Edge Impulse

Edge Impulse uses primarily two software programs to collect data from the microcontroller – one running on the microcontroller (**firmware**) and one on your computer (**daemon**):

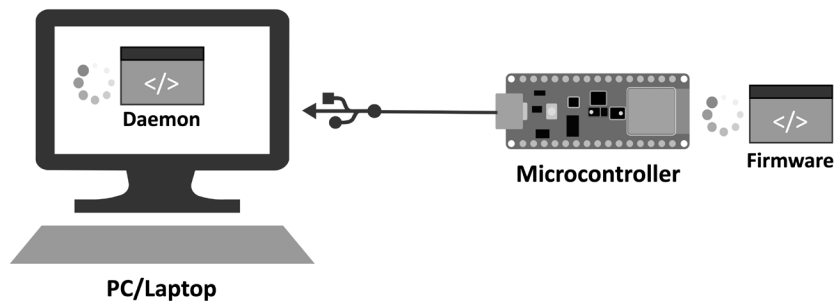


Figure 4.11: Firmware runs on the microcontroller while the daemon runs on your computer

Edge Impulse provides these two programs for the following purposes:

- The **firmware** gathers data from the sensor, such as the microphone, and transmits it over USB
- The **daemon** retrieves data transmitted by the microcontroller, creates the file, and uploads it to the Edge Impulse

With these two software programs, you have everything you need to collect data directly from your Arduino Nano.

How to do it...

In Edge Impulse, open the **Collect data** section, click on the **Connect a fully supported development board** option, and then click on the **Arduino Nano 33 BLE Sense** board (<https://docs.edgeimpulse.com/docs/arduino-nano-33-ble-sense>). Now follow the steps to upload the firmware on the microcontroller and install the daemon on your computer to record audio clips with the Arduino Nano:

Step 1:

Install the software dependencies reported in the Edge Impulse guide. At the time of writing, the dependencies are:

- **Edge Impulse CLI:** <https://docs.edgeimpulse.com/docs/edge-impulse-cli/cli-installation>
- **Arduino CLI:** <https://arduino.github.io/arduino-cli>

We recommend you check the dependencies reported in the <https://docs.edgeimpulse.com/docs/arduino-nano-33-ble-sense> guide for proper configuration.

Step 2:

Open the terminal and install the Arduino cores for Arduino Nano by running the following command:

```
$ arduino-cli core install arduino:mbed_nano
```

The cores contain the necessary libraries and software packages required to use the board in the Arduino IDE and enable the uploading of programs to the microcontroller.

Step 3:

Connect the Arduino Nano board to your computer through the micro-USB data cable and press the RESET button on the platform twice quickly:

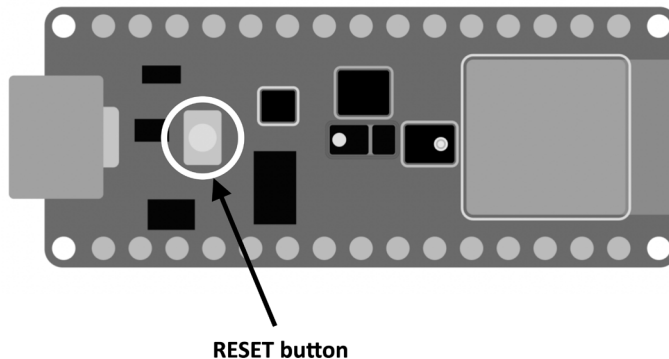


Figure 4.12: RESET button on Arduino Nano 33 BLE Sense board

The built-in LED should start pulsating to communicate that the device is in bootloader mode.

Step 4:

Download the Edge Impulse firmware to upload on the Arduino Nano. Then, unzip the file.



At the time of writing, you can download the firmware from the following link: <https://cdn.edgeimpulse.com/firmware/arduino-nano-33-ble-sense.zip>.

In the unzipped folder, execute the *flash script* to upload the firmware on the Arduino Nano. You should use the script accordingly with your **operating system (OS)**—for example, `flash_linux.sh` for Linux.

Once the firmware has been uploaded on the Arduino Nano, you can press the **RESET** button to launch the program.

Step 5:

Open the terminal and run `edge-impulse-daemon` using the following command:

```
$ edge-impulse-daemon
```

In the terminal, the wizard will ask you to authenticate on Edge Impulse, select the project you are working on, and give a name to your device. For example, you could call it **personal**.

The Arduino Nano should now be paired with Edge Impulse. To check whether the Arduino Nano is paired with Edge Impulse, open Edge Impulse and click on **Devices** from the left-hand side panel. In the **Devices** section, you should have something like what is reported in *Figure 4.13*:

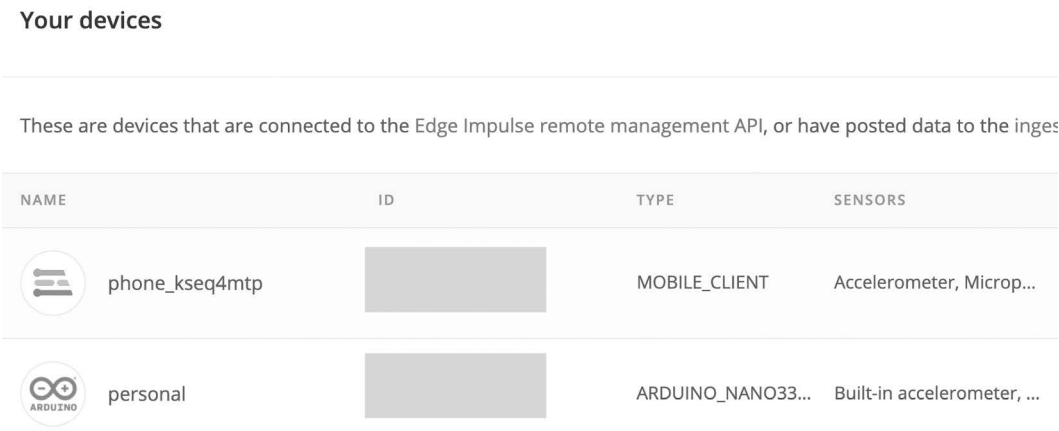
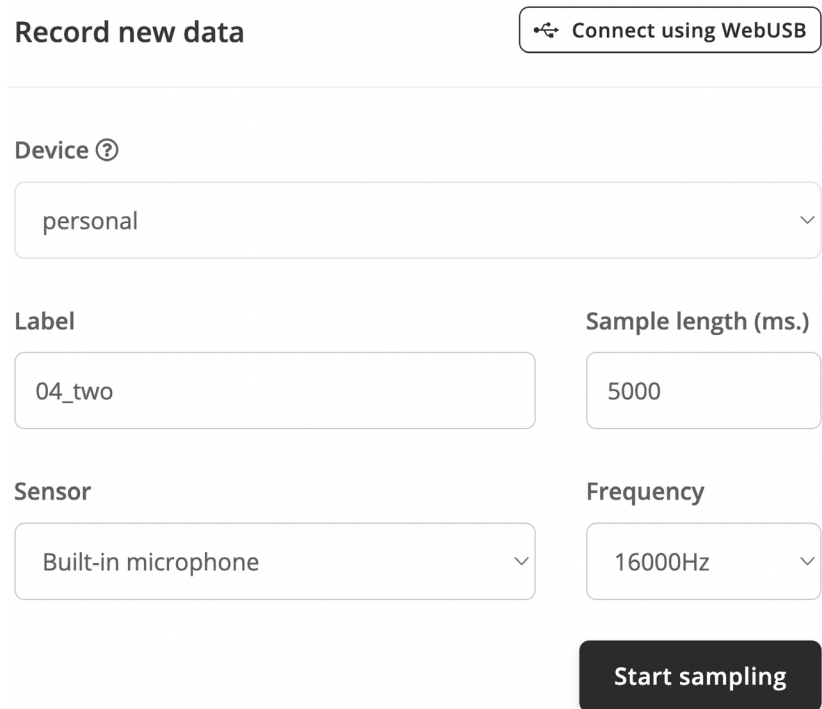


Figure 4.13: List of paired devices with Edge Impulse

As you can see from *Figure 4.13*, the Arduino Nano (**personal**) is listed in the **Your devices** section.

Step 6:

In Edge Impulse, enter the **Data acquisition** section. In the **Record new data** area, you should see your Arduino Nano device:



Record new data Connect using WebUSB

Device ?

personal

Label

04_two

Sample length (ms.)

5000

Sensor

Built-in microphone

Frequency

16000Hz

Start sampling

Figure 4.14: The Arduino Nano device (personal) should be listed in the Record new data area

Ensure your Arduino Nano is selected in the **Device** drop-down list. Then, set the **Sample length (ms.)** value to **5000** and **Frequency** to **16000Hz**.

You may now begin recording audio clips using the microcontroller by clicking the **Start sampling** button. As previously mentioned, when using the mobile phone microphone, record at least 25 audio samples for each class and extend the unknown class accordingly.

Step 7:

Split the samples between training and test datasets by clicking on the **Perform train/test split** button in the **Danger zone** area of **Dashboard**:

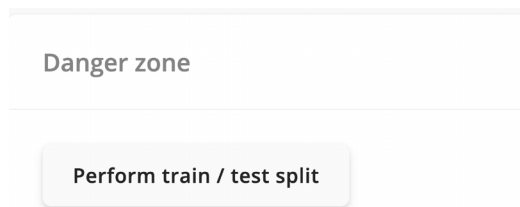


Figure 4.15: The button to perform the train and test split

Edge Impulse will ask you twice if you are sure about this action because the data shuffling is irreversible.

You should now have 70% of the samples assigned to the training/validation set and 30% to the test one.

There's more

In this recipe, we learned how to use Edge Impulse to record audio with the Arduino Nano to augment the dataset for KWS.

All the acquired samples are available in the **Data acquisition** section of Edge Impulse. In this section, you can add more training data but also export the dataset to `.json` or `.wav` audio files. By exporting the dataset, you can bring your samples to another Edge Impulse project or another ML development environment.

The training dataset is now ready, and we can proceed with the design of the ML model.

In the upcoming recipe, we will start this discussion by introducing the feature extraction algorithm used in this project to recognize spoken words with ML: the **Mel filterbank energy (MFE)** algorithm.

Extracting MFE features from audio samples

Edge Impulse relies on the **impulse** to craft all data processing tasks, including feature extraction and model inference. In this tutorial, we will see how to create an impulse to extract MFE features from our audio samples.

Getting ready

In Edge Impulse, an impulse is responsible for data processing and consists of the following two sequential computational blocks:

- **Processing block:** This is the preliminary step in any ML application, and it aims to prepare the data for the ML algorithm.
- **Learning block:** This is the block that implements the ML solution, which aims to learn patterns from the data provided by the processing block.

The processing block determines the ML effectiveness since the raw input data is often unsuitable for feeding the model directly. For example, the input signal could be noisy or have irrelevant and redundant information for training the model, just to name a couple of scenarios.

Therefore, Edge Impulse offers several pre-built processing functions, including the possibility of having custom ones.

In our case, we will use the MFE feature extraction processing block, and the following subsections will help us learn more about this.

Analyzing audio in the frequency domain

In contrast to vision applications where **convolutional neural networks (CNNs)** can make feature extraction part of the learning process, typical speech recognition models do not perform well with raw audio data. Therefore, feature extraction is required and needs to be part of the processing block.

We know from physics that *sound is the vibration of air molecules and propagates as a wave*. For example, if we played a pure single **tone**, the microphone would record a sine signal:

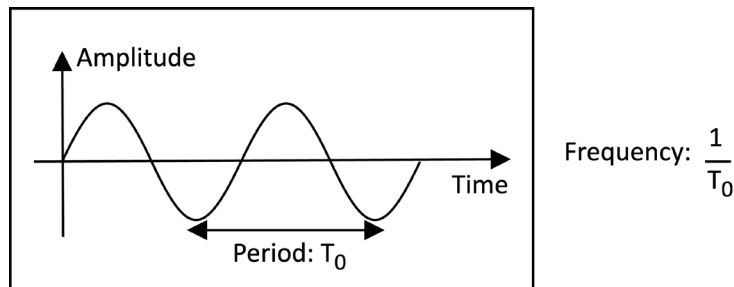


Figure 4.16: A pure single tone corresponds to a sine wave

Although the sounds in nature are far from pure, *every sound can be expressed as the sum of sine waves at different frequencies and amplitudes.*



We usually use the term **components** to refer to sine waves that make up the signal.

As sine waves are characterized by their frequency (**hertz**, or **Hz**) and peak amplitude, we commonly plot the components on a graph with the frequency on the x-axis and amplitude on the y-axis:

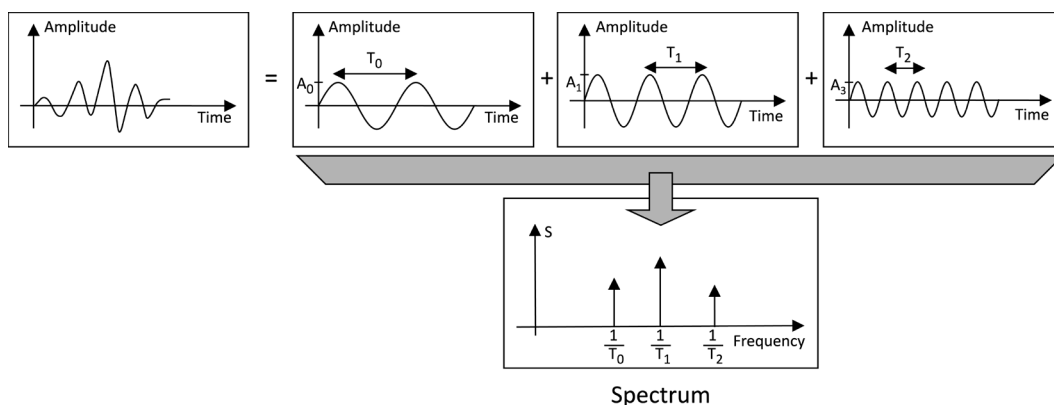


Figure 4.17: Representation of a signal in the frequency domain

The graph reporting the frequency and amplitude of each sine wave is called a **spectrum** and gives us a snapshot of the components at a specific time.



The **Discrete Fourier Transform (DFT)**, or **Fast Fourier Transform (FFT)**, is the required mathematical tool to decompose the digital audio waveform into all its constituent components.

Now that we are familiar with the frequency representation of an audio signal, let's see what we can generate as an input feature for a CNN.

Extracting the Mel-spectrogram

The spectrum obtained with the FFT does not convey any temporal information as it refers to a particular moment in time of the signal. Since temporal information is necessary to understand the evolution of the signal over time, it is essential to consider the signal's spectral content across multiple time intervals.

This spectrum representation at different time steps is called **FFT-spectrogram** and is obtained by dividing the audio signal into smaller frames and performing the FFT on each one.



The frame size is typically called the **frame length**, while the temporal distance between two frames is known as the **frame step**.

The FFTs are then stacked together to form an image, like the one shown in *Figure 4.18*:

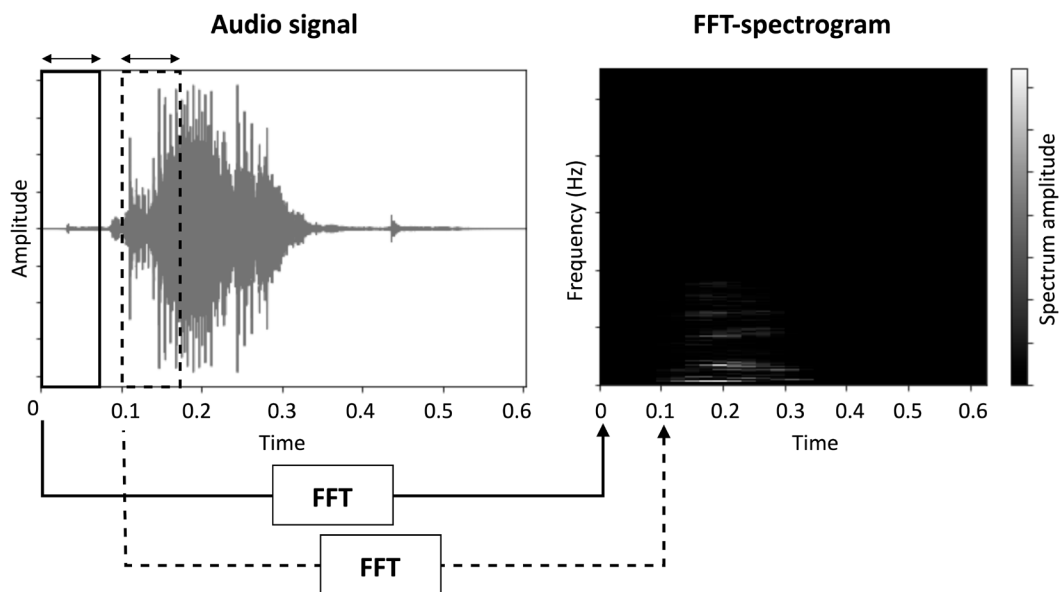


Figure 4.18: Audio signal and FFT-spectrogram of the word red

In the spectrogram, each vertical slice represents the resulting FFT – in particular:

- The width reports the time
- The height reports the frequency
- The color reports the components' amplitude, so a brighter color implies a higher amplitude

However, training an ML model for speech recognition using the FFT-spectrogram could be challenging because the relevant features are not emphasized. This aspect can be visually observed from the preceding screenshot, as the spectrogram is a flat image due to there being the same pixel intensity in almost all regions.

As a result, the spectrogram is adjusted to make the relevant features more visible. These adjustments are applied considering that *humans perceive frequencies and loudness on a logarithmic scale rather than linearly*.

The first transformation is applied to the frequencies, which are *converted from Hz to Mel using a set of overlapped triangular filters (filterbank)*. The Mel scale remaps the frequencies to make them distinguishable and perceived equidistantly. For example, if we played pure tones from 100 Hz to 200 Hz with a 1 Hz step, we could distinctly perceive all 100 frequencies. However, if we conducted the same experiment at higher frequencies (for example, between 7500 Hz and 7600 Hz), we could barely hear all tones. Therefore, since not all frequencies are equally relevant for our ears, we can use fewer components to represent the signal.



Mel components are calculated using triangular filters overlapped in the frequency domain. The number of triangular filters equals the number of Mel frequencies to extract.

Alongside frequency scaling, amplitude scaling is also required as the human brain does not perceive amplitude linearly but logarithmically. Therefore, the *amplitudes are scaled logarithmically* to make them visible in the spectrogram.

The spectrogram obtained by applying the preceding transformations is called a **Mel-spectrogram** or MFE. The MFE of the red word using 40 Mel frequencies is reported in Figure 4.19:

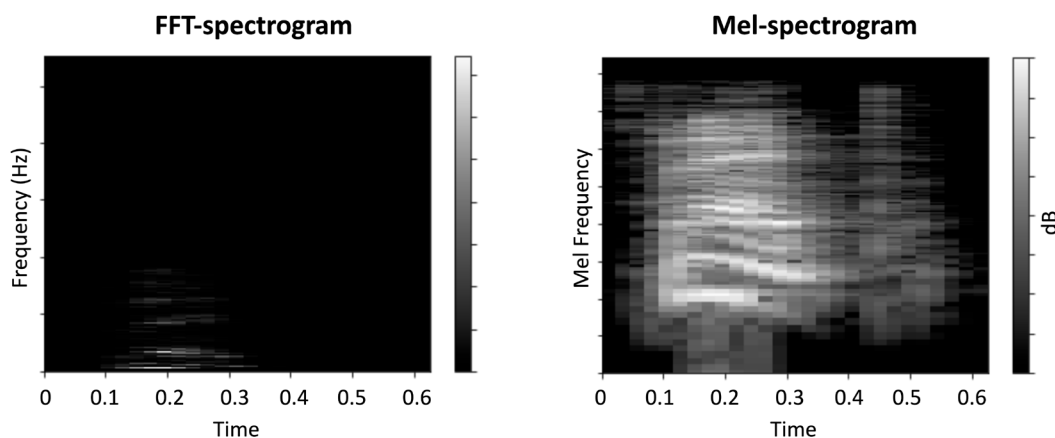


Figure 4.19: FFT-spectrogram and Mel-spectrogram of the word red

As you can see from Figure 4.19, the transformed image presents a higher contrast than the FFT-spectrogram.

How to do it...

We start designing our first impulse by clicking on the **Create impulse** option from the left-hand side menu, as shown in the following screenshot:

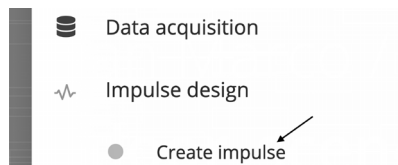


Figure 4.20: Create impulse option

In the **Create impulse** section, ensure the time-series data has the **Window size** field set to 1000 ms and the **Window increase** field to 500 ms.

These two parameters determine the length of the input sample and the temporal distance between two consecutive ML inferences, as shown in the following illustration:

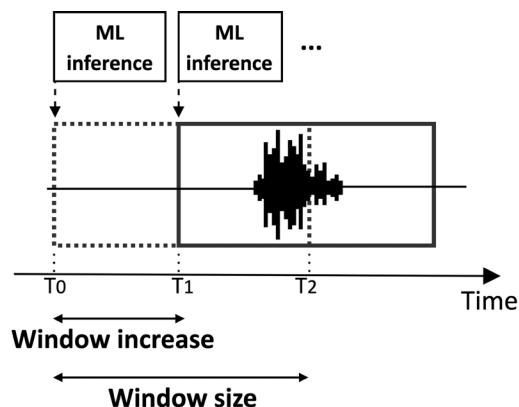


Figure 4.21: Window size versus Window increase

Therefore, the **Window size** value depends on the training sample length (1 s) and may affect the accuracy of results. On the other hand, the **Window increase** value does not impact the training results but affects the chances of getting the correct start of the utterance in the final application. In a KWS application, we generally do not know when the spoken word begins. As a result, the only way to detect the word effectively is to split the audio stream into overlapped windows (or segments) and execute ML inference on each one. The smaller the **Window increase** value, the higher the probability of detecting the word. However, the **Window increase** value must be carefully selected based on the latency of the ML inference, as each one can only begin once the previous has been completed.



Let us set the window increase to 500 ms for now. We will determine whether to increase or decrease this value based on the estimated model latency at a later stage.

At this point, follow the following steps to design a processing block for extracting MFE features from the training samples:

Step 1:

Click the **Add a processing block** button and select **Audio (MFE)** from the list.

Step 2:

Click the **Add a learning block** button and select **Classification** from the list. The **Output features** block should report the seven output classes to recognize, as shown in the following screenshot:

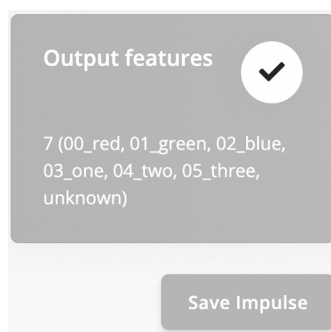


Figure 4.22: Output labels

Save the impulse by clicking on the **Save Impulse** button.

Step 3:

Click on the **MFE** option from the **Impulse design** category:

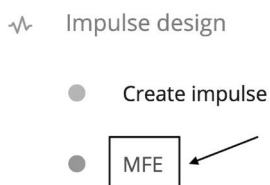


Figure 4.23: MFE option

In the new window, you should see the parameters affecting the MFE feature extraction, such as the frame length, frame step, the number of Mel frequencies to extract, and so on. You can keep the MFE parameters in this recipe at their default values.

Step 4:

Extract the MFE features from each training sample by clicking on the **Generate features** button at the top of the page:



Figure 4.24: Generate features button

Then, click the **Generate features** button to extract the MFE from all training samples. Edge Impulse will return **Job completed** in the console output at the end of this process.

There's more

In this recipe, we learned the theory behind the MFE features and how to extract them from all training samples in Edge Impulse.

Once you have extracted the MFE features from the training samples, you can examine the generated training dataset in the **Feature explorer** area, as shown in *Figure 4.25*:

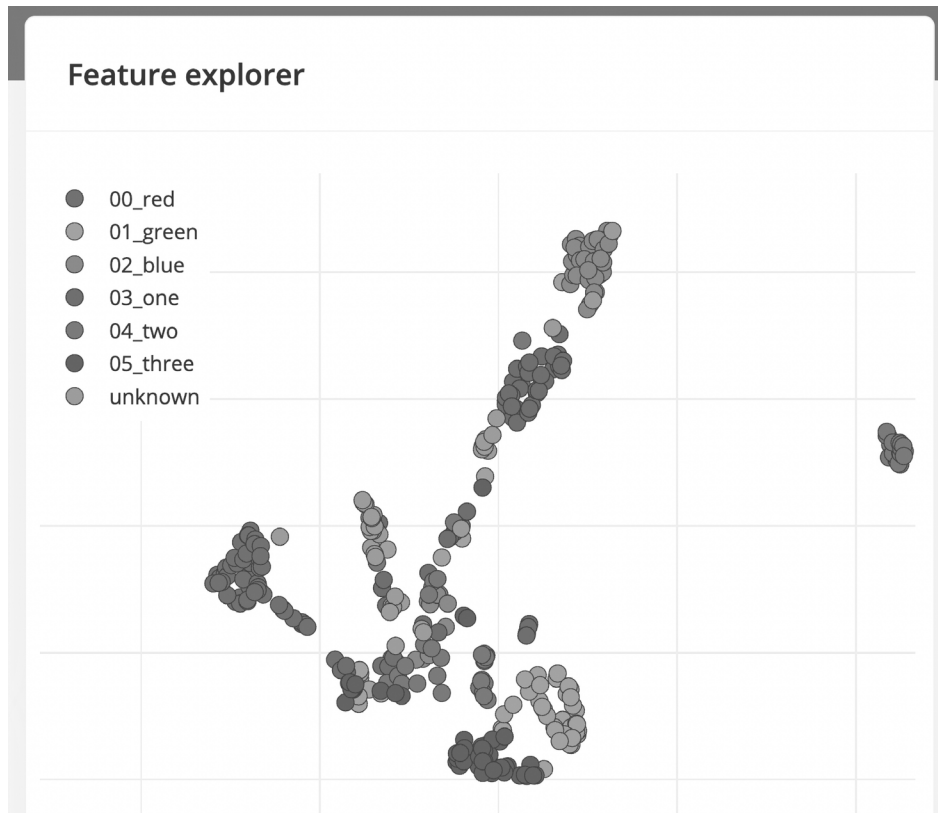


Figure 4.25: Feature explorer showing the seven output classes

The **feature explorer** chart is a **two-dimensional (2D)** scatter plot that you can use to assess whether the input features suit your problem. If so, the output classes (except the unknown output category) should be well separated.

Under the **Feature explorer** area, you can find the **On-device performance** section related to MFE computation:

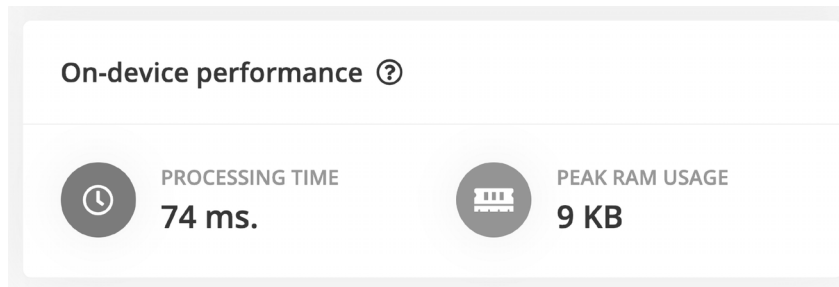


Figure 4.26: Estimated latency performance and RAM usage on the Arduino Nano 33 BLE Sense board

PROCESSING TIME (latency) and **PEAK RAM USAGE** (data memory) for the MFE computation are estimated considering the target device selected in **Dashboard | Project info**:

Project info

Project ID

Labeling method One label per d ▾

Latency calculations Arduino Nano ▾

Figure 4.27: Target device reported in Project info

Based on the estimated latency performance (**75 ms**), the value appears below the previously set **Window increase**. If the ML latency is not too high, there is a good chance that we can reduce **Window increase** further to increase the likelihood of correctly detecting the start of the spoken word in the end application.

Now that we have designed the pre-processing stage, it is time to focus on the ML model design. In the upcoming recipe, we will design a CNN in Edge Impulse for classifying words from sound.

Designing and training a CNN

In this recipe, we will be leveraging the following CNN architecture:

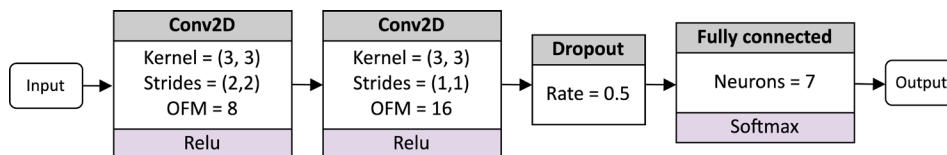


Figure 4.28: CNN architecture

The model presented in *Figure 4.28* is a modified version of what Edge Impulse will propose when designing the **neural network (NN)**. Our network has two 2D **convolution layers** with 8 and 16 **output feature maps (OFMs)**, one **dropout layer**, and one **fully connected layer**, followed by a **softmax activation**.

The network's input is the MFE feature extracted from the 1-s audio sample.

Getting ready

To get ready for this recipe, we need to understand how to design and train an ML model in Edge Impulse. Edge Impulse uses different ML frameworks for training depending on the chosen learning block. For a classification learning block, the framework employs **TensorFlow** with **Keras**. The model can be designed in two ways:

- **Visual mode (simple mode)**: This is the quickest method performed through the **user interface (UI)**. Edge Impulse provides several basic NN building blocks and architecture presets that are helpful for those new to **deep learning (DL)**.
- **Keras code mode (expert mode)**: If we require more control over the network architecture, we can edit the Keras code directly from the web browser.

In this recipe, we will use both approaches to get familiar with the Edge Impulse environment.

How to do it...

In Edge Impulse, click on the **Classification** option in the **Impulse design** section and follow the next steps to design and train the CNN presented in *Figure 4.28*:

Step 1:

Select the **2D Convolutional** architecture preset:

Neural network architecture

Architecture presets ⓘ 1D Convolutional (Default) 2D Convolutional

Figure 4.29: 2D Convolutional architecture preset

Then, ensure the network has the following layers:



Figure 4.30: Expected CNN architecture in visual mode

Then, remove the dropout layer between the two convolution layers (**2D conv / pool layer**) by clicking on the bin icon:

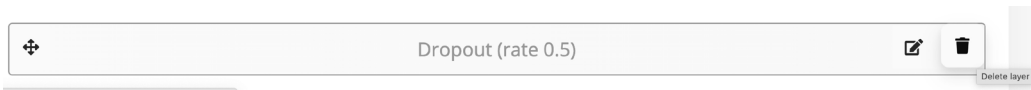


Figure 4.31: Button to remove a layer

If you need to adjust the OFM or kernel size of the 2D convolution layers, you can use the following buttons:

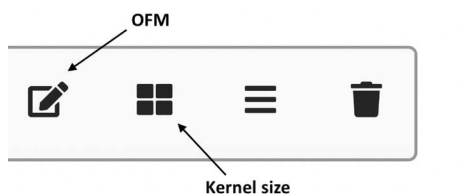


Figure 4.32: Buttons to change the OFM and kernel size

Step 2:

Switch to Keras (expert) mode by clicking on the **:** button and then clicking on the **Switch to Keras (expert) mode** option:

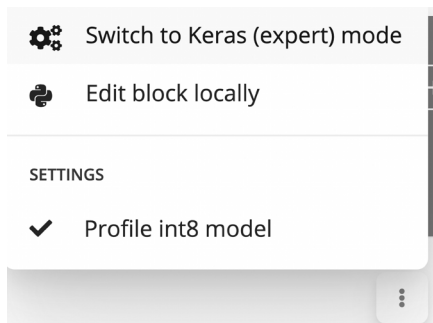


Figure 4.33: The Keras (expert) mode option

In the coding area, delete the **MaxPooling2D** layers:

```

21 rows = int(input_length / (columns * channels))
22 model.add(Reshape((rows, columns, channels), input_shape=(input_length, ))
23 model.add(Conv2D(8, kernel_size=3, kernel_constraint=tf.keras.constraints.l
    activation='relu'))
24 model.add(MaxPooling2D(pool_size=2, strides=2, padding='same')) ← Delete
25 model.add(Conv2D(16, kernel_size=3, kernel_constraint=tf.keras.constraints
    activation='relu'))
26 model.add(MaxPooling2D(pool_size=2, strides=2, padding='same')) ← Delete
27 model.add(Dropout(0.5))
28 model.add(Flatten())
29 model.add(Dense(classes, name='y_pred', activation='softmax'))
30

```

Figure 4.34: Delete the two max pooling layers from the Keras code

After, set the strides of the first convolution layer to (2,2):

```

model.add(Conv2D(8, strides=(2,2), kernel_size=3, activation='relu',
kernel_constraint=tf.keras.constraints.MaxNorm(1), padding='same'))

```

The pooling layer is a subsampling technique that reduces information propagated through the network and lowers the overfitting risk. However, this operator may increase latency and data memory usage. In memory-constraint devices such as microcontrollers, memory is a precious resource, and we need to use it as efficiently as possible. Therefore, *the idea is to adopt non-unit strides in convolution layers to reduce spatial dimensionality*. This approach is typically more performant because we skip the pooling layer computation entirely, and we can have faster convolution layers, given fewer output elements to process.

Step 3:

Launch the training by clicking on the **Start training** button:



Figure 4.35: The Start training button

The output console will report the accuracy and loss of the datasets during training after each epoch.

Upon completion of the training, Edge Impulse should provide a report on the model’s performance, including accuracy and loss metrics, as well as a confusion matrix for the validation dataset:

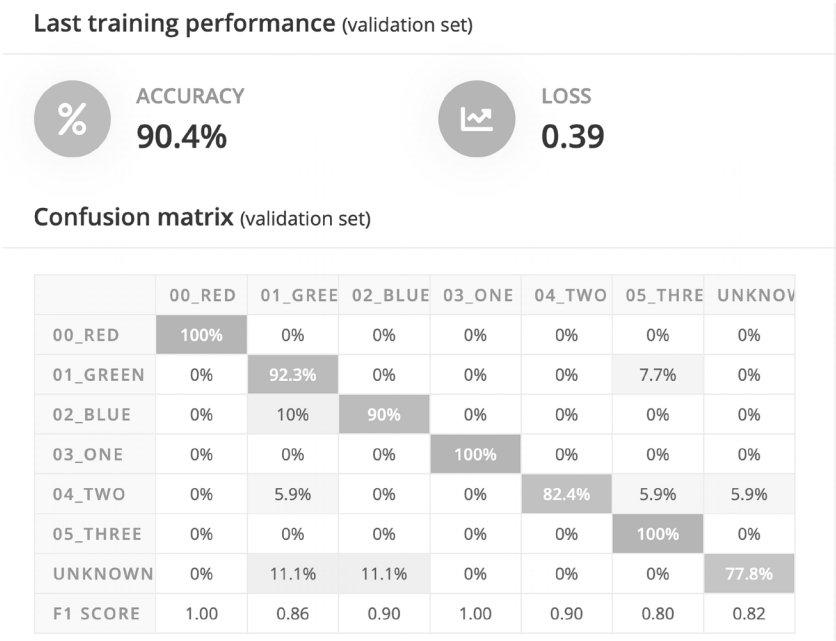


Figure 4.36: Confusion matrix

The previous screenshot shows that our KWS model achieved a very high level of performance, with an accuracy of over 90% on the validation dataset.

On the same page, you can also find the estimate of the model’s latency performance when running on the Arduino Nano:

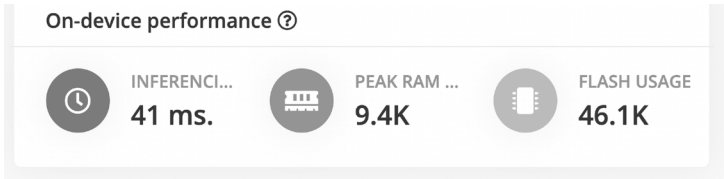


Figure 4.37: Expected on-device performance

If we combine the estimated latency performance for the ML model (41 ms) and the MFE computation (25 ms), we get a total latency of 66 ms. Based on this, reducing the Window increase to around 100 ms may be possible.

There's more

In this recipe, we learned how to design and train an ML model in Edge Impulse.

Due to the relatively limited number of samples in the dataset, there's a possibility that your model could achieve 100% accuracy, which might indicate overfitting to the training data. To mitigate this issue, Edge Impulse provides a method to augment the dataset. Data augmentation can be enabled in visual mode, allowing the generation of additional training data at runtime from the original samples. For example, these new samples can be generated by introducing noise or masking some frequency components. It is important to note that switching from visual mode to Keras code mode could result in the loss of any adjustments made in the Python code. Therefore, we recommend enabling data augmentation before entering Keras mode to avoid the need to reapply those changes in the Python code.

Now that the model is ready, how do you know whether it is the best solution for our target device?

In the upcoming recipe, we will explore how Edge Impulse can help with the EON Tuner.

Tuning model performance with the EON Tuner

In this recipe, we will use the Edge Impulse EON Tuner to find the best feature extraction method and ML architecture for KWS on the Arduino Nano.

Getting ready

Developing the most efficient ML pipeline for a given target platform is always challenging. One way to do this is through iterative experiments. For example, we can evaluate how some target metrics (latency, memory, and accuracy) change depending on the input feature generation and the model architecture. However, this process is time-consuming because several combinations need to be tested and evaluated. Furthermore, this approach requires familiarity with digital signal processing and NN architectures to know the parameters to tune.

The Edge Impulse EON Tuner (<https://docs.edgeimpulse.com/docs/eon-tuner>) is a powerful tool designed to automate discovering the most optimal ML solution for a given target platform. Unlike traditional **AutoML** tools focusing solely on finding the best model architecture, the EON Tuner also includes the processing block as part of the optimization problem. Therefore, the EON Tuner is an **E2E** optimizer for discovering the best combination of processing block and ML model for a given set of constraints, such as latency, RAM usage, and accuracy.

How to do it...

In Edge Impulse, click on the EON Tuner from the left-hand side menu:

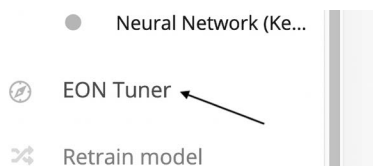


Figure 4.38: The EON Tuner option

Then, follow the next steps to learn how to find the most efficient ML-based pipeline for our KWS application:

Step 1:

Set up the EON Tuner parameters by clicking on the settings wheel icon in the **Target** area:

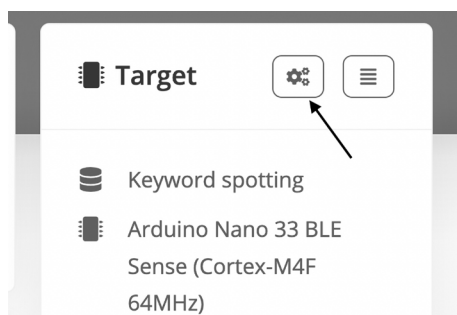


Figure 4.39: EON Tuner settings

Edge Impulse will open a new window for setting up the EON Tuner. In this window, set the **Dataset category**, **Target device**, and **Time per inference (ms)** values as follows:

- **Dataset category:** Keyword spotting
- **Target device:** Arduino Nano 33 BLE Sense (Cortex-M4F 64MHz)
- **Time per inference (ms):** 100



To explore alternative solutions with potentially higher accuracy or faster inference times than the model designed in the previous recipe, we have set the **Time for inference (ms)** value to 100 ms.

Then, save the EON Tuner settings by clicking on the **Save** button.

Step 2:

Launch the EON Tuner by clicking on the **Start EON Tuner** button. The tool will show the progress in the progress bar and report the discovered architectures in the same window, as shown in the following screenshot:

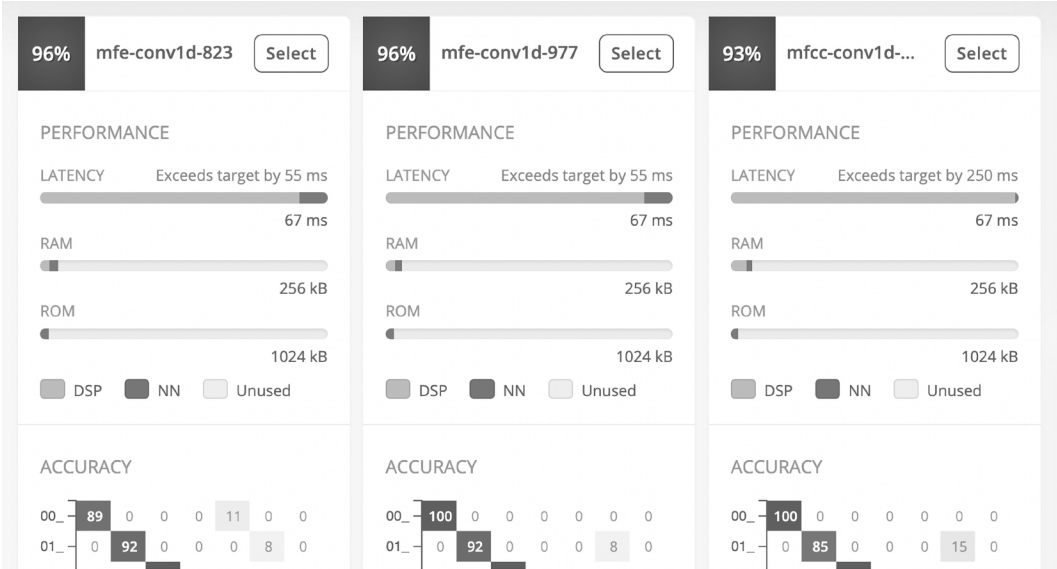


Figure 4.40: EON Tuner reports the confusion matrix and estimated on-device performance for each proposed ML solution

Upon completion of the discovery phase, the EON Tuner will present you with a collection of ML-based solutions. However, it is essential to note that Edge Impulse may also offer solutions that exceed the target latency as part of a more comprehensive evaluation, as some of these solutions may provide better accuracy.

From EON tuner findings, choose the solution with the highest accuracy and a latency of around 100 ms.

Once you have selected an architecture, Edge Impulse will ask you to update the primary model:

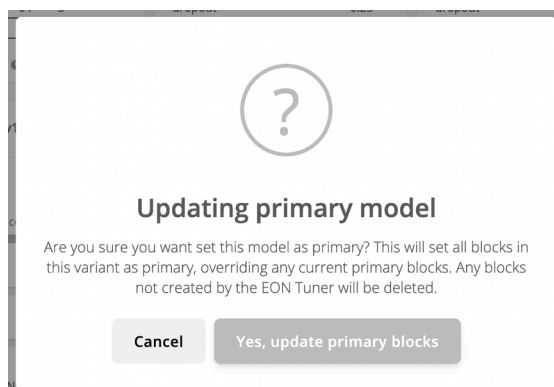


Figure 4.41: Pop-up window asking to update the model

Click on the **Yes, update primary blocks** button to override the architecture trained in the previous recipe. A pop-up window will appear, confirming that the model has been updated.

Now, click on the **Retrain model** option from the left-hand side panel:

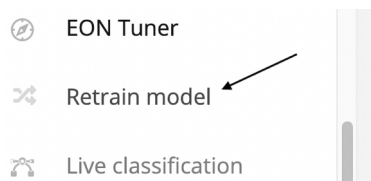


Figure 4.42: Retrain model option

In the new section, click the **Train model** button to train the network again.

There's more

In this recipe, we learned how to automate the search for the optimal feature extraction method and ML architecture for our target platform, thanks to the Edge Impulse EON Tuner.

This tool proved to be a valuable resource, especially when we are unsure how to design a model for optimal performance and memory consumption.

From the proposed solutions discovered by the EON Tuner for KWS, you may have noticed architectures based on 1D convolutions. Why can the 1D convolution be suitable in our case? Convolutional layers are designed to learn local features, and in the case of the MFE, these features present a spatial locality along the time axis. As a result, 1D convolutions, which are more performant than the 2D counterpart for the reduced computational workload, can be sufficient for learning these patterns effectively.

Having trained and tuned the model, it is now the time to test its effectiveness on unseen data, data that was not included in the training process.

In the upcoming recipe, we will assess the model accuracy on the test dataset and with live samples taken from a smartphone using Edge Impulse.

Live classifications with a smartphone

When discussing model testing, we usually refer to evaluating the trained model on the test dataset. However, model testing in Edge Impulse is more than that.

In this recipe, we will learn how to test model performance on the test dataset and show a way to perform live classifications with a smartphone.

Getting ready

In Edge Impulse, there are two ways to evaluate the accuracy of a model:

- **Model testing on the test dataset:** We assess the accuracy using the test dataset. The test dataset provides an unbiased evaluation of model effectiveness because the samples are not used directly or indirectly during training.
- **Live classification:** This is a unique feature of Edge Impulse whereby we can record new samples from a smartphone or a supported device (for example, the Arduino Nano).

The live classification approach benefits from testing the trained model in the real world before deploying the application on the target platform.

In the upcoming *How to do it...* section, we will show you how to carry out both evaluations in Edge Impulse.

How to do it...

The following steps will guide you through the accuracy evaluation of a trained model using a test dataset and a live classification tool:

Step 1:

In Edge Impulse, click on the **Model testing** option from the left-hand side panel:

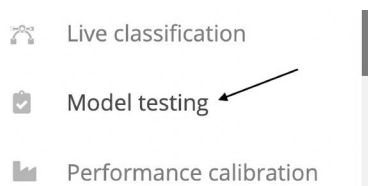


Figure 4.43: The Model testing option

On the new page, click the **Classify all** button to start the accuracy evaluation on the test dataset.

Edge Impulse will extract the MFE from the test set, run the trained model, and report the performance in the confusion matrix.

Step 2:

Click on the **Live classification** option from the left-hand side panel:

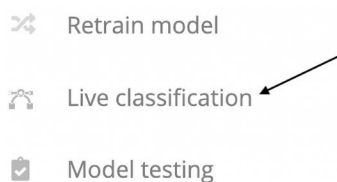


Figure 4.44: The Live classification option

Ensure the smartphone is reported in the **Device** list:

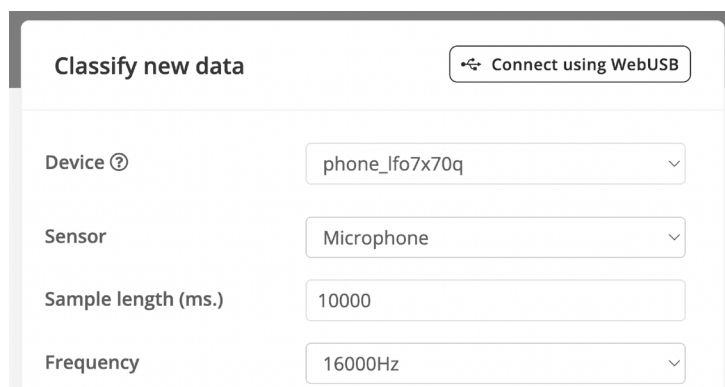
A form titled 'Classify new data' with a 'Connect using WebUSB' button. Below the title are four fields: 'Device' (dropdown menu showing 'phone_lfo7x70q'), 'Sensor' (dropdown menu showing 'Microphone'), 'Sample length (ms.)' (text input showing '10000'), and 'Frequency' (dropdown menu showing '16000Hz').

Figure 4.45: Classify new data

In the **Classify new data** section, choose **Microphone** from the **Sensor** drop-down list. Then, adjust the **Sample length (ms)** and **Frequency** to 10000 and 16000 Hz, respectively. By doing this, we will record the audio for 10 seconds, with a sampling rate of 16 kHz.

Step 3:

Click on the **Start sampling** button and then tap on **Give access to the Microphone** on your phone. Record any of our six utterances (red, green, blue, one, two, and three) for 10 seconds. The audio sample will be uploaded on Edge Impulse once you have completed the recording.

At this point, Edge Impulse will divide the recording into individual frames, each containing a one-second-long sample. The trained model will then be tested on each of these frames, and the results of the classification process will be presented in two tables. The first table will offer a **general overview** of the detection results for each output category:

CATEGORY	COUNT
blue	1
green	1
one	1
red	2
three	1
two	1
unknown	29
uncertain	1

Figure 4.46: Generic summary reporting the number of detections for each keyword

The second table, instead, will offer a more **in-depth analysis** of the results, providing information on the probability of each class at every timestamp:

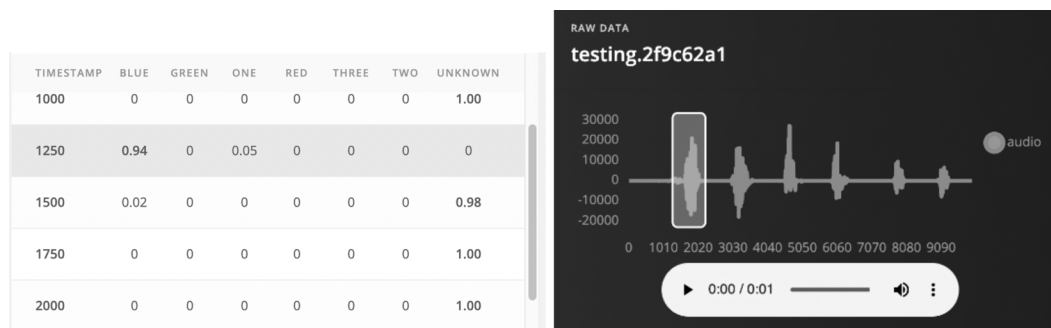


Figure 4.47: Detailed analysis reporting the probability of the classes at each timestamp

If you click on a table entry, you can visualize the corresponding audio waveform in the same window, as shown in the previous figure.

When using the **Live classification** tool, Edge Impulse will divide the recording into separate frames based on the **Window Increase** setting specified in **Impulse design**. Therefore, depending on the expected latency for the MFE computation and ML inference, you may need to adjust this value to observe changes in the detection of spoken words.



It is recommended to set the Window Increase to a value that is a multiple of the frame length utilized to calculate the MFE. The reason for that will be evident in the upcoming section.

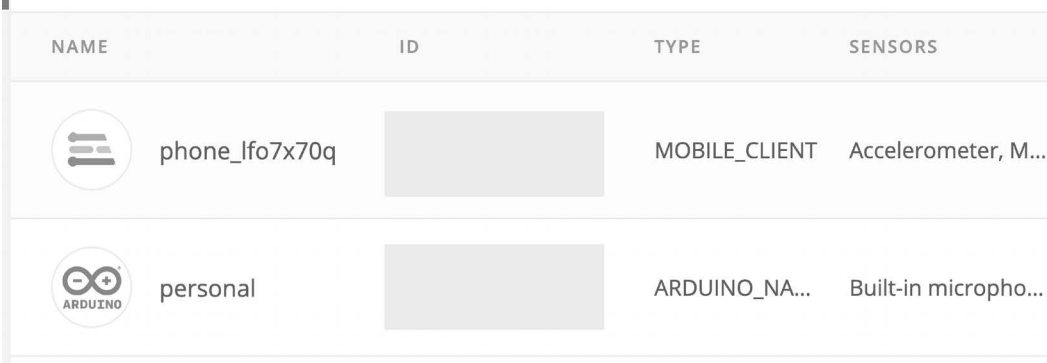
There's more

In this recipe, we learned how to assess a model's accuracy on unseen data using a test dataset and live samples acquired with a smartphone.

If you found live classification with the smartphone helpful, live classification with the Arduino Nano will be even more valuable.

Testing model performance with the sensor used in the final application is a good practice to have more confidence in the accuracy of results. Thanks to Edge Impulse, performing live classification on the Arduino Nano is possible.

To do so, you just need to ensure that the Arduino Nano is paired with Edge Impulse by clicking on **Devices** from the left-hand side panel, as shown in *Figure 4.48*:





NAME	ID	TYPE	SENSORS
 phone_lfo7x70q		MOBILE_CLIENT	Accelerometer, M...
 personal		ARDUINO_NA...	Built-in micropho...

Figure 4.48: List of devices paired with Edge Impulse

As you can see from the preceding screenshot, the Arduino Nano (**personal**) is listed in the **Your devices** section.

Now, go to **Live classification** and select the Arduino Nano 33 BLE Sense board's name (**personal**) from the **Device** drop-down list. You can now record audio samples from the Arduino Nano and check whether the model works as expected.

At this point, you have the model trained and thoroughly tested. Therefore, the only remaining step is to deploy the final application on the target device.

In the upcoming final recipe, we will build the KWS application on the Arduino Nano.

Keyword spotting on the Arduino Nano

As you might have guessed, it is time to deploy the KWS application on the Arduino Nano.

In this recipe, we will show how to do so with the help of Edge Impulse.

Getting ready

The application on the Arduino Nano will be based on the **nano_ble33_sense_microphone_continuous.cpp** example provided by Edge Impulse, which implements a real-time KWS application. Before adjusting this code sample, we want to examine how it works to get ready for this final recipe.

Learning how a real-time KWS application works

A real-time KWS application—for example, the one used in a smart assistant—should capture and process all pieces of the audio stream to never miss any events. Therefore, the application must *record audio and run inference simultaneously* so we do not miss any information.

On a microcontroller, parallel tasks can be performed in two ways:

- With a **real-time OS (RTOS)**. In this case, we can use two threads for capturing and processing the audio data.
- With a dedicated peripheral such as **direct memory access (DMA)** attached to the ADC. DMA allows data transfer without interfering with the main program running on the processor.

In this recipe, we won't deal with this aspect directly. In fact, the `nano_ble33_sense_microphone_continuous.cpp` example already provides an application where the audio recording and inference run simultaneously through a **double-buffering** mechanism. Double buffering uses two buffers of fixed size, where the following applies:

- One buffer is dedicated to the audio *recording* task.
- One buffer is dedicated to the *processing* task (feature extraction and ML inference).

Each buffer keeps the audio samples required for a window increase recording. Therefore, the buffer size can be calculated through the following formula:

$$buffer_{size} = sampling_{rate} \cdot window_{increase}$$

The preceding formula can be defined as the product of the following:

- *sampling_{rate}*: The sampling frequency in Hz. For example, 16 kHz = 16000 Hz.
- *window_{increase}*: The window increase in s. For example, 250 ms = 0.250 s.

Therefore, if we sample the audio signal at 16 kHz and the window increase is 250 ms, each buffer will have a capacity of 4,000 samples.

These two buffers are continuously switched between *recording* and *processing* tasks, and the following diagram visually shows how:

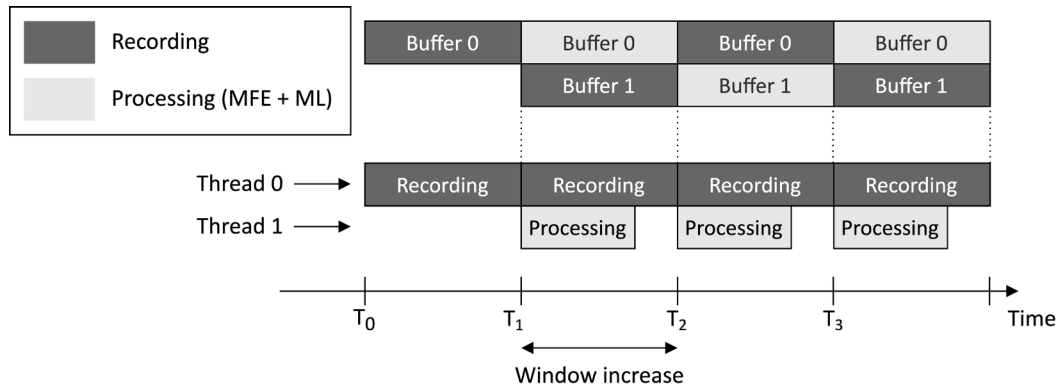


Figure 4.49: Recording and processing tasks running simultaneously

From the preceding diagram, we can observe the following:

1. At $t=T_0$, the *recording* task starts filling **Buffer 0**.
2. At $t=T_1$, **Buffer 0** is full. Therefore, the *processing* task can start the inference using the data in **Buffer 0**. Meanwhile, the *recording* task continues to record audio data in the background using **Buffer 1**.
3. At $t=T_2$, **Buffer 1** is full. Therefore, the *processing* task must have finished the previous computation before starting a new one.

Keeping the window increase as short as possible has the following benefits:

- Increases the probability of getting the correct beginning of an utterance
- Reduces the computation time of feature extraction because this is only computed on the window increase



From the second point, it should be clear why we recommend setting the **Window increase** value to a multiple of the MFE frame length. By doing so, we can extract the MFE features for each frame without any overlap or gaps.

However, the **Window increase** value should be long enough to guarantee that the processing task can be completed within this time frame.

At this point, you might have one question in mind: *If we have a Window increase of 250 ms, how can the double buffers feed the NN since the model expects a 1-s audio sample?*

The double buffers are not the NN input but the input for an additional buffer containing the samples of the 1-s audio. This buffer stores the data on a **first-in, first-out (FIFO)** basis and provides the actual input to the ML model, as shown in the following diagram:

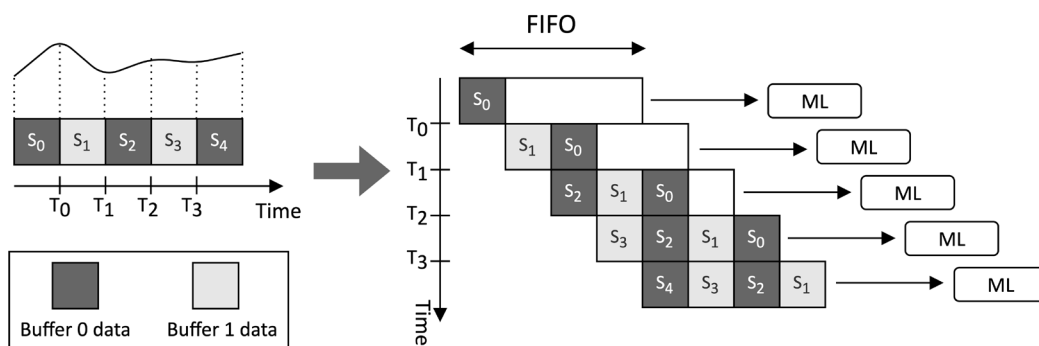


Figure 4.50: The FIFO buffer is used to feed the NN model

Therefore, every time we start a new processing task, the sampled data is copied into the FIFO queue before running the inference.

How to do it...

In Edge Impulse, click on the **Deployment** option from the left-hand side menu:



Figure 4.51: Deployment option

Then, select **Arduino Library** from **deployment options**, as shown in the following screenshot:

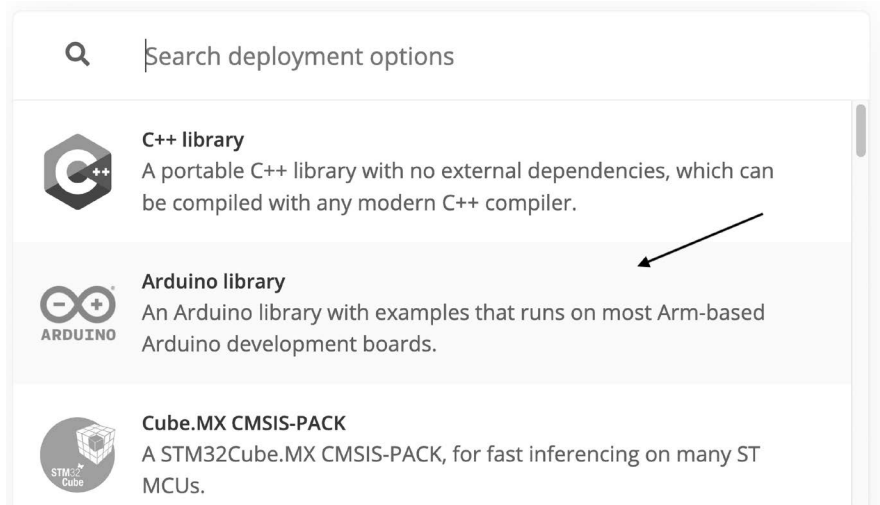


Figure 4.52: Edge Impulse deployment section

Next, click the **Build** button at the bottom of the page and save the ZIP file on your machine. The ZIP file is an Arduino library containing the KWS application, the routines for the MFE feature extraction, and a few ready-to-use examples for the Arduino Nano 33 BLE Sense board.



Now, follow the following steps to tweak the `nano_ble33_sense_microphone_continuous.cpp` file to control the built-in RGB LEDs on the Arduino Nano using our voice:

Step 1:

Open the Arduino IDE and import the library created by Edge Impulse. To do so, click on the **Libraries** tab from the left pane and then click on the **Import** button, as shown in the following screenshot:



Figure 4.53: Import library in the Arduino Web Editor

Once imported, open the **nano_ble33_sense_microphone_continuous** example from **Examples | FROM LIBRARIES | <name_of_your_project>_INFERENCEING | nano_ble33_sense**:

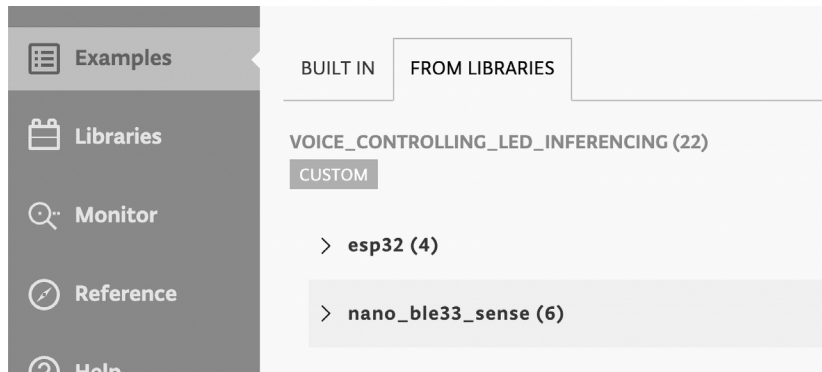


Figure 4.54: Example folder in the Arduino Web Editor

In our case, **<name_of_your_project>** is **VOICE_CONTROLLING_LED**, which matches the name given to our Edge Impulse project.



In the file, the `EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW` macro defines the window increase in terms of the number of frames processed per model window. We can keep it at the default value.

Step 2:

Include the `mbed.h` header file:

```
#include "mbed.h"
```

Then, create a global array of `mbed::DigitalOut` objects to drive the built-in RGB LEDs:

```
mbed::DigitalOut rgb[] = {p24, p16, p6};
constexpr int ON = 0;
constexpr int OFF = 1;
```

The initialization of `mbed::DigitalOut` requires the `PinName` value of the RGB LEDs. The pin names can be found in the Arduino Nano 33 BLE Sense board schematic (https://content.arduino.cc/assets/NANO33BLE_V2.0_sch.pdf):

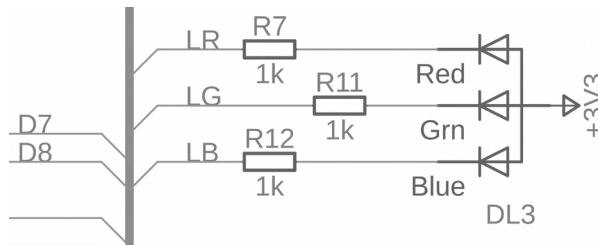


Figure 4.55: The built-in RGB LEDs are powered by a current-sinking configuration

The RGB LEDs—identified with the labels **LR**, **LG**, and **LB**—are controlled by a current-sinking circuit and are connected to **P0.24**, **P0.16**, and **P0.06**:

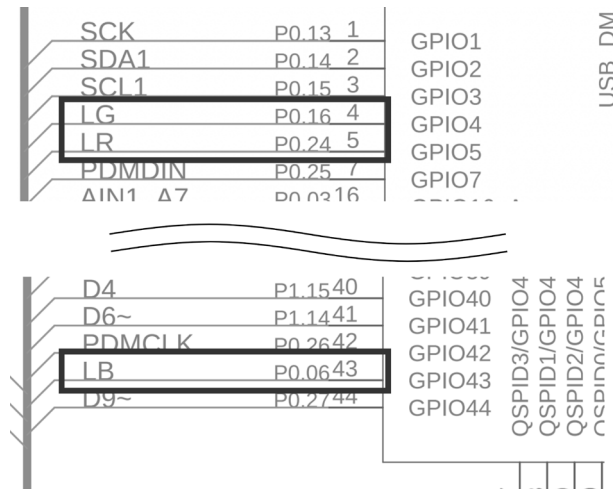


Figure 4.56: The RGB LEDs are connected to P0.24, P0.16, and P0.06

Therefore, the **general-purpose input/output (GPIO)** pins must supply 0 volts (V) (LOW) to turn on the LEDs.

Step 3:

Define an integer global variable (`current_color`) to keep track of the last detected color. Initialize it to 0 (*red*):

```
constexpr int RED = 0;
constexpr int GREEN = 1;
constexpr int BLUE = 2;
size_t current_color = RED;
```

Step 4:

Implement the utility functions to check whether the index of the output class is color or unknown:

```
constexpr size_t NUM_COLORS = 3;

bool is_color(size_t ix) {
    return ix < NUM_COLORS;
}

bool is_unknown(size_t ix) {
    return ix == (EI_CLASSIFIER_LABEL_COUNT - 1);
}
```

The preceding two functions will determine whether the recognized word is a color or a number in the application.



`EI_CLASSIFIER_LABEL_COUNT` is a C define provided by Edge Impulse and is equal to the number of output categories.

Step 5:

In the `setup()` function and after the serial peripheral initialization, initialize the built-in RGB LEDs by turning on just the `current_color` LED:

```
rgb[RED] = OFF; rgb[GREEN] = OFF; rgb[BLUE] = OFF;
rgb[current_color] = ON;
```

Step 6:

In the `loop()` function, set the **moving average (MA)** flag to false in the `run_classifier_continuous()` function:

```
run_classifier_continuous(&signal, &result, debug_nn, false);
```

The `run_classifier_continuous()` function is responsible for the model inference. The MA is disabled by passing false after the `debug_nn` parameter. However, *why do we turn off this functionality?*

The MA is an effective method to filter out false detections when the window increase is small. For example, consider the word *bluebird*. This word contains *blue*, but it is not the utterance we want to recognize. When running continuous inference with a slight window increase, processing small pieces of the audio stream at a time is beneficial to detect the right word because the word *blue* may be classified with high confidence in one piece but not in the others. So, *the goal of the MA is to average the results of classifications over time to avoid false detections.*

As you may be able to guess, the output class must have multiple high-rated classifications when using the MA. Therefore, *what happens if the window increase is large?*

When the window increase is large (for example, greater than 100 ms), we process fewer segments per second, and then the MA can filter out all the classifications. Since our window increase can be between 250 ms and 500 ms (depending on the ML architecture chosen), we recommend you turn it off to avoid filtering out the classifications.

Step 7:

In the `loop()` function, delete the code after `run_classifier_continuous()` till the end of the function, as this section will be substituted with the code required to drive the RGB LED.

Step 8:

In the `loop()` function and after `run_classifier_continuous()`, write the code to return the class with highest probability:

```
size_t ix = 0;
size_t ix_max = 0;
float pb_max = 0.0;
for(ix; ix < EI_CLASSIFIER_LABEL_COUNT; ix++) {
    if(result.classification[ix].value > pb_max) {
```

```

        ix_max = ix;
        pb_max = result.classification[ix].value;
    }
}

```

In the preceding code snippet, we iterate through all the output classes (EI_CLASSIFIER_LABEL_COUNT) and keep the index (ix) variable with the maximum classification value (result.classification[ix].value).

Step 9:

In the loop() function, write the code to control the RGB led light based on the outcome of the model classification. To do so, if the probability of the output category (pb_max) is higher than a fixed threshold (for example, 0.5) and the label is not *unknown*, stop the recording and check whether the output class is color:

```

#define PROBABILITY_THR 0.5
if(pb_max > PROBABILITY_THR && !is_unknown(ix_max)) {
    record_ready = false;
    if(is_color(ix_max)) {

```

If the label is a color different from the last one detected, turn off current_color and turn on new_color:

```

        size_t new_color = ix_max;
        if(new_color != current_color) {
            rgb[current_color] = OFF;
            rgb[new_color] = ON;
            current_color = new_color;
        }
    } // if(is_color(ix_max))

```

If the label is not a color, it means that it is a number. Therefore, blink the current_color LED for the recognized number of times:

```

    else {
        size_t num_blinks = ix_max - 2;
        for(size_t i = 0; i < num_blinks; ++i) {
            rgb[current_color] = OFF;
            delay(1000);
            rgb[current_color] = ON;

```

```
        delay(1000);  
    }  
} // else
```

Before starting a new inference, wait for 1 second and reset the recording buffer:

```
    delay(1000);  
    inference.buf_select = 0;  
    inference.buf_count = 0;  
    inference.buf_ready = 0;  
    record_ready = true;  
}
```

Compile and upload the sketch on the Arduino Nano. You should now be able to change the color of the LED or make it blink with your voice!

There's more

In this final recipe, we learned how to tweak the `nano_ble33_sense_microphone_continuous.cpp` example generated by Edge Impulse to control the RGB LED on the Arduino Nano with our voice.

If you open the Serial Monitor in the Arduino IDE, you can see the log produced by Edge Impulse during the execution of the program. If you come across the warning message:

```
Error sample buffer overrun. Decrease the number of slices per model  
window (EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW)  
ERR: Failed to record audio...
```

It means that you cannot record audio at the desired window increase rate because the latency of the model inference (which includes feature extraction and inference) exceeds the window increase duration. Therefore, to fix this issue, you simply need to reduce the value of `EI_CLASSIFIER_SLICES_PER_MODEL_WINDOW` to a lower setting. By doing so, this error message should disappear.

Summary

The recipes presented in this chapter demonstrated how to build an end-to-end KWS application with Edge Impulse and the Arduino Nano.

Initially, we learned how to prepare the dataset by recording audio samples with a smartphone and the Arduino Nano directly from Edge Impulse.

Afterward, we delved into model design. Here, we introduced the MFE (or Mel-spectrogram) as a suitable input feature for training a CNN model for KWS.

Then, we trained a generic CNN and used the Edge Impulse EON Tuner to discover more efficient model architectures for our target platform regarding accuracy, latency performance, and memory consumption.

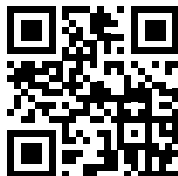
Finally, we tested the model's accuracy on the test dataset and live audio samples recorded with a smartphone and deployed the KWS application on the Arduino Nano.

In this chapter, we have started discussing how to build a tinyML application with a microphone using Edge Impulse and the Arduino Nano. With the next project, we will continue this discussion on the Raspberry Pi Pico to recognize music genres with the help of TensorFlow and the CMSIS-DSP library.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



5

Recognizing Music Genres with TensorFlow and the Raspberry Pi Pico – Part 1

The project we will develop together holds a special place in my heart because it brings back memories of my first programming experience on an **Arm Cortex-M** microcontroller.

Back in my university days, portable MP3 audio players were definitely one of the coolest things to have. I was fascinated by this technology that allowed me to carry thousands of high-quality songs in my pocket to enjoy anywhere. As a technology and music enthusiast, I wanted to learn more about it. Therefore, I undertook the challenge of building an audio player from scratch using a microcontroller and a touch screen (<https://youtu.be/LXm6-LuMmUU>).

Completing that project was fun and gave me valuable hands-on experience. One feature I hoped to include was a **machine learning (ML)** algorithm for recognizing the music genre to auto-equalize the sound and get the best listening experience.

However, **deep learning (DL)** was not advanced enough then, especially on edge devices, so I had to abandon the idea. Today, with the recent advances in **tinyML**, I am excited to demonstrate that it is feasible to bring that algorithm to life on microcontrollers.

This project aims to recognize three music genres from recordings obtained with a microphone connected to the Raspberry Pi Pico. The music genres we will classify are disco, jazz, and metal. Since the project offers many learning opportunities, it is split into two parts to give as much exposure to the technical aspects as possible.

This first part will focus on the dataset preparation and the analysis of the feature extraction technique employed for classifying music genres: the **Mel Frequency Cepstral Coefficients (MFCCs)**.

By the end of this first part, we will better understand how to connect an external microphone to the Raspberry Pi Pico, generate audio files from audio samples transmitted over the serial connection, and extract the MFCC features from audio clips using TensorFlow.

In this first part, we're going to cover the following recipes:

- Connecting the microphone to the Raspberry Pi Pico
- Recording audio samples with the Raspberry Pi Pico
- Generating audio files from samples transmitted over the serial
- Building the dataset for classifying music genres
- Extracting MFCCs from audio samples with TensorFlow

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- A Raspberry Pi Pico board
- A SparkFun RedBoard Artemis Nano (optional)
- A micro-USB data cable
- A USB-C data cable (optional)
- 1 x electret microphone amplifier – MAX9814
- 5-pin press-fit header strip (optional but recommended)
- 1 x half-size solderless breadboard
- 1 x push-button
- 8 x jumper wires
- Laptop/PC with either Linux, macOS, or Windows
- Google Drive account
- Kaggle account



The source code and additional material are available in the Chapter05_06 folder in the GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Chapter05_06

Connecting the microphone to the Raspberry Pi Pico

Our application requires a microphone to record songs and classify their music genre. Since the Raspberry Pi Pico does not have a built-in microphone, we need to employ an external one and figure out the appropriate way to connect it to the microcontroller.

This recipe will help you achieve this objective by offering a step-by-step guide on integrating a microphone into an electronic circuit alongside the microcontroller.

Getting ready

The microphone put into action in this recipe is a low-cost **electret microphone** with the **MAX9814 amplifier**, which you can buy, for example, from one of the following distributors:

- Pimoroni: <https://shop.pimoroni.com/products/adafruit-electret-microphone-amplifier-max9814-w-auto-gain-control>
- Adafruit: <https://www.adafruit.com/product/1713>

The signal generated by a microphone is generally weak, as it ranges from several microvolts to a few millivolts, depending on the intensity of the sound. For this reason, the audio signal is often amplified to ensure that it can be effectively sampled by the microcontroller's **analog-to-digital converter (ADC)**.

Our microphone has a built-in amplifier, which allows high-quality recordings thanks to its **automatic gain control (AGC)** mechanism. This mechanism ensures the sound is clear and consistently distinguishable in environments with unpredictable background noise levels.

The microphone module, as shown in *Figure 5.1*, has a design with five pinholes at its base to accommodate the header strip:

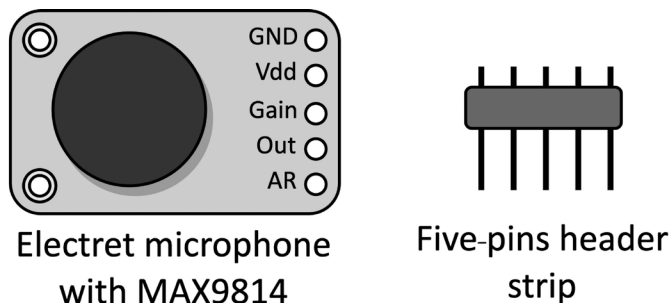


Figure 5.1: Electret microphone with MAX9814

Unfortunately, the header strip is generally unsoldered. However, you have two options to connect the header strip to the microphone module:

- Using a **press-fit header strip**. Its header pins have a metal loop at the end, which can snugly fit into a standard header pinhole. With this solution, you do not need soldering because the pins ensure a firm mechanical connection. Press-fit header strips are available from various electronic distributors, such as **DigiKey**: <https://www.digikey.com/en/products/detail/samtec-inc/PHT-102-01-L-S/6691694>.
- Soldering the header strip. If you are unfamiliar with soldering, we recommend reading the following tutorial: <https://learn.adafruit.com/adafruit-agc-electret-microphone-amplifier-max9814/assembly>.

The microphone requires a supply voltage between 2.7V and 5.5V to operate, which should be provided through the **Vdd** and **GND** pins. The device produces an analog signal voltage on the **Out** pin directly proportional to the intensity of the sounds it detects. This signal is the output of the MAX9814 amplifier and has a maximum **peak-to-peak voltage (Vpp)** of 2 Vpp on a 1.25V **direct current (DC)** bias.



The term **Vpp** refers to the voltage amplitude of a waveform's peak-to-peak value, representing the full vertical height of the waveform.

The maximum gain of the MAX9814 amplifier can be modified using the **Gain** pin. According to the microphone module's documentation, if you leave the **Gain** pin unconnected (floating), you will get a 60 dB maximum gain, while connecting it to **GND** or **Vdd** yields gains of 50 dB and 40 dB, respectively.



We advise connecting the **Gain** pin to **Vdd** to prevent an excessively noisy output signal.

With an understanding of the microphone's essential features, let's explore how to feed the output signal generated by this device into the Raspberry Pi Pico.

Connecting the microphone to the ADC pin

The voltage variations produced by the microphone require conversion to a digital format through the ADC peripheral.

The RP2040 microcontroller on the Raspberry Pi Pico has four ADCs, but only three of them can be used of the external inputs, which are shown here:

ADC name	ADC0	ADC1	ADC2
Pin	GP26	GP27	GP28

Figure 5.2: ADC pins

The expected voltage range for the ADC on the Raspberry Pi Pico is between 0V and 3.3V, perfect for the signal from our electret microphone.

How to do it...

Let's start by placing the Raspberry Pi Pico on the breadboard. You should mount the platform vertically, as we did in *Chapter 2, Unleashing Your Creativity with Microcontrollers*.

Once you have placed the device on the breadboard, make sure the USB data cable is disconnected from the microcontroller and follow the next steps to connect the microphone to the Raspberry Pi Pico.

Step 1:

Place a push-button on the breadboard and connect it to the **GP10** pin (**GPIO** pin) and **GND**:

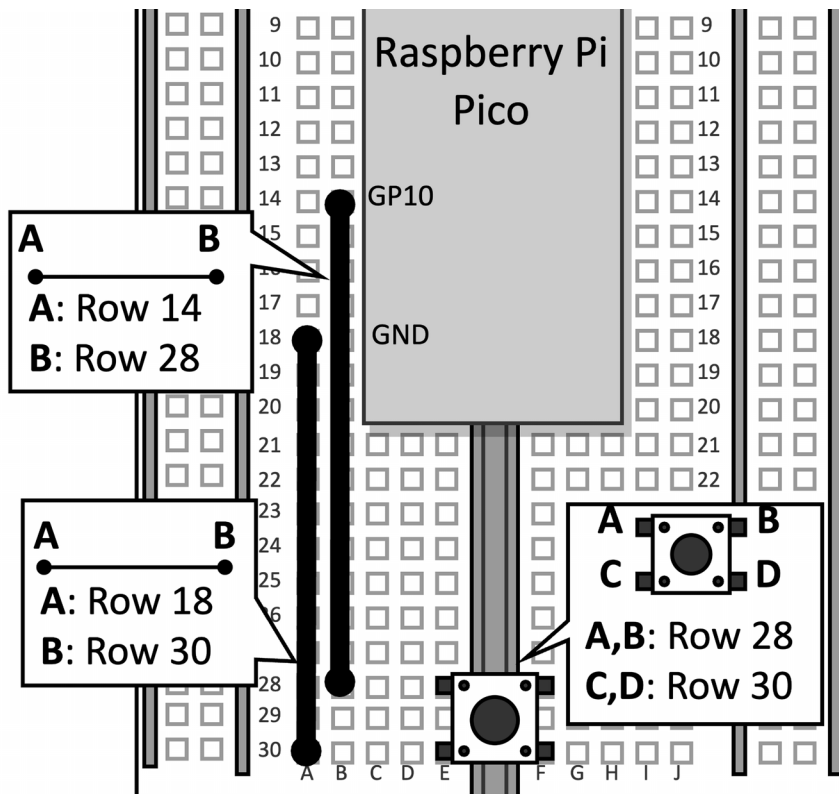


Figure 5.3: Push-button connected to GP10 and GND

The push-button will be used to start the audio recording.

Step 2:

Place the microphone on the breadboard and connect it to the Raspberry Pi Pico's pin reserved for the ADC0 (GP26):

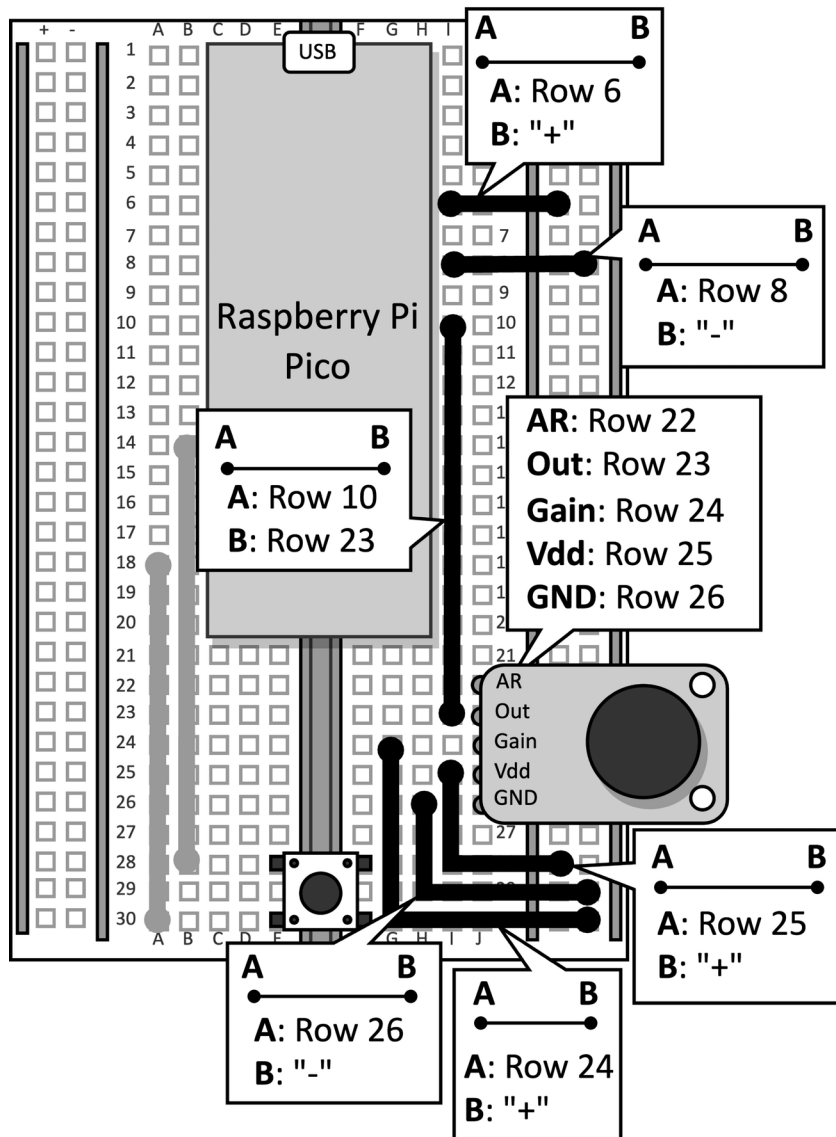


Figure 5.4: Complete electronic circuit

The preceding image shows how you should connect the **Vdd**, **GND**, **Out**, and **Gain** pins of the microphone module on the breadboard.

The **Vdd** pin is used to supply voltage to the microphone amplifier and must be stable and equal to the ADC supply voltage. These conditions are required to reduce the noise on the analog signal generated by the microphone. Therefore, we advise connecting the **Vdd** pin to the **ADC_VREF** pin, which is the 3.3V reference voltage of the ADC peripheral on the Raspberry Pi Pico.

The **GND** pin is used to supply the ground to the microphone amplifier and should be the same as the ADC peripheral. Since analog signals are more susceptible to noise than digital ones, the Raspberry Pi Pico offers a dedicated ground for ADCs: the **analog ground (AGND)**. Therefore, we recommend connecting the **GND** pin to the **AGND** pin to decouple the analog circuit from the digital one.

The microphone module uses the **Out** pin to transmit the amplified analog signal and should be connected to the ADC input pin. Since we use the ADC0 peripheral, the **Out** pin is connected to the **GP26** pin.

The following table summarizes the connections we made between the Raspberry Pi Pico and the microphone module:

Mic with MAX9814 - Pin	Vdd	GND	Out
Raspberry Pi Pico - Pin	ADC_VREF	AGND	GP26

Figure 5.5: Microphone connections with the Raspberry Pi Pico

Besides the **Vdd**, **GND**, and **Out** pins, we also connected the **Gain** pin to the **Vdd** pin. The **Gain** pin is used to adjust the gain of the amplifier. By connecting the **Gain** pin to the **Vdd** pin, we get a 40 dB amplification gain.



The **AR** pin on the microphone module defines the **attack and release** ratio and is not required in this project. However, you can discover more about the functionality offered by this pin in the MAX9814 datasheet (<https://datasheets.maximintegrated.com/en/ds/MAX9814.pdf>).

Because the circuit is completed, you can now plug the Raspberry Pi Pico into your computer through the micro-USB data cable.

There's more...

In this recipe, we learned how to connect an external microphone module to the Raspberry Pi Pico.

However, can we check whether the microphone works without connecting it to the microcontroller? The answer is yes, and you just need to read the following tutorial to discover how you can do it with a headphone jack: <https://learn.adafruit.com/adafruit-agc-electret-microphone-amplifier-max9814/wiring-and-test>.

For those using the SparkFun Artemis Nano microcontroller, there is no need for an external microphone as one is already built-in. However, this microphone differs from the one discussed in this recipe. Instead of providing an analog output, the microphone on the SparkFun Artemis Nano delivers a digital output through **pulse density modulation (PDM)**.

Now that we have the complete circuit with a microphone and the microcontroller, we are ready to start recording audio samples.

In the upcoming recipe, we will discover how to develop a sketch to record audio clips for 4 seconds.

Recording audio samples with the Raspberry Pi Pico

Like any ML project, we need to prepare a dataset. For our purposes, the audio clips will be acquired with the microphone connected to the microcontroller. This choice will help us obtain an ML model with high accuracy because the samples derive from the exact microphone used in the final application.

In this recipe, we will create an Arduino sketch to record audio clips of 4 seconds with the Raspberry Pi Pico. The audio samples will then be transmitted over the serial connection and transformed into audio files in the following recipe.

Getting ready

Microcontrollers can be employed to build fully functional digital audio recorders when paired with a microphone.

To accomplish this task with our microphone, we must develop a program to sample the audio analog signal at regular intervals, also known as the **sample rate**.

To create the correct digital representation, we must consider the following:

- The sample rate should be chosen considering the **Nyquist–Shannon sampling** theorem. According to this theorem, we must sample the signal at least twice the highest frequency we want to capture.
- The **bit depth** to represent each digital sample. Common bit depths for audio are 16, 24, or 32 bits per sample. A higher bit depth provides a more accurate representation of the analog signal, resulting in higher fidelity recordings. However, the higher the bit depth, the more storage space is needed.

Since microcontrollers have limited memory, dealing with high-quality audio data can be challenging. For example, if we consider standard high-quality audio tracks, these are often recorded with a 44.1 KHz sample rate and use a 32-bit depth for each sample. As a result, such settings would require ~176 KB of memory storage for just 1 second of audio. Given that the Raspberry Pi Pico has only 264 KB of SRAM, it is clear that recording high-quality audio clips can pose challenges.

Nevertheless, high-fidelity songs are not essential for identifying the music genre. After all, human ears can still discern the type of music played, even from low-quality songs like those played over the phone. Therefore, in this project, we will halve both the sample rate and the bit depth traditionally used for recording high-quality music. Specifically, we will record songs at a 22.05 KHz sample rate and employ the 16-bit integer format for each sample. This choice will allow us to store an audio clip of 4 seconds in the SRAM of the Raspberry Pi Pico.

The ADC peripheral is necessary for converting analog signals into digital data that the microcontroller can process.

The subsequent section will outline the method used in this project to record audio signals using this peripheral with the timer interrupt.

Recording audio with ADC and timer interrupts

As we know from *Chapter 2, Unleashing Your Creativity with Microcontrollers*, interrupts are signals the microcontroller generates to respond to specific events. A particular class of interrupts is the **timer interrupt**, which can trigger an event when the timer reaches a predefined value. Therefore, the timer interrupt can be employed to sample the microphone's analog output at a regular rate.

The `mbed::Ticker` object from Mbed OS can be leveraged to set off periodic interrupts. To start the timer interrupt, the `mbed::Ticker` object offers the `attach_us()` method, which takes the **interrupt service routine (ISR)** to execute and the desired sample rate in microseconds as input arguments.

In our setup, the ISR will be tasked with sampling the microphone's signal and store the data in an audio buffer.

To stop the timer interrupt, we must invoke the `detach()` method of the `mbed::Ticker` object.

The ADC on the RP2040 microcontroller of the Raspberry Pi Pico has a 12-bit resolution and can sample signals with a maximum frequency of 500 KHz, providing sufficient capability for most audio recording applications.

Since the ADC has a 12-bit resolution, the digital values range from 0 to 4095. Therefore, the maximum value of 4095 corresponds to 3.3V, while the minimum value of 0 represents 0V.

The ADC will be configured in *one-shot mode*, which means the ADC will provide the sample as soon as we request it. However, *how can we program this peripheral?*

Programming the ADC with the Raspberry Pi Pico SDK

Arduino IDE's official support for RP2040-based microcontrollers, such as the Raspberry Pi Pico, is facilitated through Mbed OS. Mbed OS supplies a **real-time operating system (RTOS)** and a vast collection of handy routines for programming microcontroller peripherals, including the ADC. In addition to Mbed OS, Arduino exposes other APIs and **Software Development Kits (SDKs)** to the programmer, such as:

- **Arduino API:**

<https://www.arduino.cc/reference/en/>

- **Raspberry Pi Pico SDK:**

<https://www.raspberrypi.com/documentation/pico-sdk/>

The Raspberry Pi Pico SDK is the official software library for programming RP2040-based microcontrollers and offers tools, optimized libraries, and APIs tailored for the Pico's hardware capabilities. Therefore, the Raspberry Pi Pico SDK expands the options for developers working with RP2040-based microcontrollers. In the upcoming *How to do it...* section, we will familiarize ourselves with this library by programming the ADC peripheral.

How to do it...

Open the Arduino IDE and create a new sketch. Then, take the following steps to record 4-second-long songs with the Raspberry Pi Pico and transmit them over the serial connection.


```
{
    int32_t cur_idx = 0;
    bool    is_ready = false;
    int16_t data[AUDIO_LENGTH_SAMPLES];
};

volatile Buffer buffer;
```

The audio buffer (data) is encompassed within the Buffer struct, together with another two members, which are:

- `cur_idx`: An `int32_t` variable used to keep the current index of the buffer
- `is_ready`: A `bool` variable used to indicate whether the buffer is ready to use

The audio buffer is the `int16_t` type to accommodate the 12-bit values acquired with the ADC. With a target sample rate (`SAMPLE_RATE`) of 22,050 and an audio length (`AUDIO_LENGTH_SEC`) of 4 seconds, we require a buffer capable of holding 88,200 `int16_t` samples (`AUDIO_LENGTH_SAMPLES`). This buffer is kept in data memory and accounts for 66% of the total memory, as the maximum capacity is 264 KB. Therefore, it should be evident that a 4-second audio duration is close to the maximum limit we can accommodate.



The `volatile` keyword used in the code informs the compiler that the value of the buffer variable may change outside the normal flow of the program, as it happens with the ISR. This keyword should always be used when a variable can change in the interrupt handler to avoid the compiler removing this variable from the code and leading to incorrect behavior.

Step 5:

Declare a global `mbed::Ticker` object to trigger periodic interrupts:

```
mbed::Ticker timer;
```

The timer object will be used to fire the timer interrupts at the frequency of the audio sample rate (22,050 Hz).

Step 6:

Write the ISR to sample the amplified audio signal coming from the microphone:

```
#define BIAS_MIC    1552 // (1.25V * 4095) / 3.3

void timer_ISR() {
    if(buffer.cur_idx < AUDIO_LENGTH_SAMPLES) {
        int16_t v = (int16_t)((adc_read() - BIAS_MIC));
        int32_t ix_buffer = buffer.cur_idx;
        buffer.data[ix_buffer] = (int16_t)v;
        buffer.cur_idx++;
    }
    else {
        buffer.is_ready = true;
    }
}
```

The ISR samples the microphone's signal with the `adc_read()` function from the Raspberry Pi Pico SDK. As the signal produced by the MAX9814 amplifier carries a 1.25V bias, we must subtract the corresponding digital sample (`BIAS_MIC`) from the measurement. To determine the digital value that corresponds to the 1.25V, you can apply the following formula:

$$D_{sample} = \frac{2^N - 1}{V_{ref}} \cdot V_{sample}$$

From the previous formula:

- D_{sample} is the digital sample
- N is the ADC resolution (12)
- V_{sample} is the voltage sample
- V_{ref} is the ADC supply voltage reference (for example, `ADC_VREF`).

Therefore, we should subtract 1,552 from the measurement as we have a 12-bit ADC with a V_{ref} of 3.3V.

Once we have subtracted the bias from the measurement, we can store it in the audio buffer (`buffer.data[ix_buffer] = v`) and then increment the buffer index (`buffer.cur_idx++`). When the buffer is ready, the ISR sets the `is_ready` flag to true (`buffer.is_ready = true`).

Step 7:

In the `setup()` function, initialize the serial peripheral and set the button mode to `PullUp`:

```
Serial.begin(115200);  
while(!Serial);  
button.mode(PullUp);
```

Since the button is connected to **GND** and the **GPIO** pin, we must enable the internal pull-up resistor to prevent the floating state. Therefore, the numerical value returned by `mbed::DigitalIn` will be 0 when the button is pressed.

Step 8:

In the `setup()` function, use the Raspberry Pi Pico SDK to initialize the ADC peripheral:

```
adc_init();  
adc_gpio_init(26);  
adc_select_input(0);
```

The ADC peripheral is initialized by calling the following functions:

1. `adc_init()` to initialize the ADC peripheral.
2. `adc_gpio_init(26)` to initialize the GPIO used by the ADC. This function needs the **GPIO** pin number attached to the ADC peripheral. Therefore, we pass 26 because ADC0 is attached to GP26.
3. `adc_select_input(0)` to initialize the ADC input. The ADC input is the reference number of the ADC attached to the selected GPIO. Therefore, we pass 0 because we use ADC0.

By calling the preceding three functions, we configure the ADC peripheral in one-shot mode.

Step 9:

In the `loop()` function, check whether the button is pressed:

```
if(button == PRESSED)  
{
```

If so, wait for almost a second (for example, 800 ms) to avoid recording the mechanical sound of the pressed button:

```
delay(800);
```


Then, reset the buffer and turn the built-in LED on to inform the user that the recording is about to begin:

```
buffer.cur_idx = 0;
buffer.is_ready = false;
led_builtin = ON;
```

After, initialize the timer object to fire the interrupts with a frequency of 22,050 Hz:

```
uint32_t sr_us = 1000000 / SAMPLE_RATE;
timer.attach_us(&timer_ISR, sr_us);
```

The timer object features the `attach_us()` method for initializing both the ISR to be invoked (`timer_ISR`) and the interval time in microseconds (`sr_us`) at which it should be called.

Once the timer interrupt has been launched, wait for the audio buffer to be ready:

```
while(!buffer.is_ready);
```

When the buffer is ready, stop the timer by detaching the ISR, turn the built-in LED off, and transmit all audio samples over the serial line by line:

```
timer.detach();
led_builtin = OFF;
Serial.println(SAMPLE_RATE);
Serial.println(AUDIO_LENGTH_SAMPLES);
for(int i = 0; i < AUDIO_LENGTH_SAMPLES; ++i) {
    Serial.println(buffer.data[i]);
}
```

In the previous code, the `Serial.println()` function from the Arduino API is used to transmit the sample rate (`SAMPLE_RATE`), the audio length (`AUDIO_LENGTH_SAMPLES`), and samples (`buffer.data[i]`) through serial communication.



The sample rate and the audio length will be essential in the following recipe to generate audio files from the transmitted values over the serial.

Now, compile and upload the sketch on the Raspberry Pi Pico. Once the device is ready, open the serial terminal, play music close to the microphone, and press the push-button.

After 4 seconds, you should see a sequence of integer numbers printed on the serial terminal. These numbers represent the audio samples acquired with the ADC peripheral in digital format. Speaking in front of the microphone should result in a variation of the numbers displayed. If this occurs, it is an encouraging sign that the Raspberry Pi Pico can effectively capture audio clips.

There's more...with the SparkFun Artemis Nano!

In this recipe, we learned how to record audio clips with the ADC peripheral and timer interrupts using the Raspberry Pi Pico SDK and Mbed OS API.

If you are using the SparkFun Artemis Nano, you can refer to the pre-built example (Example1_MicrophoneOutput) available in the Arduino IDE, which shows how to use the built-in PDM microphone:

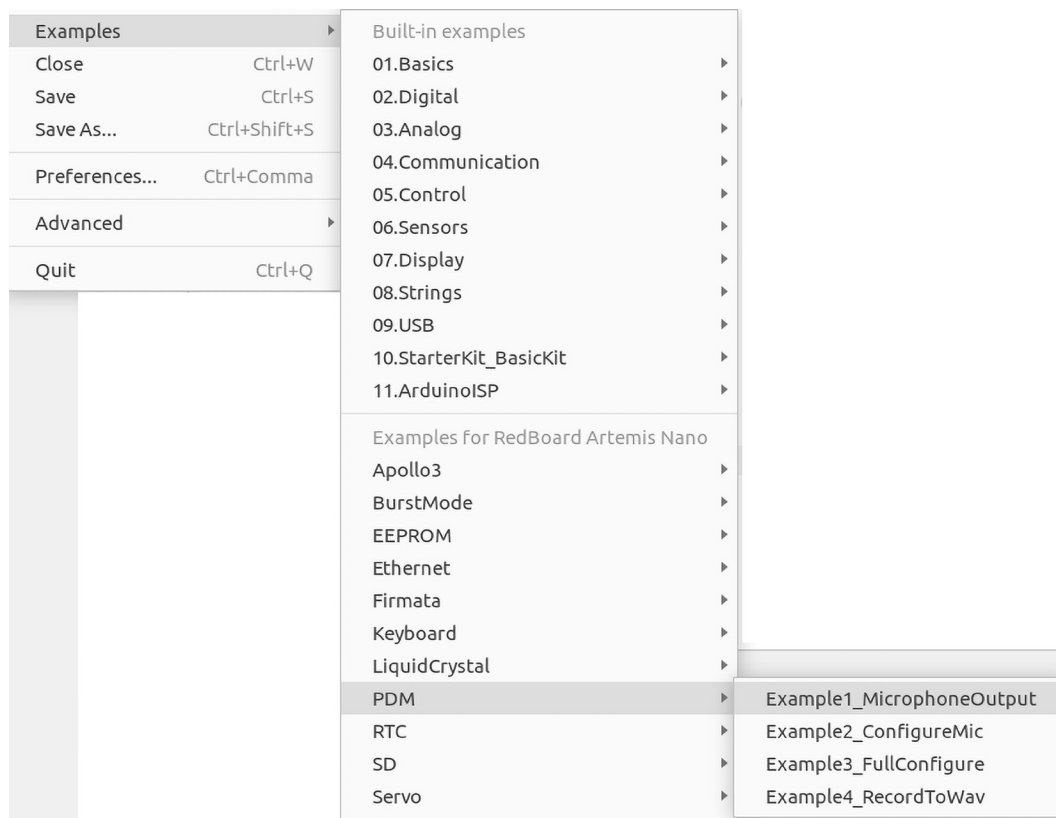


Figure 5.6: Pre-built example in the Arduino IDE to record audio with SparkFun Artemis Nano's microphone

At this point, you might wonder, how do we know that we recorded the audio clip accurately?

In the upcoming recipe, we will implement a Python script to convert the samples transmitted over the serial connection into audio files we can reproduce on our computer.

Generating audio files from samples transmitted over the serial

If the previous recipe taught us the method for recording audio using a microcontroller, our next objective is to create audio files we can reproduce on our computer.

In this recipe, we will develop a Python script locally to generate audio files in .wav format from the audio samples transmitted over the serial.

Getting ready

In this recipe, we will develop a Python script locally to create audio files from the data transmitted over the serial. To facilitate this task, we will need two main Python libraries:

- **pySerial**, which allows us to read the data transmitted serially
- **soundfile** (<https://pysoundfile.readthedocs.io/>), which enables us to read or write audio files

The soundfile library can be installed with the following pip command:

```
$ pip install soundfile
```

Using the `write()` method provided by this library, we can generate audio files in various formats from NumPy arrays.

How to do it...

On your computer, create a new Python script called `parse_audio_samples.py` and use the following steps to generate audio .wav files from the transmitted samples over the serial connection.

Step 1:

Import the necessary Python libraries:

```
import serial
import soundfile as sf
import numpy as np
```

Step 2:

Initialize **pySerial** with the port and baud rate used by the microcontroller:

```
import serial
ser = serial.Serial()
ser.port = '/dev/ttyACM0'
ser.baudrate = 115200
```

The baud rate is 115200 and matches the baud rate initialized with the Raspberry Pi Pico.

The easiest way to determine the serial port name is from the device drop-down menu in the Arduino IDE:

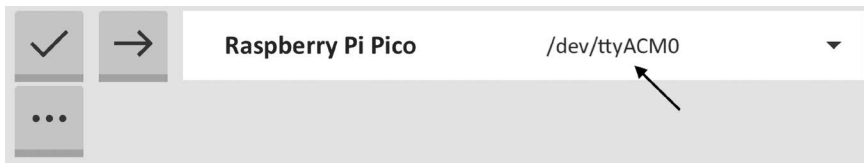


Figure 5.7: The serial port name is reported in the Arduino IDE

After, open the serial port and discard the content in the input buffer:

```
ser.open()
ser.reset_input_buffer()
```

Once the input buffer is cleared, the program can start receiving the data transmitted over the serial connection.

Step 3:

Define a **while** loop that runs indefinitely until the program ends. Within the loop, read the sample rate, audio length, and samples transmitted over the serial:

```
def serial_readline(obj):
    data = obj.readline()
    return data.decode("utf-8")

while True:
    ad_sr_str = serial_readline(ser)
    ad_len_str = serial_readline(ser)
    ad_sr = int(ad_sr_str)
```

```
ad_len = int(ad_len_str)
ad_buf = np.empty((ad_len), dtype=np.int16)
for i in range(0, ad_len):
    sample_str = serial_readline(ser)
    ad_buf[i] = int(sample_str)
```

The first two reads with the `serial_readline(ser)` function are necessary to know the sample rate (`ad_sr`) and the number of audio samples (`ad_len`) of the recorded song. After acquiring this information, the program enters a for loop to parse the transmitted audio samples and store them in the audio buffer (`ad_buf`).

Step 4:

Create a `.wav` audio file called `test.wav` from the `ad_buf` NumPy array:

```
sf.write("test.wav", ad_buf, ad_sr, subtype='PCM_16')
```

The audio file is generated by calling the `write()` method from the soundfile library, which only requires the following input arguments:

- The input NumPy array to convert to audio (`ad_buf`).
- The sample rate (`ad_sr`).
- The audio format (subtype). Since our samples are in 16-bit format, the subtype is set to `'PCM_16'`.

Now, we are ready to verify whether the Raspberry Pi Pico recorded the audio clip properly. To do so, ensure the Arduino serial monitor is closed because the serial peripheral on your computer can only communicate with one application. Then, make sure the Raspberry Pi Pico is connected to your computer and run the Python script:

```
$ python parse_audio_samples.py
```

Now, press the push-button and speak in front of the microphone. The Raspberry Pi Pico will turn the built-in LED on for 4 seconds to signify the recording. After the recording, the Python script will parse the data transmitted over the serial and create the audio file `test.wav`. Reproduce this file with your favorite media player and check whether the audio has been recorded correctly. If so, you are ready for the following recipe, where you will collect audio samples for building the training dataset.

There's more...

In this recipe, we learned how to use the soundfile library to create audio files in Python from NumPy arrays.

Although we proposed to play back the file with a media player, you could do it directly in Python, for example, using the **playsound** (<https://pypi.org/project/playsound/>) module.

The playsound library can be installed with the following pip command:

```
$ pip install playsound
```

This module only contains the function to play the audio, which has the same name as the module: `playsound()`.

Now that we can record audio clips with the Raspberry Pi Pico, we are ready to prepare the training dataset.

In the upcoming recipe, we will expand the Python script to collect audio clips for building a dataset for music genre classification in Google Drive.

Building the dataset for classifying music genres

Having been able to record audio with the Raspberry Pi Pico, we are now set to build the dataset for classifying music genres.

This recipe will walk you through collecting training samples from two sources: audio clips from the GTZAN dataset and audio recordings captured with the Raspberry Pi Pico. The collected audio clips will then be uploaded to Google Drive, ensuring easy access from Google Colab during the ML model preparation phase.

Getting ready

The dataset we need for training our model requires a substantial number of training samples for each music genre. Typically, a minimum of 100 samples per genre is recommended to yield better accuracy results.

However, the number of training samples is not the only factor to consider. In fact, the dataset must encompass a broad spectrum of songs, capturing the diverse stylistic variations within each genre.

As you might guess, collecting such a vast number of audio clips is time-consuming. Therefore, how can we make this process less tedious? Our proposal is to take the training samples from two sources, which are the following:

- Audio clips from the GTZAN dataset
- Audio recordings captured with the Raspberry Pi Pico

The following subsection will provide more details about these two sources.

Using the GTZAN dataset for music genre classification

The GTZAN dataset is one of the most used public datasets for music genre recognition.

There are ten music genres represented in the dataset: blues, classical, country, disco, hip hop, jazz, metal, pop, reggae, and rock. The dataset is well balanced as each genre is represented by precisely 100 audio clips that are 30 seconds long.

Every audio track in the dataset is stored in the .wav format and has the following:

- Sample rate of 22,050 Hz
- Mono channel
- 16-bit integer samples

Therefore, these audio clips match the format we intend to use in our application.

Since, for this project, we only need to recognize the disco, jazz, and metal music genres, we can import onto Google Drive only the training samples of interest to save drive space.

Augmenting the dataset with audio samples taken with the Raspberry Pi Pico

The recordings taken with the Raspberry Pi Pico ensure our dataset includes audio samples captured with the same microphone we plan to deploy in production. However, gathering a large volume of recordings might be challenging since we need to acquire the data manually. Nonetheless, it is advisable to select a minimum of 5 songs for each music genre and to record 10 clips, each 4 seconds long, from each track. This approach will provide a diverse range of samples while being manageable to collect.



You might consider playing the songs hosted on **Pixabay** (<https://pixabay.com/>) as they are royalty-free and suitable for commercial purposes.

You might have observed that the length of the audio clips from GTZAN and the ones captured with the Raspberry Pi Pico do not match. So, before diving into dataset preparation, we must decide on the appropriate audio length for the ML model's input. What should that be?

Choosing the suitable model input length

Microcontrollers have limited memory, and it can be challenging to work with long-duration audio. Given that the Raspberry Pi Pico has 264 KB of SRAM, it is impractical to consider both 30-second and 4-second audio inputs. In fact, the former requires over 1 MByte of memory, while the latter would demand 176.4 KB.

As a result, we need to consider shorter audio clips. In our project, the objective will be to recognize the music genre by analyzing just 1 second of audio data. This choice will help reduce the memory required to 44.1 KB when the samples are 16-bit integer values sampled with a frequency of 22,050 Hz.

How to do it...

Let's start by downloading the GTZAN dataset. You can download the dataset from Kaggle: <https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification>.

Once you have downloaded the file, unzip the dataset. Within the unzipped folder, you should have the `genres_original/` directory. This folder contains the audio clips for each of the ten music genres, as shown in the following screenshot:

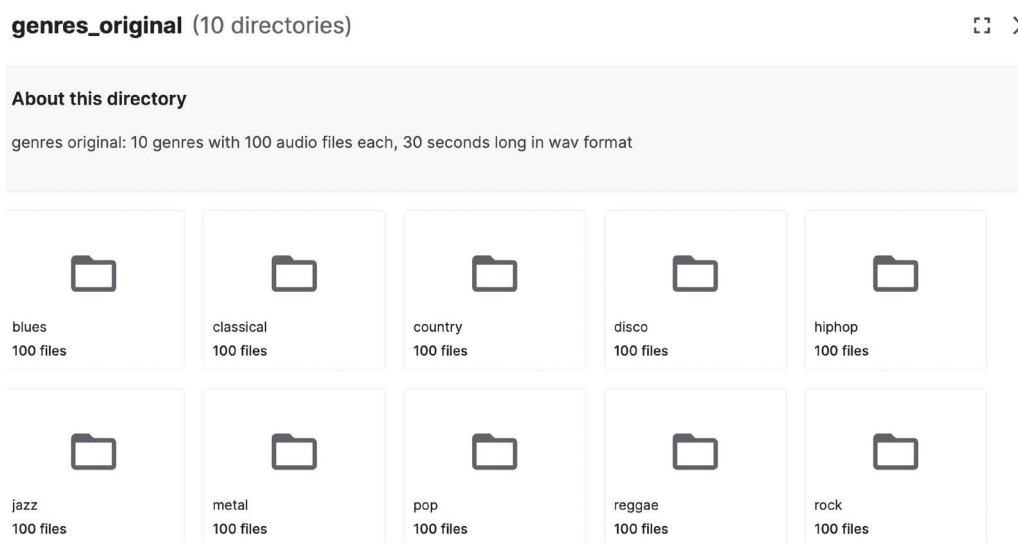


Figure 5.8: Subfolders contained in the `genres_original/` directory

After, upload the disco, jazz, and metal folders contained in the `genres_original` directory to Google Drive. To do so, create a new folder called `mgr_dataset` in Google Drive and drag the preceding three folders inside. Within your `mgr_dataset` folder, you should have the following three directories containing the GTZAN audio clips:

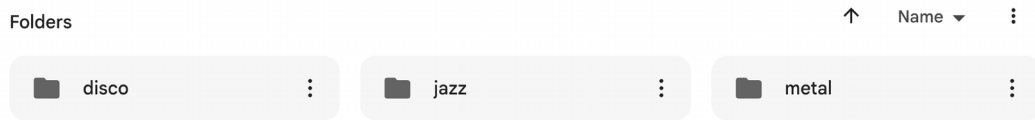


Figure 5.9: Folder structure in the `mgr_dataset` folder in Google Drive

Take note of the Google Drive file ID of the preceding three directories (`disco`, `jazz`, and `metal`), which is the last part of the **Uniform Resource Locator (URL)** of the directory when viewed in the web browser. The Google Drive ID is required for uploading the files to Google Drive with the PyDrive library.

On your computer, copy the Python script implemented in the previous recipe (`parse_audio_samples.py`) in a new file called `build_dataset.py`. Open the `build_dataset.py` file with your favorite local Python editor and remove the line where we create the `test.wav` audio file (`sf.write()`).

Now, take the following steps to adjust the Python script to upload the audio clips taken with the Raspberry Pi Pico to Google Drive.

Step 1:

Import the **UUID** Python module to produce unique filenames for the `.wav` files:

```
import uuid
```

Then, declare two variables to hold the label's name and the ID of the destination folder in Google Drive:

```
label = ""  
gdrive_id = ""
```

The `label` variable will be the prefix for the filename of the `.wav` files, while the `gdrive_id` one will hold the ID of the destination folder in Google Drive. Both variables will be assigned during the audio acquisition process.

Step 2:

Ensure you have the **PyDrive** library pip installed in your local Python development environment (`pip install pydrive`) and grant permission for your account to use the Google Drive API:

```
from pydrive.auth import GoogleAuth
gauth = GoogleAuth()
gauth.LocalWebserverAuth()
```

The preceding code will open a window in the web browser to request access to your Google Drive.



To learn more about the PyDrive library, you can refer to *Chapter 2, Unleashing Your Creativity with Microcontrollers*.

Then, use the `gauth` (**GoogleAuth**) object to create a local instance of Google Drive:

```
from pydrive.drive import GoogleDrive
drive = GoogleDrive(gauth)
```

The PyDrive library requires this **OAuth** JSON file to access Google Drive, which must be located in the same directory as your Python script and renamed `client_secrets.json`.

Step 3:

After receiving the whole audio sample over the serial, use the Python `input()` function to ask whether the audio clip can be saved. If the user types `y` from the keyboard, ask for the label's name:

```
key = input("Save audio? [y] for YES: ")
if key == 'y':
    str_label = ""
    while str_label == "":
        str_label = input("Provide the label's name or leave it empty to use\n[{}]: ".format(label))
    label_new = input(str_label)
    if label_new != '':
        label = label_new
```

If the user does not provide the label's names, the program will not modify the content of the `label` variable.

Then, save the audio clip as a .WAV file:

```
unique_id = str(uuid.uuid4())
filename = label + "_" + unique_id + ".WAV"
sf.write(filename, ad_buf, ad_sr,
         subtype='PCM_16')
```

Step 5:

Use the Python `input()` function to ask for the ID of the Google Drive destination folder:

```
str_gid = """Provide the Google Drive folder ID
or leave it empty to use
[{}]:""".format(gdrive_id)
gdrive_id_new = input(str_gid)
if gdrive_id_new != '':
    gdrive_id = gdrive_id_new
```

If the user does not provide the Google Drive destination folder ID, the program will not modify the content of the `gdrive_id` variable.

Then, upload the file to Google Drive:

```
gfile = drive.CreateFile({'parents': [{'id': gdrive_id}]})
gfile.SetContentFile(filename)
gfile.Upload()
```

Step 6:

Ensure the Raspberry Pi Pico is connected to your computer through the micro-USB data cable. Then, run the `build_dataset.py` Python script to acquire the audio clips with the microphone connected to the microcontroller:

```
$ python build_dataset.py
```

After, take your smartphone and open the web browser. There, go to <https://pixabay.com/>, tap on the drop-down menu in the search bar, and select the **Music** option.

In the search bar, enter `disco` to list all the disco songs hosted on Pixabay. Then, place your smartphone near the microphone and play your chosen track.

Press the push-button on the breadboard to start the audio recording for 4 seconds. After the recording, the Python script will ask you to provide the label (disco, in this case) and the Google Drive ID of the destination folder in Google Drive.



After the initial recording, it is unnecessary to input the label and Google Drive ID again, as the Python script will reuse the previous entries.

From the same song, try to acquire at least ten audio clips. Then, carry out the same procedure for another four disco songs.

After gathering samples for the disco class, follow the same steps for the jazz and metal music genres.

The dataset is set up at this point, and we can move forward to analyzing the feature extraction method that will help us classify the music genres: the MFCCs.

There's more...with the SparkFun Artemis Nano!

In this recipe, we learned how to prepare a dataset for classifying music genres using the GTZAN dataset and audio clips recorded with the Raspberry Pi Pico.

For those interested in deploying the application on the SparkFun Artemis Nano, we recommend augmenting the dataset with audio samples recorded on this platform. The reason is that the SparkFun Artemis Nano's microphone has different specifications, sensitivities, and frequency responses than the Raspberry Pi Pico's microphone counterpart. Therefore, by including audio samples from both devices, we enhance the likelihood of training a model that can effectively classify music genres on the Raspberry Pi Pico and SparkFun Artemis Nano.

Now that the dataset is prepared, the remaining focus of this chapter's first part will be on the feature extraction technique we intend to employ for music genre classification.

In the upcoming recipe, we will delve into how you can extract the MFCCs from the audio samples.

Extracting MFCCs from audio samples with TensorFlow

Acoustic models heavily rely on hand-crafted engineering input features to achieve high accuracy. **Mel Frequency Cepstral Coefficients (MFCCs)** are extensively utilized in audio applications and have demonstrated remarkable success in various use cases, including music genre classification.

In this recipe, we will show you how to extract MFCCs in Python using the **TensorFlow signal processing functions** (https://www.tensorflow.org/versions/r2.11/api_docs/python/tf/signal):

Getting ready

The primary goal of MFCCs is to combine the temporal information and spectral characteristics of the audio signal in a very compact manner.

In *Chapter 4, Using Edge Impulse and Arduino Nano to Control LEDs with Voice Commands*, we gave a high-level summary of this feature extraction method. Here, in this chapter, we will delve deeper into its underlying compute blocks for implementing it with the TensorFlow signal processing functions.

The computational blocks necessary to extract the MFCCs from the input signal are illustrated in *Figure 5.10*:

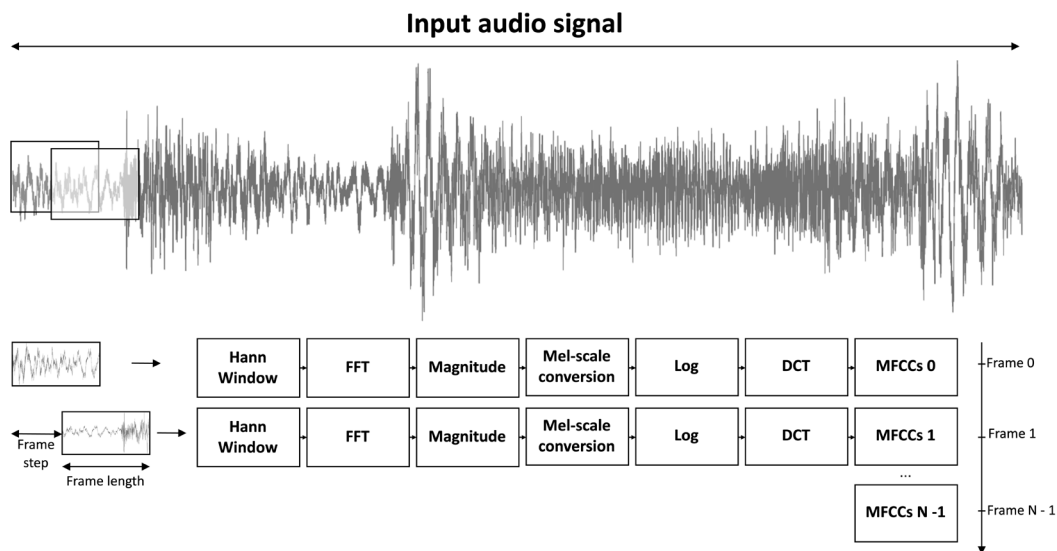


Figure 5.10: MFCCs computation

As you can see from the preceding image, the MFCCs algorithm splits the audio sample into overlapped frames and, for each one, it performs the following sequential operations:

1. **Hann** windowing
2. **Fast Fourier Transform (FFT)** calculation

3. Magnitude calculation
4. **Mel-scale** transformation
5. **Logarithmic (Log)** function
6. **Discrete Cosine Transform (DCT)** computation

Similar to many other feature extraction methods, the extracted features by MFCCs depend on some hyperparameters, which are:

- **Frame length:** This is the duration of each frame extracted from the audio signal.
- **Frame step:** This is the distance between two consecutive frames
- **FFT length:** This is the number of samples used in the FFT to compute the spectrum. Typically, the FFT length is the same as the frame length.
- **Mel-spectrogram attributes:** These are the necessary attributes to compute the Mel-spectrogram, such as the minimum and maximum frequency to represent and the number of output Mel frequencies.
- **Number of MFCCs to extract:** This is the number of DCT coefficients to keep after the DCT computation.

To determine the appropriate value for the hyperparameters, we need to get familiar with all the operations performed by MFCCs. As such, the following subsections will assist in comprehending these operations in detail.

Applying the Hann window

The first operation performed by MFCCs is typically **Hann windowing**, a filter applied in the time domain to reduce inaccuracies in the calculation of the **Fourier transform**. If we were signal processing experts, we would say *windowing is necessary to minimize spectral leakage*. Without this filtering, the Fourier transform may be inaccurate because it assumes that the input signal is periodic and extends to infinity.

To apply the Hann windowing filter, we need to multiply the input frame element-wise with the Hann window coefficients, which are determined through the following equation:

$$W_{Hann}(i) = 0.5 \cdot \left(1 - \cos\left(\frac{2 \cdot \pi \cdot i}{N - 1}\right)\right)$$

Where:

- i is the sample index in the frame

- $W_{Hann}(i)$ is the Hann coefficient for the sample index in the frame
- N is the frame length



The TensorFlow signal processing function responsible for calculating the Hann window coefficients is `tf.signal.hann_window` (https://www.tensorflow.org/versions/r2.11/api_docs/python/tf/signal/hann_window).

The following distribution shows that the Hann window function has a maximum value of 1 at the center of the frame, while the minimum value of 0 is given at both ends:

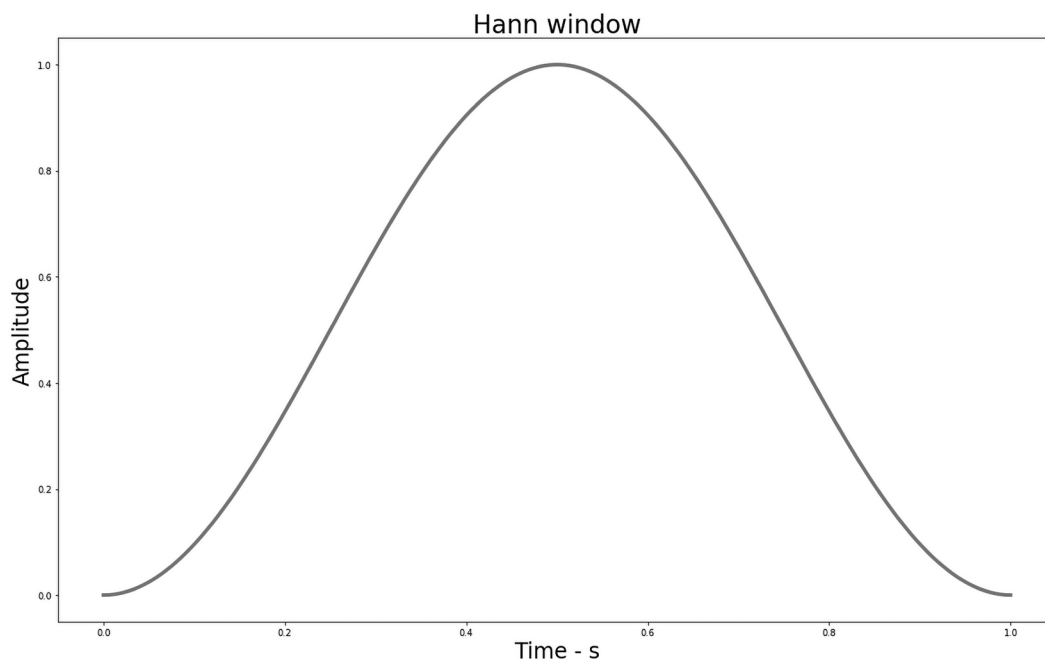


Figure 5.11: Hann window function

The resulting windowed signal is then processed using the **Real Fast Fourier Transform (RFFT)** to extract the frequency components.

Leveraging RFFT

The **R** prefix stands for a **real** input signal. This prefix distinguishes it from the standard **FFT** applied to complex inputs.

Complex numbers are represented using the following notation:

$$z = re + j \cdot im$$

Where:

- z is the complex number
- re is the real part
- im is the imaginary part

Mathematically, there is no difference between the RFFT and the FFT with the imaginary component set to 0. Both functions produce as many frequencies as the number of input samples as complex values.



FFT generates positive and negative frequencies, meaning the number of distinct frequencies is half the frame length.

So, why do we use RFFT?

RFFT is more computationally advantageous than FFT because it can use fewer arithmetic operations by exploiting the symmetry of the output spectrum. In fact, as shown in the following figure, the negative frequencies of the FFT can be obtained by **complex conjugating** the positive ones:

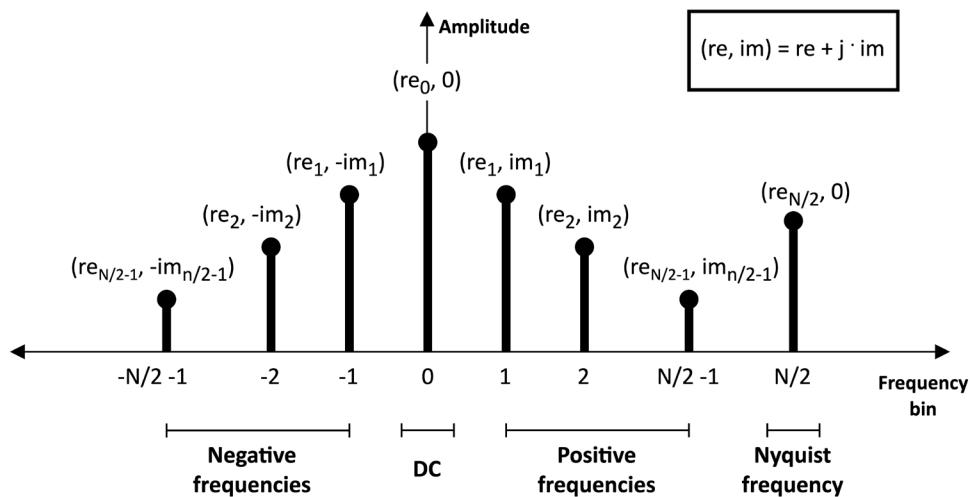


Figure 5.12: The FFT on a real signal produces negative frequencies symmetric to the positive ones



If a complex value a is the complex conjugate of b , it means that both the real and imaginary components of a and b are the same. However, the imaginary part of a has the opposite sign of the imaginary part of b .

As you can observe from the values of the output spectrum, the component in the middle, corresponding to 0 Hz, is unique and always has an imaginary part equal to 0. This component is called **DC** and *represents the average value of the input signal*.



Pay attention to the preceding output spectrum. The DC component is not the only frequency with an imaginary part equal to 0. The last component, located at $N/2$, also has an imaginary part equal to 0. This frequency is known as the **Nyquist frequency**, which corresponds to the *highest frequency of the spectrum (sample rate / 2)*.

The distance between two consecutive frequencies in the FFT output is known as **frequency resolution**, and it depends upon the input frame length: *the shorter the frame length, the lower the frequency resolution*. Therefore, since there is a trade-off between temporal and spectral resolution, the appropriate frame length choice depends on the specific target application.

For speech recognition tasks, a frame length between 20 ms and 30 ms is typically adequate to preserve temporal resolution and prevent missing utterances. On the other hand, for applications like ours, where fine-grained spectral details are crucial to capture harmonics and pitch, longer frame lengths ranging from 90 ms to 100 ms are generally needed.



In our case, we will consider 2,048 and 1,024 for the frame length and frame step. Therefore, for audio signals sampled at 22,050 Hz, the frame length and frame step correspond to 93 ms and 45.5 ms

Now that we know how the frame length and frequency resolution are related, we can continue with the next computation block that follows the RFFT: the magnitude calculation.

Calculating the FFT magnitude

The RFFT produces a vector of complex values representing the signal's frequency components. The magnitude of these components is of particular interest because it conveys information about the energy distribution, which tells us about their relevance in the representation of the input signal.

As we are working with complex numbers, the magnitude of each component is calculated by applying the following formula:

$$abs(x) = |x| = \sqrt{re^2 + im^2}$$

Where:

- $abs(x)$ is the absolute value of the component
- re and im are the real and imaginary parts of the component



The TensorFlow signal processing function responsible for computing the absolute value is `tf.math.abs` (https://www.tensorflow.org/versions/r2.11/api_docs/python/tf/math/abs).

Once we have calculated the magnitude of the FFT, the next step is to reduce the number of components using the **Mel scale**. The following subsection will discuss the process of converting the Hz frequencies to Mel.

The Mel scale conversion

In *Chapter 4, Using Edge Impulse and Arduino Nano to Control LEDs with Voice Commands*, we learned that our auditory system perceives frequency changes on a logarithmic scale rather than a linear one. Inspired by that, the Mel scale was introduced to apply a logarithmic transformation to the FFT frequencies.

To apply this frequency conversion, we must first define the following:

- **Fmin:** *This is the lower frequency of the Mel scale.* This frequency must be greater than or equal to the Mel frequency corresponding to 0 Hz.
- **Fmax:** *This is the higher frequency of the Mel scale.* This frequency must be less than or equal to the Mel frequency corresponding to the Nyquist frequency.
- **Number of Mel frequencies:** *This is the number of evenly spaced frequencies between Fmin and Fmax on the Mel scale.* Since these frequencies are equally distanced on the Mel scale, they are distinguishable by the human ear.

Due to the logarithmic nature of this conversion, we can use fewer Mel frequencies than FFT frequencies. The reason for that is that many lower Mel frequencies would correspond to the same Hz frequency. As a result of this, we can represent the spectrum with fewer components by converting the FFT representation to the Mel scale.

For many audio applications, the number of Mel frequencies tends to be between 20 and 40. In our case, we will consider 40 Mel frequencies to capture as much spectral information as possible.

Before we explain how to calculate the Mel frequencies, let us examine the following image, which illustrates the relationship between Mel frequencies, Mel frequencies converted to Hz, and FFT frequencies:

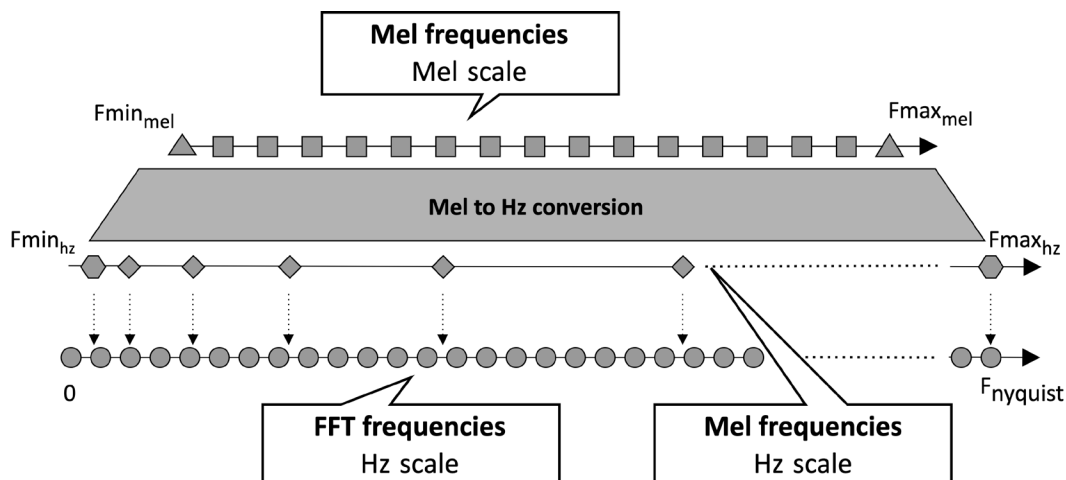


Figure 5.13: Relationship between Mel frequencies, Mel frequencies in Hz, and FFT frequencies

From the previous image, we can observe that:

- The Mel frequencies are evenly spaced in the Mel scale
- The Mel frequencies are fewer than the FFT frequencies
- The Mel frequencies converted to Hz are not evenly spaced
- The Mel frequencies converted to Hz do not map precisely to the FFT frequencies

Since the Mel frequencies do not correspond to FFT frequencies and are fewer in number, the question arises: *what is the relationship between the magnitude of a Mel frequency and the magnitude of FFT frequencies?*

The magnitude of each Mel frequency is obtained by applying a triangular filter in the Hz scale to capture the relevant FFT magnitudes that contribute to it.

These triangular filters, which are known as **triangular filter banks**, are for each Mel frequency and have been designed to:

- Be equally spaced on the Mel scale

- Have the apex of the triangle on each Mel frequency
- Have the two neighboring Mel frequencies that are adjacent to the center frequency of the filter as the vertices of the triangle's base

The following image shows how triangular filters on the Mel scale correspond to those on the Hz scale:

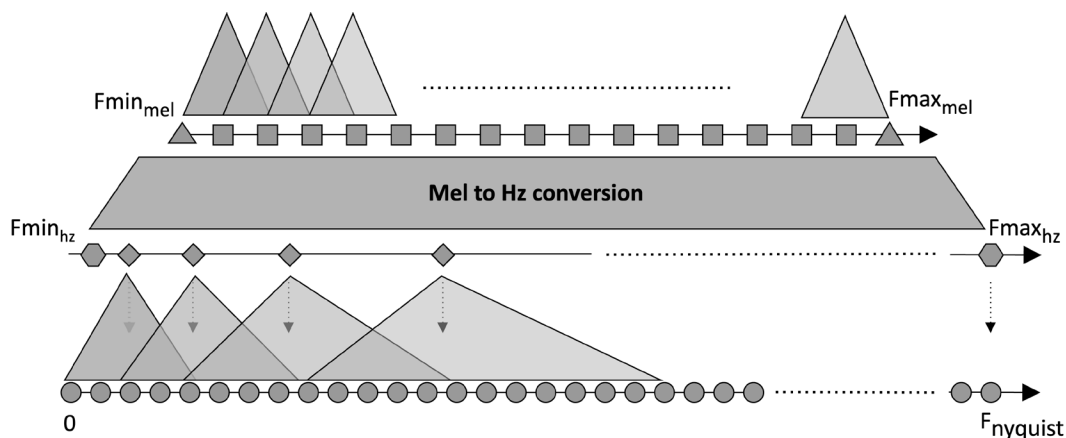


Figure 5.14: The triangular filters all have the same width on the Mel scale

By examining the preceding image, we can see that the filters on the Hz scale are narrow and more closely spaced at lower frequencies. As the frequencies increase, the filters become wider and more sparsely spaced. This indicates that more FFT frequencies contribute to the filter's energy at higher frequencies. In contrast, at lower frequencies, a smaller number of FFT frequencies do so.

The purpose of the filter is to assign a *weight* (*Mel weight*) to each FFT frequency. The triangle's apex corresponds to the highest *Mel weight*, set to 1. Instead, the base vertices of the triangle correspond to the lowest *Mel weight*, set to 0.



The TensorFlow signal processing function to calculate the *Mel weight* is `tf.signal.linear_to_mel_weight_matrix` (https://www.tensorflow.org/versions/r2.11/api_docs/python/tf/signal/linear_to_mel_weight_matrix).

After determining the *Mel weight* for each FFT frequency across all Mel frequencies, we can convert the FFT component to the Mel scale.

This operation is accomplished by carrying out a matrix multiplication between the FFT magnitude and the *Mel-weight* matrix, as depicted in the following image:

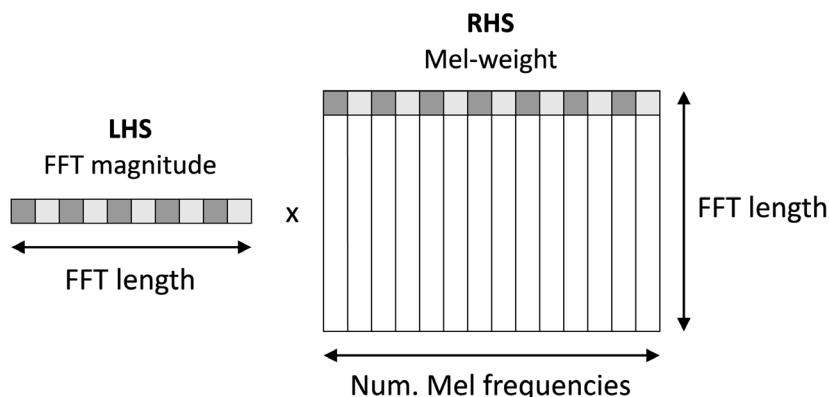


Figure 5.15: Mel scale conversion as a vector-by-matrix multiplication

In Figure 5.15:

- The **left-hand side (LHS)** operand is the FFT magnitude
- The **right-hand side (RHS)** operand is the matrix that keeps the *Mel weight* to assign to each FFT frequency for a given Mel component

After the Mel scale conversion, the next step is to apply the logarithmic function element-wise at each Mel component, which feeds the final stage of the MFCCs pipeline.



If we apply the Log-Mel conversion to a single input frame, we do not obtain the **Log-Mel spectrogram**. The reason is that this single conversion focuses on a particular moment in time, whereas a spectrogram depicts the signal's frequency variations over time. Therefore, the Log-Mel spectrogram is obtained only after applying the Log-Mel transformation across all input frames. In our situation, while storing the spectrogram could speed up the following operation in Python, it's not crucial. Our decision not to retain the spectrogram is strategic for minimizing memory usage, which is fundamental when deploying the algorithm on the microcontroller.

Computing the DCT coefficients

The DCT is used in the final stage of MFCCs computation to obtain less but highly unrelated information.



The TensorFlow signal processing function used to calculate the DCT after the Log-Mel conversion is `tf.signal.mfccs_from_log_mel_spectrograms` (https://www.tensorflow.org/versions/r2.11/api_docs/python/tf/signal/mfccs_from_log_mel_spectrograms).

MFCCs are the first few DCT coefficients that describe the spectrum's shape. The first DCT coefficient reflects the average power in the spectrum, while the higher-order coefficients provide more detailed spectral information, such as the pitch. Generally, 8–13 MFCCs are adequate for speech recognition tasks as they are insensitive to pitch variations. However, more coefficients may be required to capture pitch and tone nuances for applications like music genre recognition. In our case, *we will aim to extract 18 MFCCs from each frame*.

Having introduced all the necessary building blocks of the MFCCs, let's briefly analyze its memory requirements to see whether it is suitable for microcontroller deployment.

Evaluating the SRAM usage to run MFCCs

The MFCCs computation performs sequential operations that need to hold intermediate results in temporary buffers. Temporary buffers are kept in SRAM, which is usually very limited compared to the program memory.

Calculating the minimum SRAM utilization for a data pipeline like ours, where we have sequential operations with a single input and a single output, is relatively simple. In this scenario, *the minimum SRAM usage equals the two largest intermediate buffers the operations require*.

This result can be derived from the fact that only one buffer is required to hold the input, and another is needed to keep the output. Therefore, the intermediate stages of the pipeline can be computed circularly using these two buffers.

The following figure reports the memory requirement for each intermediate buffer required in the MFCCs computation:

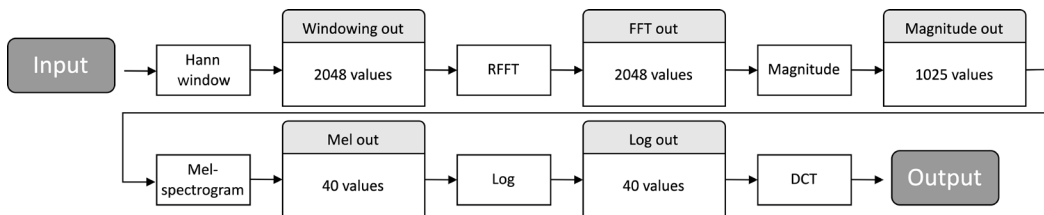


Figure 5.16: MFCCs data pipeline

By looking at the data pipeline, it should be evident that the Hann windowing and RFFT operations require the most significant intermediate buffers. Assuming that each value is stored in a 32-bit floating-point format, the total SRAM usage for these operations would be approximately 16 KB. As a result, this memory utilization is considerably smaller than the 264 KB of SRAM available on the Raspberry Pi Pico.

How to do it...

Open the web browser and create a new Colab notebook. Then, follow the steps to extract the MFCCs from a 1-second audio clip using the TensorFlow signal processing functions.

Step 1:

Define the MFCCs hyperparameters:

```
SAMPLE_RATE = 22050
FRAME_LENGTH = 2048
FRAME_STEP = 1024
FFT_LENGTH = 2048
FMIN_HZ = 20
FMAX_HZ = SAMPLE_RATE / 2
NUM_MEL_FREQS = 40
NUM_MFCCS = 18
```

The previous variables hold the MFCCs hyperparameters considered for this project, which have been discussed in the *Getting ready* section of this recipe.

Step 2:

Define the function interface to extract MFCCs from the input signal using the TensorFlow signal processing functions. This function should accept the input signal, sample rate, and MFCCs hyperparameters as input arguments:

```
import numpy as np
import tensorflow as tf

def extract_mfccs_tf(
    ad_src,
    ad_sample_rate,
    num_mfccs,
```

```
frame_length,  
frame_step,  
fft_length,  
fmin_hz,  
fmax_hz,  
num_mel_freqs):
```

The variable names for the input arguments should be self-explanatory to know the purpose of each one. Please note that we have imported the NumPy and TensorFlow libraries because they are required to perform the MFCCs feature extraction.

Step 2:

Within the `extract_mfccs_tf()` function, calculate the total number of frames that can be extracted from the input signal:

```
n = ad_src.shape[0]  
num_frames = int((n - frame_length) / frame_step)  
num_frames += int(1)
```

In the previous code snippet, we applied the formula reported in the TensorFlow `tf.signal.frame()` (https://www.tensorflow.org/versions/r2.11/api_docs/python/tf/signal/frame) function for the `pad_end = False` case, which only considers the frames that fit entirely within the input signals.

Step 3:

Within the `extract_mfccs_tf()` function, initialize the function's output with zeros:

```
output = np.zeros(shape=(num_frames, num_mfccs))
```

The preceding code creates a 2D NumPy array of zeros with `num_mfccs` columns and `num_frames` rows. The output array is used to store the MFCCs computed for each frame of the audio signal.



Theoretically, we could create a 2D NumPy array with `num_frames` columns and `num_mfccs` rows to store the MFCCs. However, as we will discover later in the chapter, our choice of array dimensions will be aligned with the expected input format of the model.

Step 4:

Within the `extract_mfccs_tf()` function, iterate over each frame and extract the corresponding MFCCs. To accomplish this task, write a for loop to iterate over each frame index:

```
for i in range(num_frames):
```

Then, extract the corresponding frame from the input signal:

```
    idx_s = i * frame_step
    idx_e = idx_s + frame_length
    src = ad_src[idx_s:idx_e]
```

In the previous code, the starting (`idx_s`) and ending (`idx_e`) indices of the frame to extract are computed based on the current frame index, `i`.

Once you have the frame, apply the Hann windowing:

```
    hann_coef = tf.signal.hann_window(frame_length)
    hann = src * hann_coef
```

The preceding code employs the TensorFlow `tf.signal.hann_window()` function to compute the Hann window coefficients (`hann_coef`), which are then multiplied element-wise with the input audio frame (`src`).

After the Hann windowing, calculate the FFT magnitude:

```
    fft_spect = tf.signal.rfft(hann)
    fft_mag_spect = tf.math.abs(fft_spect)
```

The TensorFlow `tf.signal.rfft()` function returns an output with half as many complex values as the generic FFT of the same size. Consequently, the `fft_spect` array contains frame length / 2 complex values.

The FFT magnitude is the input of the Mel scale conversion. This domain transformation is done via a vector-by-matrix multiplication between the FFT magnitude and the Mel weight matrix.

Therefore, generate the Mel weight matrix with the TensorFlow `tf.signal.linear_to_mel_weight_matrix()` function:

```
    num_fft_freqs = fft_mag_spect.shape[0]
    mel_weigh_mtx = tf.signal.linear_to_mel_weight_matrix(
        num_mel_freqs,
```

```
num_fft_freqs,  
ad_sample_rate,  
fmin_hz,  
fmax_hz)
```

The TensorFlow `tf.signal.linear_to_mel_weight_matrix()` function generates the Mel weight matrix, which has `num_mel_freqs` columns and `num_fft_freqs` rows.

Now, perform the vector-by-matrix multiplication between the FFT magnitude and the Mel weight matrix:

```
mel_spect = np.matmul(fft_mag_spect, mel_weigh_mtx)
```

Then, apply the logarithmic function on each Mel component:

```
log_mel_spect = np.log(mel_spect + 1e-6)
```

Finally, compute the DCT with the TensorFlow `tf.signal.mfccs_from_log_mel_spectrograms()` function and store the first `num_mfccs` coefficients in the output array. Then, return the output array to the user when the MFCCs features have been extracted from all input frames:

```
dct = tf.signal.mfccs_from_log_mel_spectrograms(  
    log_mel_spect)  
  
output[i] = dct[0:num_mfccs]  
return output
```

As you can see from the previous code snippet, the MFCCs features (`dct[0:num_mfccs]`) are stored at different rows in the output array.

Step 5:

Mount the top-level Google Drive directory:

```
from google.colab import drive  
drive.mount('/content/drive/')
```

The `mount()` function will open a window in the web browser to log in to your Google account. Once logged in, the root directory of Google Drive will be mounted in Colab, and you can access its contents.

Step 6:

Create a string variable and assign it the path to the Google Drive folder containing the disco, jazz, and metal subfolders:

```
train_dir = "drive/MyDrive/mgr_dataset"
```

Step 7:

Load an audio file from either the disco/, jazz/, or metal/ folder (for example, disco/disco.00002.wav) using the read() function provided by the soundfile library:

```
import soundfile as sf

filepath = train_dir + '/disco/disco.00002.wav'
ad, sr = sf.read(filepath)
```

The read() function takes the path to the audio file to open and returns the audio samples in a NumPy array (ad) along with the corresponding sample rate (sr). The audio samples are represented as floating-point values and normalized to the range of $[-1, 1)$.



The notation $[-1, 1)$ refers to the interval range of the audio samples. In this context, the values are greater than or equal to -1 and strictly less than 1.

It's worth noting that the input audio files from the dataset have 16-bit integer samples. Therefore, the soundfile library performs an internal normalization on the audio data, which is done by dividing the 16-bit integer values by 32,768 (2¹⁵). The 32,768 value corresponds to the maximum absolute amplitude value for a 16-bit integer. Since 16-bit integer values range from -32,768 to +32,767, dividing by 32,768 ensures that the resulting floating-point values fall within the range of $[-1, 1)$.

Step 8:

Extract a 1-second segment from the ad array and play the audio with IPython.display.Audio() from the IPython.display Python module:

```
SAMPLE_RATE = 22050
test_ad = ad[0:SAMPLE_RATE]
```

```
import IPython.display as ipd
ipd.Audio(test_ad, rate=sr)
```

In the preceding code, we extract the first second of the song in the `test_ad` array. This operation takes the first `SAMPLE_RATE` values, corresponding to the number of samples in 1 second.

Once the audio segment has been extracted, the `IPython.display.Audio()` function displays the following HTML5 audio widget, which allows you to reproduce the song directly within the Colab environment:



Figure 5.17: The HTML5 audio widget

The reason for extracting a 1-second audio clip is that our goal is to recognize the music genre just from 1 second of audio data. Therefore, the MFCCs should be derived from this 1-second snippet rather than the complete audio track.

Step 9:

Extract the MFCCs from the `test_ad` audio sample using the `extract_mfccs_tf()` function:

```
mfccs_tf = extract_mfccs_tf(
    test_ad,
    SAMPLE_RATE,
    NUM_MFCCS,
    FRAME_LENGTH,
    FRAME_STEP,
    FFT_LENGTH,
    FMIN_HZ,
    FMAX_HZ,
    NUM_MEL_FREQS)
```

Step 10:

Implement a function to visualize the MFCCs as an image:

```
import matplotlib.pyplot as plt
from matplotlib import cm
```

```
def display_mfccs(mfccs_src):
    fig, ax = plt.subplots()
    cax = ax.imshow(mfccs_src, interpolation='nearest', cmap=cm.gray,
                    origin='lower')
    ax.set_title('MFCCs')
    plt.xlabel('Frame index - Time')
    plt.ylabel('Coefficient index - Frequency')
    plt.colorbar(cax)
    plt.show()
```

Then, use the `display_mfccs()` function to display the `mfccs_tf` array as an image:

```
display_mfccs(mfccs_tf.T)
```

In the previous code snippet, the `mfccs_tf` array is transposed using the `.T` method solely to visualize the temporal information along the *x*-axis.

If you have used the `disco/disco.00002.wav` audio clip, the expected output should be as follows:

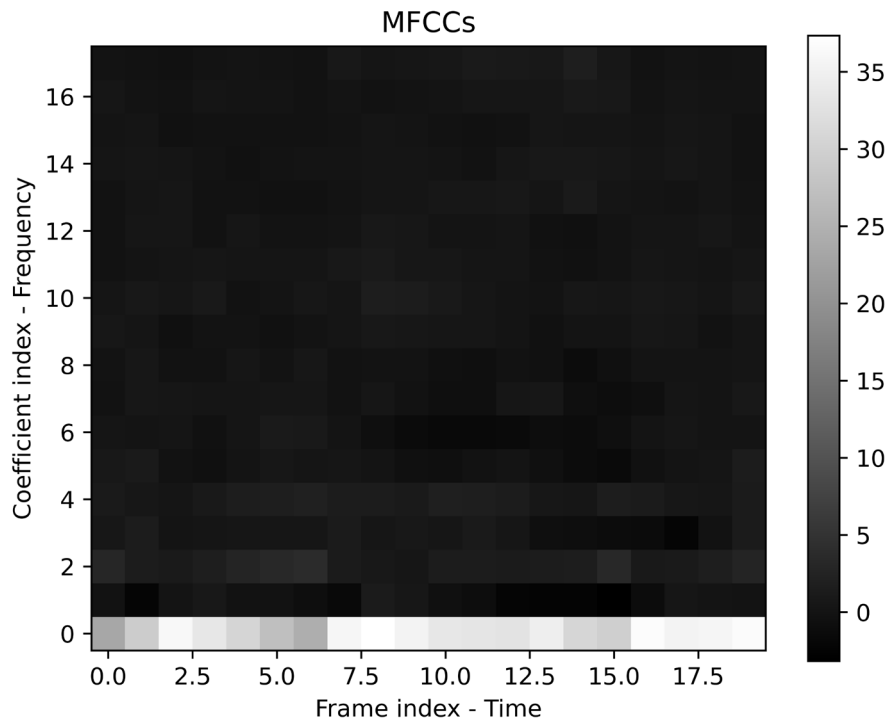


Figure 5.18: MFCCs visualization

If you look at the preceding image, you can notice that the color of the first row at the bottom is very different from the others because the values are much higher. This row corresponds to the 0th coefficient of MFCCs, representing the audio signal's overall energy. Unlike the other coefficients, the first one is influenced by the overall loudness of the audio signal. Therefore, these values are expected to be larger than the others.



The first MFCC does not carry relevant information to the overall shape of the spectrum. Consequently, most developers choose to eliminate this feature while training a model. However, in our case, we have decided to retain it. Therefore, feature scaling will be crucial to ensure this feature does not influence the ML model more.

There's more...

In this recipe, we learned about the core compute blocks behind the MFCCs feature extraction computation and how to leverage them using the TensorFlow signal processing functions.

TensorFlow is not the only option to extract MFCCs from an input signal, though.

For example, the **Librosa** (<https://librosa.org/doc/latest/index.html>) library provides the `librosa.feature.mfcc()` function (<https://librosa.org/doc/0.10.0/generated/librosa.feature.mfcc.html>), which can be used to perform the MFCCs feature extraction easily.

In the Colab file available in the `Chapter05_06` folder in the GitHub repository, we have included the code to demonstrate how to extract the MFCCs with Librosa from the `test_ad` variable containing the audio test sample. Upon displaying the resulting MFCCs, you can observe that the dimensions are transposed, and the resulting image's color scheme differs from the one obtained with TensorFlow. The reason for this difference is due to the underlying implementations.

Although we might have various Python libraries for extracting MFCCs features, we might not have the same options when deploying the model on the microcontroller. Therefore, when choosing the library for computing the MFCCs feature extraction, always consider how to leverage its computation on the target device because the same MFCCs algorithm implemented in Python should be deployed on the microcontroller to avoid accuracy loss.

Summary

In the first part of this chapter, we walked through the steps of recording audio clips using an external microphone with the Raspberry Pi Pico and analyzed the compute build blocks of the MFCCs feature extraction algorithm.

Our practical journey started by learning to connect the microphone to the Raspberry Pi Pico and record audio clips using the ADC peripheral and timer interrupts.

Then, we crafted a Python script to create audio files from the samples transmitted by the microcontroller over the serial connection. This script was then extended to upload the audio files to Google Drive, laying the foundation for building the training dataset. Given the large number of samples required for training the ML model, we collected the training data from the GTZAN dataset and audio recordings captured with the Raspberry Pi Pico. After the dataset preparation, we finally analyzed and implemented the MFCCs feature extraction using TensorFlow.

In the upcoming second part of this project, the focus will move toward the model design for efficient deployment on memory-constrained devices. Particularly, we will explore how to tailor the MFCCs algorithm for microcontrollers, design an LSTM model to classify music genres, and deploy the final application on the Raspberry Pi Pico!

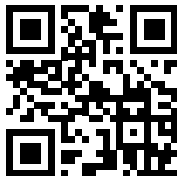
References

- Tzanetakis, G., Essl, G., & Cook, P. (2001). *Automatic musical genre classification of audio signals*. The International Society for Music Information Retrieval: <http://ismir2001.ismir.net/pdf/tzanetakis.pdf>

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



6

Recognizing Music Genres with TensorFlow and the Raspberry Pi Pico – Part 2

The first part of this project gave us the prerequisites to train a music genre recognition model. Now that we have obtained the dataset and have gotten acquainted with implementing the MFCCs feature extraction, we can delve into the model design and the application deployment. Although we might consider leaving the deployment for the end, it should never be the case when building tinyML applications. Given its limited computational and memory capabilities, the target device must always be at the center of our design choice, from the feature extraction to the model design.

For this reason, this second part will amply discuss how the target device influences the implementation of the MFCCs feature extraction.

We will start our discussion by tailoring the MFCCs implementation for the Raspberry Pi Pico. Here, we will learn how **fixed-point arithmetic** can help minimize the latency performance and show how the **CMSIS-DSP** library provides tremendous support in employing this limited numerical precision in feature extraction.

After reimplementing the algorithm to extract the MFCCs using fixed-point arithmetic, we will design an ML model capable of recognizing music genres with a **long short-term memory (LSTM) recurrent neural network (RNN)**.

Finally, we will test the model accuracy on the test dataset and deploy a music genre classification application on the Raspberry Pi Pico, with the help of **TensorFlow Lite for Microcontrollers (tflite-micro)**.

By the end of this second part, we will know how to leverage fixed-point arithmetic with the CMSIS-DSP library, design an LSTM model for music genre classification, and deploy the MFCCs algorithm and the TensorFlow Lite model on the microcontroller.

In this second part, we're going to cover the following recipes:

- Computing the FFT magnitude with fixed-point arithmetic using the CMSIS-DSP library
- Implementing the MFCCs feature extraction with the CMSIS-DSP library
- Designing and training an LSTM RNN model
- Evaluating the accuracy of the quantized model on the test dataset
- Deploying the MFCCs feature extraction algorithm on the Raspberry Pi Pico
- Recognizing music genres with the Raspberry Pi Pico

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- A Raspberry Pi Pico
- A SparkFun RedBoard Artemis Nano (optional)
- A micro-USB data cable
- A USB-C data cable (optional)
- 1 x electret microphone amplifier – MAX9814
- A 5-pin press-fit header strip (optional but recommended)
- 1 x half-size solderless breadboard
- 6 x jumper wires
- A laptop/PC with either Linux, macOS, or Windows
- A Google Drive account



The source code and additional material are available in the Chapter05_06 folder on the GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Chapter05_06.

Computing the FFT magnitude with fixed-point arithmetic using the CMSIS-DSP library

In the previous chapter, we discovered that the Raspberry Pi Pico has enough memory to handle the data pipeline to extract the MFCCs, using floating-point arithmetic. However, this data format does not offer the best computational efficiency for our desired target platform.

In this recipe, we will uncover why floating-point arithmetic is inefficient on the Raspberry Pi Pico and propose the **16-bit fixed-point (Q15)** arithmetic as a more practical alternative. To provide a hands-on understanding of Q15, we will guide you through calculating the FFT magnitude, using this data type with the **CMSIS-DSP** Python library in the Colab notebook. This approach will simplify the transition of the code to the Raspberry Pi Pico in subsequent chapters.

Getting ready

In the previous recipe, we learned how to extract MFCCs from an audio sample using TensorFlow. Nevertheless, it is necessary to consider whether this implementation is efficient for our device. To make this assessment, *we must have a good understanding of the microcontroller's hardware capabilities*. Therefore, the subsequent subsection will take a closer look at the computational resources of the Raspberry Pi Pico.

Evaluating the hardware capabilities of the Raspberry Pi Pico

The Raspberry Pi Pico features a low-power microcontroller, with a dual-core **Arm Cortex-M0+** CPU clocked at 133 MHz. As a low-power device, the processor has limited computational resources compared to more powerful CPUs. For example, the CPU does not have floating-point hardware acceleration like many other microcontrollers. Therefore, floating-point operations are all software-emulated and can take considerably longer than integer ones. As such, *we should consider using integer arithmetic as much as possible to improve the algorithm's speed*.

This recipe aims to show how you can use the **16-bit integer fixed-point** arithmetic to improve the performance of one of the most compute-intensive parts of the MFCCs pipeline: the calculation of the FFT magnitude.

However, besides knowing what we need to do, it is also fundamental to understand how to do it. Hence, the subsequent subsection will introduce the **CMSIS-DSP** library. This software library will be essential to optimize the MFCCs computation easily for the Raspberry Pi Pico or any other Arm-based microcontrollers.

Using the CMSIS-DSP Python library

Developing an algorithm from scratch and optimizing it for a specific platform can be tedious and challenging. It requires a deep understanding of device computing capabilities and optimization techniques we may be unfamiliar with. The implementation becomes even more complicated when we adopt data formats that differ numerically from the floating-point reference implementation, such as fixed-point ones. In this scenario, it is also necessary to develop the Python implementation that matches the numerical behavior of the algorithm deployed on the platform to avoid any potential accuracy losses.

When targeting Arm-based microcontrollers, such as the Raspberry Pi Pico, Arduino Nano, or SparkFun Artemis Nano, implementing an algorithm can be made simple and efficient using the C functions available in the **CMSIS-DSP** library (<https://github.com/ARM-software/CMSIS-DSP>).

The CMSIS-DSP library provides optimized functions for various applications and **Arm Cortex-M** CPUs, *ensuring portability and high performance across different platforms*. Some examples of these routines include:

- *Math functions* to perform arithmetic operations, such as addition and multiplication between two memory buffers
- *Matrix functions* to perform operations like matrix-by-matrix multiplication or vector-by-matrix multiplication
- *Transform functions* to perform operations like the **Fast Fourier Transform (FFT)** or **Real Fast Fourier Transform (RFFT)**



For the complete list of functions, you can consult the official CMSIS-DSP documentation: <https://www.keil.com/pack/doc/CMSIS/DSP/html/index.html>.

The CMSIS-DSP library is also available as a Python library, enabling us to test and develop an algorithm in a Python environment that closely resembles the final implementation on the microcontroller.

The Python library is known as `cmsisdsp` (<https://pypi.org/project/cmsisdsp>) and can be installed by using the following pip Python package manager command:

```
$ pip install numpy==1.23.5
$ pip install cmsisdsp==1.9.6
```

In this project, we will use the `cmsisdsp` 1.9.6 release, which depends on the NumPy 1.23.5 release. The CMSIS-DSP library provides tremendous assistance when writing an algorithm that utilizes numerical formats different from floating-point arithmetic. In fact, many of the routines provided in the CMSIS-DSP library can be accelerated through the integer fixed-point arithmetic, improving the performance of algorithms on CPUs without floating-point hardware acceleration.

Therefore, if we intend to use the fixed-point numerical format, this library will allow us to write the algorithm in Python using the same numerical precision as the microcontroller’s implementation.

The following subsection will provide further details about the lower numerical precision format we intend to use in this recipe: the 16-bit fixed-point format.

Representing numbers in 16-bit fixed-point format

Many functions in the CMSIS-DSP library support different numerical formats, such as 32-bit floating-point (F32), 16-bit integer fixed-point (Q15), and 32-bit integer fixed-point (Q32).

The **fixed-point integer format** provides an incredible computational advantage in representing real values when the processor does not have floating-point hardware acceleration. In fact, *this numerical format can represent real numbers through integer values.*

Conceptually, the integer representation of real numbers is straightforward. To represent a range of real numbers, *we apply a scale factor to the real values to map them to the integer range:*

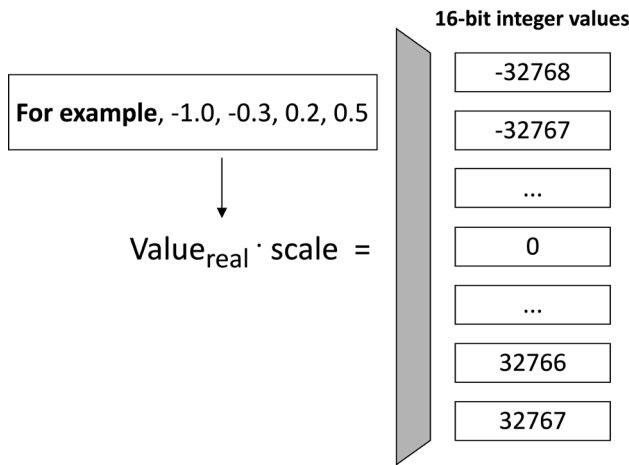


Figure 6.1: Representation of real numbers with integer values

From this conversion, it should be evident that real numbers are equidistantly on the fixed-point scale. Therefore, it implies that *the distance between two representable real values is always the same*.



The distance between two representable real values is called **precision**.

To map a range of real numbers to a fixed-point format, we should use an appropriate scale factor that accounts for the full range of desired values.



This scale factor is often chosen as a power of two for computational efficiency.

For example, suppose you want to represent the $[-2, 1.5]$ range using 16-bit fixed-point numbers. In this case, 32768 would not be a suitable scale factor because it would not be possible to represent both extremities of the range with a 16-bit integer number. Specifically, a 16-bit integer can only represent values between -32768 and 32767, and a scale factor of 32768 would cause the extremities (-2 and 1.5) to exceed this range. In contrast, a scale factor of 16,384 could be an excellent candidate to represent the entire range.

The applied scale factor is also referred to as a **binary point** because it acts in the same way as the decimal point in decimal arithmetic. Therefore, like the decimal point, the binary point separates the **integer (m)** and **fractional (n)** parts in the fixed-point value, as shown in Figure 6.2:

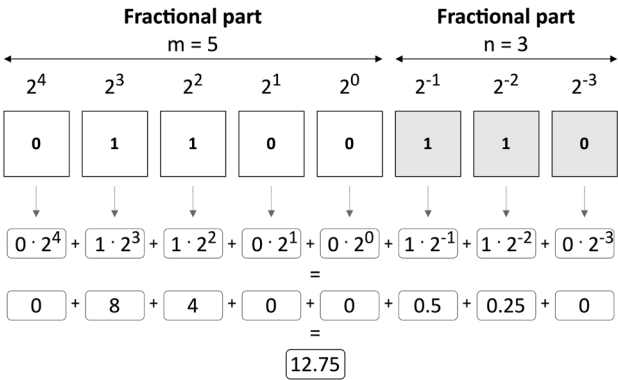


Figure 6.2: Example of how the 12.75 value can be represented with 8-bit fixed-point using 3 bits for the fractional part

Since *the scale factor determines the fixed position of the binary point*, we can deduce that it is defined as:

$$scale = 2^n$$

Where n is the number of bits reserved for the fractional part of the number.

One widely used fixed-point format is **Q1.15**, or simply **Q15**, with the name derived from the 15 bits used for the fractional part and 1 bit for the integer one. The Q15 format is relatively standard in digital signal processing applications when the numbers are in the $[-1, 1)$ range. In fact, this format can represent numbers in the range of $[-1, 0.9999]$ with a precision of $1/32768$ (0.0000305175).



For simplicity, we can assume that the Q15 format is suitable for representing values in the range $[-1, 1]$ instead of $[-1, 1)$, which is the more accurate definition.

Since the binary point can be placed wherever we want through the scale factor, it is essential to specify the format of the fixed-point number to ensure clarity when working with this numerical type. The commonly used convention to define the fixed-point type is **Qm.n**, where **m** is the number of bits for the integer part, and **n** is the number of bits for the fractional part.



A helpful tool for playing with fixed-point arithmetic is available at the following link: <https://chummersone.github.io/qformat.html#converter>. This tool enables you to explore how the placement of the binary point affects the numerical range and precision.

Converting a floating-point number to a fixed-point number is straightforward, and, as we know, it can be accomplished using the following formula:

$$V_{fixed} = V_{float} \cdot scale$$

Where:

- $scale$ is the scale factor
- V_{fixed} is the fixed-point value
- V_{float} is the floating-point value

On the other hand, if you need to convert a fixed-point number to a floating-point number, you can use the following one:

$$V_{float} = \frac{V_{fixed}}{scale}$$

As we will see in the following *How to do it...* section, the CMSIS-DSP library provides the routines to facilitate the conversion of numbers between these two formats.

How to do it...

Continue working in the Colab notebook of the previous chapter, and proceed with the following steps to calculate the FFT magnitude using 16-bit fixed-point arithmetic through the CMSIS-DSP library:

Step 1:

Import the `cmsisdsp` Python library:

```
import cmsisdsp as dsp
```

In the previous code, we import the `cmsisdsp` library as `dsp` to use a shorter name when calling CMSIS-DSP functions.

Step 2:

Implement a function that computes the RFFT in the 16-bit fixed-point format using the CMSIS-DSP library. To do so, define a function interface that accepts a NumPy array as an input argument that holds values in Q15 format:

```
def rfft_q15(src):
```

The FFT computation with the CMSIS-DSP library consists of three steps:

1. The creation of the **RFFT instance** through the `arm_rfft_instance_q15()` function
2. The **initialization** of the instance through the `arm_rfft_init_q15()` function
3. The **computation** of the RFFT through the `arm_rfft_q15()` routine



You can learn more about the Q15 real FFT using the CMSIS-DSP library at the following link: https://arm-software.github.io/CMSIS-DSP/main/group__RealFFTQ15.html.

Therefore, within the function, create the `arm_rfft_instance_q15()` instance:

```
inst = dsp.arm_rfft_instance_q15()
```

Then, initialize the instance with the `arm_rfft_init_q15()` function:

```
src_len = src.shape[0]
stat = dsp.arm_rfft_init_q15(inst, src_len, 0, 1)
```

The `arm_rfft_init_q15()` function accepts the following input arguments:

- The instance to initialize (`inst`)
- The FFT length (`src_len`). The RFFT function in the CMSIS-DSP library only supports the following lengths: 32, 64, 128, 256, 512, 1024, 2048, 4096, and 8192
- The flag (`0`) to specify the transform direction: `0` for *forward* and `1` for *inverse*
- The flag (`1`) to enable the bit reversal of the output: `1` for the output to be in the regular order or `0` to be in the bit reversed order.

Once you have initialized the instance, compute the RFFT through the `arm_rfft_q15()` routine:

```
fft_q15 = dsp.arm_rfft_q15(inst, src)
```

The `arm_rfft_q15()` function takes the following input arguments:

- The RFFT instance (`inst`) to define the compute properties of the RFFT
- The Q15 input array (`src`) to apply the RFFT

According to the official documentation, the 16-bit fixed-point output (`fft_q15`) is not in Q15 format. Instead, the fixed-point output format depends on the FFT length. In our case, where the FFT length is **2048**, it is **Q12.4**. Therefore, 4 bits are reserved for the fractional part of the number, while 12 bits are used for the integer part.

The output format produced by the `arm_rfft_q15()` function differs from what we previously obtained using the `tf.signal.rfft()` TensorFlow signal processing function. The following image shows how the CMSIS-DSP library stores the FFT output values in memory:

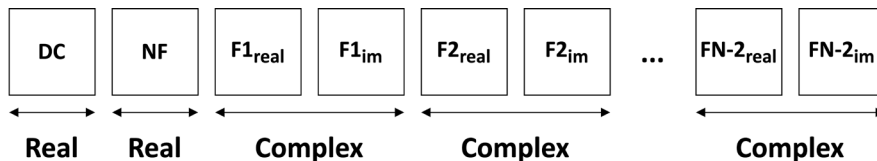


Figure 6.3: FFT output format when using the CMSIS-DSP library

Therefore:

- The output contains twice the samples included in the input frame
- The first value (**DC**) corresponds to the **DC** component, which is represented as a real number only
- The second value (**NF**) corresponds to the **Nyquist** component, which is defined as a real number only
- Starting from the third position of the FFT output, the remaining **N-2** components are represented as complex numbers, where **N** is the FFT length. Each complex number comprises a pair of values: one representing the real part and the other representing the imaginary part

Given the symmetry property of the real FFT, the **DC** component and the first **N/2** complex values are enough to reconstruct the entire spectrum.

Hence, make the function return only the first **N/2 + 1** complex values of the FFT output:

```
return fft_q15[:src_len + 1]
```

As the first two elements are real numbers and each of the following are complex numbers represented with a pair of values, retrieving the first `src_len + 1` values of the output is necessary.



As you may have observed, the output format of the RFFT is less obvious than the one obtained with the TensorFlow function counterpart. CMSIS-DSP adopts this format to minimize memory usage, which is crucial for microcontroller deployment.

Step 3:

Implement a function to calculate the FFT magnitude in Q15 format using the CMSIS-DSP library.

To do so, define a function interface that accepts a NumPy array as an input argument:

```
def mag_q15(src):
```

Although the function was named `mag_q15`, the input argument `src` can be a NumPy array of numbers in any 16-bit fixed-point format. As a result, it can operate on Q15, Q12.4, or any other 16-bit fixed-point types.

Within the function, compute the magnitude with the `arm_cmlx_mag_q15()` function:

```
f0 = src[0],
```

```
fn = src[1],  
fx = dsp.arm_cmplx_mag_q15(src[2:])  
return np.concatenate((f0, fx, fn))
```

In the previous code:

- The values stored in the `f0` and `f1` arrays correspond to the magnitude of the DC and Nyquist components. Since their imaginary part is zero, the magnitude is the same as the real part
- The values stored in the `fx` array are the magnitudes of the remaining frequencies in the spectrum. These magnitudes are calculated starting from the third element of the `src` array, using the `arm_cmplx_mag_q15()` CMSIS-DSP function

As per the official documentation, the `arm_cmplx_mag_q15()` function employs Q15 by Q15 multiplications internally and returns an output converted to Q2.14 format. Since the binary point can be disregarded when multiplying two fixed-point numbers with the same type, the operands are treated as integer values. Therefore, the Q12.4 or any other fixed-point format can safely be used as input. However, as the output is converted to Q2.14, it implies that the resulting fixed-point type will be Q13.3 and not Q12.4 anymore.

In the final line of the code snippet, the function returns an array that concatenates the magnitudes of the different components.

Step 4:

Take `FRAME_LENGTH` (2048) samples from the test audio sample (`test_ad`):

```
src = test_ad[0:FRAME_LENGTH]
```

In the preceding code, the `src` array contains the first 2048 (`FRAME_LENGTH`) samples of the `test_ad` array, creating the input test for the FFT magnitude calculation.

Step 5:

Compute the FFT magnitude using the 16-bit fixed-point arithmetic.

To accomplish this task, you need first to convert the audio sample from floating-point format to Q15 fixed-point format, using the `arm_float_to_q15()` CMSIS-DSP function:

```
src_q15 = dsp.arm_float_to_q15(src)
```

Since the amplitude of audio samples is in the $[-1, 1)$ range, the Q15 format can be safely adopted to represent these values.



The preceding function simply multiplies each input floating-point sample by 32768, which is the scale factor of the Q15 format.

Once you have converted the audio samples to Q15, calculate the FFT magnitude using the functions created in *Step 2* and *Step 3*:

```
cmsis_fft_q15 = rfft_q15(src_q15)
cmsis_fft_mag_q15 = mag_q15(cmsis_fft_q15)
```

After obtaining the FFT magnitude in fixed-point format, we can evaluate the numerical behavior of the proposed method by comparing it with the floating-point TensorFlow implementation.

Step 6:

Convert the FFT magnitude from fixed-point format to floating-point:

```
scale = float(1 << 3)
cmsis_fft_mag = cmsis_fft_mag_q15 / scale
```

This conversion is necessary to compare the FFT magnitude derived from the CMSIS-DSP library against the one obtained using the TensorFlow signal processing functions. In the previous code, the conversion from fixed-point to floating-point is performed by dividing the FFT magnitude by 8 ($1 \ll 3$), the scale factor of the Q13.3 format.

Step 7:

Calculate the FFT magnitude with TensorFlow signal processing functions:

```
tf_fft = tf.signal.rfft(src)
tf_fft_mag = tf.math.abs(tf_fft)
```

Step 8:

Calculate the absolute difference between the output obtained with the TensorFlow signal processing functions and the CMSIS-DSP implementation:

```
abs_diff = np.abs(tf_fft_mag - cmsis_fft_mag)
```

Then, print the minimum, maximum, mean, and standard deviation of the absolute difference in the output log:

```
print("DIFF:\n",
      "min:", np.min(abs_diff),
      "max:", np.max(abs_diff),
      "mean:", np.mean(abs_diff),
      "std:", np.std(abs_diff))
```

The statistical information in the output log will indicate that the two implementations are not numerically identical. This numerical difference can be attributed to their different numerical precisions. Nevertheless, since the training data for our model will come from the CMSIS-DSP Python version, this discrepancy in output values will not influence the model's accuracy.

There's more...

In this recipe, we learned how to leverage the Q15 fixed-point arithmetic in the FFT magnitude calculation using the CMSIS-DSP Python library.

For those interested in improving the performance of the FFT magnitude calculation at the expense of some accuracy, the CMSIS-DSP `arm_cmplx_mag_fast_q15()` function is an alternative option to perform the complex magnitude computation. The output of this function is still in Q2.14 format, but it tends to be less accurate due to the clamping of small values to zero.

Having gained knowledge about the fixed-point arithmetic and the CMSIS-DSP library, we can now redesign the MFCCs feature extraction using the Q15 data type.

In the upcoming recipe, we will demonstrate how to implement the entire MFCCs data pipeline employing the CMSIS-DSP library.

Implementing the MFCCs feature extraction with the CMSIS-DSP library

The practical use of Q15 fixed-point arithmetic in computing the FFT magnitude provided the foundational knowledge to build functions using fixed-point arithmetic.

In this recipe, we will exploit this knowledge to rewrite the implementation of the MFCCs feature extraction using the Q15 data format.

Getting ready

In the preceding recipe, we learned that the Raspberry Pi Pico, like many other microcontrollers, does not have hardware acceleration for floating-point arithmetic. Despite this limitation, we now know we can leverage the computation with the fixed-point format and rely solely on integer arithmetic.

However, the MFCCs computation can be optimized even further on the Raspberry Pi Pico by exploiting an incredible additional feature offered by this device: the large program memory size.

As we know, the program memory is reserved for storing the program. However, this memory can also be utilized to store constant data. Since the microcontroller comes with 2 MB of program memory, a possible optimization is to precompute parts of the algorithm that are repeated during each execution and store them in lookup tables. As you can guess, this approach can eliminate redundant computations and significantly improve performance.

In the computation of the MFCCs, we have plenty of opportunities to leverage this optimization technique. For example, the coefficients of the Hann window are identical for all frames. Therefore, we can pre-calculate them to skip the costly computation of the cosine. Similarly, this optimization technique can benefit the Mel-scale conversion, logarithm function, and **Discrete Cosine Transform (DCT)**.

However, we haven't extensively discussed the DCT, so it might be worthwhile seeing how it works to understand what can be precomputed.

Extracting the DCT coefficients

As we have seen with the MFCCs implementation with TensorFlow, we can use the TensorFlow `tf.signal.mfccs_from_log_mel_spectrograms()` signal processing function to compute the DCT after the Log-Mel-scale conversion. By examining the implementation code (https://github.com/tensorflow/tensorflow/blob/v2.11.0/tensorflow/python/ops/signal/mfcc_ops.py#L27), it becomes evident that TensorFlow calculates the **DCT-II** coefficients using the following formula:

$$W_{DCT}(k) = \sqrt{\frac{2}{N}} \cdot \sum_{n=0}^{N-1} x(n) \cdot \cos\left(\frac{\pi}{N} \cdot (n + 0.5)\right) \cdot k$$

Where:

- $W_{DCT}(k)$ is the DCT coefficient at index k
- N is the number of Mel frequencies (40, in our case)
- n is the index of the Mel frequency
- k is the index of the DCT coefficient that we want to calculate

Therefore, as we saw in the Mel-scale conversion, the DCT computation can also be performed through a vector-by-matrix operation.

In this case, the LHS operator is the output of the logarithmic function, while the RHS one is the *DCT-weight* matrix, as shown in Figure 6.4:

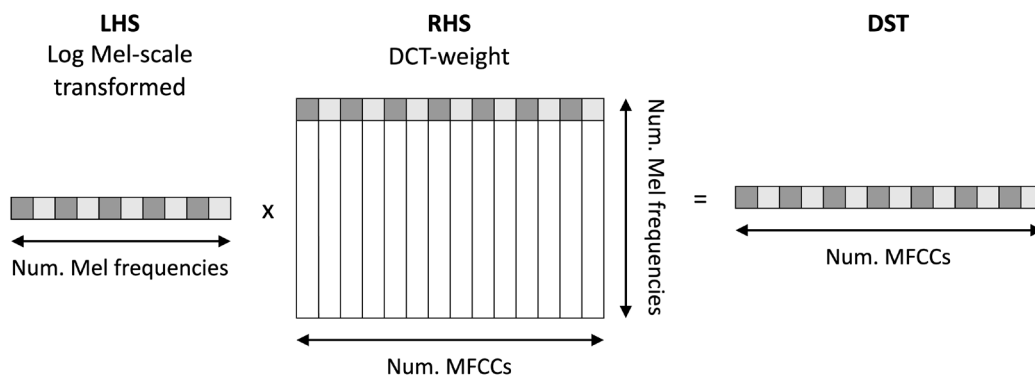


Figure 6.4: DCT computation as a vector-by-matrix multiplication

The *DCT-weight* matrix is identical for all frames and has the number of columns corresponding to the number of MFCCs we want to extract.

How to do it...

Continue working in the Colab notebook, and follow these steps to implement the MFCCs pipeline with the CMSIS-DSP library using 16-bit fixed-point arithmetic:

Step 1:

Implement a function to precompute the Hann window coefficients in the Q15 format:

```
def gen_hann_lut_q15(frame_len):
    hann_lut_f32 = tf.signal.hann_window(frame_len)
    return dsp.arm_float_to_q15(hann_lut_f32)
```

The Python function accepts the frame length (`frame_len`) as the input argument. The frame length is then passed to the `tf.signal.hann_window()` TensorFlow signal processing function to obtain the Hann window coefficients. The coefficients are converted to Q15 format using the `arm_float_to_q15()` CMSIS-DSP routine.



The extremes of the input floating signal range, which are 0 and 1, are safely representable with the Q15 format.

Step 2:

Implement a function to precompute the *Mel-weight* matrix in Q15 format. This function should accept the necessary attributes to apply the Mel-scale conversion. These attributes are the *sample rate*, *fmin in Hz*, *fmax in Hz*, *number of Mel frequencies*, and *number of FFT frequencies*:

```
def gen_mel_weight_mtx(sr, fmin_hz, fmax_hz,
                       num_mel_freqs, num_fft_freqs):
```

The variable's names for the input arguments should be self-explanatory to know the purpose of each one.

Within the function, use the `tf.signal.linear_to_mel_weight_matrix()` TensorFlow routine to produce the *Mel-weight* matrix. Then, convert each value to Q15 format:

```
m_f32 = tf.signal.linear_to_mel_weight_matrix(
    num_mel_freqs,
    num_fft_freqs,
    sr,
    fmin_hz,
    fmax_hz)
m_q15 = dsp.arm_float_to_q15(m_f32)
```

Before returning the *Mel-weight* matrix, reshape the `m_q15` array into a matrix, as the `arm_float_to_q15()` CMSIS-DSP function collapses all dimensions to the first one:

```
return m_q15.reshape((m_f32.shape[0],
                      m_f32.shape[1]))
```

Afterward, the *Mel-weight* matrix will have the number of columns corresponding to the number of Mel frequencies and the number of rows equal to the FFT frequencies.



The values stored in the *Mel-weight* matrix are between 0 and 1. Therefore, they can be safely represented with the Q15 format.

Step 3:

Implement a function to precompute the logarithmic function. This function should accept the 16-bit fixed-point scale factor as the input argument:

```
def gen_log_lut_q(q_scale):
```

The `q_scale` input variable will allow the conversion from 16-bit fixed-point to floating-point, and vice versa.

Within the function, we iterate over all positive 16-bit integer values:

```
    max_int16 = np.iinfo("int16").max
    log_lut = np.zeros(shape=(max_int16), dtype="int16")

    for i16 in range(0, max_int16):
```

As the logarithmic function is not defined for negative values, we can precompute the logarithmic function only for half of all possible 16-bit integer values.

Convert every 16-bit positive integer value (`i16`) to floating-point format, apply the logarithmic operation, and convert the result back to fixed-point format:

```
        q16 = np.array([i16,], dtype="int16")
        f_v = q16 / float(q_scale)
        log_f = np.array(np.log(f_v + 1e-6),)
        log_q = log_f * float(q_scale)
        log_lut[i16] = int(log_q)
    return log_lut
```

In the previous code, the `q16 / float(q_scale)` expression is used to convert the 16-bit fixed-point integer value to floating-point, while the `log_f * float(q_scale)` expression is employed to convert the result of the logarithmic back to fixed-point.

Step 4:

Implement a function to precompute the *DCT-weight* matrix in the Q15 format. The function should accept the number of Mel frequencies and the number of MFCCs to extract as input arguments:

```
import math
def gen_dct_weight_mtx(num_mel_freqs, num_mfccs):
```

Within the `gen_dct_weight_mtx()` function, create an `int16` matrix filled with zeros. The matrix dimensions must correspond to those of the *DCT-weight* matrix. Therefore, the number of columns should be equal to the number of MFCCs, while the number of rows should be equal to the number of Mel frequencies:

```
mtx_q15 = np.zeros(shape=(num_mel_freqs, num_mfccs),
                      dtype="int16")
```

The `mtx_q15` matrix will contain the *DCT-weight* matrix in the Q15 format. Once you have initialized with zeros, calculate the values of the *DCT-weight* matrix by applying the formula presented in the *Getting ready* section of this recipe:

```
scale = np.sqrt(2.0 / float(num_mel_freqs))
pi_div_mel = (math.pi / num_mel_freqs)

for n in range(num_mel_freqs):
    for k in range(num_mfccs):
        v = scale * np.cos(pi_div_mel * (n + 0.5) * k)
        v_f32 = np.array([v,], dtype="float32")
        mtx_q15[n][k] = dsp.arm_float_to_q15(v_f32)

return mtx_q15
```

In the previous code, we used the `cos()` function from the NumPy Python library to perform the cosine function. Since the result of this operation is in floating-point format, we use the `arm_float_to_q15()` CMSIS-DSP function to convert it to Q15.



Since the scale (`scale`) is less than one and the result of the cosine is in the `[-1, 1]` range, the Q15 format can be used to represent values in the *DCT-weight* matrix.

Step 5:

Precompute the Hann window coefficients, *Mel-weight* matrix, logarithmic function, and *DCT-weight* matrix:

```
num_fft_freqs = int((FFT_LENGTH / 2) + 1)
q13_3_scale = 8
# Precompute the Hann window coefficients
hann_lut_q15 = gen_hann_lut_q15(FRAME_LENGTH)

# Precompute the Mel-weight matrix
mel_wei_mtx_q15 = gen_mel_weight_mtx(SAMPLE_RATE,
                                     FMIN_HZ,
                                     FMAX_HZ,
                                     NUM_MEL_FREQS,
                                     num_fft_freqs)

# Precompute the Log function for Q13.3
log_lut_q13_3 = gen_log_lut_q(q13_3_scale)

# Precompute the DCT-weight matrix
dct_wei_mtx_q15 = gen_dct_weight_mtx(NUM_MEL_FREQS,
                                     NUM_MFCCS)
```

After precomputing the necessary parts of the MFCCs pipeline, determine the amount of program memory necessary to store them:

```
num_bytes = 2
mem_usage = 0
mem_usage += np.size(hann_lut_q15) * num_bytes
mem_usage += np.size(mel_wei_mtx_q15) * num_bytes
mem_usage += np.size(log_lut_q13_3) * num_bytes
mem_usage += np.size(dct_wei_mtx_q15) * num_bytes
print("Program memory usage: ", mem_usage, "bytes")
```

The expected output should be as follows:

```
Program memory usage: 153070 bytes
```

Figure 6.5: MFCCs program memory utilization

Thus, precomputing the Hann window coefficients, *Mel-weight* matrix, logarithmic function, and *DCT-weight* matrix will necessitate approximately 153 KB of program memory, which accounts for roughly 8% of the total program memory available.

Step 6:

Implement a function to compute the MFCCs with 16-bit fixed-point arithmetic. The function's input parameters should include the following arguments:

- The audio sample to split into frames
- The number of MFCCs to extract
- The frame length
- The frame step
- The precomputed Hann window coefficients, Mel-weight matrix, logarithmic function, and DCT-weight matrix

```
def extract_mfccs_cmsis(  
    ad_src,  
    num_mfccs,  
    frame_length,  
    frame_step,  
    hann_lut_q15,  
    mel_wei_mtx_q15,  
    log_lut_q13_3,  
    dct_wei_mtx_q15):
```

The variable's names for the input arguments should be self-explanatory to know the purpose of each one.

Within the function, initialize the output array used to hold the MFCCs:

```
    n = ad_src.shape[0]  
    num_frames = int((n - frame_length) / frame_step)  
    num_frames += int(1)  
  
    output = np.zeros(shape=(num_frames, num_mfccs))
```

Then, iterate over each frame:

```
for i in range(num_frames):
    idx_s = i * frame_step
    idx_e = idx_s + frame_length
    frame = ad_src[idx_s:idx_e]
```

For each frame, use the 16-bit fixed-point arithmetic to extract the MFCCs. To accomplish this task, first convert the frame from floating-point to Q15 format:

```
frame_q15 = dsp.arm_float_to_q15(frame)
```

Since the audio samples in the frame are expected to be in the $[-1, 1)$ range, it is safe to convert them to Q15.

Afterward, apply the Hann window on the input frame. To do so, perform the element-wise multiplication between the Hann window coefficients and the input frame using the `arm_mult_q15()` CMSIS-DSP function:

```
frame_q15 = dsp.arm_mult_q15(frame_q15,
                             hann_lut_q15)
```

Once you have applied the Hann window, calculate the FFT magnitude:

```
fft_spect_q15 = rfft_q15(frame_q15)
fft_mag_spect_q15 = mag_q15(fft_spect_q15)
```

As we know from the previous recipe, the fixed-point format for the FFT magnitude is Q13.3.

After the calculation of the FFT magnitude, perform the Mel-scale conversion with the `arm_mat_vec_mult_q15()` CMSIS-DSP routine:

```
log_mel_spect_q15 = dsp.arm_mat_vec_mult_q15(
    mel_we_i_mtx_q15.T,
    fft_mag_spect_q15.T)
```

In the preceding code, the matrix-by-vector routine (`arm_mat_vec_mult_q15`) has been used because the vector-by-matrix one is not included in the CMSIS-DSP library.

Therefore, the *Mel-weight* matrix becomes the LHS operand, while the FFT magnitude becomes the RHS one, as shown in the following figure:

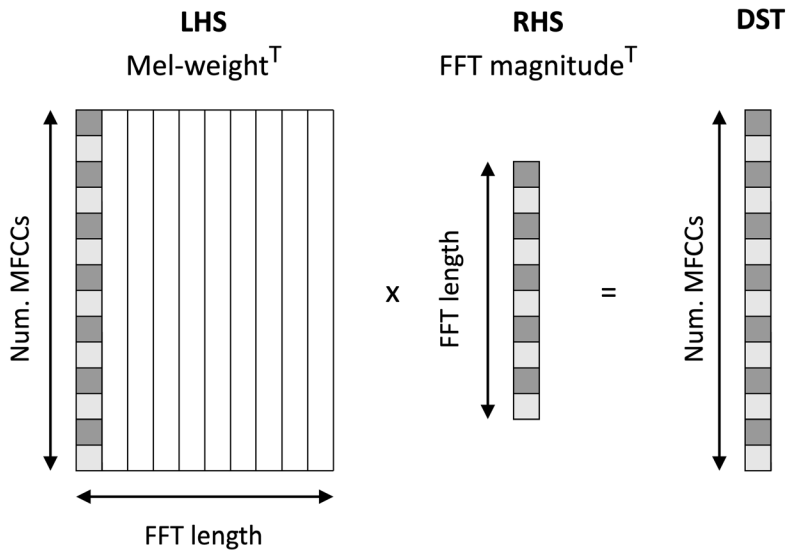


Figure 6.6: Mel-scale conversion as a matrix-by-vector multiplication

Since LHS and RHS matrices are swapped with respect to what we had with the TensorFlow implementation, we need to transpose them ($.T$) to use the matrix-by-vector function.



You have probably noticed that the inputs have different 16-bit fixed-point types, as the LHS matrix is Q15, while the RHS matrix is Q13.3. Therefore, what is the output fixed-point format after the multiplication? Since the maximum representable floating-point value in absolute terms for the Q15 type is 1, the output format is still Q13.3.

Once you have performed the Mel-scale conversion, apply the logarithmic function using the precomputed lookup table:

```
for idx, v in enumerate(log_mel_spect_q15):
    log_mel_spect_q15[idx] = log_lut_q13_3[v]
```

Then, calculate the MFCCs with the `arm_mat_vec_mult_q15()` CMSIS-DSP routine:

```
mfccs = dsp.arm_mat_vec_mult_q15(
    dct_weigh_mtx_q15.T,
    log_mel_spect_q15)
```

Similar to what we have seen for the Mel-scale conversion, we use the matrix-by-vector routine (`arm_mat_vec_mult_q15`). Therefore, the *DCT-weight* matrix is the LHS operand, while the output of the Log-Mel-scale conversion (the Log-Mel-scale transformed) is the RHS one. However, only the *DCT-weight* matrix needs to be transposed in this case because the Log-Mel-scale input is already in the vector-column format, as illustrated in the following figure:

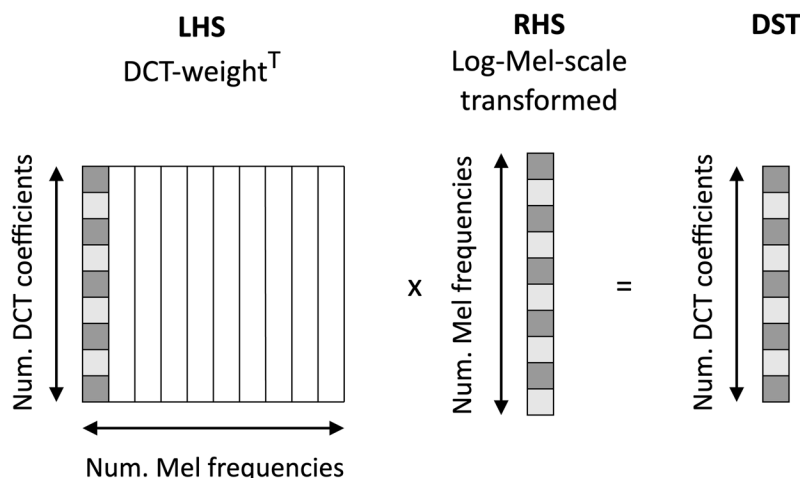


Figure 6.7: DCT computation as a matrix-by-vector multiplication

Finally, convert the MFCCs to floating-point, since this function will be used to generate the training samples:

```
output[i] = mfccs.T / float(8)
return output
```

As you can see in the preceding code snippet, the `mfccs` array is transposed with the `.T` method to match the data layout returned by the MFCCs implementation, developed with TensorFlow.

Step 7:

Extract the MFCCs from the `test_ad` audio sample using the `extract_mfccs_cmsis()` function:

```
mfccs_cmsis = extract_mfccs_cmsis(
    test_ad,
    NUM_MFCCS,
    FRAME_LENGTH,
    FRAME_STEP,
    hann_lut_q15,
```

```
mel_wei_mtx_q15,  
log_lut_q13_3,  
dct_wei_mtx_q15)
```

Then, use the `display_mfccs()` function to display the `mfccs_cmsis` array as an image:

```
display_mfccs(mfccs_cmsis.T)
```

In the previous code snippet, the `mfccs_cmsis` array is transposed using the `.T` method solely to visualize the temporal information along the *x*-axis.



The array holding the MFCCs was also transposed in the first part of this project, when using the TensorFlow signal processing functions to extract the MFCCs.

If you have used the `disco/disco.00002.wav` audio clip, the expected image should be almost identical to what was obtained using the TensorFlow implementation.

This result demonstrates our successful employment of 16-bit fixed-point arithmetic to speed up the MFCCs feature extraction.

Now, you are ready to train the model for music genre classification.

There's more...

In this recipe, we learned how to leverage the computation of the MFCCs feature extraction with the 16-bit fixed point arithmetic using the CMSIS-DSP library.

The visual representation from the MFCCs array aligns closely with the TensorFlow version's output. However, we know that the fixed-point method is numerically different from the floating-point approach. Therefore, what are the areas with more significant numerical differences?

If you are interested in visualizing the areas of the image where the 16-bit fixed-point MFCCs implementation is less accurate, you can display the absolute difference between the output result obtained with TensorFlow and CMSIS-DSP:

```
abs_diff = np.abs(mfccs_tf - mfccs_cmsis)  
display_mfccs(abs_diff.T)
```

The expected output should be as follows:

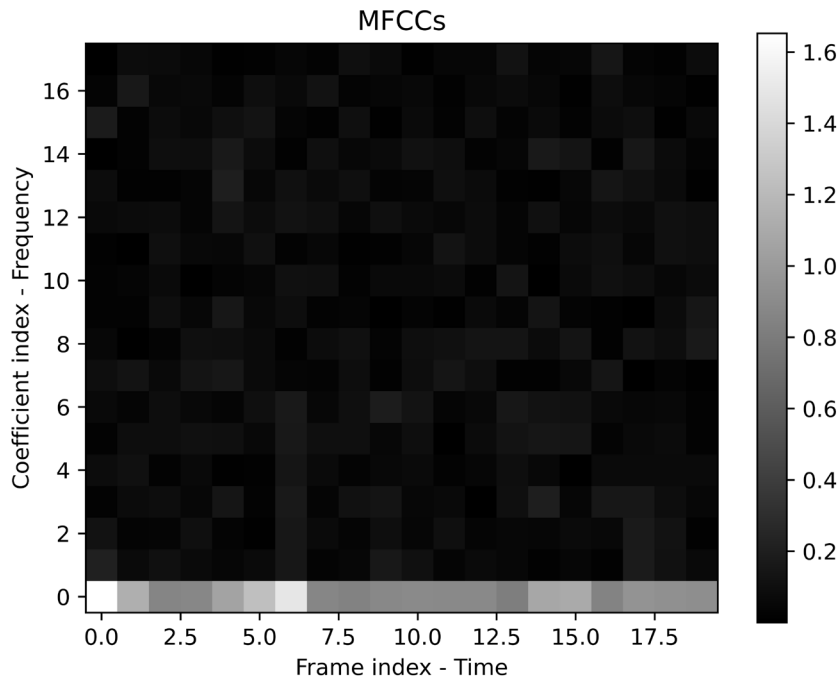


Figure 6.8: Visualization of the numerical differences between the CMSIS-DSP and TensorFlow implementation

In the preceding image, the darker the color, the lower the numerical difference. Therefore, the most significant numerical differences are concentrated in the first bottom row, corresponding to the 0th MFCC. This result confirms what we could already guess: that the error is proportional to the value's magnitude. Consequently, when trying to represent larger values, the error tends to be higher than when representing smaller ones.

Now that we have successfully implemented the feature extraction algorithm with the fixed-point arithmetic, we can train the model.

In the upcoming recipe, we will design and train an LSTM model to classify music genres.

Designing and training an LSTM RNN model

In this project, the model designed for classifying music genres is an LSTM RNN, as illustrated in the following diagram:

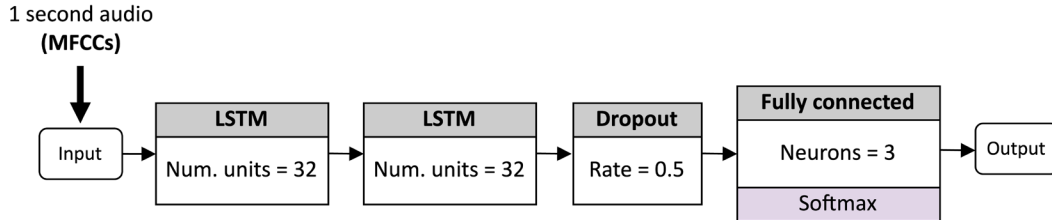


Figure 6.9: LSTM recurrent neural network for music genre classification

As shown in the previous image, the MFCCs extracted from 1 second of raw audio are the input for the model, which consists of the following layers:

- 2 x **LSTM** layers with 32 number of units each (**Num. units**)
- 1 x **Dropout** layer with a 50% rate (0.5)
- 1 x **Fully connected** layer with three output neurons, followed by a **Softmax** activation function

In this recipe, we will design and train this LSTM model with TensorFlow.

Getting ready

In *Chapter 4, Using Edge Impulse and the Arduino Nano to Control LEDs with Voice Commands*, we addressed an audio classification problem using a standard **convolutional neural network (CNN)** that learned visual patterns from the **Mel-filterbank energy (MFE)** extracted from raw audio data. Since **MFCCs** are just a compressed representation of MFE, the problem we aim to solve here is similar to what we previously faced.

Even though MFE and MFCCs can be treated as images, they inherently carry temporal information. In fact, as we discovered in the first part of this project, the MFCCs are features extracted at a particular time step. Therefore, this temporal information suggests we can explore an alternative architecture to CNN, better suited for time series analysis: RNNs.

Time series analysis with RNNs

RNNs represent a unique category of artificial neural networks tailored for sequential data, such as time series or text.

Some areas where RNNs have been found very successful include:

- **Natural language processing (NLP)**: for example, in the context of machine translation and text generation.
- **Time-series analysis**: for example, to predict weather patterns
- **Speech recognition**: for example, for voice searching or translating speech to text

While traditional feedforward neural networks like CNNs have connections that only move forward without cycles, RNNs are architecturally distinct. RNNs have connections that can cycle back, enabling them to maintain an internal state (**history**) of the previous input, as shown in *Figure 6.10*:

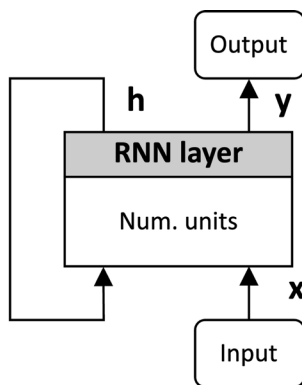


Figure 6.10: RNN high-level representation

In *Figure 6.10*, x is the input, y is the output, h is the internal state, and **Num. units** is the internal state and output size.

The **RNN layer** is a trainable layer that makes this network capable of solving prediction problems from input, with features gathered over multiple time steps. Unlike traditional feedforward neural networks that process input data simultaneously, *the RNN layer takes the features sequentially* based on their time position. Particularly, this layer iterates over all time steps, and for each one, it generates an output (y_t) and updates its history (h_t) using two inputs:

- The set of features at that specific time step (x_t)
- The history of the previous input (h_{t-1})

The following image provides a visual representation of this iterative process, which allows the RNN layer to identify patterns throughout the sequence of data:

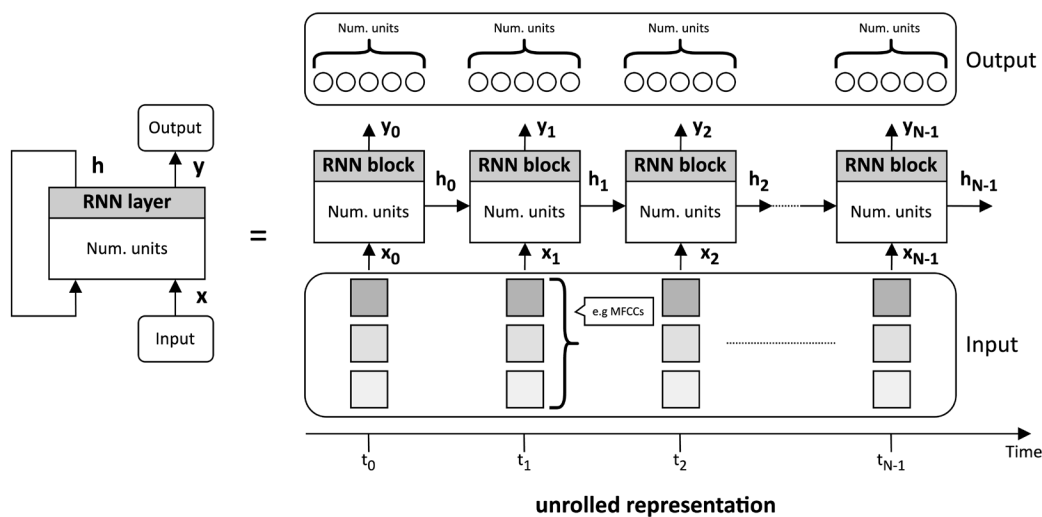


Figure 6.11: RNN layer high-level computation (unrolled representation)

The **unrolled representation** shown in Figure 6.11 should provide a more precise visualization of how data flows through the network, and how the history from a time step influences the result at the next.

The **RNN block** handles the processing for one iteration and is the same for each time step. Therefore, its *learned weights are shared over time space*.

The simplest version of the RNN block, called **vanilla RNN**, updates the history and generates the output using the following formulas:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Where,

- h_t is the history at time t
- h_{t-1} is the history at time $t - 1$
- x_t is the set of input features at time t
- y_t is the output at time t

- W_{hh} is the weight matrix used to project the history at time $t - 1$
- W_{xh} is the weight matrix used to project the input at time t
- W_{hy} is the weight matrix used to generate the output at time t

Figure 6.12 shows the operations performed with the preceding two equations:

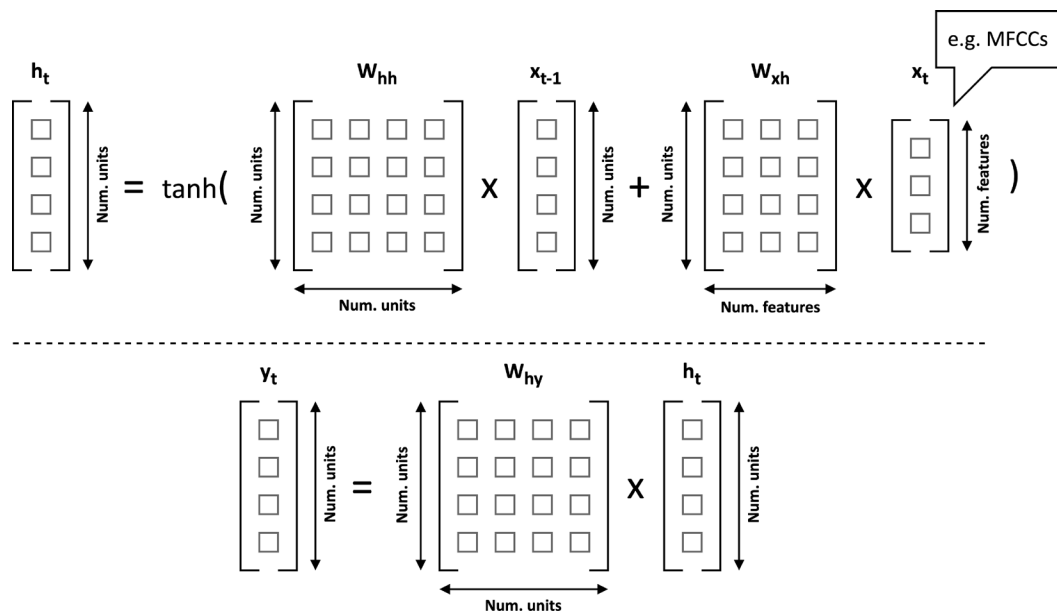


Figure 6.12: Visual representation of the vanilla RNN formulas for updating the history and generating the output

Since the **Num. units** parameter defines the internal state and output size, we can conclude that it affects the number of trainable parameters.

In theory, vanilla RNNs could remember long-term dependencies across a sequence of input data. However, in practice, there is a hitch.

When we train vanilla RNNs for a long data sequence using back-propagation, the gradients can diminish toward zero (**vanishing gradient**) and impede training the model effectively.

Therefore, **LSTM** cells were designed to address this problem.



The LSTM layer in the Keras API is documented at the following link: https://keras.io/keras_core/api/layers/recurrent_layers/lstm/. The Keras API takes a 3D input tensor consisting of feature columns, time step rows, and a batch size dimension. In our project, the number of features corresponds to the number of MFCCs obtained at every time step.

Here, we won't delve into how this operator addresses the vanishing gradient issue because it is not required for this project and falls outside the book's scope. What is crucial to understand is that our RNN employs the LSTM cell, and the primary hyperparameter to consider is **Num. units**, which influences the history and output size.



A recommended reference to understand how the LSTM works is from the *Deep Learning* book (*Chapter 10*) by Ian Goodfellow, Yoshua Bengio, and Aaron Courville, available at the following link: <https://www.deeplearningbook.org/contents/rnn.html>.

Before proceeding with the model training, we want to discuss the rationale that brought us to design the RNN architecture presented in *Figure 6.9*.

Designing a many-to-one RNN for music genre classification

Let's start this discussion by considering the RNN unrolled representation shown in *Figure 6.11*. This representation is also simplified and presented in the following image:

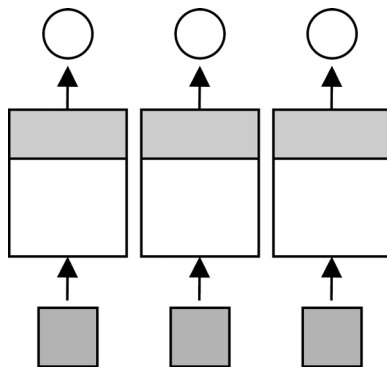


Figure 6.13: A many-to-many RNN architecture

In the preceding unrolled representation, the model takes an input and yields **Num. units** output at every time step. As a result, this RNN architecture generates another sequence.



The LSTM Keras API returns a 3D output tensor with **Num. units** columns, time steps rows, and an optional batch size dimension ([batch, time steps, **Num. units**]).

This model architecture is called **many-to-many** and is standard for solving **sequence-to-sequence** tasks, such as speech recognition or language translation.

However, RNNs are very versatile to solve tasks of various types, and this flexibility is given by the alternative ways they can take the input and generate the output. For example, consider the following alternative RNN architecture, called **many-to-one**, which still consumes the input at each time step but generates the **Num. units** output only at the final time step:

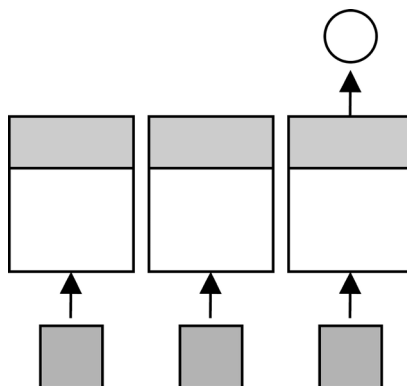


Figure 6.14: A many-to-one RNN architecture

Many-to-one architecture is commonly used for classification problems and, for this reason, is adopted in this project.



The many-to-many and many-to-one are just a few of the possible RNN architecture variants. You can discover more about the other RNN configurations at the following link: <https://cs231n.github.io/rnn/>.

Generally, a stacked LSTM architecture, consisting of two or more LSTM layers, is recommended to learn complex feature representation of the input and then improve accuracy. To build such architecture, it is necessary that all LSTM layers, except the last one, return the output at each time step rather than a single output for the final time step.



In TensorFlow, the LSTM layer returns the output at each time step when its `return_sequences` parameter is set to `True`.

The last thing worth mentioning before demonstrating how to design and train this model is an issue you can find when training LSTM RNNs. These networks can easily overfit. Therefore, including a dropout layer with a large dropout rate (for example, 0.5) is highly recommended to minimize this risk.

How to do it...

Continue working in the Colab notebook, and follow these steps to design and train the LSTM RNN to classify music genres:

Step 1:

Extract the MFCCs features from all training samples. To do so, import the `os` Python module to use its `listdir()` method, which lists all the files in a specified directory:

```
import os
```

The `listdir()` directory will be utilized later on to retrieve the name of the audio files stored in the training dataset.

Then, initialize two empty lists to store the MFCCs extracted from the 1-second audio clip (`x`) and the corresponding label (`y`):

```
x = []  
y = []
```

After, iterate over all audio samples stored in the training dataset:

```
LIST_GENRES = ['disco', 'jazz', 'metal']  
  
for genre in LIST_GENRES:  
    folder = train_dir + "/" + genre  
  
    list_files = os.listdir(folder)  
  
    for song in list_files:
```

The preceding code snippet iterates through each subfolder in the dataset directory that holds the audio samples. These folders (`disco/`, `jazz/`, and `metal/`) have the name that matches the audio genre of the songs they contain. For each of these folders, we gather all the files and then loop through each one.

At this point, load each file using the `read()` function provided by the `soundfile` library:

```
filepath = folder + "/" + song
try:
    ad, sr = sf.read(filepath)
```

We recommend you use the `try` block to catch possible exceptions when reading the files (for example, reading a corrupted audio file).

Given that songs may exceed a duration of 1 second, divide the audio files into 1-second segments. Then, extract the MFCCs from each, using the MFCCs function implemented with the CMSIS-DSP library:

```
# Training audio length in the number of samples
TRAIN_AUDIO_LENGTH_SAMPLES = 22050

num_it = int(len(ad) / TRAIN_AUDIO_LENGTH_SAMPLES)

for i in range(num_it):
    s0 = TRAIN_AUDIO_LENGTH_SAMPLES * i
    s1 = s0 + TRAIN_AUDIO_LENGTH_SAMPLES
    src_audio = ad[s0 : s1]

    mfccs = extract_mfccs_cmsis(
        src_audio,
        NUM_MFCCS,
        FRAME_LENGTH,
        FRAME_STEP,
        hann_lut_q15,
        mel_wei_mtx_q15,
        log_lut_q13_3,
        dct_wei_mtx_q15)
```


Finally, store the MFCCs and the corresponding class index in the `x` and `y` list. If an exception arises from the `read()` function, proceed with the loop's next iteration:

```
x.append(mfccs.tolist())
y.append(LIST_GENRES.index(genre))

except Exception as e:
    continue
```

If you have built the dataset as described in the first part of this project, this step will extract MFCCs features from approximately 10,000 samples for each music genre.

Step 2:

Convert the `x` and `y` list in NumPy arrays:

```
x, y = np.array(x), np.array(y)
```

Step 3:

Split the dataset into train (60%), validation (20%), and test (20%) datasets using the **scikit-learn** Python library:

```
from sklearn.model_selection import train_test_split

# Split 1 (60% vs 40%)
x_train, x0, y_train, y0 = train_test_split(x, y,
                                           test_size=0.40,
                                           random_state = 1)

# Split 2 (50% vs 50%)
x_test, x_validate, y_test, y_validate =
    train_test_split(x0, y0,
                    test_size=0.50,
                    random_state = 3)
```

Step 4:

Design the many-to-one LSTM RNN architecture illustrated in *Figure 6.9* using TensorFlow's Keras API:

```
import tensorflow as tf
from tensorflow.keras import activations
```

```

from tensorflow.keras import layers

input_shape = (x_train.shape[1], x_train.shape[2])

norm_layer = layers.Normalization(axis=-1)

# Learn mean and standard deviation from dataset
norm_layer.adapt(x_train)

input = layers.Input(shape=input_shape)
x = norm_layer(input)
x = layers.LSTM(32, return_sequences=True)(x)
x = layers.LSTM(32)(x)
x = layers.Dropout(0.5)(x)
x = layers.Dense(len(LIST_GENRES),
                  activation='softmax')(x)

model = tf.keras.Model(input, x)

```

From the design of the model, you can see that we have included a normalization layer (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Normalization) between the input and the LSTM layer. This layer shifts and scales MFCCs into a distribution centered around zero with a standard deviation of one, by applying the Z-score normalization formula:

$$value_{new} = \frac{value_{old} - \mu}{\sigma}$$

The Z-score normalization technique, discussed in *Chapter 3, Building a Weather Station with TensorFlow Lite for Microcontrollers*, requires the input features' mean and standard deviation to adjust the original data's distribution. In this context, TensorFlow determines the mean and standard deviation at runtime when the `adapt()` function of the Keras normalization layer is invoked.

The reason why we normalize the MFCCs is because the coefficients do not have the same numerical range. For example, as we know from the first part of this project, the 0th coefficient generally has large values because the overall loudness of the audio signal influences it. Therefore, by normalizing the MFCCs, we reduce the risk that coefficients with larger values may affect the model more.

Another point we want to highlight is setting the `return_sequences` parameter to `True` in the LSTM layer. This setting enables the LSTM layer to return the output at every time step.

Step 5:

Print the model's summary to gain an understanding of the number of trainable parameters required by the model:

```
model.summary()
```

The preceding code will yield an output like the following:

```
Model: "model_4"

```

Layer (type)	Output Shape	Param #
input_9 (InputLayer)	[(None, 20, 18)]	0
normalization_8 (Normalization)	(None, 20, 18)	37
lstm_8 (LSTM)	(None, 20, 32)	6528
dropout_8 (Dropout)	(None, 20, 32)	0
flatten_4 (Flatten)	(None, 640)	0
dense_12 (Dense)	(None, 32)	20512
dense_13 (Dense)	(None, 3)	99

```

=====
Total params: 27176 (106.16 KB)
Trainable params: 27139 (106.01 KB)
Non-trainable params: 37 (152.00 Byte)
=====

```

Figure 6.15: Model summary returned by `model.summary()`

The information displayed in the output log shows that the LSTM model comprises 14,984 trainable parameters. When quantized to 8-bit, these parameters consume 14,984 bytes, or approximately 14 KB, of program memory. Considering that the Raspberry Pi provides 2 MB of program memory, the model will use roughly 1% of its capacity.

Step 6:

Train the model with 30 epochs and a batch size of 50:

```
optimiser = tf.keras.optimizers.Adam(learning_rate=0.001,
```

```
        beta_1=0.9,
        beta_2=0.999)

model.compile(optimizer=optimiser,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

NUM_EPOCHS = 30
BATCH_SIZE = 50

history = model.fit(
    x_train, y_train,
    epochs= NUM_EPOCHS,
    batch_size=BATCH_SIZE,
    validation_data=(x_validate, y_validate))
```

Training will take a few minutes if you have a dataset of about 10,000 samples for each class. Using our dataset, we achieved an accuracy of approximately 90% and a loss of 0.09 on the validation dataset.

Step 7:

Save the TensorFlow model following the *TensorFlow RNN conversion to TensorFlow Lite* guide provided at the following link: <https://www.tensorflow.org/lite/models/convert/rnn>:

```
run_model = tf.function(lambda x: model(x))

BATCH_SIZE = 1
STEPS = x_train.shape[1]
FEATURES = x_train.shape[2]

concrete_func = run_model.get_concrete_function(
    tf.TensorSpec([BATCH_SIZE, STEPS, FEATURES],
                  model.inputs[0].dtype))

TF_MODEL = 'music_genre'

model.save(TF_MODEL, save_format="tf", signatures=concrete_func)
```

Unfortunately, saving the TensorFlow model entails more steps than what was discussed in *Chapter 3, Building a Weather Station with TensorFlow Lite for Microcontrollers*. These additional steps are essential to make the LSTM operator compatible with the TensorFlow Lite's fused LSTM layer.



TensorFlow Lite's fused LSTM operation exists to minimize memory usage and maximize the performance of its underlying implementation.

The code presented in this step is derived from the sample code provided by the TensorFlow team in the following Colab notebook: https://colab.research.google.com/github/tensorflow/tensorflow/blob/master/tensorflow/lite/examples/experimental_new_converter/Keras_LSTM_fusion_Codelab.ipynb#scrollTo=5pGyWlkJDpMQ.

Now that we have the TensorFlow model saved, we can proceed with the model quantization using the TensorFlow Lite converter, and its accuracy evaluation using the test dataset.

There's more...

In this recipe, we learned how to design and train a many-to-one LSTM model with TensorFlow to classify music genres.

As discussed in the *Getting ready* section, a stacked LSTM architecture is generally recommended to learn complex feature representation of the input and then improve accuracy.

One question that might arise is whether we could replace the second LSTM layer with a fully connected layer that has the same number of neurons as **Num. units**. Indeed, this change can be done and yields roughly the same level of accuracy and loss. However, it's worth noting that this alternative architecture is less compact, as it necessitates roughly twice the number of trainable parameters compared to our original model. This alternative architecture is provided within the Colab notebook uploaded in the `Chapter05_06` folder on the GitHub repository.

Having trained the model, the following step involves its quantization to make it suitable for the microcontroller deployment.

In the upcoming recipe, we will employ the TensorFlow Lite converter to quantize the model to 8-bit and then evaluate its accuracy on the test dataset.

Evaluating the accuracy of the quantized model on the test dataset

After training the model using TensorFlow, we are ready to make it suitable for microcontroller deployment.

In this recipe, we will quantize the trained model to 8-bit using the TensorFlow Lite converter tool and then assess its accuracy with the test dataset. After evaluating the model's accuracy, we will use the xxd tool to convert the TensorFlow Lite model to a C-byte array, preparing it for deployment on the microcontroller.

Getting ready

Quantization is a pivotal technique in the ML world on microcontrollers because it makes the model storage-efficient and boosts its latency inference.

The quantization adopted in this book involves converting 32-bit floating-point numbers to 8-bit integers. While this technique offers a model reduction of four times and a latency improvement, we could lose accuracy because of the reduced numerical precision. For this reason, it is paramount to evaluate the quantized model's accuracy whenever we quantize the model.

Unfortunately, TensorFlow Lite lacks a built-in function to assess the model's accuracy in the same way we typically do in TensorFlow with the `evaluate()` method:

```
# Accuracy evaluation with TensorFlow
loss, accuracy = model.evaluate(x_test, y_test)
```

As a result, we need to implement a Python script in Colab to execute the model inference to check whether each input test sample has been correctly classified. Now, you might wonder how we run a TensorFlow Lite model in Python. The solution lies in the **Python TensorFlow Lite interpreter**. This Python interpreter offers an interface akin to what we use to run the model on the microcontroller, as we will see in the following *How to do it...* section.

How to do it...

Continue working in the Colab notebook, and follow these steps to quantize the model to 8-bit using the TensorFlow Lite converter, and evaluate its accuracy on the test dataset:

Step 1:

Select a few hundred samples randomly from the test dataset to calibrate the quantization:

```
def representative_data_gen():
    data = tf.data.Dataset.from_tensor_slices(x_test)
    for i_value in data.batch(1).take(100):
        i_value_f32 = tf.dtypes.cast(i_value, tf.float32)
        yield [i_value_f32]
```

Step 2:

Quantize the TensorFlow model to 8-bit using the TensorFlow Lite converter, while retaining the floating-point format for both the model's input and output:

```
converter = tf.lite.TFLiteConverter.from_saved_model(TF_MODEL)
converter.representative_dataset = tf.lite.
RepresentativeDataset(representative_data_gen)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_
INT8]

tfl_model = converter.convert()
```

Keeping the input in floating-point format enables us to directly feed the model with MFCCs, eliminating the need for an additional intermediate buffer. Instead, retaining the output in floating-point format removes the need to dequantize the output results.

Step 3:

Initialize the TensorFlow Lite interpreter:

```
interp = tf.lite.Interpreter(model_content=tfl_model)
```

The Python-based TensorFlow Lite interpreter works similarly to the one we used in tflite-micro. Therefore, it is employed to load the TensorFlow Lite models and execute them on the intended device, which, in this case, is the machine hosting the Colab environment.

Step 4:

Allocate the tensors, and get the input and output details:

```
interp.allocate_tensors()
```

```
i_details = interp.get_input_details()[0]
o_details = interp.get_output_details()[0]
```

Step 5:

Implement a function to run the model inference using the TensorFlow Lite Python interpreter. To do so, define a function called `classify` that accepts the test sample as an input argument:

```
def classify(i_value):
```

Inside the function, add a new dimension to the test sample at position 0 (`axis=0`) to align with the 3D tensor shape expected by the TensorFlow Lite model:

```
    i_value_f32 = np.expand_dims(i_value, axis=0)
```

This shape expansion is required because the LSTM layer, which is the first layer of the model, expects a 3D input tensor consisting of feature columns, time step rows, and a batch size dimension of one (`[1, time steps, features]`).

Then, copy the input sample into the model's input tensor using the `set_tensor()` method of the TensorFlow Lite interpreter:

```
    i_value_f32 = tf.cast(i_value_f32, dtype=tf.float32)
    interp.set_tensor(i_details["index"], i_value_f32)
```

Once the input tensor is initialized, invoke the model inference:

```
    interp.invoke()
```

Finally, reset the state of the TensorFlow Lite fused LSTM operator and return the output tensor:

```
    interp.reset_all_variables()

    return interp.get_tensor(o_details["index"])[0]
```

Using the `reset_all_variables()` method from the interpreter is crucial because the Python TensorFlow Lite fused LSTM operator is **stateful**. Being stateful implies that the LSTM internal variables are not reset and can influence the output of future model inferences. By calling the `reset_all_variables()` method, we clear the LSTM operator's internal state and ensure that each inference is not influenced by the previous one.

Step 6:

Evaluate the accuracy of the quantized TensorFlow Lite model. To do so, declare two variables to keep track of the number of correct classifications:

```
num_correct_samples = 0
```

Then, iterate through all the test samples:

```
for i_value, o_value in zip(x_test, y_test):
```

For each input test sample (*i_value*), run the model inference and return the index of the maximum value in the output tensor:

```
o_pred_f32 = classify(i_value)
o_res = np.argmax(o_pred_f32)
```

Then, check whether the classification result corresponds to the class of the input sample. If so, increment the *num_correct_samples* variable by one:

```
if o_res == o_value:
    num_correct_samples += 1
```

Finally, outside the for loop, print the proportion of accurately classified samples to the overall number of samples:

```
num_total_samples = len(x_test)
print("Accuracy:", num_correct_samples/num_total_samples)
```

Using our test dataset, the quantized model achieved an accuracy nearly identical to the TensorFlow one.

Step 7:

Convert the TensorFlow Lite model to a C-byte array with the xxd tool:

```
TFL_MODEL_FILE = 'model.tflite'
open(TFL_MODEL_FILE, "wb").write(tfl_model)

!apt-get update && apt-get -qq install xxd
!xxd -i $TFL_MODEL_FILE > model.h
!sed -i 's/unsigned char/const unsigned char/g' model.h
!sed -i 's/const/alignas(8) const/g' model.h
```

The command generates a C header file containing the TensorFlow Lite model as a constant `unsigned char` array.



Remember the final two lines where we employ the `sed` command to make the array containing the model as `alignas(8) const`. These changes guarantee that the model resides in the program memory (Flash) and that the array is aligned to the 8-byte boundary.

You can now download the `model.h` file from Colab's left pane and proceed with the first step toward model deployment: implementing the MFCCs feature extraction algorithm in C using the CMSIS-DSP library.

There's more...

In this recipe, we learned how to assess the accuracy of the quantized LSTM RNN using the TensorFlow Lite Python interpreter.

For those who have also implemented the many-to-one RNN architecture with TensorFlow, consider quantizing this model and evaluating its accuracy. Also, in this case, you should obtain an accuracy nearly identical to what is obtained with TensorFlow.

This recipe concludes the design phase of the ML model, paving the way to model deployment.

In the upcoming recipe, we will deal with the first ingredient required to run the trained model on the Raspberry Pi Pico: the C implementation of the MFCCs feature extraction algorithm using the CMSIS-DSP library.

Deploying the MFCCs feature extraction algorithm on the Raspberry Pi Pico

The final two recipes of this chapter will guide us through the development of the music genre classification application on the microcontroller.

In this particular recipe, we will deploy the MFCCs feature extraction algorithm using the CMSIS-DSP on the Raspberry Pi Pico.

Getting ready

Extracting MFCCs from raw audio data is the first stage of our computing chain to classify music genres. Since we developed this algorithm in Python using the CMSIS-DSP library, transitioning to a C language implementation should be relatively straightforward.

The C version of the CMSIS-DSP library mirrors its Python counterpart, offering the same functions and, often, an identical API, simplifying the migration to the final implementation on a board.

The only ingredients needed to deploy the MFCCs feature extraction algorithm on the Raspberry Pi Pico or other Arm-based microcontrollers are the following:

- The Arduino CMSIS-DSP library
- A Python method to convert the precomputed components of the MFCCs algorithm from NumPy arrays to constant C arrays
- A Python method to convert the Python constants of the MFCCs algorithm to C variables

To allow you to concentrate solely on the algorithm deployment, we have made the preceding components available in the `Chapter05_06` folder of the `TinyML-Cookbook_2E` GitHub repository.



If you are curious about how to build the Arduino CMSIS-DSP library from source code, you can refer to the following guide provided in the `TinyML-Cookbook_2E` GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Docs/build_arduino_cmsisdsp_lib.md.

Now, we are probably left with a challenge to address before jumping into the *How to do it...* section: how do we validate the functionality of the C algorithm?

The easiest way is to check whether the C implementation returns the same output as the Python counterpart.

How to do it...

Continue working in the Colab notebook, and follow these steps to transform the precomputed parts of the MFCCs algorithm to C arrays:

Step 1:

Copy into Colab the `to_c_array()` Python function provided at the following link: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Chapter05_06/PythonScripts/to_c_array.py.

This function aims to convert a NumPy array into a C header file containing an array, with the NumPy array's content and shape dimensions. The `to_c_array()` Python function needs the following input arguments to generate the C header file:

- The NumPy array to transform (`data`)

- The desired C data type as a string (`c_type`)
- The name of the output header file (`filename`)

Additionally, an optional fourth input argument (`num_cols`) allows you to adjust the number of array values written in a single row. Its default setting is 12.

Step 2:

Copy into Colab the `to_c_consts()` Python function provided at the following link: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Chapter05_06/PythonScripts/to_c_consts.py.

This function aims to convert a list of Python variables into C constants stored in a C header file.

The `to_c_const()` Python function needs the following input arguments to generate the C header file:

- The list of Python variables to convert (`data`)
- The name of the output header file (`filename`)

Each entry in the Python list consists of a three-element tuple containing the following information:

- The Python variable to convert
- The name of the C variable as a string
- The desired C data type as a string (`c_type`)

After calling this function, all Python variables will be converted to C constants and stored in the same header file.

Step 3:

Generate the C arrays for all the precomputed components of the MFCCs feature extraction algorithm:

```
to_c_array(hann_lut_q15, 'int16_t',
           'hann_lut_q15')
to_c_array(mel_wei_mtx_q15.T, 'int16_t',
           'mel_wei_mtx_q15_T')
to_c_array(log_lut_q13_3, 'int16_t',
           'log_lut_q13_3')
```

```
to_c_array(dct_wei_mtx_q15.T, 'int16_t',
           'dct_wei_mtx_q15_T')
```

The `mel_wei_mtx_q15` and `dct_wei_mtx_q15` arrays are transposed to avoid transposing them when running the MFCCs algorithm on the microcontroller.

Step 4:

Generate the C arrays for the Q15 input audio test (`test_ad`) and the expected MFCCs in floating-point format (`mfccs_cmsis`):

```
test_src_q15 = dsp.arm_float_to_q15(test_ad)

to_c_array(test_src_q15, 'int16_t', 'test_src')
to_c_array(mfccs_cmsis, 'float', 'test_dst')
```

The `test_src` and `test_dst` arrays hold the test audio sample and the corresponding MFCCs. These arrays will be used in the Arduino sketch to verify the proper functionality of the MFCCs feature extraction algorithm.

Step 5:

Generate the C constants required by the MFCCs feature extraction algorithm:

```
NUM_FRAMES = int(((TRAIN_AUDIO_LENGTH_SAMPLES - FRAME_LENGTH) / FRAME_
STEP) + 1)
NUM_FFT_FREQS = int((FFT_LENGTH / 2) + 1)

vars = [
    (FRAME_LENGTH, 'FRAME_LENGTH', 'int32_t'),
    (FRAME_STEP, 'FRAME_STEP', 'int32_t'),
    (NUM_FRAMES, 'NUM_FRAMES', 'int32_t'),
    (FFT_LENGTH, 'FFT_LENGTH', 'int32_t'),
    (NUM_FFT_FREQS, 'NUM_FFT_FREQS', 'int32_t'),
    (NUM_MEL_FREQS, 'NUM_MEL_FREQS', 'int32_t'),
    (NUM_MFCCS, 'NUM_MFCCS', 'int32_t')
]

to_c_consts(vars, 'mfccs_consts')
```

Now, you have everything you need to deploy the MFCCs feature extraction algorithm on the microcontroller.

Therefore, download all the generated C header files (`mfccs_consts.h`, `dct_wei_mtx_q15_T.h`, `hann_lut_q15.h`, `log_lut_q13_3.h`, `mel_wei_mtx_q15_T.h`, `test_src.h`, `test_dst.h`) from Colab's left pane and open the Arduino IDE.

In the Arduino IDE, create a new sketch and follow the following steps to run the MFCCs feature extraction algorithm on the microcontroller:

Step 1:

Import all the generated C header files in the Arduino IDE. As shown in *Figure 6.16*, click on the **Tab** button with the upside-down triangle and click on **Import File into Sketch**:

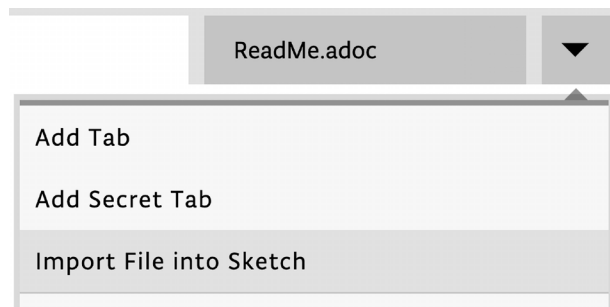


Figure 6.16: Tab button in the Arduino IDE to import the C header files

Step 2:

Download the Arduino CMSIS-DSP library from the TinyML-Cookbook_2E GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/ArduinoLibs/Arduino_CMSIS_DSP.zip.

After downloading the ZIP file, import it into the Arduino IDE.

Step 3:

In the sketch, include the `arm_math.h` header file to access the functions and data types provided by the CMSIS-DSP library:

```
#include "arm_math.h"
```

Then, include the C header files required by the MFCCs feature extraction algorithm:

```
#include "mfccs_consts.h"
#include "dct_wei_mtx_q15_T.h"
#include "hann_lut_q15.h"
#include "log_lut_q13_3.h"
#include "mel_wei_mtx_q15_T.h"
#include "test_src.h"
#include "test_dst.h"
```

Step 4:

Create a C++ class called MFCC_Q15 to implement the MFCCs feature extraction algorithm:

```
class MFCC_Q15 {
public:
    // Public members
private:
    // Private members
};
```

Inside the private section, declare the two Q15 (q15_t) buffers required for the MFCCs calculation, the CMSIS-DSP instances for the RFFT (arm_rfft_instance_q15), and the Mel-weight and DCT-weight matrices (arm_matrix_instance_q15):

```
// Private members
q15_t          _bufA[FRAME_LENGTH];
q15_t          _bufB[FRAME_LENGTH * 2];
arm_rfft_instance_q15 _rfft_inst;
arm_matrix_instance_q15 _mel_wei_mtx_inst;
arm_matrix_instance_q15 _dct_wei_mtx_inst;
```

As we know from the first part of this project, the MFCCs calculation involves a series of steps that require storing intermediate results. In this processing flow, each operation has one input and one output. As a result, only two buffers are necessary (_bufA and _bufB), specifically designed to accommodate the two intermediate results that require the most memory. By analyzing the MFCCs Python implementation with the CMSIS-DSP library, it should be evident that the two largest intermediate buffers are needed for the RFFT function (arm_rfft_q15), as the input has FRAME_LENGTH values, and the output has twice the values of the input (FRAME_LENGTH * 2).

Besides the intermediate buffers, the private section also keeps the CMSIS-DSP instances for the RFFT function (`arm_rfft_instance_q15`) and the matrices with the Mel-weight and DCT-weight (`arm_matrix_instance_q15`). These instances are C structures, essential for setting up the compute parameters of the RFFT and the matrix-by-vector functions. The initialization of these data structures will be done only once and within the default constructor.

Step 5:

In the public section of the MFCC_Q15 class, implement the default constructor to initialize the CMSIS-DSP instances:

```
// Public members
MFCC_Q15() {
    // RFFT instance
    arm_rfft_init_q15(&rfft_inst, FFT_LENGTH, 0, 1);

    // Mel-weight matrix instance
    _mel_wei_mtx_inst.numRows = mel_wei_mtx_q15_T_dim1;
    _mel_wei_mtx_inst.numCols = mel_wei_mtx_q15_T_dim0;
    _mel_wei_mtx_inst.pData =
        (q15_t*)&mel_wei_mtx_q15_T_data[0];

    // DCT-weight matrix instance
    _dct_wei_mtx_inst.numRows = dct_wei_mtx_q15_T_dim1;
    _dct_wei_mtx_inst.numCols = dct_wei_mtx_q15_T_dim0;
    _dct_wei_mtx_inst.pData =
        (q15_t*)&dct_wei_mtx_q15_T_data[0];
}
```

The RFFT (`arm_rfft_q15`) and matrix-by-vector multiplication (`arm_mat_vec_mult_q15`) CMSIS-DSP functions require an instance to know how to perform the computation.

The RFFT instance is initialized the same way we did using the CMSIS-DSP Python function. However, the initialization of `arm_matrix_instance_q15` is a fresh step for us, as it was not necessary for the `arm_mat_vec_mult_q15()` function in the Python context. This instance is essential in C because the `arm_mat_vec_mult_q15()` function cannot directly determine the shape of the input arguments from the data pointer alone.

Hence, this function requires that the matrix (the left-hand side operand) be provided along with this instance, which carries the matrix's dimension information (rows and columns) and the data pointer. From this data structure, the function can infer the shape of both operands.

It is worth noting that these instances are initialized only once, when the MFCC_Q15 object is created, because these data structures do not change at runtime.

Step 6:

In the public section of the MFCC_Q15 class, implement a method to extract MFCCs from raw audio in Q15 format. To do so, define a method called `run` that accepts the data pointers to the raw audio (`src`) and the output tensor (`dst`) as input arguments:

```
void run(const q15_t* src, float* dst) {
```

Inside the `run()` function, use a `for` loop to iterate over each frame:

```
for(int i = 0; i < NUM_FRAMES; ++i ) {
```

For each frame, apply the Hann window on the input frame using the element-wise multiplication function (`arm_mult_q15`) and store the result in the `_bufA` buffer:

```
    arm_mult_q15(&src[i * FRAME_STEP],
                hann_lut_q15_data,
                _bufA,
                FRAME_LENGTH);
```

The element-wise multiplication is performed between the precomputed Hann window coefficients (`hann_lut_q15_data`) and the `FRAME_LENGTH` input values belonging to a single frame.

Once you have applied the Hann window, calculate the FFT and store the result in the `_bufB` buffer:

```
    arm_rfft_q15(&rfft_inst, _bufA, _bufB);
```

Then, calculate the magnitude and store the result in the `_bufA` buffer:

```
    _bufA[0] = _bufB[0];
    _bufA[NUM_FFT_FREQS - 1] = _bufB[1];
    arm_cmplx_mag_q15(&_bufB[2],
                     &_amp;_bufA[1],
                     NUM_FFT_FREQS - 2);
```

After calculating the FFT magnitude, perform the Mel-scale conversion with the `arm_mat_vec_mult_q15()` CMSIS-DSP function and store the result in the `_bufB` buffer:

```
arm_mat_vec_mult_q15(&_mel_wei_mtx_inst,
                    _bufA,
                    _bufB);
```

The only difference between the CMSIS-DSP Python and C APIs is in the matrix instance for the left-hand side operand, which wasn't necessary in Python.

Here, the transposition of the Mel-weight matrix is not required because this operation has already been done in Python, when its corresponding C header file was generated.

Once you have performed the Mel-scale conversion, apply the logarithmic function using the precomputed lookup table and store the result in the `_bufA` buffer:

```
for(int idx = 0; idx < NUM_MEL_FREQS; ++idx) {
    int16_t val = (int16_t)_bufB[idx];
    _bufA[idx] = log_lut_q13_3_data[val];
}
```

Then, calculate the MFCCs with the `arm_mat_vec_mult_q15()` CMSIS-DSP function and store the result in the `_bufB` buffer:

```
arm_mat_vec_mult_q15(&_dct_wei_mtx_inst,
                    _bufA,
                    _bufB);
```

Finally, convert the MFCCs to floating-point and store the result in the output buffer (`dst`):

```
for(int k = 0; k < NUM_MFCCS; ++k) {
    dst[k + i * NUM_MFCCS] = _bufB[k] / (float)(8);
}
```

In contrast to the Python version, the MFCCs buffer (`_bufB`) doesn't need transposition because the vector occupies a continuous memory block, whether in a **row-vector** or **column-vector** form. As a result, transposition does not alter the actual sequence of values in memory.

Once you have stored the MFCCs in the output buffer, you can process a new frame or exit the for loop if all frames have been processed:

```
    } // for(int i = 0; i < NUM_FRAMES; ++i )
} // run()
```

Step 7:

Define a global MFCC_Q15 object called `mfccs`:

```
MFCC_Q15 mfccs;
```

When creating this object, the constructor of the `MFCC_Q15` class gets invoked, initializing the CMSIS-DSP instances.

Step 8:

Define an array to hold the extracted MFCCs from all frames of the input audio sample:

```
#define DST_SIZE (NUM_FRAMES * NUM_MFCCS)
float dst[DST_SIZE];
```

Step 9:

In the `setup()` function, initialize the serial peripheral with a 115200 baud rate:

```
void setup() {
    Serial.begin(115200);
    while (!Serial);
}
```

The serial peripheral will be employed to communicate whether the MFCCs feature extraction has been successfully executed.

Step 10:

In the `loop()` function, extract the MFCCs from the input test sample stored in the `test_src.h` header file:

```
void loop() {
    mfccs.run(test_src_data, dst);
```

Then, check whether the output returned by the preceding function (`dst`) matches the expected result:

```
for(int i = 0; i < DST_SIZE; ++i) {
    float v0 = dst[i];
    float v1 = test_dst_data[i];
    if(abs(v0 - v1) > 0.125f) {
```

```
        Serial.println("TEST FAILED");
        while(1);
    }
}

Serial.println("TEST PASSED");
while(1);
```

As you can see from the preceding code snippet, we iterate through all the values stored in the `dst` array to check whether their expected result in the `test_dst_data` array differs more than 0.125 in absolute terms from them. If even a single value has a deviation greater than this threshold, we transmit the `TEST FAILED` message. Otherwise, we communicate the `TEST PASSED` message to indicate the successful execution of the algorithm.

The reason why we chose the 0.125 threshold rather than 0 is that the Python and C CMSIS-DSP libraries are not bit-accurate with each other, due to different rounding methods when computing the fixed-point calculations. Since the rounding can only flip the least significant bit of a fixed-point number, this 1-bit difference translates to the precision of the Q13.3 format of the MFCCs, which corresponds to $0.125 (2^{-3})$.

Now, compile and upload the sketch on the Raspberry Pi Pico. Once the device is ready, open the serial terminal. After a few seconds, you should see the `TEST PASSED` message, indicating the successful implementation of the MFCCs feature extraction algorithm!

At this point, the only step left is deploying the model on the Raspberry Pi Pico.

There's more...with the SparkFun Artemis Nano!

In this recipe, we learned how to implement the MFCCs feature extraction algorithm in C using the CMSIS-DSP library.

The same sketch is compatible with any Arm-based microcontrollers programmed in the Arduino IDE, thanks to the CMSIS-DSP library. However, the benefit of the CMSIS-DSP library is not limited to code portability. In fact, the library can leverage specific optimized routines depending on the target platform as well. Therefore, what do you think about testing the MFCCs feature extraction algorithm on the SparkFun Artemis Nano microcontroller and evaluating its latency performance?

To accomplish this task, you should do the following:

1. Create a new Arduino project.
2. Import the Arduino CMSIS-DSP library in the Arduino IDE.

3. Ensure you have commented out `libs.extra`, and remove it from `libs.all` in the `platform.txt` file, as reported in the *Setting up the SparkFun Artemis Nano board in the Arduino IDE* guide (https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Docs/setup_sparkfun_artemis_nano.md). This step is necessary to avoid compilation errors when building a sketch based on the CMSIS-DSP library.
4. Tweak the sketch to profile the `mfcc.run()` execution.

The `mfcc.run()` execution can be easily profiled using the `mbed::Timer` interface (<https://os.mbed.com/docs/mbed-os/v6.16/apis/timer.html>). A demonstration of how to employ this interface to profile a generic function called `func()` is provided in the following code snippet:

```
#include "mbed.h"

mbed::Timer timer;
timer.start();

// Function to profile
func();

auto t0 = timer.elapsed_time();

using std::chrono::duration_cast;
using std::chrono::microseconds;
auto t_diff = duration_cast<microseconds>(t0);
```

The solution for this exercise is available in the `Chapter05_06` folder on the GitHub repository.

The compilation process on the SparkFun Artemis Nano might take several minutes, primarily due to the CMSIS-DSP library being compiled. Nevertheless, once you have compiled successfully for the first time, the Arduino IDE will cache the binaries, ensuring faster subsequent builds.

On the SparkFun Artemis Nano, extracting the MFCCs should take approximately 250ms, whereas on the Raspberry Pi Pico, it should be around 590ms. Therefore, the SparkFun Artemis Nano is more than twice as fast as the Raspberry Pi Pico. This performance difference lies in the specialized **digital signal processing (DSP)** instructions on the Arm Cortex-M4 CPU of the SparkFun Artemis Nano, which are unavailable on the Arm Cortex-M0+ CPU of the Raspberry Pi Pico.

Now that we have deployed the MFCCs feature extraction algorithm on the microcontroller, we are ready to complete our project.

In the upcoming final recipe, we will finalize the application to classify music genres from 1-second audio, recorded with the external microphone connected to the Raspberry Pi Pico.

Recognizing music genres with the Raspberry Pi Pico

Here we are, ready to finalize our application on the Raspberry Pi Pico.

In this final recipe, we will deploy the TensorFlow Lite model to recognize the music genre from audio clips recorded with the microphone connected to the microcontroller.

Getting ready

The application we will design in this recipe aims to continuously record a 1-second audio clip and run the model inference, as illustrated in the following image:

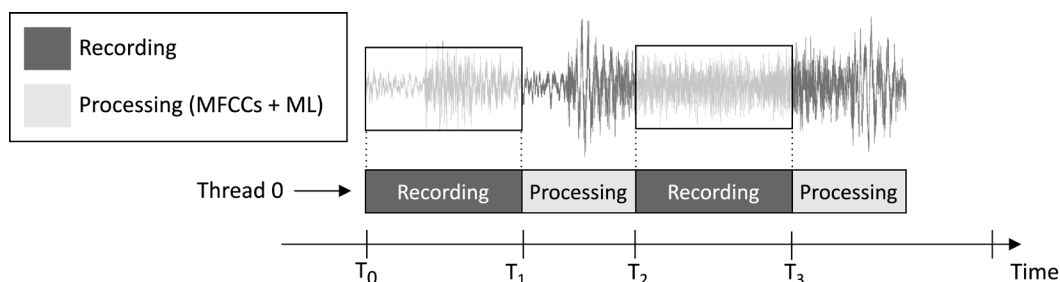


Figure 6.17: Recording and processing tasks running sequentially

From the task execution timeline shown in the preceding image, you can observe that the feature extraction and model inference are always performed after the audio recording and not concurrently. Therefore, it is evident that we do not process some segments of the live audio stream.

Unlike a real-time **keyword spotting (KWS)** application, which should capture and process all pieces of the audio stream to never miss any spoken word, here, we can relax this requirement because it does not compromise the effectiveness of the application.

As we know, the input of the MFCCs feature extraction is the 1-second raw audio in Q15 format. However, the samples acquired with the microphone are represented as 16-bit integer values. Hence, *how do we convert the 16-bit integer values to Q15?* The solution is more straightforward than you might think: converting the audio samples is unnecessary.

To understand why, consider the Q15 fixed-point format. This format can represent floating-point values within the $[-1, 1]$ range. Converting from floating-point to Q15 involves multiplying the floating-point values by 32,768 (2^{15}). Nevertheless, because the floating-point representation originates from dividing the 16-bit integer sample by 32,768 (2^{15}), it implies that the 16-bit integer values are inherently in Q15 format.

How to do it...

Take the breadboard with the electronic circuit built in the previous chapter, with the microphone attached to the Raspberry Pi Pico. Disconnect the data cable from the microcontroller, and remove the push-button and its connected jumpers from the breadboard, as they are not required for this recipe. *Figure 6.18* shows what you should have on the breadboard:

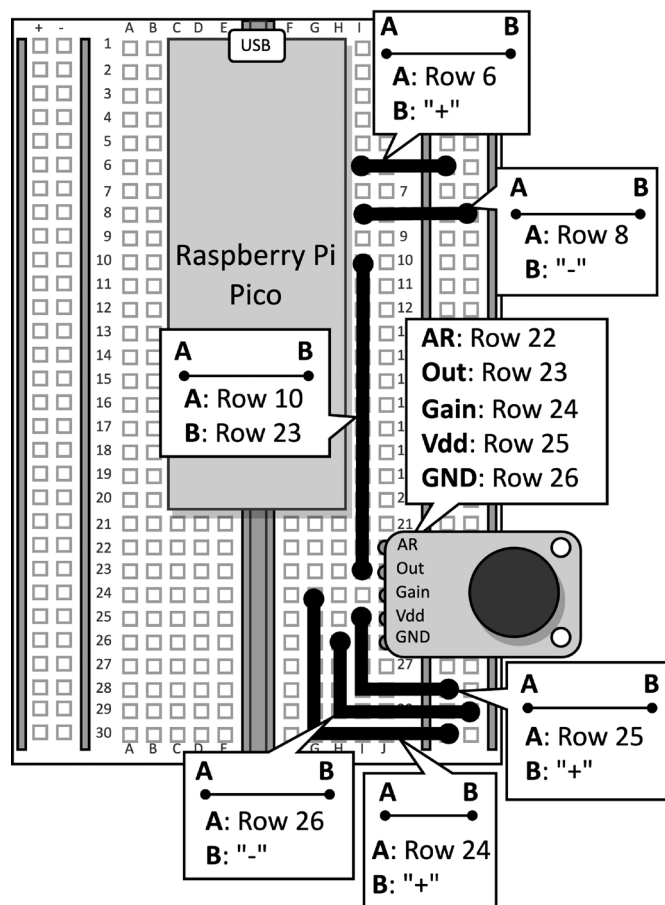


Figure 6.18: The electronic circuit built on the breadboard

After removing the push-button from the breadboard, open the Arduino IDE and create a new sketch.

Now, follow the following steps to develop the music genre recognition application on the Raspberry Pi Pico:

Step 1:

Download the Arduino TensorFlow Lite library from the TinyML-Cookbook_2E GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/ArduinoLibs/Arduino_TensorFlowLite.zip.

After downloading the ZIP file, import it into the Arduino IDE.

Step 2:

Import all the generated C header files required for the MFCCs feature extraction algorithm in the Arduino IDE, excluding `test_src.h` and `test_dst.h`.

Step 3:

Copy the sketch developed in the previous recipe for implementing the MFCCs feature extraction, excluding the `setup()` and `loop()` functions.

Remove the inclusion of the `test_src.h` and `test_dst.h` header files. Then, remove the allocation of the `dst` array, as the MFCCs will be directly stored in the model's input.

Step 4:

Copy the sketch developed in *Chapter 5, Recognizing Music Genres with TensorFlow and the Raspberry Pi Pico – Part 1*, to record audio samples with the microphone, excluding the `setup()` and `loop()` functions.

Once you have imported the code, remove any reference to the LED and the push-button, as they are no longer required. Then, change the definition of `AUDIO_LENGTH_SEC` to record audio lasting 1 second:

```
#define AUDIO_LENGTH_SEC 1
```

Step 5:

Import the header file containing the TensorFlow Lite model (`model.h`) into the Arduino project.

Once the file has been imported, include the `model.h` header file in the sketch:

```
#include "model.h"
```

Step 5:

Include the necessary header files for `tf-lite-micro`:

```
#include <TensorFlowLite.h>
#include <tensorflow/lite/micro/all_ops_resolver.h>
#include <tensorflow/lite/micro/micro_interpreter.h>
#include <tensorflow/lite/micro/micro_log.h>
#include <tensorflow/lite/micro/system_setup.h>
#include <tensorflow/lite/schema/schema_generated.h>
```

The header files are the same ones described in *Chapter 3, Building a Weather Station with TensorFlow Lite for Microcontrollers*.

Step 6:

Declare global variables for the `tf-lite-micro` model and interpreter:

```
const tf::Model* tflu_model = nullptr;
tf::MicroInterpreter* tflu_interpreter = nullptr;
```

Then, declare the TensorFlow Lite tensor objects (`TfLiteTensor`) to access the input and output tensors of the model:

```
TfLiteTensor* tflu_i_tensor = nullptr;
TfLiteTensor* tflu_o_tensor = nullptr;
```

The global variables declared in this step are those described in *Chapter 3, Building a Weather Station with TensorFlow Lite for Microcontrollers*.

Step 7:

Declare a buffer (tensor arena) to store the intermediate tensors used during the model execution:

```
constexpr int tensor_arena_size = 16384;
uint8_t tensor_arena[tensor_arena_size] __attribute__((aligned(16)));
```

The size of the tensor arena has been determined through empirical testing, as the memory needed for the intermediate tensors varies, depending on how the LSTM operator is implemented underneath. Our experiments on the Raspberry Pi Pico found that the model only requires 16 KB of RAM for inference.

Step 8:

In the `setup()` function, initialize the serial peripheral with a 115200 baud rate:

```
Serial.begin(115200);  
while (!Serial);
```

The serial peripheral will be employed to transmit the recognized music genre over the serial communication.

Step 9:

In the `setup()` function, load the TensorFlow Lite model stored in the `model.h` header file:

```
tflu_model = tflite::GetModel(model_tflite);
```

Then, register all the DNN operations supported by tflite-micro, and initialize the tflite-micro interpreter:

```
tflite::AllOpsResolver tflu_ops_resolver;  
  
static tflite::MicroInterpreter static_interpreter(  
    tflu_model,  
    tflu_ops_resolver,  
    tensor_arena,  
    tensor_arena_size);  
tflu_interpreter = &static_interpreter;
```

Step 10:

In the `setup()` function, allocate the memory required for the model, and get the memory pointer of the input and output tensors:

```
tflu_interpreter->AllocateTensors();  
tflu_i_tensor = tflu_interpreter->input(0);  
tflu_o_tensor = tflu_interpreter->output(0);
```

Step 11:

In the `setup()` function, use the Raspberry Pi Pico SDK to initialize the ADC peripheral:

```
adc_init();
adc_gpio_init(26);
adc_select_input(0);
```

Step 12:

In the `loop()` function, prepare the model's input. To do so, record an audio clip for 1 second:

```
// Reset audio buffer
buffer.cur_idx = 0;
buffer.is_ready = false;

constexpr uint32_t sr_us = 1000000 / SAMPLE_RATE;
timer.attach_us(&timer_ISR, sr_us);

while(!buffer.is_ready);

timer.detach();
```

After recording the audio, extract the MFCCs:

```
mfccs.run((const q15_t*)&buffer.data[0],
          (float *)&tflu_i_tensor->data.f[0]);
```

As you can see from the preceding code snippet, the MFCCs will be stored directly in the model's input.

Step 13:

Run the model inference, and return the classification result over the serial communication:

```
tflu_interpreter->Invoke();

size_t ix_max = 0;
float pb_max = 0;
for (size_t ix = 0; ix < 3; ix++) {
    if(tflu_o_tensor->data.f[ix] > pb_max) {
        ix_max = ix;
    }
}
```

```
        pb_max = tflu_o_tensor->data.f[ix];
    }
}

const char *label[] = {"disco", "jazz", "metal"};

Serial.println(label[ix_max]);
```

Now, plug the micro-USB data cable into the Raspberry Pi Pico. Once you have connected it, compile and upload the sketch on the microcontroller.

Afterward, open the serial monitor in the Arduino IDE and place your smartphone near the microphone to play a disco, jazz, or metal song. The application should now recognize the song's music genre and display the classification result in the serial monitor!

There's more...with the SparkFun Artemis Nano!

In this final recipe, we learned how to deploy a trained model for music genre classification on the Raspberry Pi Pico, using `tfllite-micro`.

If you have successfully recorded audio clips using the PDM microphone on the SparkFun Artemis Nano, you might consider adapting the sketch just implemented for this microcontroller board. What you need to do is remove the references to the Raspberry Pi Pico's ADC peripheral and modify the code to capture audio samples with the built-in microphone on the SparkFun Artemis Nano.

In the `Chapter05_06` folder on the GitHub repository, we have uploaded the sketch skeleton to guide your code changes.

Summary

In this chapter, we have completed our music genre recognition application on the Arduino Nano and Raspberry Pi Pico. Here, our focus has been optimizing the MFCCs feature extraction algorithm through software optimizations, using fixed-point arithmetic. Additionally, we have developed, trained, and deployed an RNN model that relies on the LSTM operator to classify music genres.

Our practical journey started by getting acquainted with fixed-point arithmetic and its role in accelerating the extraction of MFCCs from audio clips. To achieve this objective, the Python CM-SIS-DSP played a crucial role, helping us to develop an algorithm in a Python environment that closely resembles the final implementation on the microcontroller.

Following the implementation of the MFCCs feature extraction, our attention shifted to model design. Here, we designed and trained an RNN model based on the LSTM layer with TensorFlow to classify music genres.

Once the model was trained, we quantized it to 8-bit using the TensorFlow Lite converter and assessed its accuracy on the test dataset.

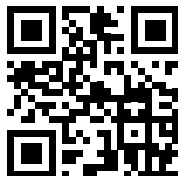
Finally, we developed an Arduino sketch to test the MFCCs algorithm, which was implemented using the C CMSIS-DSP library on the Raspberry Pi Pico. Then, we extended the sketch to deploy the trained model using tflite-micro.

With the conclusion of this chapter, we can close the part related to tinyML with audio sensors on microcontrollers. So, what's next? In the upcoming chapter, we will explore vision sensors by implementing an object detection application with Edge Impulse.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



7

Detecting Objects with Edge Impulse Using FOMO on the Raspberry Pi Pico

Undoubtedly, **image classification** is one of the most fundamental and well-known tasks in **machine learning (ML)** and **computer vision**. This task is crucial to many applications across domains such as medical diagnosis, autonomous vehicles, security surveillance, and entertainment to automate visual data analysis.

However, we only get insights into what we have in the image when it comes to image classification. In fact, if we want to figure out where objects are located within the scene, especially if there's more than one, we need to turn to a more advanced technique in computer vision known as **object detection**.

Object detection is more computationally intensive than image classification because it involves performing the classification task in multiple regions and scales of the input image simultaneously to identify the objects and their respective locations.

Despite this challenge, this chapter will showcase the successful deployment of an object detection application on microcontrollers using **Edge Impulse** and the **Faster Objects, More Objects (FOMO)** ML algorithm.

Our objective will be to develop an application capable of detecting cans. The chapter will begin with dataset preparation, demonstrating how to acquire images with a webcam and label them in Edge Impulse.

Next, we will design an ML model based on the FOMO algorithm. In this part, we will explore the architectural features of this novel ML solution that allows us to deploy object detection on highly constrained devices.

Finally, we will test the model using the Edge Impulse **Live classification** tool and implement the cans detection application for the **Raspberry Pi Pico**.

The Raspberry Pi Pico will not be connected to any camera module. Instead, the microcontroller will receive images captured with a webcam connected to the computer through the serial interface.

This chapter is intended to show how to develop an object detection application using Edge Impulse and get familiar with the leading techniques behind FOMO that make this algorithm suitable for highly constrained devices.

In this chapter, we're going to cover the following recipes:

- Acquiring images with the webcam
- Designing the Impulse's pre-processing block
- Transfer learning with FOMO
- Evaluating the model's accuracy
- Using OpenCV and pySerial to send images over the serial interface
- Reading data from the serial port with Arduino-compatible platforms
- Deploying FOMO on the Raspberry Pi Pico



The source code and additional material are available in the Chapter07 folder on the GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Chapter07.

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- A Raspberry Pi Pico
- A SparkFun RedBoard Artemis Nano (optional)
- A micro-USB data cable

- A USB-C data cable (optional)
- A USB webcam
- Laptop/PC with either Linux, macOS, or Windows
- Edge Impulse account

Acquiring images with the webcam

In this first recipe, we will prepare the dataset by acquiring images of cans using the USB webcam and labeling them using Edge Impulse.

Getting ready

As with all ML projects, our first step is dataset preparation. In this project, the dataset will be created from scratch using Edge Impulse.

The data acquisition for images does not differ from what we experienced in *Chapter 4, Using Edge Impulse and Arduino Nano to Control LEDs with Voice Commands*, for building the KWS dataset.

To prevent overfitting and improve model robustness, you should create a more diverse training dataset. For example, you might consider the following to ensure that the model is reliable and effective in real-world scenarios:

- Different backgrounds
- Different distances from the camera
- Different angles
- Different light conditions



We recommend using a USB webcam rather than the laptop's built-in one because capturing images from various angles and distances is more practical.

It is not necessary to take one can per image sample. You have the flexibility to capture as many cans as you want in a single image because, thanks to Edge Impulse, you can crop and label them individually during the labeling process.

However, how many samples do we need for training our model? Let's uncover this aspect in the upcoming subsection.

Building a dataset for FOMO

Two factors mainly influence the number of samples to acquire for training a model:

- The nature of the problem we aim to solve
- The ML technique used for training the model

Regarding the second point, employing a **transfer learning** algorithm for a computer vision task can dramatically reduce the required training samples. The idea behind this technique is to exploit the learned features of a model already trained on a large dataset for addressing a problem similar to its original intent.



Transfer learning will be discussed in *Chapter 8, Classifying Desk Objects with TensorFlow and the Arduino Nano*.

In Edge Impulse, we can employ transfer learning for object detection using a pre-trained model tailored for memory-constrained devices called FOMO.

As a result, we won't need thousands of images that we would typically require for training a computer vision model from scratch but rather something in the order of 30 to 50.

How to do it...

Open Edge Impulse and create a new project called `object_detection`. Then, follow the next steps to acquire image data with the USB webcam connected to your computer:

Step 1:

In the Edge Impulse **Dashboard** section, go to the **Project info** section and select **Bounding boxes (object detection)** for **Labeling method**:

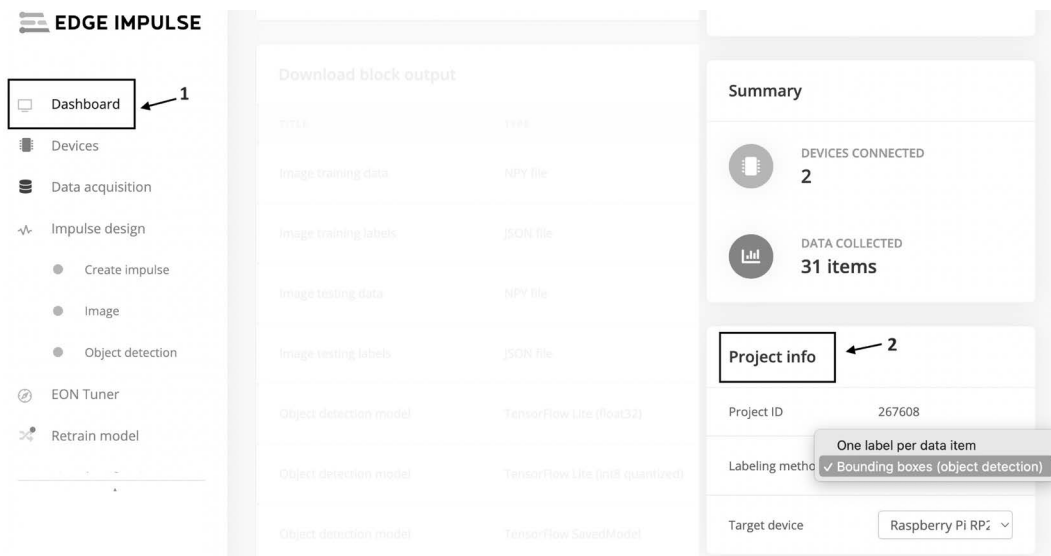


Figure 7.1: Labeling method is in the Project info section

Step 2:

In the Edge Impulse **Dashboard** section, click on the **Collect new data** option:

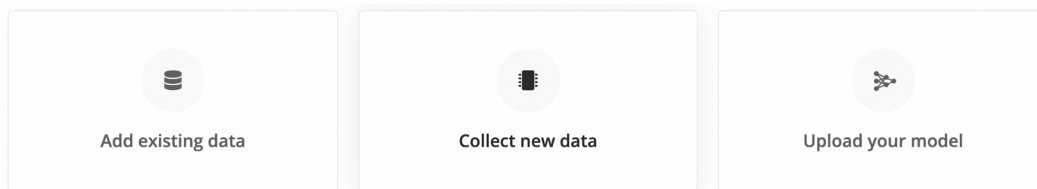


Figure 7.2: Click on the Collect new data button to start acquiring data

Edge Impulse will open a pop-up window asking how you want to collect the data. Select the **Connect to your computer** option to start capturing images from your laptop:

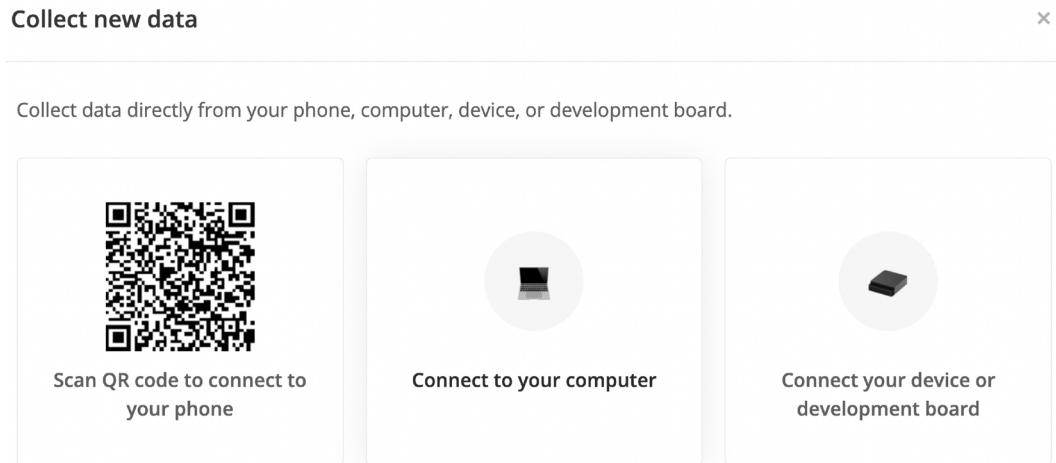



Figure 7.3: Click on the **Connect to your computer** button to start acquiring image data from your computer

After clicking on the button, a new page in the web browser will be opened. On the new page, select the **Collecting images** option to acquire images with the webcam.



In principle, using the smartphone's camera for dataset creation is possible. Nevertheless, integrating the smartphone camera introduces additional complexity when supplying input images to the microcontroller. Therefore, in the interest of maintaining the project's simplicity, we have opted for the use of a USB webcam.

Step 3:

Click on the **Give access to the camera** button to give permission to acquire images with the webcam:

Permission required

Give access to the camera

Figure 7.4: The **Give access to the camera** button

After clicking the button, you should see the camera preview. Enter the label (for example, can), and you are set to acquire the images for the dataset.

Step 4:

Use the **Capture** button to acquire at least 30 images with cans, ensuring the backgrounds, zoom levels, and angles are varied. Feel free to take pictures with multiple cans, as we can crop them later during labeling. The images will be automatically uploaded in the Edge Impulse **Dataset** section.

Step 5:

In the Edge Impulse **Dataset** section, click on the **Labeling queue** button:



Figure 7.5: Labeling queue button

Then, use your mouse to drag a rectangular box around each can in the sample image and label them as cans. After labeling, click the **Save labels** button to proceed to the following sample. Edge Impulse should be able to automatically put the bounding box around the cans of the subsequent images. As you can guess, this tool is AI-powered to speed up the labeling process. If required, make any adjustments to the bounding boxes to ensure accurate labeling. Once you are happy with the adjustments, click the **Save labels** button again to proceed with the following sample. Continue this process until you have labeled all the images in the queue.

If you go to the Edge Impulse **Dataset** section, you should now see all samples with labels:

Training (23) Test (17) ⌵ ☒ ⌵

SAMPLE NAME	LABELS	ADDED	
can.489ndmcj	can, can	Today, 01:17:24	⋮
can.489ndku5	can, can	Today, 01:17:22	⋮
can.489ndd35	can, can	Today, 01:17:14	⋮

Figure 7.6: Samples with labels

Now, go to the Edge Impulse **Dashboard** section to rebalance the dataset. To perform this operation, click on the **Perform train / test split** button in **Danger zone**:

Danger zone

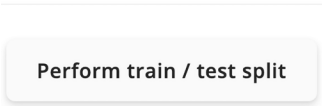


Figure 7.7: The rebalancing dataset button

After this operation, the dataset should have roughly 80% of samples for training and 20% for testing the model (**test dataset**).

There’s more...

In this recipe, we learned how to acquire images with the webcam and label them using Edge Impulse. However, as mentioned in the *How to do it...* section, you can also use the smartphone’s camera for this task. To do so, click on the **Scan QR code to connect to your phone** button in the **Collect new data** section, and use your smartphone to scan the **Quick Response (QR)** code displayed on the screen. Then, your device will be paired with Edge Impulse, and you can begin acquiring images immediately with your phone.

Once we have prepared the dataset, we are ready to proceed with the model’s design phase.

In the upcoming recipe, we will start this phase by preparing the pre-processing stage.

Designing the Impulse's pre-processing block

Now that we have the dataset ready, let's delve into the Impulse preparation.

In this recipe, our initial focus will be on the design of the pre-processing block responsible for preparing the input image feeding the ML model.

Getting ready

The object detection pipeline we are going to build with the help of Edge Impulse is the following:

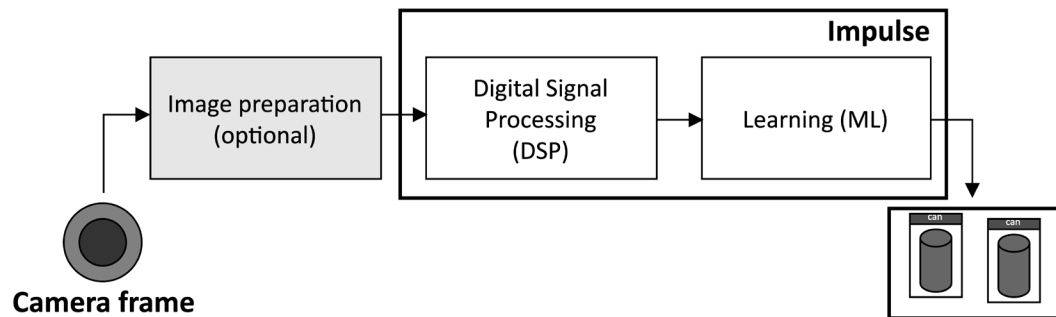


Figure 7.8: The object detection pipeline

The image preparation block reported in the preceding figure encompasses all the essential operations required to transform the camera frame into the expected input format for the **Learning** block.

In this project, we won't use any camera module with the microcontroller. Instead, we will emulate the functionality of an actual camera by transmitting the frames captured with the webcam from the computer over the serial connection.

To avoid the image preparation stage reported in *Figure 7.8* and make the implementation of the object detection pipeline simpler, we can assume the following for the camera frame:

- The image resolution of the camera frame matches the resolution of Impulse's input.
- The camera frame is in the same color format as the Impulse's input.

However, what input image resolution and color format should we consider for our problem?

Choosing the correct input image resolution

Since the content of the input image changes at runtime, it is evident that it must be stored in data memory (**SRAM**) and not in the program memory (**Flash**). The SRAM capacity of Raspberry Pi Pico is 264 Kbytes, so we need to carefully choose the correct input image resolution to avoid exceeding this limit.

Considering the pipeline illustrated in *Figure 7.8* and omitting the **Image preparation** block, there are three significant memory chunks contributing to the SRAM utilization: the camera frame (**Camera**), the input model (**Input**), and the ML model's temporary tensors (**Temp**):

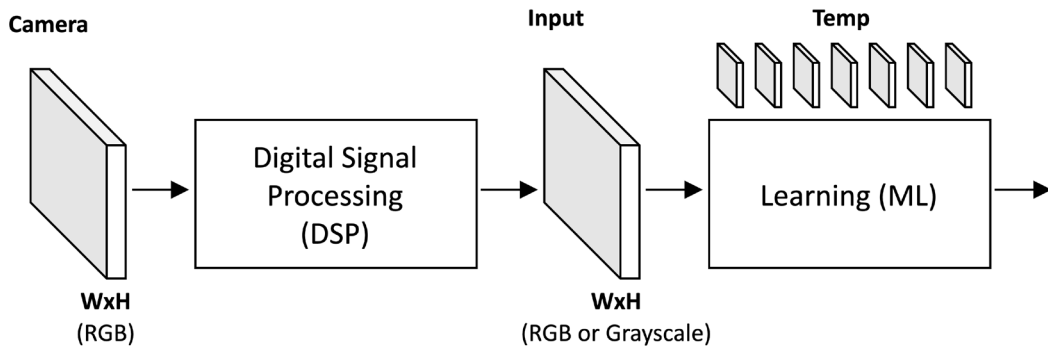


Figure 7.9: The Impulse's data processing

To ensure that our design stays within the SRAM constraints, we need to consider the following:

$$Camera_{memory} + Input_{memory} + Temp_{memory} < SRAM_{memory}$$

The temporary memory required by the ML model depends on the input image size. As ballpark figures, using the default FOMO model provided by Edge Impulse, a resolution between 64x64 and 86x86 will result in an SRAM utilization of around 200 Kbytes.



The FOMO model only works on squared input images.

The FOMO model supports two color formats for the input image: **RGB** and **grayscale**. Opting for grayscale can save one-third of the memory compared to RGB images.

Therefore, after these considerations, we have opted for a grayscale image with a resolution of 76x76 pixels. This decision should ensure that our object detection pipeline consumes approximately less than 200 Kbytes of SRAM and fits within the memory constraints of the Raspberry Pi Pico.

How to do it...

Continue working in Edge Impulse and take the following steps to prepare the set of features from the dataset:

Step 1:

Click on the **Create impulse** option from the left-hand side menu and ensure you have the **Image data** block for the input data, as shown in the following screenshot:

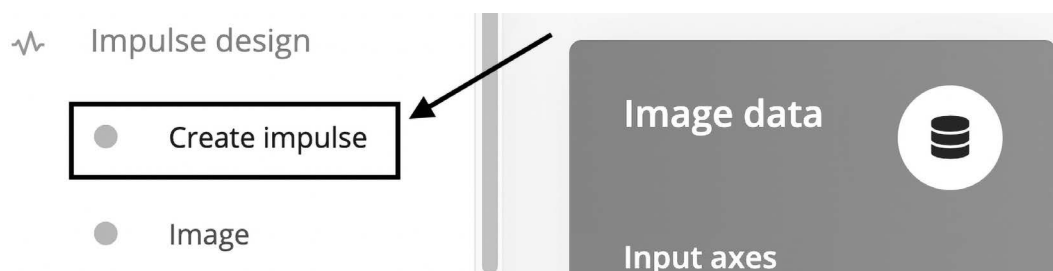


Figure 7.10: The Image data block

After, enter the values 76 and 76 in the image resolution fields (**Image width** and **Image height**):

Image width	Image height
76	76


Figure 7.11: Image resolution fields

The image resolution specified with these two fields is for the input of the Impulse, which will feed the pre-processing block.

Then, ensure the **Fit shortest axis** option is selected in the **Resize mode** drop-down menu. This selection allows for the cropping of rectangular input images using the shorter dimension of the original image, typically the image's height.

Step 2:

Click on the **Add a processing block** option and, in the new pop-up window, select the **Image** option by clicking on the **Add** button:

 **Add a processing block** ×

Did you know? You can bring your own DSP code.




DESCRIPTION	AUTHOR	RECOMMENDED
<p>Image</p> <p>Preprocess and normalize image data, and optionally reduce the color depth.</p>	Edge Impulse	 

Figure 7.12: The Image processing block

The processing block is responsible for preparing the raw input image produced by the camera for feeding the model. Therefore, the processing block can perform tasks like image normalization or color format, just to name a few.

Step 3:

Click on the **Add a learning block** option and, in the new pop-window, select the **Object Detection (Images)** option by clicking on the **Add** button:

 **Add a learning block** ×

Did you know? You can bring your own model in PyTorch, Keras or scikit-learn.



DESCRIPTION	AUTHOR	RECOMMENDED
<p>Object Detection (Images)</p> <p>Fine tune a pre-trained object detection model on your data. Good performance even with relatively small image datasets.</p>	Edge Impulse	 

Figure 7.13: The Object Detection learning block

As you can see from the **Object Detection** learning block description, the ML algorithm is based on a pre-trained model.

Upon including the learning block, you will see the **Output features** block, indicating the target object we intend to detect, which, in this case, is **can**:

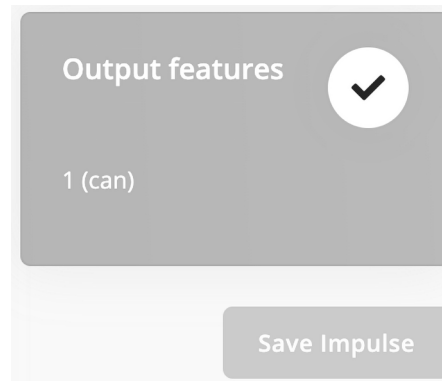


Figure 7.14: The Save Impulse button

Finally, click on the **Save Impulse** button.

Step 4:

Click on the **Image** option from the left-hand side menu:

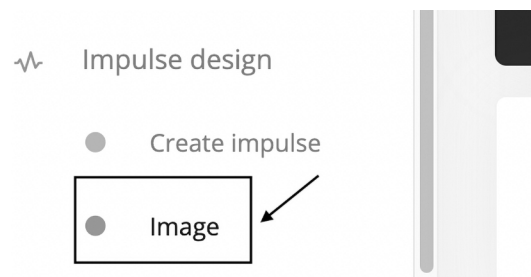
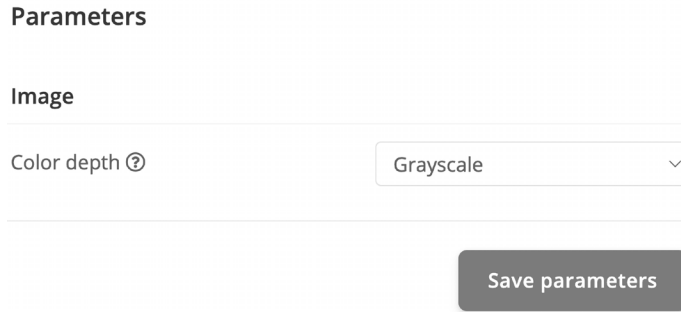


Figure 7.15: The Image option

In the new window, select **Grayscale** from the **Color depth** drop-down menu:



The screenshot shows a web interface titled "Parameters". Under the "Image" section, there is a label "Color depth ?" followed by a dropdown menu. The dropdown menu is open, showing "Grayscale" as the selected option. Below the dropdown menu is a button labeled "Save parameters".

Figure 7.16: Select Grayscale from the Color depth drop-down menu

Then, click on the **Save parameters** button.

In the new window, click the **Generate features** button to generate the features from each training sample. After a few seconds, you will see the **Job completed** message in the output log and be ready to train the ML model exploiting the FOMO algorithm.

There's more...with the SparkFun Artemis Nano!

In this recipe, we have learned how to choose the appropriate input image resolution for the object detection pipeline, considering the 264 Kbytes of SRAM available on the Raspberry Pi Pico. However, it is worth noting that the SparkFun Artemis Nano board offers a larger SRAM capacity of 384 Kbytes, allowing us to experiment with higher image resolutions, such as 96x96 pixels.

Having decided on the input image resolution and built the pre-processing stage, let's design the ML architecture.

In the upcoming recipe, we will leverage transfer learning with FOMO to train an object detection model suitable for memory-constrained devices.

Transfer learning with FOMO

After designing the pre-processing block, it is time to train the ML model.

In this recipe, we will discuss the features that make the FOMO model suitable for highly constrained devices and show how to train it in Edge Impulse.

Getting ready

The design of the FOMO architecture, leveraged in this project to enable object detection on Raspberry Pi Pico, demonstrates that by approaching problems from a different and simple perspective, we can turn the seemingly impossible into reality. tinyML developers should always think this way if they want to unlock novel solutions on microcontrollers, as the computational capabilities of these devices are certainly not the same as those of the cloud, laptops, or smartphones.

In the following subsection, we will dive deep into the technical details of FOMO to learn how this model works and be inspired by its underlying ideas.

Behind the design of FOMO

If you are an ML developer, I am confident that you will have come across popular object detection architectures like **MobileNet SSD** (<https://arxiv.org/abs/1512.02325>) and **YOLO** (<https://arxiv.org/abs/1506.02640>).

These two architectures have shown remarkable accuracy in object detection tasks, capable of detecting objects that even the human eye might find challenging.

However, how many parameters do these models have? These networks take 15 **million (M)** parameters for MobileNet v2 SSD and 21.2 M for YOLOv5m. Even though there are smaller and quantized versions of these models, the parameter reduction is generally insufficient for efficient deployment and execution on microcontrollers.

So, how did the FOMO algorithm manage to make this task possible? The solution lies in the output type of the object detection model. In numerous scenarios where an object detection model is used, the main goal is counting objects within the scene or determining their location. Therefore, *what is indeed required in such cases is not the bounding box itself but rather the centroid.*

The following image shows the output difference between an object detection model based on a bounding box (left image) and a centroid (right image):

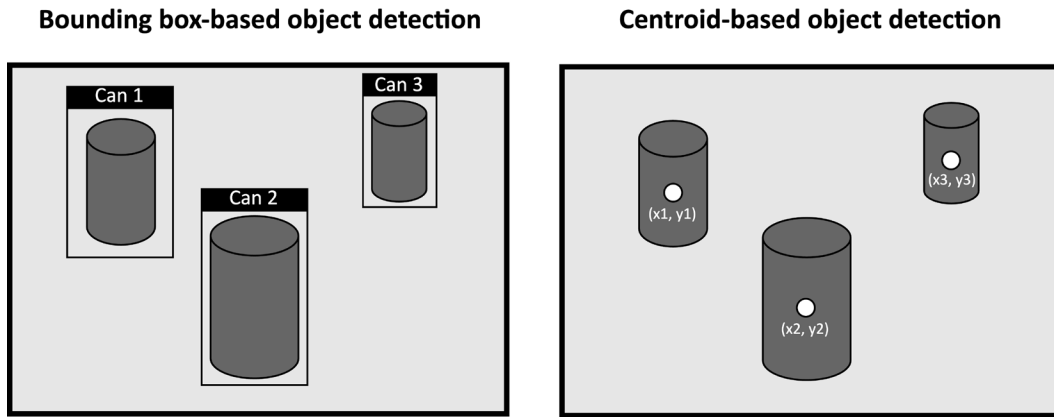


Figure 7.17: Bounding box versus centroid object detection

If we are interested in the object's centroid, the problem becomes less complex because this information can be extracted from the learned features of an image classification model. Using an image classification model, for example, **MobileNet v2**, makes FOMO require much less processing power and memory than MobileNet v2 SSD or YOLOv5m. Are you curious how we can determine the centroid using an image classification model? The following subsection will explain how we can detect objects by generating heat maps from the learned features.

Locating objects from heat maps

The nature of an image classification model is just to tell us the content of an image without telling us about the location of the objects in the scene. Even though the output model does not include the localization of the object, this information is retained in the previous convolutional layers. Indeed, each layer can be perceived as an image wherein the pixels are activated when features corresponding to the objects of interest are detected. As we know, the deeper we go in a **convolutional neural network (CNN)**, the lower the resolution is to avoid an unmanageable number of trainable parameters. Therefore, the object locality vanishes as we approach the network's end. To visualize what has just been described, consider the following figure, where the original image has a 32x32 resolution, and each layer reduces the resolution by a quarter:

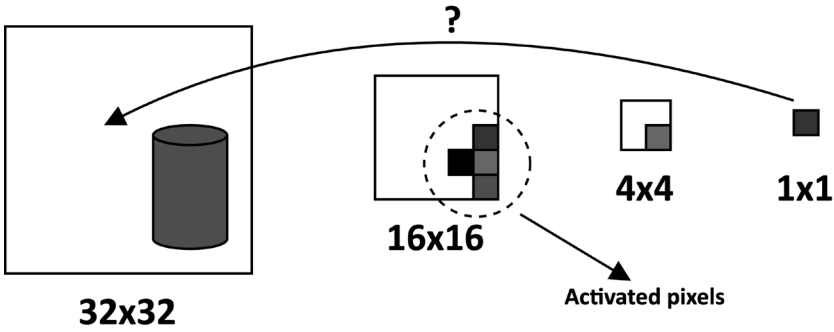


Figure 7.18: Example of activation maps in a CNN

From Figure 7.18, we can observe that the deeper we go, the more uncertainty on the locality of the object there is when attempting to retrace the activated pixels from the lower image back to the input one. The most challenging layer is the last, where we have a single pixel that cannot be confidently mapped back to the initial image.

The approach implemented in FOMO to determine the centroid's position of the objects was to cut off the last layers of MobileNet v2, one of the most efficient models for constrained devices, including microcontrollers. The cut-off layers are then replaced with a per-region probability map and a custom loss function to generate a **heat map**. This heat map (right image) is what is used to locate the objects in the scene, as shown in Figure 7.19:

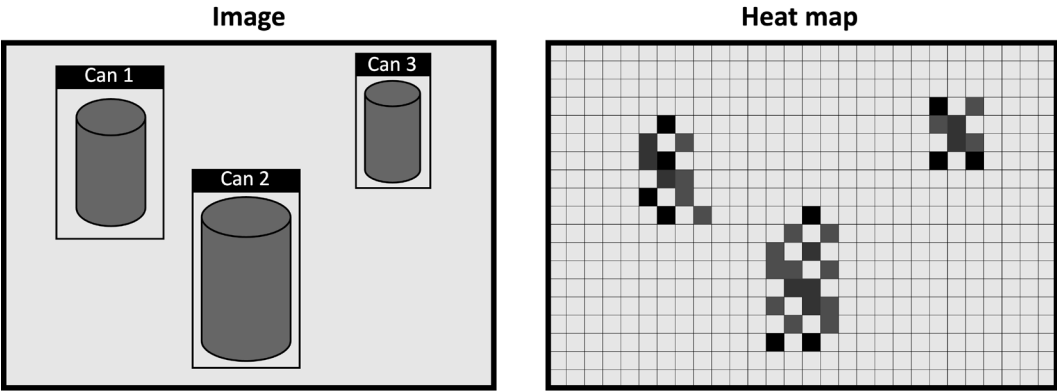


Figure 7.19: Example of an image of cans with corresponding the heat map

To produce an accurate heat map, it is crucial to strike a balance when determining the cutting-off point in the network. If the truncation point is too near the first layer, there might not be sufficient learned features to locate the object precisely. On the other hand, if it is too close to the network's end, the locality information could be diminished. In FOMO, the truncation point is chosen when the output layer of the MobileNet v2 model corresponds to 1/8 of the input image resolution. This choice ensures a balance between having enough learned features and preserving the locality.

How to do it...

Continue working in Edge Impulse and take the following steps to train the ML model based on FOMO:

Step 1:

Click on the **Object detection** option from the left-hand side menu:

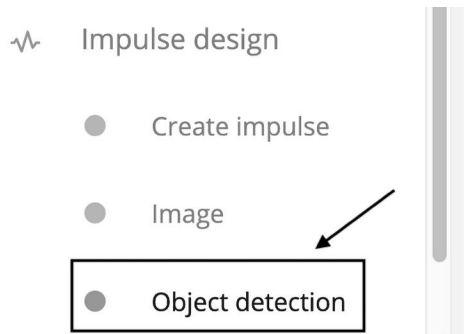


Figure 7.20: The Object detection option

On the new page, you should find the **Training settings**, **Advanced training settings**, and **Neural Network architecture** sections. In the **Training settings** section, keep the **Number of training cycles** and **Learning rate** at their default values and enable the **Data augmentation** option to include artificially created samples in the training dataset.

In the **Advanced training settings**, ensure the **Validation set size** is 20% to reserve 20% of the training samples for the validation dataset.

In the **Neural network architecture** section, ensure you use the **FOMO MobileNet v2 0.35** pre-trained model to train the object detection model, as shown in *Figure 7.21*:

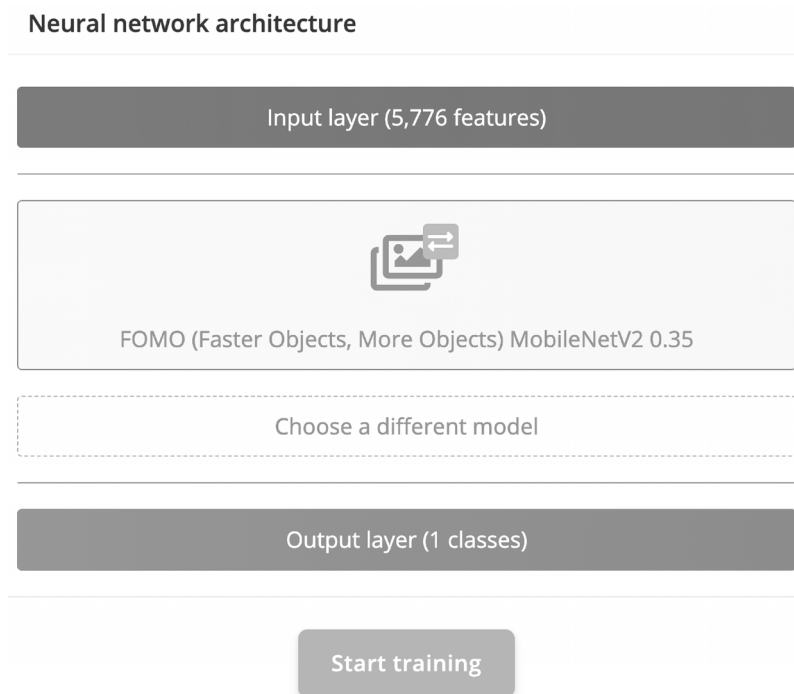


Figure 7.21: Model architecture

As you can see from the **Neural network architecture** section, the model expects an input with **5,776** features, which corresponds to a grayscale image with a resolution of 76x76 pixels. This input feeds the pre-trained **FOMO** model based on **MobileNetv2** with a **0.35 alpha value**.



In *Chapter 8, Classifying Desk Objects with TensorFlow and the Arduino Nano*, you will learn more about the MobileNet v2 model and the importance of the alpha value to reduce the number of trainable parameters.

To start the training, click on the **Start training** button. As the training progresses, you can monitor the **Loss**, **Precision**, **Recall**, and **F1** after each epoch in the **Training output**.

Upon training completion, you can visualize the accuracy performance in the **Confusion matrix** obtained on the validation set. This information can be found in the **Model** section, as shown in *Figure 7.22*:

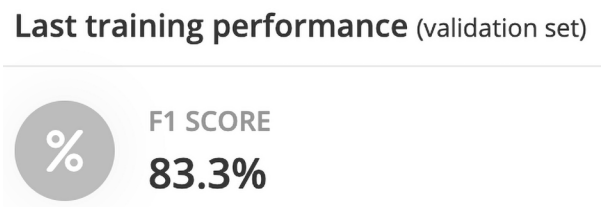


Figure 7.22: The model’s accuracy reported with the F1 score metric

In our case, we obtained an **F1** score of **83.3%**, indicating that our quantized model (**Quantized (int8)**) effectively detects the cans with a good balance between **Precision** and **Recall**.

Step 2:

Evaluate the on-device model’s performance. To accomplish this task, you first need to select the Raspberry Pi Pico (**Raspberry Pi RP2040**) as the target device from the **Target** drop-down menu located at the top of the page:

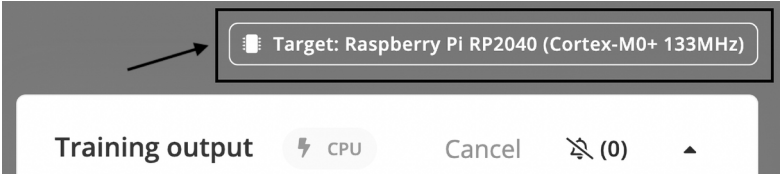


Figure 7.23: Target device drop-down menu

Afterward, evaluate the **peak SRAM** and **Flash usage** in the **On-device performance** section. The values should be similar to what is shown in *Figure 7.24*:

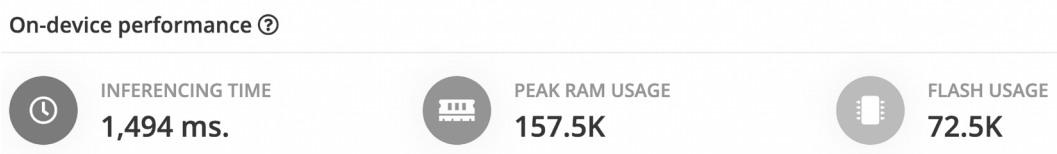


Figure 7.24: On-device performance

As you can see from the estimated on-device figures, the Impulse will need around **157.5** Kbytes of SRAM, accounting for about 60% of the total available on the Raspberry Pi Pico.

The estimated inference time indicates that a single inference may take more than a second. As a result, the solution we have developed might be suitable only for applications where we do not have constraints on the latency performance.

There's more...

In this recipe, we have learned how to train the FOMO model in Edge Impulse. From the estimated on-device performance, we have seen that the model should take roughly 157.5 Kbytes of SRAM and 72.5 Kbytes of Flash. However, is there a way to reduce the memory consumption even further?

Indeed, there is a way to reduce it by tweaking the underlying Keras implementation of the model just trained. To do so, click on the **Switch to Keras (expert) mode** option, accessible via the three dots in the **Neural Network settings**:

Neural Network settings

Training settings

Number of training cycles (?)

- ⚙️ Switch to Keras (expert) mode
- ✚️ Edit block locally

Figure 7.25: Keras (expert) mode option

After switching to Keras mode, Edge Impulse will reveal the underlying Python code of the trained model. Within this code, there are two areas where we can make some adjustments to reduce memory usage and improve latency performance.

The first area is on the heat map pixel classifier, which assigns the pixels of the heat map to their respective classes. This classifier is located after the cutting-off point of the MobileNet v2 model, as highlighted in *Figure 7.26*:

```

39     if type(layer) == BatchNormalization:
40         layer.momentum = 0.9
41     #! Cut MobileNet where it hits 1/8th input resolution; i.e. (HW/8, HW/8, C)
42     cut_point = mobile_net_v2.get_layer('block_6_expand_relu')
43     #! Now attach a small additional head on the MobileNet
44     model = Conv2D(filters=32, kernel_size=1, strides=1,
45                   activation='relu', name='head')(cut_point.output)
46     logits = Conv2D(filters=num_classes, kernel_size=1, strides=1,
47                   activation=None, name='logits')(model)
48     return Model(inputs=mobile_net_v2.input, outputs=logits)
49 
```

Figure 7.26: The cutting-off point

As you can see from the previous snapshot, the heat map pixel classifier is composed of two Convolution 2D (Conv2D) layers, each with a unit kernel size. The first Conv2D operation utilizes 32 filters, which then feed into the second Conv2D with `num_classes` filters (in this case, two, corresponding to the detection of the can and the background). One idea to reduce the number of trainable parameters and decrease the memory consumption is to lower the filters in the first Conv2D layer from 32 to 16.

In our test, implementing this change led to a reduction of 1 Kbyte in SRAM usage and 2.5 Kbytes in program memory, as shown in *Figure 7.27*:

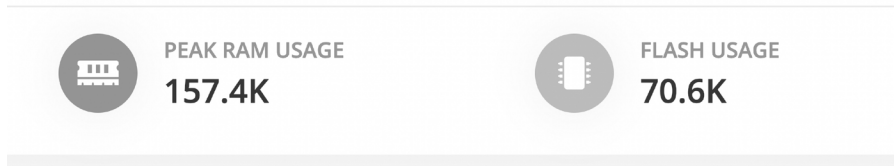


Figure 7.27: Estimated RAM and Flash usage using 16 kernels

This tweak should not impact the model's accuracy because we only classify one object type. Generally, more convolution filters are necessary when classifying multiple object types.

The second area that allows further reduction in memory consumption and latency performance involves modifying the layer where the cutting-off point occurs. By default, Edge Impulse performs the cut-off in the MobileNet v2 model when the image resolution of the layer decreases to 1/8 in relation to the input image. In the MobileNet v2 model, this layer is **block_6_expand_relu**. A potential approach is shifting this cut-off point closer to the initial layer, like **block_5_expand_relu** or **block_3_expand_relu**. However, it is worth noticing that while this tweak could result in significant memory savings, it may also lead to a substantial drop in accuracy due to fewer learned features available for object detection. Nonetheless, it is worth experimenting to see what happens.

The model is now trained, and we only need to test its effectiveness on unseen data during training before we can proceed with the model deployment on the Raspberry Pi Pico.

In the upcoming recipe, we will assess the model's accuracy in Edge Impulse using the test dataset and the Live Classification tool.

Evaluating the model's accuracy

The model's accuracy experimented on the validation dataset seems to be promising. However, we can only confidently confirm the model's suitability for our needs after evaluating its performance on unseen data.

In this recipe, we will carry out this evaluation with the **Model testing** and **Live classification** tools of Edge Impulse.

Getting ready

The test dataset provides an unbiased evaluation of the model's accuracy since these samples are not used during training. Evaluating how the model behaves on unseen data is essential to determine its alignment with our expectations. For example, this assessment might unveil that the model struggles to identify objects against specific backgrounds. If such a situation arises, it could be due to the training dataset's limited size. Therefore, we may retrain the model using additional images to address the issue.

However, what extra steps should we take if we have trained the model with additional training samples and cannot detect our objects in all backgrounds?

If we can still not detect objects with some backgrounds, the low input image resolution could likely be the culprit, and the trained model could not be suitable for all applications. At this point, we must know where the target device will be physically located. For example, suppose we know that the application will be employed within an environment characterized by a fixed background (such as an industrial setting). In that case, there is a good chance that the current model will remain suitable. Nonetheless, if the feasibility of deploying the application in an environment with a fixed background is impossible, the only way to ensure the model's functionality is to train a new model that processes higher-resolution input images. As a result, we must consider an alternative target device with more computational power and memory capabilities.

Figure 7.28 visually summarizes the essential steps that we should always keep in mind for building a practical tinyML application:

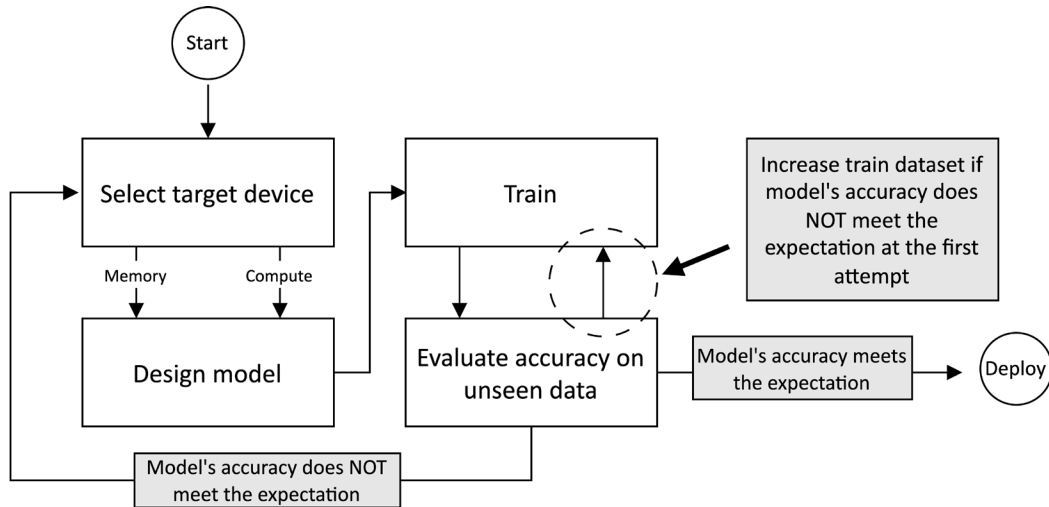


Figure 7.28: Steps for designing an ML model for memory-constrained devices

As shown in Figure 7.28, it is crucial to design the model while considering the computational resources available on the target device. If we fail to do so, we may be unable to deploy the model effectively. However, it is only after evaluating the model's accuracy on unseen data that we can discover whether the target device suits our needs. In fact, if the model's accuracy does not meet our expectations, we should consider an alternative device to design a new and larger model.

How to do it...

Continue working in Edge Impulse and take the following steps to evaluate the accuracy of the ML model using the **Model testing** and **Live classification** tools:

Step 1:

Click on the **Model testing** option from the left-hand side menu:

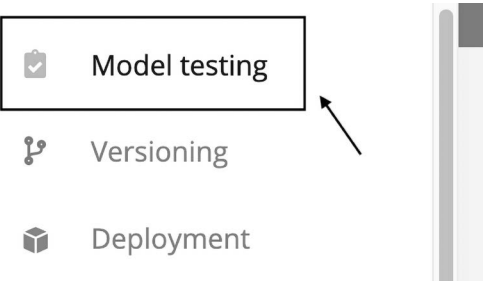


Figure 7.29: The Model testing option

In the new section, you should see the list of samples belonging to the test dataset with the expected labels.

Step 2:

Click on the **Classify all** button to evaluate the accuracy of the test dataset. After clicking the button, you will see the progress of this evaluation in the **Model testing output** log.

Once the accuracy evaluation on the test dataset is completed, you will find the overall accuracy result showcased in the **Model testing results** section. In addition to this result, each test sample will report the **F1 score** and be marked in either green or red, indicative of the correct or wrong detection, as shown in *Figure 7.30*:

SAMPLE NAME	EXPECTED OUTCOME	LENGTH	F1 SCORE	RESULT
can.489pk9rj	can	-	66%	...
can.489pk4sp	-	-	0%	...
can.489pk0jh	-	-	0%	...
can.489ndd35	can, can	-	100%	...

Figure 7.30: Test data results

In our case, we obtained an accuracy of just 60% on the test dataset, significantly lower than what we got on the training and validation datasets (83%).

Step 3:

If you have obtained a low accuracy result on the test dataset, examine the misclassified samples. To do so, click on the vertical ellipsis (three dots) of each image with a low accuracy result and then click on the **Show classification** option:

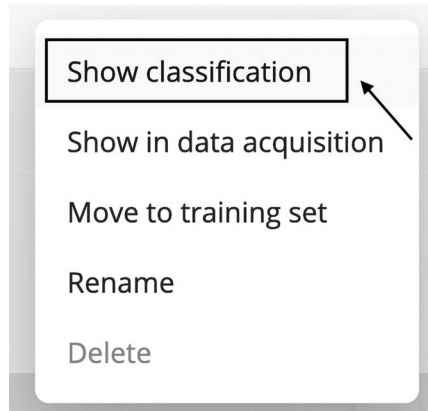


Figure 7.31: The Show classification option

For every sample, you should find the expected bounding boxes and the model's actual output.

To improve the model's accuracy, consider adding new samples to the training dataset. To do this, click the **Data acquisition** option and collect more images with cans. Then, split them among the training and test datasets, proceed to generate the features, and train the model again.

Step 4:

Test the model using the **Live classification** tool. To do so, click on the **Live classification** option from the left-hand side menu:

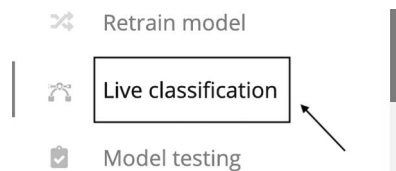


Figure 7.32: The Live classification option

In the new window, click on the icon with the microchip image to test the model on the live stream video acquired with your webcam:

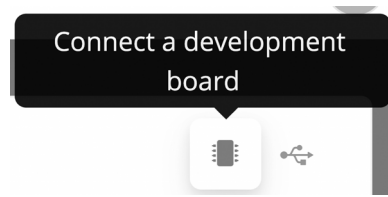


Figure 7.33: The microchip icon

In the new window, click on the **Connect to your computer** button. Edge Impulse will then open a new tab in the web browser, requesting permission to access the webcam. Grant permission by clicking on the **Give access to the camera** button and then click on the **Switch to classification mode** button, located at the bottom of the page:



Switch to classification mode

Figure 7.34: The classification mode button

Edge Impulse will then load the model and build a web application that demonstrates the object detection model's functionality through the live video stream captured by your webcam!

There's more...

In this recipe, we have learned how critical model evaluation on unseen data is to discover whether the target device is suitable for our needs.

In our experiments, we have observed that the object detection model performs optimally when the background is fixed. Although we may perceive this model as quite basic, it is worth considering that this background condition is frequently encountered in industrial settings. An example is the conveyor belt, where this model could be used for counting and classifying objects to improve quality control in production lines.

This recipe ends the model's preparation, and from now on, our attention will shift to the application deployment on the microcontroller.

In the upcoming recipe, we will begin this project phase by developing a Python script to read camera frames from the webcam and transmit them over the serial interface.

Using OpenCV and pySerial to send images over the serial interface

In this recipe, we will emulate a camera sensor that can communicate with the Raspberry Pi Pico through the serial interface using a local Python script.

This Python script will be able to capture images from the webcam using OpenCV and, upon the microcontroller's request, transmit its pixels over the serial interface.

Getting ready

The Python script we will develop in this recipe can be considered a software program that emulates the functionality of a camera sensor we may intend to use in production. Our target platform is often not equipped with all the required hardware components during the initial prototyping stage. Nevertheless, running the model on the microcontroller might still be necessary. Some of these reasons might be for testing the model functionality or gather preliminary latency performance data.

The Python script we aim to develop in this recipe exploits the **OpenCV** (<https://opencv.org/>) library to capture images from the webcam and use the serial interface to transmit pixel data to the microcontroller, as shown in the following diagram:

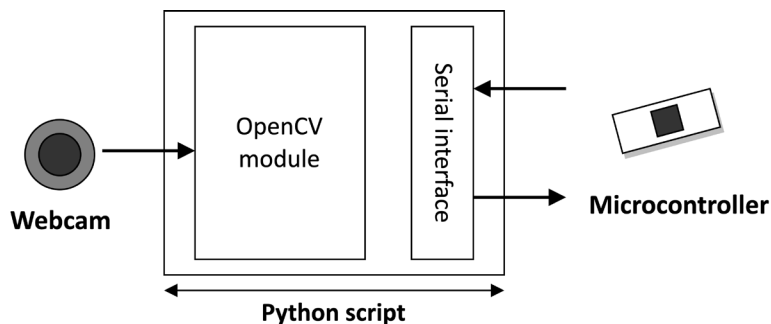


Figure 7.35: The camera software model

The OpenCV library can be installed with the following pip command:

```
$ pip install opencv-python
```

Capturing live frames from a camera with OpenCV is trivial, and it requires only the following two functions:

- `VideoCapture()`, for the camera initialization
- `read()`, for capturing the frame from the camera and storing it in a NumPy array

To replicate the functionality of a camera sensor, the camera frame must be captured only upon a read request. In this recipe, this read request will be performed when the microcontroller transmits the `<cam-read>` string over the serial. When the Python script receives this command, it can capture the image from the webcam using OpenCV. However, before sending the image pixel data to the microcontroller, the program must rescale the image to 76x76 and change its color space to grayscale to match the ML model's input format. All the necessary image processing operations required to prepare the input image will be carried out with OpenCV and discussed in the *How to do it...* section of this recipe.



To learn more about the functionalities of the OpenCV functions employed in this recipe, we recommend you refer to the official OpenCV documentation provided at the following link: <https://docs.opencv.org/4.x/>.

However, in the following subsection, we want to discuss one aspect we haven't encountered so far: data transmission over serial communication using the `pySerial` library.

Sending bytes over the serial with `pySerial`

Up to this point, we have employed the `pySerial` library exclusively to receive data sent by the microcontroller. Nonetheless, this library also allows us to transmit data serially.

The `pySerial` library provides the `write()` method for this scope, which only takes the data to be sent as an input argument. This function sends bytes over the serial, which means that the input data must be of type `bytes`, as also reported in the official documentation of `pySerial` (https://pyserial.readthedocs.io/en/latest/pyserial_api.html). Since our camera frame is stored in a NumPy array, there are at least two operations to be performed before transmitting data, which are the following:

- Casting the NumPy array to `uint8`. This cast is essential because the pixels of the camera frame could be stored as an `int32` or `float` data type. As we know, the ML model's input expects a grayscale image, where each pixel represents the intensity as a single 8-bit value. Therefore, this explicit data type conversion is required to match the data type the microcontroller expects.

- Convert the NumPy array into an array of bytes. This operation can be easily performed using the `bytearray()` Python function.

After the preceding two operations, we can send data over the serial using the `write()` method of `pySerial`.

How to do it...

In your local Python **Virtual Environment** (`virtualenv`), create a new Python script called `emulated_camera.py`. Then, take the following steps to capture images from the webcam using OpenCV, resize them to 76x76, change the color format to grayscale, and transmit their pixel values over the serial:

Step 1:

Import the necessary Python libraries:

```
import numpy as np
import serial
import cv2
```

In the preceding code snippet, the `cv2` is imported to use the OpenCV library.

Step 2:

Initialize the serial communication with the port and baud rate (for example, 115200) used by the microcontroller:

```
ser = serial.Serial()
ser.port = '/dev/ttyACM0'
ser.baudrate = 115200
```

The easiest way to determine the serial port name is from the device drop-down menu in the Arduino IDE:

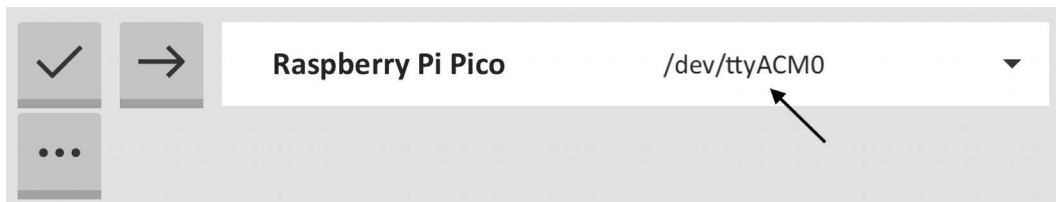


Figure 7.36: The serial port name is reported in the Arduino IDE

After the initialization of the serial communication, open the serial port and discard the content in the input buffer:

```
ser.open()  
ser.reset_input_buffer()
```

Once the input buffer is cleared, the program can receive and transmit data over the serial.

Step 3:

Define the OpenCV video camera object to start capturing video frames from the webcam:

```
cam = cv2.VideoCapture(0)
```

In the preceding code snippet, the `VideoCapture()` function is employed to initialize the OpenCV camera object (`cam`) to allow us to capture pictures with the webcam. If you pass `0` to the `VideoCapture()` function, the default camera will be used. However, if you have multiple cameras connected to your laptop, you may need to replace the `0` value with the index corresponding to the desired camera. For example, if you wanted to use the second camera, you would substitute `0` with `1`.

For most cameras, OpenCV employs a default camera configuration featuring a resolution of 640x480 and a BGR888 color format. These settings imply that the resulting image has 640 pixels in width and 480 pixels in height, and each pixel is represented as a triplet of three 8-bit integer values, representing the intensity of the blue, green, and red channels.

Step 4:

Open an infinite while loop. Within the loop, read data from the serial port. Then, verify whether the `<cam-read>` string has been transmitted, indicating the read request:

```
def serial_readline(obj):  
    data = obj.readline()  
    return data.decode("utf-8").strip()  
  
while True:  
    data_str = serial_readline(ser)  
    if str(data_str) == "<cam-read>":
```

Step 5:

Within the `if` statement, capture a frame from the webcam with OpenCV:

```
ret, img_bgr = cam.read()
```

The `read()` function returns two values, which are the following:

- `ret`: A Boolean value indicating whether the frame has been acquired successfully.
- `img_bgr`: The pixels of the captured frame in a NumPy array. Each element in the array corresponds to a pixel's value. Since the default color format in OpenCV is BGR888, each entry in the array contains three 8-bit values corresponding to the intensity of the blue, green, and red channels.

Step 6:

Within the `if` statement, resize the frame to match the image resolution of the ML model's input using the `resize()` OpenCV function:

```
h0 = img_bgr.shape[0]
w0 = img_bgr.shape[1]
h1 = min(h0, w0)
w1 = h1
img_cropped = img_bgr[0:w1, 0:h1]
img_resized = cv2.resize(
    img_cropped,
    (76, 76),
    interpolation = cv2.INTER_LINEAR)
```

Since the captured frame has a different aspect ratio than the ML model's input image, it is necessary to crop the frame to prevent distortion artifacts during resizing. The cropping is simply accomplished by taking a slice from the original image. In our case, the resulting cropped image has a width and height equal to the shortest dimension of the original image (`min(h0, w0)`).

After cropping, we can resize the image with the `resize()` function, which takes the cropped image (`img_cropped`), the new image resolution (76, 76), and the interpolation policy (`interpolation=cv2.INTER_LINEAR`) as input arguments.

Step 7:

Within the `if` statement, convert the BGR888 image to grayscale using the `cvtColor()` OpenCV function:

```
img_gray = cv2.cvtColor(img_resized,
                        cv2.COLOR_BGR2GRAY)
```

The `cvtColor()` function converts images from one color format to another. In this case, it converts the resized image (`img_resized`) from BGR888 to grayscale (`cv2.COLOR_BGR2GRAY`).

Step 8:

Within the `if` statement, transmit the image pixel data contained in the `img_gray` NumPy array over the serial:

```
data = bytearray(img_gray.astype(np.uint8))
ser.write(data)
```

As you can see from the preceding code snippet, we first cast the NumPy array holding the grayscale image to `uint8` (`np.uint8`) and then convert it to an array of bytes using the `bytearray()` function. These two operations allow us to send the correct bytes over the serial interface.

Step 9:

Within the `if` statement, display the BGR888 resized image (`img_resized`) in a window using the `imshow()` OpenCV function:

```
cv2.imshow("Captured image", img_resized)
```

The `imshow()` function only requires two input arguments, which are the following:

1. The title of the window displaying the camera frame (Captured image)
2. The image to display (`img_resized`)

Step 10:

Outside the `if` statement, use the `waitKey()` OpenCV function to wait for at least 33 **milliseconds (ms)** before displaying another frame in the window:

```
key = cv2.waitKey(33)
```

The `waitKey()` function takes the waiting time in ms as an input argument and returns the key pressed during that time. For example, providing a value of 33 would result in the images being displayed on the screen at a frame rate of approximately 30 ($1 / 0.033$) **frames per second (FPS)**.



The frame rate won't be 30 FPS in our scenario due to the high model inference time on the microcontroller.

If you provide a value of 0 to the `waitKey()` function, this function waits indefinitely for a key to be pressed.

However, the purpose of this function is not solely to wait for the program for a certain amount of time and identify whether a key was pressed. In fact, the `waitKey()` function is mandatory to correctly display the image on the screen when using the `imshow()` function. A common error among newcomers to OpenCV is neglecting the use of `waitKey()`. If this function is omitted, nothing will be shown on the screen.

Step 11:

Exit the program when the user presses *q* on the keyboard:

```
if key == ord('q'):
    break

cv2.destroyAllWindows()
cam.release()
```

If the *q* key is pressed in the preceding code, we exit the infinite `while` loop. However, before ending the program execution, we must close all OpenCV windows with the `destroyAllWindows()` function and release the camera resource with the `release()` function.

At this point, our emulated software camera is prepared for use with the microcontroller!

There's more...

In this recipe, we have learned how to use OpenCV to read camera frames, perform image processing operations, and transmit pixel values over the serial interface with the `pySerial` module.

In the way we have implemented the Python script, the microcontroller must know in advance the transmitted image's format to interpret the serial bytes correctly.

As a result, any alterations made to the Python program about image resolution or color format require adjustments in the Arduino sketch.

Given this limitation, you may consider improving the communication protocol. One idea could be to encompass the image resolution and color format when sending data serially. In this way, the microcontroller won't need pre-knowledge about the image format because this information is included in the transmitted data package.

Now that the emulated camera is ready, we must test it to ensure it receives the read request from the microcontroller and can capture frames from the webcam.

In the upcoming recipe, we will implement an Arduino sketch to send the camera read command over the serial and verify the correct functionality of the webcam.

Reading data from the serial port with Arduino-compatible platforms

The Python script implemented in the previous recipe allows us to transmit images over the serial. However, before deploying the model on the microcontroller, it is worth verifying whether the serial communication with the device works and whether we can display camera frames on the screen.

Therefore, in this recipe, we will implement an Arduino sketch to send a read camera request and retrieve the image pixel data transmitted by the Python script.

Getting ready

The only prerequisite for accomplishing this recipe's objective is reading the data from the serial port with an Arduino compatible platform.

Arduino provides a handy function for reading bytes sent over the serial: `readBytes()` (<https://www.arduino.cc/reference/en/language/functions/communication/serial/readbytes/>).

The `readBytes()` function is a method of the `Serial` object and takes two input arguments, which are the following:

- **Buffer:** The buffer to store the bytes read from the serial port
- **Length:** The number of bytes to read from the serial port

This function only returns when either all bytes have been read from the serial port or there is a **timeout**, which is by default 1,000 ms (<https://www.arduino.cc/reference/en/language/functions/communication/serial/settimeout/>).

The user must allocate the `Buffer` to accommodate the number of bytes read from the serial port. Since the Python script will only transmit the pixel values of the grayscale image with 76x76 resolution, this buffer should accommodate at least 5,776 bytes (76*76).

How to do it...

Open the Arduino IDE and create a new sketch. Then, take the following steps to implement the Arduino program to send the read camera request and subsequently retrieve the image pixel data transmitted over the serial:

Step 1:

Declare a `uint8_t` buffer to accommodate the 76x76 grayscale image sent by the Python script over the serial:

```
#define FRAME_SIZE (76*76)
uint8_t frame[FRAME_SIZE];
```

Since a single 8-bit value represents each pixel of the grayscale image, the total number of bytes of the transmitted image is equal to the number of width pixels (76) by height pixels (76).

Step 2:

In the `setup()` function, initialize the serial peripheral with the same baud rate used in the Python script (115200):

```
void setup() {
  Serial.begin(115200);
  while (!Serial);
}
```

Step 3:

In the `loop()` function, transmit the `<cam-read>` string over the serial to initiate a read camera frame request:

```
void loop() {
  Serial.println("<cam-read>");
```

Step 4:

In the `loop()` function, use the Arduino `readBytes()` function to read all the bytes of the grayscale image sent by the Python script:

```
Serial.readBytes(frame, FRAME_SIZE);
```

The function will return when all bytes (`FRAME_SIZE`) of the image have been read from the serial port and saved into the frame buffer.

Now, compile and upload the sketch on the Raspberry Pi Pico. Then, run the Python script:

```
$ python emulated_camera.py
```

If the Raspberry Pi Pico and the Python script can communicate over the serial, you will see a window on your screen displaying the captured frame from the webcam in grayscale format.

There's more...with the SparkFun Artemis Nano!

In this recipe, we have learned how to read data from the serial port using Arduino and verified the functionalities of the Python script.

The sketch developed in this recipe works with any Arduino compatible platform, including the SparkFun Artemis Nano. Therefore, you might consider compiling the program for this platform to check whether you can still communicate with the emulated camera.

Despite the low bit rate of serial communication, the video stream in the window should not have a visible lag in displaying the camera frames. Unfortunately, this smoothness will significantly deteriorate in the following recipe because of the long inference time. Nonetheless, the video preview will help visualize the detected objects in the scene.

Everything is now ready for deploying the model on the microcontroller.

In the upcoming final recipe, we will implement an Arduino sketch to detect cans using the Raspberry Pi Pico with the help of Edge Impulse.

Deploying FOMO on the Raspberry Pi Pico

Here we are, ready to deploy the FOMO model on the Raspberry Pi Pico.

In this recipe, we will develop a sketch to run the model inference with the **Edge Impulse Inference SDK** and transmit the centroid coordinates of the detected objects over the serial.

These coordinates will be read in the Python script developed previously to highlight the detected objects within the video stream.

Getting ready

Deploying the model trained with Edge Impulse is easy on any Arduino-compatible platform, thanks to the Arduino library generated by Edge Impulse.

This library contains everything we need to run the model inference successfully on the device, such as the following:

- The trained model in TensorFlow Lite format.
- Model parameters, such as the input image resolution and color format or the maximum number of possible detections in a single frame.
- A library containing a set of functions for **Digital Signal Processing (DSP)** operations and ML inferencing, called the **Edge Impulse Inference SDK**.

Among the plethora of functions provided by the Edge Impulse Inference SDK (<https://github.com/edgeimpulse/inferencing-sdk-cpp>), the `run_classifier()` function is the only one required for the model deployment. This function aims to execute the entire Impulse, encompassing the preparation of the model's input through the DSP block and the subsequent execution of the ML inference, as shown in the following image:

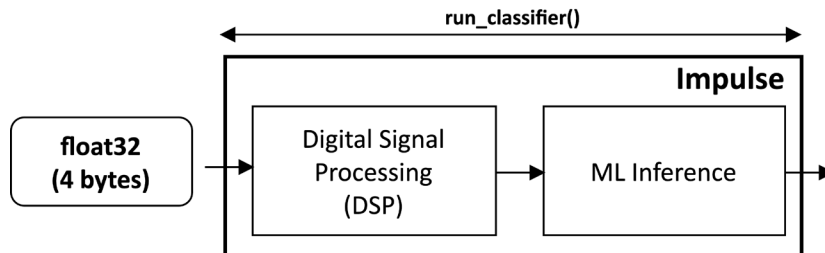


Figure 7.37: The `run_classifier()` function

As depicted in the preceding image, the Impulse always expects the input to be of the floating-point (**float32**) data type, and, in the context of vision-based tasks, each floating-point value packs a single pixel in RGB888 format:

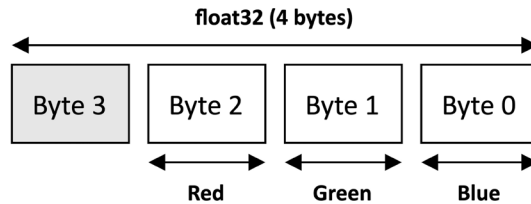


Figure 7.38: Each floating-point value packs a single pixel in RGB888 format

As a result, one byte of the floating-point (**Byte 3**) is not utilized.

The `run_classifier()` function takes three input arguments, which are the following:

- `signal_t`: The data structure used to provide the correct input to the Impulse
- `ei_impulse_result_t`: The model's output that will contain the centroids of the detected objects
- `debug flag`: To print some debug information on the output terminal, such as the content of the ML model input

Out of the three input arguments, `signal_t` and `ei_impulse_result_t` certainly demand a more profound discussion to grasp how to supply the correct input to the Impulse and how to interpret the model's output.

The following subsection will delve into the `signal_t` data structure to discover how to initialize this data structure in the context of vision-based tasks.

Initializing the `signal_t` data structure

The `signal_t` data structure is solely responsible for feeding the Impulse's input. Therefore, this data structure is the bridge between the camera frame and the Impulse, as illustrated in Figure 7.39:

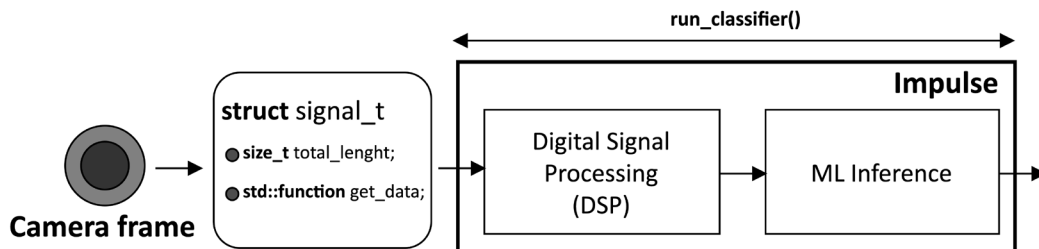


Figure 7.39: The `signal_t` structure is the bridge between the camera frame and the Impulse

The `signal_t` data structure is a C structure consisting of two fields that require initialization from the user.

The first field is `total_length`, corresponding to the total number of pixels of the ML model's input. In our case, it is 5,776 (76*76).

The second field is a function object (`std::function`) named `get_data`, which the DSP block calls when we execute the `run_classifier()` function. The `get_data` function object has the following signature:

```
std::function<int(size_t offset,  
                size_t length,  
                float *out_ptr)>
```

In the preceding code snippet, the function object takes three input arguments, which are the following:

- `offset`: The offset in the camera buffer
- `length`: The total amount of data to load from the camera buffer
- `out_ptr`: The pointer to the floating-point buffer, which corresponds to the Impulse's input



The function returns an integer value (`int`) to indicate potential error conditions. The 0 value indicates error-free execution.

The user must implement a function with the same signature as the `get_data` function object that does the following:

1. Load the data from the camera buffer accordingly with the `offset` and `length` input arguments.
2. Convert the data into the floating-point format. Therefore, if we have a grayscale image, the pixels will need to be converted to RGB888 and packed into a floating-point variable.
3. Store the floating-point variables in the output buffer (`out_ptr`).

Once the function has been implemented, we can assign it to the `get_data` field of the `signal_t` data structure.

At this point, you may have one question: is converting from grayscale to RGB888 inefficient, given that RGB888 pixels occupy 2 bytes more than a grayscale pixel?

This conversion is not as inefficient as it may appear because the color format transformation doesn't occur all at once but in slices to minimize the impact on memory resources. Therefore, Edge Impulse guarantees a consistent input format for all vision-based tasks at the cost of small extra memory.

Now that we have learned how to initialize the `signal_t` data structure, let's proceed to discover how to interpret the output results from the `ei_impulse_result_t` object.

Reading the output results from `ei_impulse_result_t`

The `ei_impulse_result_t` structure is a C structure that holds the output of the ML model. Unlike `signal_t`, this object does not require initialization, as its fields are populated with the outcomes of the model inference.

In the context of object detection, `ei_impulse_result_t` includes a field named `bounding_boxes`, which is a pointer pointing to the list that holds details about all the detected objects. Each detected object is encapsulated within the `ei_impulse_result_bounding_box_t` data structure, which provides the following information:

- The label of the detected object.
- The score value.
- The x and y coordinates of the top-left corner of the bounding box
- The width and height of the bounding box

However, you might wonder why `ei_impulse_result_t` contains information about the bounding box (`ei_impulse_result_bounding_box_t`) when FOMO only provides centroids. The answer lies in the versatility of the `ei_impulse_result_t` C structure. By including bounding boxes, Edge Impulse ensures that this structure also aligns with object detection models that yield bounding boxes. In the context of FOMO, the bounding box represents the heat-map area where the object is identified. As a result, the centroid simply denotes the intersection point of the bounding box's diagonals.

The `bounding_box` list has a predetermined size that matches the value reported in the `bounding_boxes_count` field within the `ei_impulse_result_t` structure. Unfortunately, no field indicates the number of objects detected after model inference. Therefore, it is necessary to iterate through each element and evaluate the `score` value to identify the reported detected objects within this list. If the `score` value is 0, it means that the entry in the list is not an object, whereas a score greater than 0 indicates the presence of an object.

Since the model output returns a list of coordinates, it might be challenging to interpret the accuracy of these detections without displaying their position relative to the camera frame. Therefore, we propose transmitting these coordinates over the serial. Then, use the Python script to grab these coordinates and mark the detected objects in the video stream accordingly. The following subsection will briefly describe the data format for transmitting the centroids.

Transmitting the centroid coordinates to the Python script

The data format considered for the transmission of the centroids is described in the following figure:

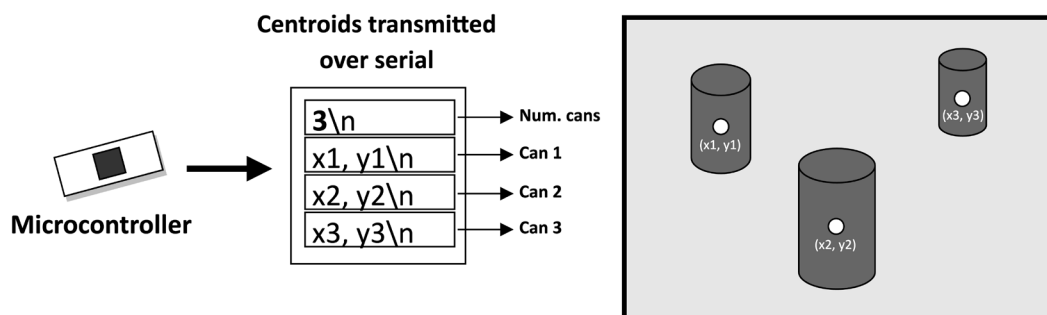


Figure 7.40: Data format for the transmission of the centroids

As you can see from *Figure 7.40*, the microcontroller first sends the number of detected objects (Num. cans) terminated with a newline (`\n`) character. Then, it transmits the coordinates of the centroids (x and y) for each detected object, which are comma-separated and terminated with a newline character.

How to do it...

In Edge Impulse, click on **Deployment** from the left-hand side menu and select **Arduino Library** from the **Search deployment options**, as shown in the following screenshot:

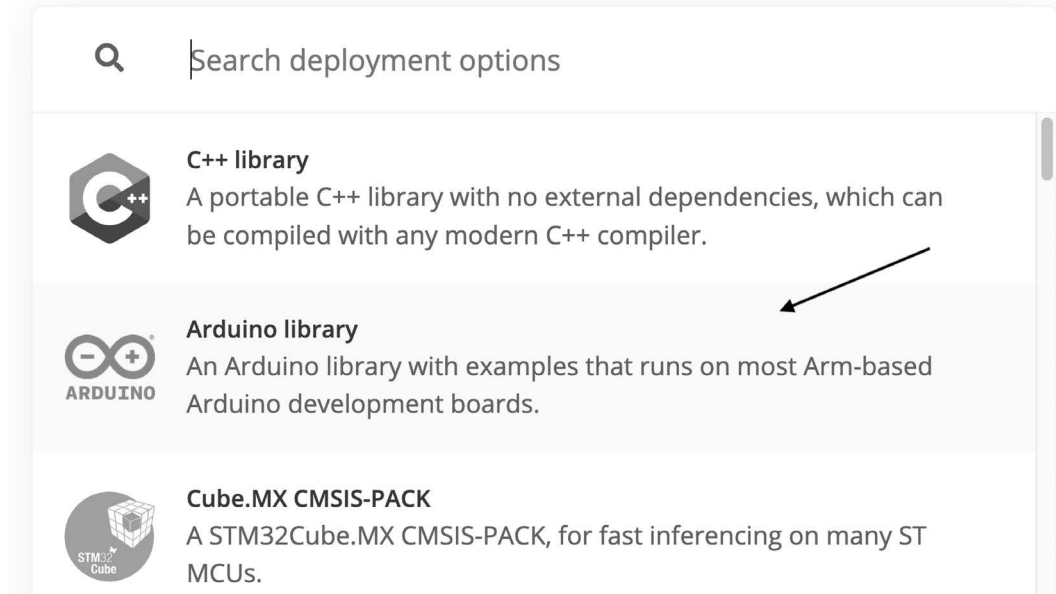
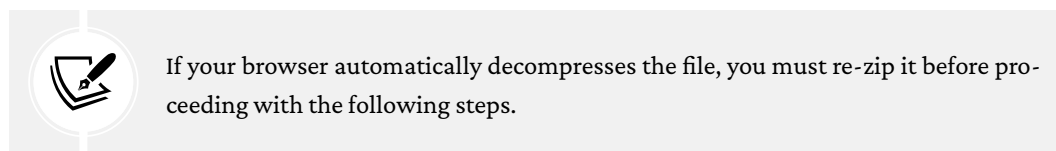


Figure 7.41: Edge Impulse deployment section

Then, click the **Build** button at the bottom of the page and save the ZIP file on your computer.



Now, open the Arduino sketch developed in the previous recipe and take the following steps to run the model inference with the Edge Impulse Inferencing SDK and transmit the coordinates of the centroids for the cans detected in the input image:

Step 1:

Import the library created with Edge Impulse.

Step 2:

Include the `<edge_impulse_project_name>_inferencing.h` header file in the sketch. For example, if the Edge Impulse project's name is `object_detection`, you should include the following file:

```
#include <object_detection_inferencing.h>
```

Including this header file is the only prerequisite for accessing the functions, constants, and C macros provided in the Edge Impulse Inferencing SDK.

Step 3:

Implement the function to load the data pixels from the camera frame with the signature matching the function `object` in the `signal_t` structure. To do so, create a function called `get_data_from_camera()` that accepts the `offset`, `length`, and `out_ptr` input arguments:

```
int get_data_from_camera(size_t offset,
                        size_t length,
                        float *out_ptr) {
```

Within this function, open a for loop to iterate over all pixels to process (`length`):

```
for(size_t i = 0; i < length; ++i) {
```

Within the for loop, load the grayscale pixel from the camera frame and form an RGB888 pixel:

```
uint8_t gray = frame[i + offset];
uint8_t r = gray;
uint8_t g = gray;
uint8_t b = gray;
```

As you can see from the previous code snippet, the pixel conversion from grayscale to RGB888 is straightforward since the red (r), green (g), and blue (b) channels have the same intensity and are equal to the grayscale value (gray).

Please note the use of the `offset` argument when accessing the grayscale pixel from the camera frame (`frame[i + offset]`). This offset is mandatory because the conversion from grayscale to RGB888 will be performed in slices to minimize the impact on memory resources. Therefore, the `offset` variable is required to know the position of the first pixel to be processed.

After obtaining the RGB888 channels, pack them in a floating-point variable and store the value in the output buffer:

```
float pixel_f = (r << 16) + (g << 8) + b;
out_ptr[i] = pixel_f;
```

Finally, close the for loop and return 0 to signify no errors in the conversion from grayscale to RGB888:

```
    } // Close for loop
    return 0;
}
```

Step 4:

In the `loop()` function, initialize the `signal_t` structure after reading the camera frame transmitted over the serial:

```
ei::signal_t signal;
signal.total_length = FRAME_SIZE;
signal.get_data = &get_data_from_camera;
```

As you can observe from the `signal_t` initialization, the `total_length` field is initialized with the total number of pixels in the camera frame (`FRAME_SIZE`), which matches the number of pixels of the ML model input.

Step 5:

In the `loop()` function, declare the `ei_impulse_result_t` structure to hold the output result:

```
ei_impulse_result_t result = { 0 };
```

Step 6:

In the `loop()` function, run the model inference using the `run_classifier()` function from the Edge Impulse Inferencing SDK:

```
run_classifier(&signal, &result, false);
```

The `run_classifier()` function will run the required DSP operations to prepare the model's input and execute the inference.



You might consider adding an LED toggle before each inference so that you can have a feeling of its execution time.

Step 7:

In the `loop()` function and after the model inference, save the centroids of the detected objects in an array. To do so, declare an array that can hold the centroid coordinates for the maximum number of possible detections:

```
uint32_t xy[EI_CLASSIFIER_OBJECT_DETECTION_COUNT][2];
```

The preceding array (`xy`) is declared statically and allows us to store the `x` and `y` coordinates for the maximum number of possible objects that the model can detect (`EI_CLASSIFIER_OBJECT_DETECTION_COUNT`). The `EI_CLASSIFIER_OBJECT_DETECTION_COUNT` is a C macro in the `model_metadata.h` file available in the Arduino library generated with Edge Impulse.

After the array declaration, declare an integer variable to keep track of the detected objects in the camera frame and initialize it to 0:

```
uint32_t num_objs = 0;
```

After, open a for loop to iterate over the list containing the detected objects:

```
uint32_t max_x = EI_CLASSIFIER_OBJECT_DETECTION_COUNT;
for (size_t ix = 0; ix < max_x; ix++) {
```

Within the for loop, check whether the score value of the potential detected object is 0. If so, skip the execution of the statements in the current loop iteration:

```
    auto bb = result.bounding_boxes[ix];
    if (bb.value == 0) {
        continue;
    }
```

If the score value is not 0, calculate the centroid coordinates, store them in the `xy` array, and increment the variable holding the number of detected objects (`num_objs`):

```
xy[num_objs][0] = bb.x + bb.width / 2;
xy[num_objs][1] = bb.y + bb.height / 2;
num_objs++;
```

Finally, close the for loop that iterates over the list of detected objects:

```
} // for (size_t ix = 0; ix < max_x; ix++)
```

Step 8:

In the `loop()` function, transmit the centroid coordinates over the serial:

```
Serial.println(num_objs);
for(uint32_t i = 0; i < num_objs; ++i) {
    Serial.print(xy[i][0]);
    Serial.print(",");
    Serial.println(xy[i][1]);
}
```

As you can see from the preceding code snippet, we first send the total number of detected objects in the camera frame and then transmit the centroid coordinates right after as comma-separated values.

Step 9:

Open the Python script with the implementation of the emulated camera (`emulated_camera.py`). Then, just before the `cv2.waitKey(33)` call, use the `pySerial` module to read the centroid coordinates transmitted over the serial and, for each, draw a circle with the OpenCV library in the camera frame to visualize their position. To do so, use the `serial.readline()` function to read the total number of detected objects in the camera frame:

```
data_str = serial_readline(ser)
num_objs = int(data_str)
```

After, use the `serial_readline()` function to retrieve the centroid coordinates for all detected objects:

```
for x in range(num_objs):
    data_str = serial_readline(ser)
    xy_str = data_str.split(",")
    xy = [int(xy_str[0]), int(xy_str[1])]
```

Before closing the for loop, draw a solid red circle in the resized BGR888 camera frame (`img_resized`) with the `circle()` OpenCV drawing function:

```
cv2.circle(img_resized, xy, 4, (0,0,255), -1)
```

The four mandatory input arguments for the `circle()` drawing function are the following:

- `img`: The image where the circle shape is drawn (`img_resized`).
- `center`: The center of radius in pixel coordinates. In our case, the centroid of the detected object (`xy`) is the center of radius.
- `radius`: The radius of the circle in pixels (for example, 4).
- `color`: The color of the circle in the same color format as the input image (for example, you can use `(0, 0, 255)` to draw a red circle in the BGR888 color format).

After drawing the circles, you can display the camera frame on the screen:

```
key = cv2.waitKey(33)
```

Now, compile and upload the sketch on the Raspberry Pi Pico. Then, launch the Python script:

```
$ python emulated_camera.py
```

After a few seconds, you will see the window with the camera frame on your screen. In contrast to the camera preview we have seen when testing the communication with the microcontroller, the video should be pretty laggy because of the high model inference time.

Now, take a can and place it in front of your webcam. After a few seconds, you should see a red circle drawn in the camera preview near the can, confirming the correct deployment of the FOMO model on the Raspberry Pi Pico!

There's more...with the SparkFun Artemis Nano!

In this recipe, we have learned how to run model inference using the Edge Impulse Inferencing SDK and display the centroids of the detected objects on the screen.

The same code can work on all Arduino compatible platforms with at least 256 Kbytes of RAM.

Therefore, what do you think about testing the application on the SparkFun Artemis Nano microcontroller?

However, it is essential to note that, at the time of writing, there is a known issue when compiling the project for the SparkFun Artemis Nano. The problem is due to a function name collision, which throws the following error message: `error: macro "F" passed 2 arguments, but takes just 1`.

The discussion around this problem, as well as the simple workaround, can be found in the Edge Impulse forum at the following link: <https://forum.edgeimpulse.com/t/macro-f-passed-2-arguments-but-takes-just-1/3026>. This function name collision can be easily solved by changing the template argument `T` in the `src/edge-impulse-sdk/tensorflow/lite/kernels/internal/reference/comparisons.h` within the Arduino Edge Impulse library to a different name (for example, `TFLiteFunc`). To do so:

1. Unzip the Arduino Edge Impulse library downloaded from Edge Impulse.
2. Open the `src/edge-impulse-sdk/tensorflow/lite/kernels/internal/reference/comparisons.h` with a text editor.
3. Inside the `comparisons.h` file, replace `F>` with `TFLiteFunc>`.
4. Inside the `comparisons.h` file, replace `F(` with `TFLiteFunc(`.
5. Compress the Arduino Edge Impulse library again with the zip tool and import the compressed file into the Arduino IDE.

By running the sketch on the SparkFun Artemis Nano, you will likely observe a slightly smoother video stream within the camera preview because of the better model's inference time on this device. The reason for that can be attributed to the SparkFun Artemis Nano's Arm Cortex-M4 CPU, which has more computing capabilities than the Arm Cortex-M0+ CPU of the Raspberry Pi Pico.

Summary

The recipes presented in this chapter demonstrated how to build an object detection application for microcontrollers with the help of Edge Impulse using a pre-trained FOMO model.

Initially, we learned how to prepare the dataset by acquiring camera frames with the webcam. Afterward, we delved into the model design. Here, we discussed how to choose a suitable image resolution and color format for an object detection model based on the FOMO architecture. Then, we explored the FOMO architecture to learn why it is ideal for memory-constrained devices.

After introducing the FOMO architecture, we trained the model and tested its accuracy on the test dataset and live images acquired with the webcam.

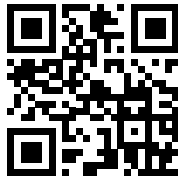
Finally, we implemented a Python script to emulate a microcontroller camera module and deployed the object detection model on the Raspberry Pi Pico using the Edge Impulse Inferencing SDK.

In this chapter, we have started discussing how to build a tinyML application with a vision sensor using Edge Impulse and the Raspberry Pi Pico. With the next project, we will continue working with this type of sensor on the Arduino Nano by using an actual camera module to recognize desk objects.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



8

Classifying Desk Objects with TensorFlow and the Arduino Nano

Convolutional neural networks (CNNs) have gained popularity because of their ability to unlock challenging **computer vision** tasks such as image classification, object recognition, scene understanding, and pose estimation, once considered impossible to solve. Nowadays, many modern camera applications are powered by these deep learning algorithms, and we just need to open the camera app on a smartphone to see them in action. However, computer vision tasks are not restricted to smartphones or cloud-based systems. In fact, these algorithms can now be accelerated in microcontrollers, albeit with their limited computational capabilities.

In this chapter, we will see the benefit of adding sight to our tiny devices by classifying two desk objects with the OV7670 camera module, in conjunction with the Arduino Nano.

In the first part, we will learn how to acquire images from the OV7670 camera module. Then, we will focus on the model design, applying transfer learning with the Keras API to recognize two objects we typically find on a desk: a mug and a book.

Finally, we will deploy the quantized TensorFlow Lite model on an Arduino Nano with the help of **TensorFlow Lite for Microcontrollers** (**tf-lite-micro**).

The goal of this chapter is to show you how to apply transfer learning with TensorFlow and learn the best practices of using a camera module with the Arduino Nano.

In this chapter, we will cover the following topics:

- Taking pictures with the OV7670 camera module
- Grabbing camera frames from the serial port with Python
- Acquiring QQVGA images with the YCbCr422 color format
- Building the dataset to classify desk objects
- Transfer learning with Keras
- Quantizing and testing the trained model with TensorFlow Lite
- Fusing the pre-processing operators for efficient deployment

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- An Arduino Nano 33 BLE Sense
- A micro-USB data cable
- 1 x half-size solderless breadboard
- 1 x OV7670 camera module
- 1 x push-button
- 18 x jumper wires (male to female)
- Laptop/PC with either Linux, macOS, or Windows
- Google Drive account



The source code and additional material are available in the Chapter08 folder in the GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Chapter08.

Taking pictures with the OV7670 camera module

In this first recipe, we will build an electronic circuit to take pictures with the OV7670 camera module, using the Arduino Nano. After assembling the circuit, we will use the pre-built `CameraCaptureRawBytes` sketch in the Arduino IDE to transmit the pixel values over the serial.

Getting ready

The **OV7670** camera module, illustrated in the following figure, is the main ingredient required in this recipe to take pictures with the Arduino Nano:

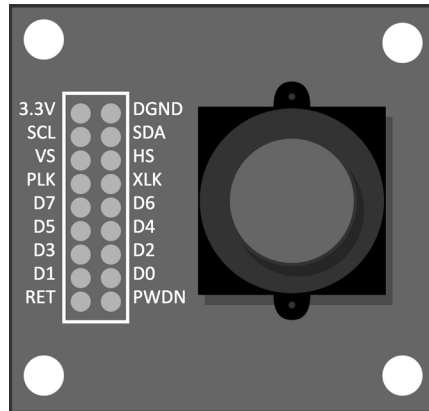



Figure 8:1: The OV7670 camera module

This vision sensor is one of the most affordable for tinyML applications, as you can buy it from various electronic distributors for less than \$10. Cost is not the only reason we went for this sensor, though. Other factors make this device our preferred option, such as the following:

- **Low frame resolution support:** Since microcontrollers have limited memory, we should consider cameras capable of transferring low-resolution images. The OV7670 camera unit is a good choice because it can output **QVGA** (320x240) and **QQVGA** (160x120) pictures.
- **Color format support:** The color format defines how the picture encodes the pixels and impacts image memory occupancy. The OV7670 camera module offers support for various color formats, such as **RGB565** and **YUV422**. We will explore these two-color formats later in the chapter to understand how they save memory and impact image quality.
- **Software library support:** Cameras can be complicated to control without a software driver. The OV7670 provides a software library for the Arduino Nano 33 BLE Sense board to easily capture images with a single function called `readFrame()`. This function captures and stores the image as an array that needs to be passed as an input argument.

The previous factors, in addition to the *voltage supply*, *power consumption*, *frame rate*, and *interface type*, are generally pondered when choosing a vision module for tinyML applications.

If you are using the Arduino Web Editor, the library required to control the OV7670 camera module with the Arduino Nano 33 BLE Sense microcontroller is pre-installed.



For those working with the local Arduino IDE, you will need to install the OV7670 Arduino library. To do so, you can follow the instructions provided at the following link: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Docs/install_OV7670_arduino_lib.md.

How to do it...

Let’s start this recipe by taking the breadboard and mounting the Arduino Nano vertically among the left and right terminal strips, as shown in the following figure:

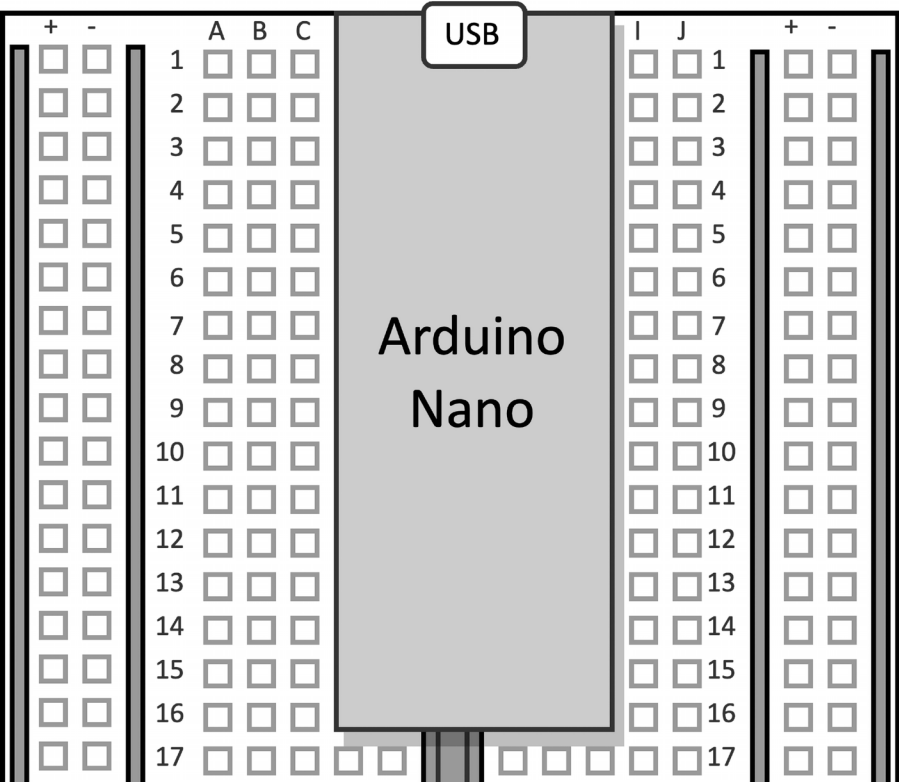


Figure 8.2: The Arduino Nano is mounted vertically between the left and right terminal strips.

Then, take the following steps to connect the Arduino Nano with the OV7670 camera module and a push-button:

Step 1:

Connect the OV7670 camera module to the Arduino Nano by using 16 male-to-female jumper wires, as illustrated in the following diagram:

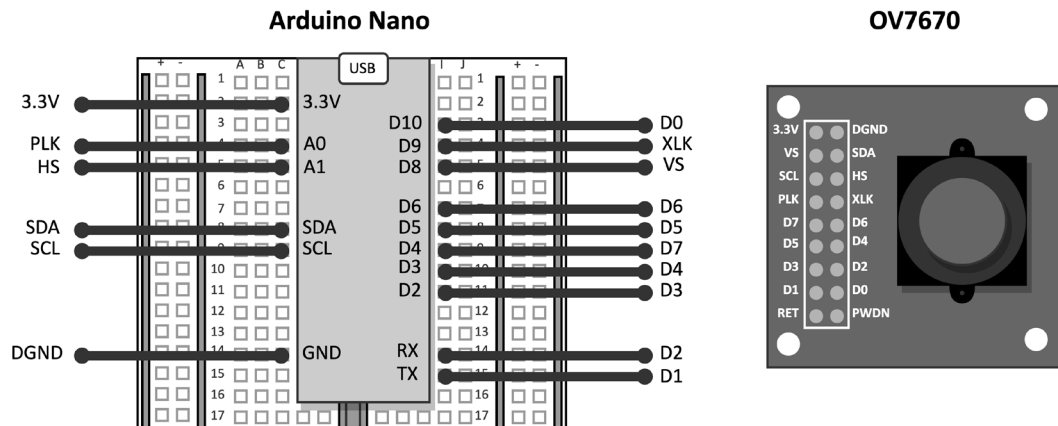


Figure 8.3: Wiring between the Arduino Nano and OV7670

The OV7670 camera module is connected to the Arduino Nano following the arrangement needed for the **Arduino_OV767X** support library, which can be found at the following link: https://github.com/arduino-libraries/Arduino_OV767X/blob/master/src.



Attention

Various manufacturers offer the OV7670 camera module with different pin configurations. For example, in some cases, the VS and SCL pins may be interchanged.

Step 2:

Add a push-button on the breadboard and connect it to the **P0.30** pin of Arduino Nano and **GND**:

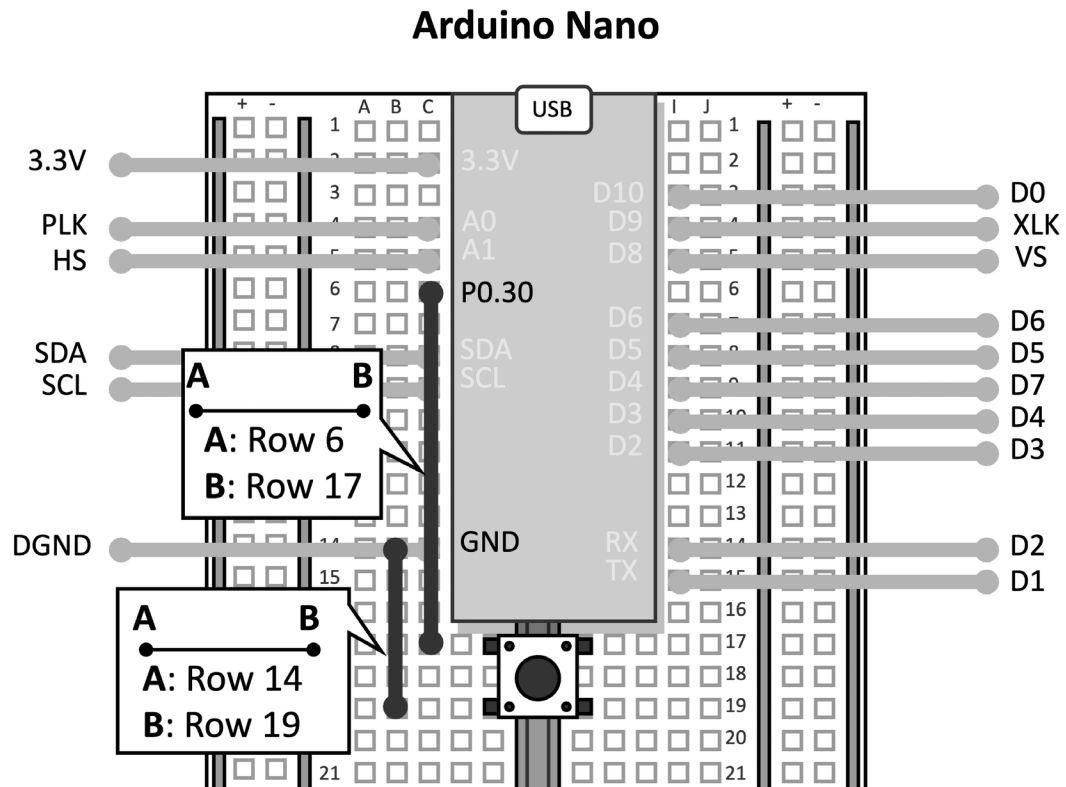


Figure 8.4: The push-button is connected between the P0.30 pin of Arduino Nano and GND

The push-button does not need the pull-up resistor because we will employ the one in the microcontroller.

Now, open the Arduino IDE to develop a sketch to take pictures whenever we press the push-button.

Step 3:

Open the `CameraCaptureRawBytes` sketch from **Examples -> FROM LIBRARIES -> ARDUINO_OV767X**:

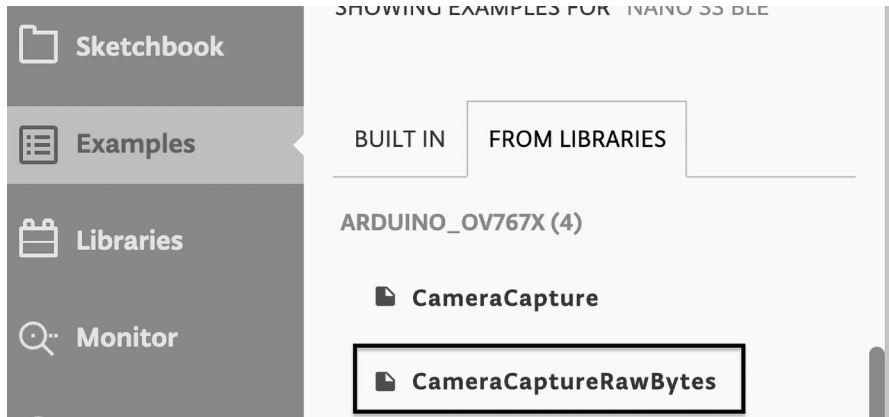


Figure 8.5: `CameraCaptureRawBytes` sketch

Then, copy the content of the `CameraCaptureRawBytes` file into a new sketch.

Step 4:

In the sketch, include the `mbed.h` header file to access the Mbed OS functionalities:

```
#include "mbed.h"
```

Then, declare and initialize a global object of type `mbed::DigitalIn` to read the state of the push-button:

```
mbed::DigitalIn button(p30);  
constexpr int PRESSED = 0;
```

In the previous code, the button Mbed object was initialized with the pin number argument of `p30` because the push-button is connected to the **P0.30** pin of the Arduino Nano.

Step 5:

Rename `bytesPerFrame` to `bytes_per_frame` to keep consistency with the lowercase naming convention used in the book.

Step 6:

At the beginning of the `setup()` function, set the button mode to `PullUp`:

```
void setup() {  
    button.mode(PullUp);  
}
```

After, ensure the baud rate of the serial peripheral is initialized to 115200:

```
Serial.begin(115200);  
while (!Serial);
```

The serial peripheral will be used in the `loop()` function to transmit the pixels of the captured image one by one through serial communication.



Do not alter or delete the remaining code that follows the serial peripheral initialization, since it is responsible for initializing the OV7670 camera module.

Step 7:

In the `loop()` function, add an `if` statement before `Camera.readFrame(data)` to check whether the push-button is pressed. If the button is pressed, take a picture from the OV7670 camera and send the pixel values over the serial:

```
if(button == PRESSED) {  
    Camera.readFrame(data);  
    Serial.write(data, bytes_per_frame);  
}
```

The `readFrame(data)` method captures an entire frame from the camera module and stores it in the `data` array, which needs enough memory to hold all pixels. In this sketch, this array has been declared and initialized globally to store a QVGA image with pixels encoded in the **RGB565** format. This format uses 2 bytes per pixel.

Compile and upload the sketch to the Arduino Nano. Now, you can open the serial monitor by clicking the **Monitor** option from the **Editor** menu. From there, you will see all the pixel values transmitted whenever you press the push-button.

There's more...

In this recipe, we learned how to take a picture with the OV7670 camera and transmit its pixels using the serial peripheral. However, *how long does it take to transmit an entire image?* The transmission time ($T_{transmission}$) depends on the number of bits to transfer and the baud rate of the serial peripheral, and it can be estimated using the following equation:

$$T_{transmission} = \frac{\text{Num bits to transfer}}{\text{Baud rate}}$$

Considering our case, where we have a QVGA image and a baud rate of 115200, the transmission time should be approximately 10 seconds, as the image size is 320x240x2 bytes, resulting in a total of 1,228,800 bits.

However, how do we know the camera operates correctly and captures images as intended?

In the upcoming recipe, we will validate the camera's functionality by developing a Python script to display the image on the screen from the pixel data transmitted over the serial.

Grabbing camera frames from the serial port with Python

In the previous recipe, we showed how to take images from the OV7670, but we didn't provide a method to display them. Therefore, it is now the time to implement a Python script locally to read the pixel values transmitted serially and show the resulting image on the screen.

Getting ready

In this recipe, we will develop a Python script locally to create images from the data transmitted over the serial. To facilitate this task, we will need two main Python libraries:

- **pySerial**, which allows us to read the data transmitted serially
- **Pillow** (<https://pypi.org/project/Pillow>), which enables us to create image files from the received data

The Pillow library is a fork of the **Python Imaging Library (PIL)** and can be installed with the following pip command:

```
$ pip install Pillow
```

Using the `fromarray()` method provided by this library, we can generate images from NumPy arrays that hold pixels in the **RGB888** format. This format uses 3 bytes to encode each pixel, assigning 8 bits to each color component (red, green, and blue), as shown in the following figure:

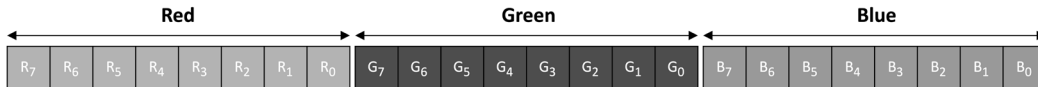


Figure 8.6: RGB888 color format

However, the sketch developed in the previous recipe doesn't send pixels in the RGB888 format but in the RGB565 format. Therefore, we are presented with two choices: perform the conversion in Python or within the Arduino sketch. We have opted for the latter, meaning we will modify the Arduino sketch to transmit images in the RGB888 format. This choice will allow us to practice color conversion in C.

Introducing the RGB565 color format

Before capturing a live frame using the OV7670 camera, the device needs to be initialized in the `setup()` function. The initialization is performed through the `begin()` method of the Camera object, which requires three input arguments: image resolution, color format, and camera **frame per second (FPS)**.



The only frame rates that can work with the Arduino Nano 33 BLE Sense board are 1 and 5 FPS.

In the **CameraCaptureRawBytes** sketch, the camera is initialized with a QVGA image resolution, RGB565 color format, and a frame rate of 1 FPS, as shown in the following code snippet:

```
Camera.begin(QVGA, RGB565, 1)
```

The high memory requirements of the RGB888 color format make it uncommon for microcontrollers, which is why the OV7670 software library does not support it. As a result, we need to write a routine to perform the conversion from RGB565 to RGB888.

RGB565 packs the pixel in 2 bytes, reserving 5 bits for the red and blue components and 6 bits for the green one:

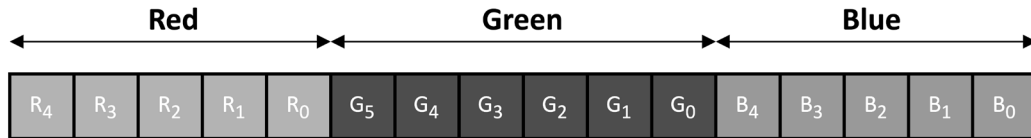


Figure 8.7: RGB565 color format

This color format is mainly used in embedded systems with limited memory capacity since it reduces the image size. However, it is worth noticing that *memory reduction is achieved by reducing the dynamic range of the color components*.

The current method of transmitting images over the serial has a drawback, wherein the Python script must know the image resolution beforehand. To address this challenge, we plan to send the picture with **metadata** that includes the image resolution.

Transmitting images over the serial

The metadata considered for this project will provide the following information:

- *The beginning of the image transmission:* We send the `<image>` string to signify the beginning of the communication.
- *The image resolution:* We send the resolution as a string of digits to say how many RGB pixels will be transmitted. The width and height will be sent on two different lines.
- *The completion of the image transmission:* Once we have sent all pixel values line by line, we transmit the `</image>` string to notify the end of the communication.

As you can guess, this metadata has been structured to validate the number of bytes transmitted by the microcontroller.

The pixel values will be sent right after the image resolution metadata and following the top-to-bottom, left-to-right order, also known as the **raster scan order**:

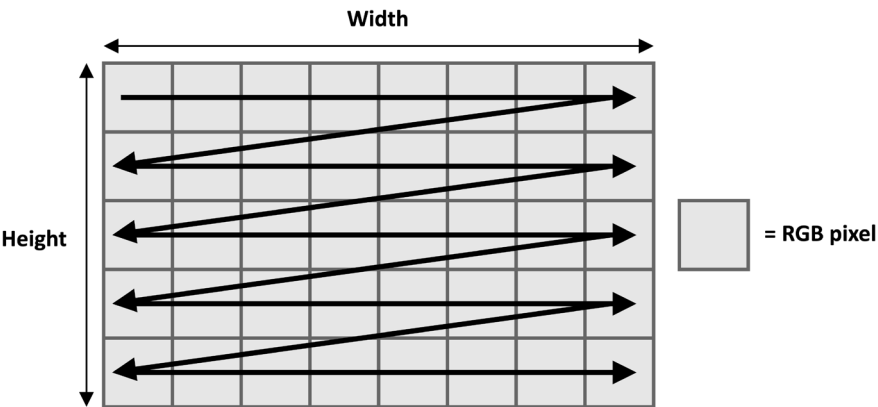


Figure 8.8: Raster scan order

The color components will be sent as strings of digits, terminated with a newline character (`\n`) following the RGB ordering. Therefore, the red channel comes first and the blue one last, as shown in the following diagram:

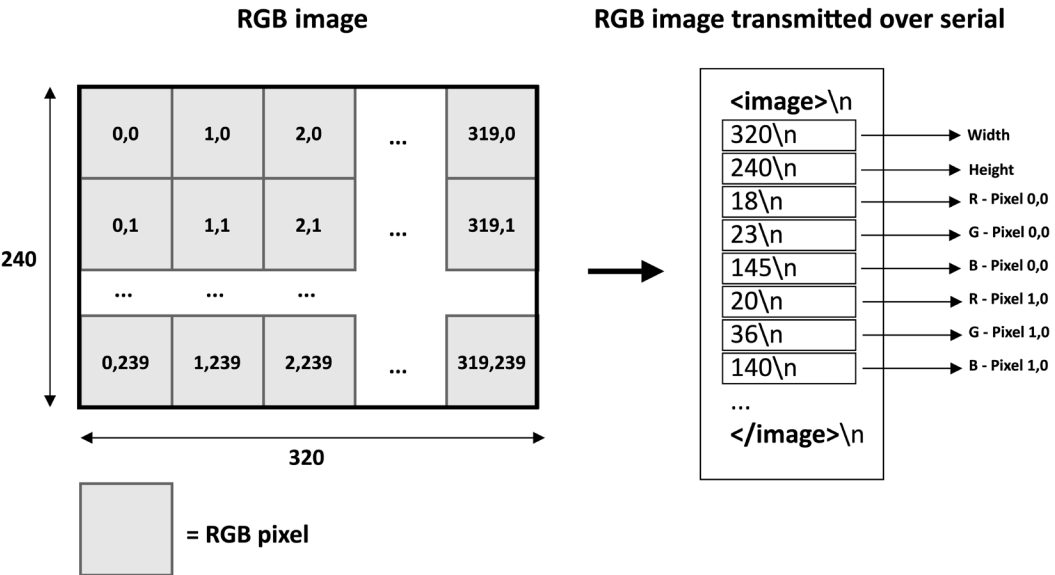


Figure 8.9: Communication protocol for the serial transmission of an RGB image

As you can observe from the preceding illustration, each color component is sent as a string of digits, terminated with a newline character (`\n`).

How to do it...

Open the Arduino sketch developed in the previous recipe and take the following steps to transmit the image in the format described in the preceding *Getting ready* section:

Step 1:

Write a function to convert the RGB565 pixel to RGB888:

```
void rgb565_rgb888(uint8_t* in, uint8_t* out) {
    uint16_t p = (in[0] << 8) | in[1];
    out[0] = ((p >> 11) & 0x1f) << 3;
    out[1] = ((p >> 5) & 0x3f) << 2;
    out[2] = (p & 0x1f) << 3;
}
```

The function takes 2 bytes from the input buffer to form the 16-bit RGB565 pixel. The first byte (`in[0]`) is left-shifted by eight positions to place it in the higher half of the `uint16_t p` variable. The second byte (`in[1]`) is set in the lower part:

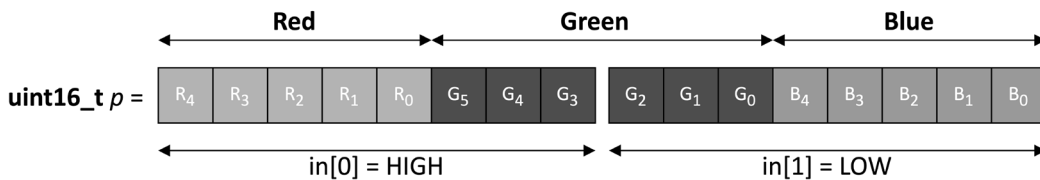


Figure 8.10: The `uint16_t` RGB565 pixel

After obtaining the 16-bit pixel in the `uint16_t p` variable, we extract the 8-bit color components as follows:

- *Red component* (`out[0]`): This is extracted by right-shifting the `p` variable by 11 positions, making the `R0` bit the least significant. Then, we clear all the non-red bits by applying a bitwise AND operation with `0x1F` (all bits are cleared except the first five).
- *Green component* (`out[1]`): This is extracted by right-shifting the `p` variable by 5 positions, making the `G0` bit the least significant. Then, we clear all the non-green bits by applying a bitwise AND operation with `0x3F` (all bits are cleared except the first six).

- *Blue component* (out[2]): This is extracted without right-shifting because the B0 bit is already the least significant. Therefore, we must clear the non-blue bits by applying a bitwise AND operation with 0x3F (all bits are cleared except the first five).

Lastly, the left-shift operation is performed on all color components to position the R4, G5, and B5 bits in the most significant bit of the corresponding 8-bit variables.

Step 2:

In the `setup()` function, enable the **test pattern** mode of the OV7670 camera after its initialization:

```
Camera.testPattern();
```

The test pattern mode allows us to verify the proper functioning of the camera. By enabling this feature, the camera module always returns a static image with color bands, which we can use to determine whether the camera is working as expected.

Step 3:

In the `loop()` function, replace the `Serial.write(data, bytes_per_frame)` statement with the routine to transmit the RGB888 image and metadata over the serial:

```
uint8_t rgb888[3];
Serial.println("<image>");
Serial.println(Camera.width());
Serial.println(Camera.height());
int32_t bytes_per_pixel = Camera.bytesPerPixel();
int32_t i = 0;
for(; i < bytes_per_frame; i+=bytes_per_pixel) {
    rgb565_rgb888(&data[i], &rgb888[0]);
    Serial.println(rgb888[0]);
    Serial.println(rgb888[1]);
    Serial.println(rgb888[2]);
}
Serial.println("</image>");
```

The communication starts by sending the `<image>` string and the resolution of the image (`Camera.width()` and `Camera.height()`) over the serial.

Next, we iterate all bytes stored in the camera buffer and apply the RGB565-to-RGB888 conversion with the `rgb565_rgb888()` function. Every color component is then sent as a string of digits with the newline character (`\n`).

Once we have iterated through all the pixels, we send the `</image>` string to indicate the end of the data transmission.

Now, you can compile and upload the sketch to the Arduino Nano.

Step 4:

On your computer, create a new Python script called `parse_camera_frame.py` and import the necessary Python libraries:

```
import numpy as np
import serial
from PIL import Image
```

Then, initialize the serial communication with the port and baud rate (115200) used by the microcontroller:

```
ser = serial.Serial()
ser.port = '/dev/ttyACM0'
ser.baudrate = 115200
```

The easiest way to determine the serial port name is from the device drop-down menu in the Arduino IDE:

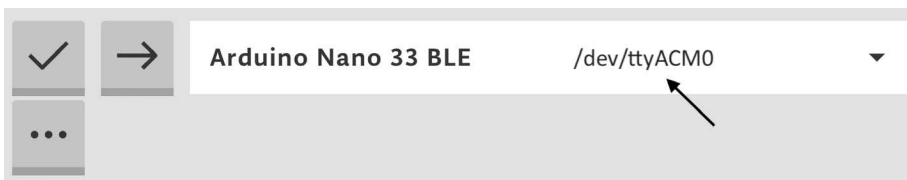


Figure 8.11: The serial port name is reported in the Arduino IDE

After the initialization of the serial communication, open the serial port and discard the content in the input buffer:

```
ser.open()
ser.reset_input_buffer()
```

Once the input buffer is cleared, the program can start receiving the data transmitted over the serial.

Step 5:

Open a while loop that runs indefinitely until the program ends. Within the loop, read data from the serial port. Then, verify whether we have received the <image> string indicating the beginning of the image transmission:

```
def serial_readline(obj):  
    data = obj.readline()  
    return data.decode("utf-8").strip()  
  
while True:  
    data_str = serial_readline.ser)  
    if str(data_str) == "<image>":
```

Step 6:

Upon detecting the <image> string, read the frame resolution (width and height) and create a NumPy array of the appropriate size to hold the image data:

```
w_str = serial_readline.ser)  
h_str = serial_readline.ser)  
w = int(w_str)  
h = int(h_str)  
c = int(3) # Number of color components  
image = np.empty((h, w, c), dtype=np.uint8)
```

Once the array is created, we can populate it with the image data received from the serial port.

Step 7:

Read the pixel values transmitted over the serial and store them in the NumPy array:

```
for y in range(0, h):  
    for x in range(0, w):  
        for d in range(0, c):  
            data_str = serial_readline.ser)  
            image[y][x][d] = int(data_str)
```

For a more efficient solution, you may consider the following alternative code, which does not use nested for loops:

```
for i in range(0, w * h * c):
    d = int(i % c)
    x = int((i / c) % w)
    y = int((i / c) / w)
    data_str = serial_readline(ser)
    image[y][x][d] = int(data_str)
```

Step 8:

Check if the last line contains the `</image>` string. If so, display the image on the screen:

```
data_str = serial_readline(ser)
if str(data_str) == "</image>":
    image_pil = Image.fromarray(image)
    image_pil.show()
```

Ensure the Arduino serial monitor is closed because the serial peripheral on your computer can only communicate with one application. Then, make sure the Arduino Nano is connected to your computer and run the Python script:

```
$ python parse_camera_frame.py
```

Now, whenever you press the push-button, the Python script will parse the data transmitted over the serial and, after a few seconds, display an image with eight color bands:

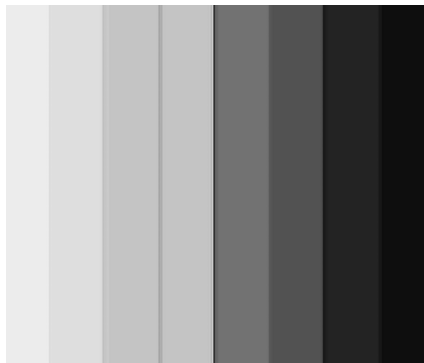


Figure 8.12: Expected output from the OV7670 camera module



The color version of the preceding image can be found at the following link: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Chapter08/Images/test_qvga_rgb565.png.

If you cannot obtain the previous test pattern, we recommend verifying the wiring between the camera and the Arduino Nano before proceeding to the following recipe.

There's more...

In this recipe, we learned how to convert images from RGB565 to RGB888 using the C language, as well as how to use the Pillow library to create and display images in Python.

Our Python script is designed only to receive pixels in the RGB888 format from the serial port. Nonetheless, transmitting images in the RGB565 format over the serial port and converting them to RGB888 within the Python script is a viable option. If you choose this path, we advise including the color format details within the image metadata, ensuring that the Python script can determine the correct color conversion method.

Having validated the camera's functionality, we will now focus on optimizing the camera acquisition for devices with limited memory.

In the upcoming recipe, we will tweak the Arduino sketch to capture images at a lower resolution and adopt YCbCr422. This color format is not only memory-efficient but also offers superior image quality compared to RGB565.

Acquiring QQVGA images with the YCbCr422 color format

When compiling the previous sketch for the Arduino Nano, you may have noticed the following warning message in the IDE's output log: *Low memory available, stability may occur*. This warning message appears because the QVGA image in the RGB565 color format requires a buffer of 153.6 KB, equivalent to roughly 60% of the available SRAM in the microcontroller.

In this recipe, we will show how to acquire an image at a lower resolution and use the YCbCr422 color format to reduce memory requirements, without compromising image quality.

Getting ready

Images are well known to require big chunks of memory, which might be a problem when dealing with microcontrollers.

Lowering the image resolution is a common practice to reduce the image memory size. Common image resolutions for microcontrollers are smaller than the QVGA (320x240) previously used, such as **QQVGA** (160x120) or **QQQVGA** (80x60).



Even lower-resolution images exist, but they are not always suitable for computer vision applications.

As we saw in the previous recipe, the RGB565 format saves memory by lowering the color components' dynamic range. However, the OV7670 camera module offers an alternative and more efficient color encoding: **YCbCr422**. Let's explore in the following subsection how this format compresses color information and how we can perform a conversion to RGB888, the color format commonly used by the inputs in most ML-based vision models.

Converting YCbCr422 to RGB888

YCbCr422 is a digital color encoding that does not express the pixel color in terms of red, green, and blue intensities but, rather, in terms of **brightness (Y)**, **blue-difference (Cb)**, and **red-difference (Cr)** chroma components.

The OV7670 camera module can output images in the YCbCr422 format, where *Cb and Cr components are shared between two adjacent pixels on a single scanline*. As a result, a single YCbCr422 unit is made of 4 bytes to encode 2 pixels, as shown in the following figure:

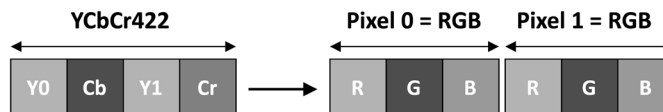


Figure 8.13: 4 bytes in the YCbCr422 format packs 2 RGB888 pixels

Therefore, a YCbCr422 image has the same number of bytes as an RGB565 image but provides better image quality, comparable to RGB888. The only downside is that converting YCbCr422 to RGB888 requires more arithmetic operations.

This drawback should be evident from the following tables, where we have reported the formulas to perform the color conversion from YCbCr422 to RGB888 using integer arithmetic only:

$\begin{aligned} \text{Cb} &= \text{Cb} - 128 \\ \text{Cr} &= \text{Cr} - 128 \end{aligned}$
Red
$R_i = Y_i + \text{Cr} + (\text{Cr} \gg 2) + (\text{Cr} \gg 3) + (\text{Cr} \gg 5)$
Green
$G_i = Y_i - ((\text{Cb} \gg 2) + (\text{Cb} \gg 4) + (\text{Cb} \gg 5)) - ((\text{Cr} \gg 1) + (\text{Cr} \gg 3) + (\text{Cr} \gg 4) + (\text{Cr} \gg 5))$
Blue
$B_i = Y_i + \text{Cb} + (\text{Cb} \gg 1) + (\text{Cb} \gg 2) + (\text{Cb} \gg 6)$

Figure 8.14: Formulas to extract two RGB888 pixels from one YCbCr422 unit

In Figure 8.14, the i subscript in R_i , G_i , B_i , and Y_i represents the RGB pixel index, either 0 (the first pixel) or 1 (the second pixel).

How to do it...

Open the Arduino sketch written in the previous recipe, and follow the steps to acquire images with a QQVGA resolution, using the YCbCr422 color format from the OV7670 camera module:

Step 1:

Resize the camera buffer (data) to accommodate a QQVGA image in the YCbCr422 color format:

```
uint8_t data[160 * 120 * 2];
```

The QQVGA resolution makes the frame buffer four times smaller than the one used in the previous recipe.

Step 2:

Write a function to clamp a value between 0 and 255:

```
template <typename T>
inline T clamp_0_255(T x) {
    return std::max(std::min(x, (T)(255)), (T)(0));
}
```

The `clamp_0_255()` function will be employed in the next step to guarantee that the color channels of the RGB888, derived from the **Y**, **Cb**, and **Cr** values, are between 0 and 255.

Step 3:

Write a function to extract an RGB888 pixel from the **Y**, **Cb**, and **Cr** components, using the formulas reported in the *Getting ready* section of this recipe:

```
void ycbcr422_rgb888(int32_t Y, int32_t Cb,
                    int32_t Cr, uint8_t* out) {
    Cr = Cr - 128;
    Cb = Cb - 128;
    int32_t r;
    int32_t g;
    int32_t b;

    r = Y + Cr + (Cr >> 2) + (Cr >> 3) + (Cr >> 5);
    g = Y - ((Cb >> 2) + (Cb >> 4) + (Cb >> 5)) -
        ((Cr >> 1) + (Cr >> 3) + (Cr >> 4));
    b = Y + Cb + (Cb >> 1) + (Cb >> 2) + (Cb >> 6);
    out[0] = clamp_0_255(r);
    out[1] = clamp_0_255(g);
    out[2] = clamp_0_255(b);
}
```

It is important to note that the preceding function returns only one RGB888 pixel, since it accepts only one brightness component (**Y**) as an input argument. Therefore, to obtain two RGB888 pixels, we must invoke this function twice, passing in two different **Y** values as input for each call.

Step 4:

In the `setup()` function, initialize the OV7670 camera to capture QQVGA frames with the YCb-Cr422 (YUV422) color format:

```
if (!Camera.begin(QQVGA, YUV422, 1)) {
    Serial.println("Failed to initialize camera!");
    while(1);
}
```

Unfortunately, the OV7670 driver interchanges YCbCr422 with YUV422, leading to some confusion. The main difference between these two formats is that YUV422 is for analog TV. Therefore, although we pass YUV422 to the `begin()` method, we actually initialize the device for YCbCr422.

Step 5:

In the `loop()` function, remove the statement that iterates over the RGB565 pixels stored in the camera buffer. Then, write a routine to read 4 bytes from the YCbCr422 camera buffer, perform the conversion to RGB888, and transmit the pixels over the serial:

```
int32_t step_bytes = Camera.bytesPerPixel() * 2;
int32_t i = 0;
for(; i < bytes_per_frame; i+=step_bytes) {
    const int32_t Y0 = data[i + 0];
    const int32_t Cr = data[i + 1];
    const int32_t Y1 = data[i + 2];
    const int32_t Cb = data[i + 3];
    ycbcr422_rgb888(Y0, Cb, Cr, &rgb888[0]);
    Serial.println(rgb888[0]);
    Serial.println(rgb888[1]);
    Serial.println(rgb888[2]);
    ycbcr422_rgb888(Y1, Cb, Cr, &rgb888[0]);
    Serial.println(rgb888[0]);
    Serial.println(rgb888[1]);
    Serial.println(rgb888[2]);
}
```

In the preceding code, we voluntarily swapped **Cb** and **Cr** because the *OV7670 driver returns the Cr component before the Cb one*.

Now, compile and upload the sketch on the Arduino Nano. Then, run the `parse_camera_frame.py` Python script implemented in the previous recipe:

```
$ python parse_camera_frame.py
```

Then, press the push-button on the breadboard. After a few seconds, you should see on the screen, again, an image with eight color bands. This time, the image should be smaller but with more vivid colors than the one captured with the RGB565 format.

There's more...

In this recipe, we learned how to lower the image resolution and use a more memory-efficient method.

In some applications, **grayscale** can be used as an even more aggressive color format to reduce the memory size of images. When using grayscale, each pixel is represented by a single brightness value, ranging from 0 (black) to 255 (white). Therefore, there is no color information. By using one channel instead of three, the image size can be reduced by three times, making it a desirable option in situations where the color information may not be relevant.

Now that we can capture frames from the camera, it is time to create the dataset to classify desk objects.

In the upcoming recipe, we will extend the `parse_camera_frame.py` Python script to save images as `.png` files and upload them to Google Drive.

Building the dataset to classify desk objects

In this recipe, we will build the dataset by collecting images of the mug and book, with the OV7670 camera and the Arduino Nano. The image files will then be uploaded to Google Drive to train the ML model in the next recipe.

Getting ready

Training a deep neural network from scratch for image classification commonly requires a dataset with 1,000 images per class. As you might guess, collecting such a vast number of pictures would be time-consuming. To overcome this challenge, we will employ the technique we applied in the previous *Chapter 7, Detecting Objects with Edge Impulse Using FOMO on the Raspberry Pi Pico: transfer learning*.

This ML technique, which we will exploit in the following recipe, allows us to build a dataset with just 20 samples per class.

How to do it...

Before implementing the Python script, remove the test pattern mode (`Camera.testPattern()`) in the Arduino sketch so that you can acquire live images. After that, compile and upload the sketch on the platform.

Then, copy the Python script implemented in the previous recipe in a new file called `build_dataset.py`. Open the `build_dataset.py` file and take the following steps to adjust the Python script to save the captured images as `.png` files and upload them to Google Drive:

Step 1:

Import the **UUID** Python module to produce unique filenames for the `.png` files:

```
import uuid
```

Then, declare two variables to hold the label's name and the ID of the destination folder in Google Drive:

```
label = ""  
gdrive_id = ""
```

The `label` variable will be the prefix for the filename of the `.png` files, while the `gdrive_id` one will hold the ID of the destination folder in Google Drive. Both variables will be assigned during the image acquisition process.

Step 2:

Ensure you have the **PyDrive** library pip installed in your local Python development environment (`pip install pydrive`), and grant permission for your account to use the Google Drive API:

```
from pydrive.auth import GoogleAuth  
gauth = GoogleAuth()  
gauth.LocalWebserverAuth()
```

The preceding code will open a window in the web browser to request access to your Google Drive.



To learn more about the PyDrive library, you can refer to *Chapter 2, Unleashing Your Creativity with Microcontrollers*.

Then, use the `gauth` **GoogleAuth** object to create a local instance of Google Drive:

```
from pydrive.drive import GoogleDrive  
drive = GoogleDrive(gauth)
```

The PyDrive library requires this **OAuth** JSON file to access Google Drive, which must be located in the same directory as your Python script and renamed `client_secrets.json`.

Step 3:

After receiving the whole image over the serial, crop it into a square shape and display it on the screen:

```
if str(data_str) == "</image>":
    image_pil = Image.fromarray(image)
    crop_area = (0, 0, h, h)
    image_cropped = image_pil.crop(crop_area)
    image_cropped.show()
```

We crop the acquired image into a square shape because the pre-trained model will consume an input with a square aspect ratio. The cropping is done by taking an area with dimensions matching the height of the original picture, as shown in the following figure:

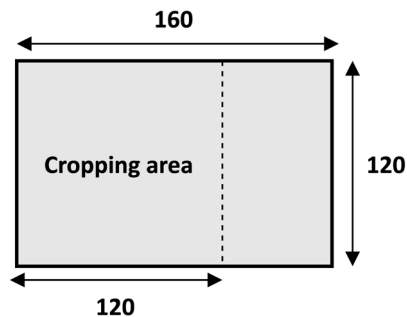


Figure 8.15: Cropping area

The picture is cropped using the `crop()` method of the Pillow library and then displayed on the screen.

Step 4:

Use the Python `input()` function to ask whether the image can be saved. If the user types `y` on the keyboard, ask for the label's name:

```
key = input("Save image? [y] for YES: ")
if key == 'y':
    str_label = ""Provide the label's name or
```



```

        leave it empty to use
        [{}]:"".format(label)
    label_new = input(str_label)
    if label_new != '':
        label = label_new

```

If the user does not provide the label's names, the program will not modify the content of the `label` variable.

Then, save the image as a `.png` file:

```

unique_id = str(uuid.uuid4())
filename = label + "_" + unique_id + ".png"
image_cropped.save(filename)

```

Step 5:

Use the Python `input()` function to ask for the ID of the Google Drive destination folder:

```

str_gid = """Provide the Google Drive folder ID
or leave it empty to use
[{}]:"".format(gdrive_id)
gdrive_id_new = input(str_gid)
if gdrive_id_new != '':
    gdrive_id = gdrive_id_new

```

If the user does not provide the Google Drive destination folder ID, the program will not modify the content of the `gdrive_id` variable.

Then, upload the file to Google Drive:

```

gfile = drive.CreateFile({'parents': [{'id':
                                     gdrive_id}]})
gfile.SetContentFile(filename)
gfile.Upload()

```

Step 6:

Launch the web browser and access Google Drive. There, create a new folder called `dataset` to store the images captured with the OV7670 camera module.

Inside the dataset folder, create two subfolders named `book` and `mug`. These folder names match the labels of the objects we aim to identify. As a result, the images acquired with the camera will be placed within the subfolder, corresponding to its associated label.

Besides the `book` and `mug` subfolders, create another directory called `unknown` to save images that do not represent a mug or a book. Now, within your dataset folder, you should have the following three directories:

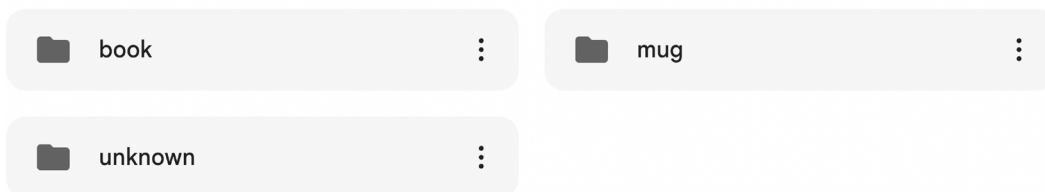


Figure 8.16: Folder structure in Google Drive

Take note of the Google Drive file ID of the preceding three directories (`book`, `mug`, and `unknown`), which is the last part of the **Uniform Resource Locator (URL)** of the directory when viewed in the web browser. The Google Drive ID is required to upload the files to Google Drive with the PyDrive library.

Now, run the `build_dataset.py` Python script to acquire 20 images of a mug and a book with the OV7670 camera:

```
$ python build_dataset.py
```

When you press the push-button on the breadboard, you should see the captured image on the screen after a few seconds. For each sample, you must provide the label (`book`, `mug`, or `unknown`) and the Google Drive ID of the destination folder in Google Drive.



Since we only capture a limited number of images per class, we suggest taking them from various angles.

Once you have captured these samples, take 20 pictures for the `unknown` class, representing cases where you have neither a mug nor a book.

The dataset is set up at this point, and we can move forward with training the model.

There's more...

In this recipe, we learned how to upload image samples acquired with the OV7670 camera to Google Drive to build the training dataset.

Although our current goal is to identify just two objects, the approach we have showcased to prepare the dataset in Google Drive is scalable, meaning adding more classes to the dataset can be done seamlessly if required. Therefore, you could consider adding another object, such as a smartphone, alongside the existing ones for mugs and books.

Now that the dataset is prepared, we can focus on the model design and training.

In the upcoming recipe, we will demonstrate how you can leverage transfer learning using TensorFlow to build a model to classify desk objects.

Transfer learning with Keras

Transfer learning is an effective technique to train a model when dealing with small datasets.

In this recipe, we will exploit it alongside the MobileNet v2 pre-trained model to recognize our two desk objects.

Getting ready

The basic principle behind transfer learning is to *exploit features learned for one problem to address a new and similar one*. Therefore, the idea is to take layers from a previously trained model, commonly called a **pre-trained model**, and add some new trainable layers on top of them:

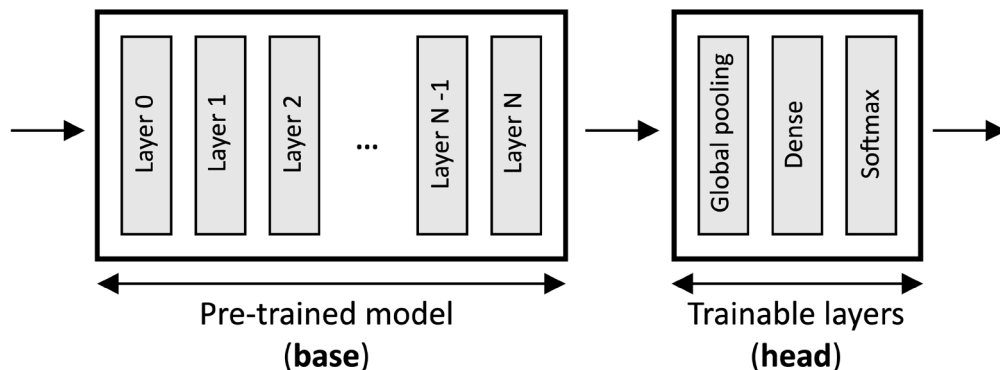


Figure 8.17: Model architecture with transfer learning

The pre-trained model's layers are frozen, meaning their weights cannot change during training. These layers are the **base** (or **backbone**) of the new architecture and aim to extract features from the input data. These features feed the trainable layers, the only layers to be trained from scratch.

The trainable layers are the **head** of the new architecture, and for an image classification problem, they aim to perform classification using the following standard sequence of operations:

1. Global pooling layer
2. Dense layer (fully connected layer)
3. Softmax layer

Since the layers composing the feature extraction are already trained, we can train the new model quickly and with a few samples in the dataset.

Keras provides different pre-trained models, such as **VGG16**, **ResNet50**, **InceptionV3**, **MobileNet**, and so on. Therefore, which one should we use?

When considering a pre-trained model for tinyML, model size is one of the key metrics to keep in mind to fit the deep learning architecture into memory-constrained devices.

From the list of Keras' pre-trained models, **MobileNet v2** is the network with fewer parameters, and it is tailored to be deployed on target devices with reduced computational power.



The list of Keras' pre-trained models is available at the following link: <https://keras.io/api/applications>.

Behind the MobileNet network design choices

MobileNet v2 is the second generation of MobileNet networks and compared to the previous one (**MobileNet v1**), it has half as many operations and higher accuracy.

This model is the perfect place to MobileNet network design choices learn from the architectural choices that made MobileNet networks small, fast, and accurate for edge inferencing. One of the successful design choices that made the first generation of MobileNet networks suitable for edge inferencing was the adoption of **depthwise convolution** layers, which expedited convolution computation.

Convolution layers are well known for their high computational and memory requirements, especially when dealing with 3x3 or larger kernel sizes. In such cases, additional temporary buffers and auxiliary data transformation may be necessary to reduce the computation to a matrix multiplication routine.



The most popular methods to optimize the computation of convolutional layers with kernel sizes larger than 1x1 are the im2col, indirect-gemm, and Winograd algorithms.

The idea behind MobileNet v1 was to replace the convolution operation with **depthwise separable convolution**, which consists of the following:

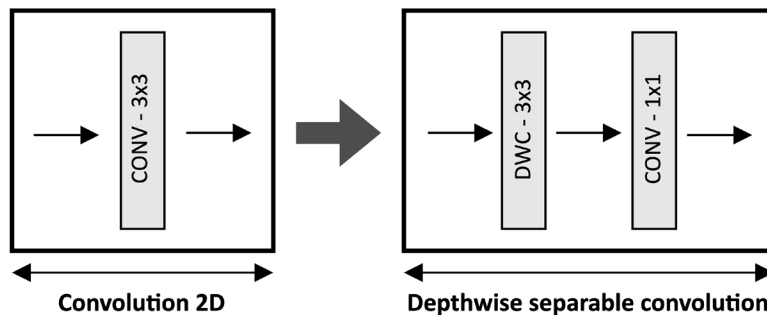


Figure 8.18: Depthwise separable convolution

As you can observe from the preceding illustration, depthwise separable convolution consists of a depthwise convolution with a 3x3 filter size (**DWC – 3x3**), followed by a convolution layer with a 1x1 kernel size (**CONV – 1x1**). This solution brings less trainable parameters, less memory usage, and a lower computational cost.

MobileNet v2 is built on top of the ideas of MobileNet v1, but it reduces the computational cost further *by carrying out convolutions on tensors with fewer channels*. From an ideal computational standpoint, it would be advantageous for all layers to operate on tensors with fewer channels (**feature maps**) to improve model latency. However, in practice, this must be balanced against accuracy, as it is crucial to maintain relevant features for the given problem.

Depthwise separable convolution alone cannot help because reducing the number of feature maps typically causes a drop in the model accuracy. Therefore, MobileNet v2 introduced the **inverted residual block**, a variation of the **residual block** used in ResNet, to help achieve this goal:

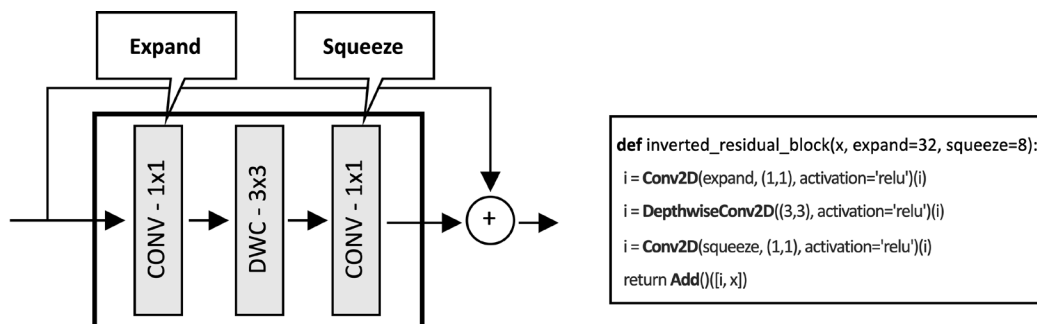


Figure 8.19: Bottleneck residual block

The inverted residual block acts as a **feature compressor**. As illustrated in the preceding diagram, the input is processed by the pointwise convolution, which increases the number of feature maps. Then, the convolution's output feeds the depthwise separable convolution layer to compress the features in fewer output channels.

Keras provides more than one variant of MobileNet v2, which are different in terms of:

- The input image resolution
- The alpha value

The **alpha value** is a floating-point number that controls the number of feature maps in each network layer. When this value is set to 1, each layer has the default number of feature maps, while smaller values reduce them proportionally. Since we want to obtain a model suitable for microcontrollers, the smallest possible alpha value should be preferred, which is 0.35.



The list of all Keras MobileNet v2 pre-trained models is available at the following link: https://github.com/keras-team/keras-applications/blob/master/keras_applications/mobilenet_v2.py.

How to do it...

Open the web browser and create a new Colab notebook. Then, follow the steps to apply transfer learning with the MobileNet v2 pre-trained model:

Step 1:

Mount the top-level Google Drive directory:

```
from google.colab import drive
drive.mount('/content/drive/')
```

The `mount()` function will open a window in the web browser to log in to your Google account. Once logged in, the root directory of Google Drive will be mounted in Colab, and you can access its contents.

Step 2:

Create a string variable, and assign it the path to the folder within Google Drive, with the training dataset containing the book, mug, and unknown subfolders:

```
train_dir = "drive/MyDrive/dataset"
```

Then, use the `image_dataset_from_directory()` function from Keras (https://www.tensorflow.org/versions/r2.11/api_docs/python/tf/keras/utils/image_dataset_from_directory) to prepare the training and validation datasets with a 80/20 split:

```
import tensorflow as tf

ds = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    validation_split=0.2,
    subset="both",
    seed=123,
    interpolation="bilinear",
    image_size=(48, 48))

train_ds = ds[0]
val_ds = ds[1]
```

The `image_dataset_from_directory()` function is used to generate the `tf.data.Dataset` data object from the image files stored in the training directory.

Since the directory structure includes a subfolder for each output class, this function automatically assigns the appropriate labels to each image, using the name of the folder it resides in.

The dataset is split between the train (80%) and validation (20%) sets through the `validation_split` and `subset` input arguments.

The `validation_split` argument defines the fraction of data to reserve to the validation set. Since we want to assign 20% of the sample to the validation dataset, it must be set to `0.2`. Meanwhile, the `subset` argument defines the dataset type to return. Since we aim to obtain the train and validation dataset, it must be set to `"both"`.

The `image_dataset_from_directory()` function also allows us to resize the train image samples to the model input resolution. MobileNet v2 supports any input image resolution larger than 32x32. Adopting a small input resolution may be tempting to reduce memory usage. However, we must consider that lower input resolutions can lead to low model accuracy. For this project, we have opted for an input model size of 48x48, as it offers a good tradeoff between accuracy and memory usage. The train images are resized through the `image_size` and `interpolation` arguments, which define the target image resolution (48, 48) and the sampling interpolation policy (bilinear).

Step 3:

Get the number of classes from the training dataset:

```
class_names = train_ds.class_names
num_classes = len(class_names)
```

Step 4:

Rescale the pixel values from [0, 255] to [-1, 1]:

```
rescale = tf.keras.layers.Rescaling(1./255, offset= -1)
train_ds = train_ds.map(lambda x, y: (rescale(x), y))
val_ds   = val_ds.map(lambda x, y: (rescale(x), y))
```

The reason for rescaling the pixel values from [0, 255] to [-1, 1] is that the pre-trained model expects an input image with pixel values between -1 and 1.

Step 5:

Import the MobileNet v2 pre-trained model, with the weights trained on the **ImageNet** dataset and `alpha=0.35`. Furthermore, set the input image size to (48, 48, 3) and exclude the top layers:

```
from tensorflow.keras.applications.mobilenet_v2 import MobileNetV2

base_model = MobileNetV2(input_shape=(48, 48, 3),
```



```
include_top=False,  
weights='imagenet',  
alpha=0.35)
```

The `include_top` argument determines whether to include or exclude the top layers of the MobileNet v2 model, which are responsible for classification. In the transfer learning case, we generally wish to utilize the pre-trained model purely as a feature extractor, so it is crucial to exclude the top layers. By doing so, we can append new trainable layers tailored to solving a different problem.

In the output log, you may have noticed the following warning message:

```
WARNING:tensorflow:'input_shape' is undefined or non-square, or 'rows' is  
not in [96, 128, 160, 192, 224]. Weights for input shape (224, 224) will  
be loaded as the default.
```

This warning message is reported because Keras provides pre-trained models for specific input image resolutions. Since the input resolution used in our project is not included in Keras' pre-trained models, Keras informs us that it will use weights trained on images with a resolution of 224x224.

Step 6:

Freeze the weights so that you do not update these values during training:

```
base_model.trainable = False  
feat_extr = base_model
```

The `base_model.trainable` option is set to `false` to make all layers of the pre-trained model non-trainable, meaning their weights will not change during training. It is crucial to set this option to prevent them from being updated or overwritten during the training process.

To verify that we have set all weights to non-trainable, we can print the following information:

```
print("num. weights:",  
      len(base_model.weights))  
print("num. trainable_weights:",  
      len(base_model.trainable_weights))  
print("num. non_trainable_weights:",  
      len(base_model.non_trainable_weights))
```

The previous code should report the number of total weights, trainable weights, and non-trainable weights for the pre-trained model in the output log. If all weights are set to non-trainable, the number of non-trainable weights should match the total number of weights.

Step 7:

Augment the input data:

```
preproc = tf.keras.layers.experimental.preprocessing
augmen = tf.keras.Sequential([
    preproc.RandomRotation(0.2),
    preproc.RandomFlip('horizontal'),
])

train_ds = train_ds.map(lambda x, y: (augmen(x), y))
val_ds    = val_ds.map(lambda x, y: (augmen(x), y))
```

Since we don't have a large dataset, this step is recommended to artificially apply some random transformations on the images to prevent overfitting.

Step 8:

Prepare the classification head with a global pooling, followed by a dense layer with a softmax activation:

```
layers = tf.keras.layers
global_avg_layer = layers.GlobalAveragePooling2D()
dense_layer = layers.Dense(num_classes,
                           activation='softmax')
```

Then, finalize the model architecture:

```
x = global_avg_layer(feet_extr.layers[-1].output)
x = layers.Dropout(0.2)(x)
outputs = dense_layer(x)
model = tf.keras.Model(inputs=feet_extr.inputs,
                       outputs=outputs)
```

Step 9:

Compile the model using a 0.0005 learning rate:

```
lr = 0.0005
opt_f = tf.keras.optimizers.Adam(learning_rate=lr)
```

```
loss_f = tf.losses.SparseCategoricalCrossentropy(from_logits=False)

model.compile(
    optimizer=opt_f,
    loss=loss_f,
    metrics=['accuracy'])
```

By default, TensorFlow employs a learning rate of 0.001. The reason for reducing the learning rate to 0.0005 is to prevent overfitting.

Step 10:

Train the model with 10 epochs:

```
model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=10)
```

Training should take a few seconds if you have around 20 images for each class. Using our dataset, we achieved an accuracy of approximately 85% on the validation dataset.

Step 11:

Save the TensorFlow model:

```
model.save("desk_objects_recognition")
```

The model is now ready to be quantized with the TensorFlow Lite converter.

There's more...

In this recipe, we learned how to leverage transfer learning to train an image classification model tailored for memory-constrained devices.

In transfer learning, **fine-tuning** is an optional step we might consider to help improve the model's accuracy.

This step is performed once the model has been trained and consists of unfreezing the weights of the pre-trained model (or part of it) and re-training it with a low learning rate. To fine-tune the model, you can follow the instructions detailed in the Keras documentation, available at the following link: https://keras.io/guides/transfer_learning/.

Now that we have the TensorFlow model, our next step is to quantize it before deploying it on the microcontroller.

In the upcoming recipe, we will quantize the TensorFlow to 8 bits using the TensorFlow Lite converter and assess its accuracy on unseen data.

Quantizing and testing the trained model with TensorFlow Lite

As we know from previous recipes, the model should be quantized to 8 bits to operate effectively on microcontrollers. Nonetheless, how do we know if the quantized model preserves the accuracy of the floating-point variant?

This question will be answered in this recipe, where we will show you how to evaluate the accuracy of the quantized model generated by the TensorFlow Lite converter. After this analysis, we will convert the TensorFlow Lite model to a C-byte array for deploying it on the Arduino Nano in the next recipe.

Getting ready

Quantization is an essential technique to reduce model size and significantly improve latency. However, adopting arithmetic with limited precision may alter a model's accuracy. As a result, evaluating the quantized model's accuracy is critical to ensure that the application performs as intended. Unfortunately, TensorFlow Lite does not provide a built-in function to assess the accuracy of the test dataset. Therefore, we must use the Python **TensorFlow Lite interpreter** to run the quantized model on the test samples to determine the number of correct classifications.

How to do it...

Let's start by collecting some test samples with the OV7670 camera module. The procedure for doing so is identical to the one outlined in the earlier *Building the dataset to classify desk objects* section. Simply capture a few images (for example, 10) for each output class, and upload them to a designated Google Drive folder for test samples. Remember to organize the image files into subfolders that match their respective classes, just like we did for the training dataset.

Next, take the following steps to evaluate the accuracy of the quantized model:

Step 1:

Build the test dataset:

```
test_dir = "drive/MyDrive/test"
```

```
test_ds = tf.keras.utils.image_dataset_from_directory(
    test_dir,
    interpolation="bilinear",
    image_size=(48, 48))
```

The `image_dataset_from_directory()` function, which we previously used to build the training and validation datasets, can also be used to obtain the test dataset. In this scenario, there is no necessity to specify the `validation_split` and `subset` input parameters.

Step 2:

Rescale the pixel values from `[0, 255]` to `[-1, 1]`:

```
test_ds = test_ds.map(lambda x, y: (rescale(x), y))
```

Step 3:

Convert the TensorFlow model to the TensorFlow Lite format. Apply the 8-bit quantization to the entire model except for the output layer, using the TensorFlow Lite converter:

```
repr_ds = test_ds.unbatch()

def representative_data_gen():
    for i_value, o_value in repr_ds.batch(1).take(60):
        yield [i_value]

TF_MODEL = "desk_objects_recognition"

converter = tf.lite.TFLiteConverter.from_saved_model(TF_MODEL)
converter.representative_dataset = tf.lite.
RepresentativeDataset(representative_data_gen)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_
INT8]
converter.inference_input_type = tf.int8

tfl_model = converter.convert()
```

The conversion is done as we did in *Chapter 3, Building a Weather Station with TensorFlow Lite for Microcontrollers*, except for the output data type. In this case, the output is kept in floating-point format to avoid the dequantization of the output result.

Step 4:

Initialize the TensorFlow Lite interpreter:

```
interp = tf.lite.Interpreter(model_content=tf1_model)
```

The Python-based TensorFlow Lite interpreter works similarly to the one we used in TensorFlow Lite for Microcontrollers. Therefore, it is employed to load the TensorFlow Lite models and execute them on the intended device, which, in this case, is the machine running the Colab environment.

Step 5:

Allocate the tensors and get the input quantization parameters:

```
interp.allocate_tensors()

i_details = interp.get_input_details()[0]
o_details = interp.get_output_details()[0]

i_quant = i_details["quantization_parameters"]
i_scale = i_quant['scales'][0]
i_zero_point = i_quant['zero_points'][0]
```

The quantization parameters will be used in the next step to quantize the input test samples.

Step 6:

Evaluate the accuracy of the TensorFlow Lite model. To do so, run the model inference on each test sample and verify the output result:

```
import numpy as np

test_ds0 = test_ds.unbatch()

num_correct_samples = 0
num_samples = len(list(test_ds0.batch(1)))

for i_value, o_value in test_ds0.batch(1):
    i_value = (i_value / i_scale) + i_zero_point
    i_value = tf.cast(i_value, dtype=tf.int8)
```

```

interp.set_tensor(i_details["index"], i_value)
interp.invoke()

o_pred = interp.get_tensor(o_details["index"])[0]
if np.argmax(o_pred) == o_value:
    num_correct_samples += 1

print("Accuracy:", num_correct_samples/num_samples)

```

The previous code uses a for loop to process each test sample. During each iteration, the input sample is quantized using the quantization parameters (`i_scale` and `i_zero_point`) obtained in the previous step. The quantized input is then copied into the model's input tensor using the `set_tensor()` method of the TensorFlow Lite interpreter.

The model inference is then invoked using the `invoke()` method, and the resulting output tensor is retrieved using the `get_tensor()` method.

If the predicted class matches the expected output value (`o_value`), the number of correctly classified test samples (`num_correct_samples`) is incremented.

Step 7:

Convert the TensorFlow Lite model to a C-byte array with the `xxd` tool:

```

open("model.tflite", "wb").write(tfl_model)
!apt-get update && apt-get -qq install xxd
!xxd -i model.tflite > model.h
!sed -i 's/unsigned char/const unsigned char/g' model.h
!sed -i 's/const/alignas(8) const/g' model.h

```

The command generates a C header file, containing the TensorFlow Lite model as a constant `unsigned char` array.



Remember the final two lines, where we employ the `sed` command to make the array containing the model as `alignas(8) const`. These changes guarantee that the model resides in the program memory (flash) and that the array is aligned to the 8-byte boundary.

You can now download the `model.h` file from Colab's left pane and proceed with the final recipe of this chapter, where we will deploy it on the Arduino Nano.

There's more...

In this recipe, we learned how to assess the accuracy of the quantized TensorFlow Lite model using the TensorFlow Lite Python interpreter.

The `model.h` C header file, which was obtained by converting the `.tflite` file to a C-byte array, contains the model architecture and the weights of the trainable layers. Since the array in the `model.h` file is defined as a constant, the model will be kept in the microcontroller program memory.

The memory usage of the model can be simply determined by assessing the size of the TensorFlow Lite model, which can be done as follows:

```
size_tfl_model = len(tfl_model)
print(len(tfl_model), "bytes")
```

In the preceding code, `tfl_model` is the TensorFlow Lite model obtained with the TensorFlow Lite converter.

The expected model size should be around 600 Kbytes, accounting for roughly 63% of the program memory available on the Arduino Nano.

The model is now ready, and our final step is its deployment on the microcontroller.

In the upcoming recipe, we will develop a sketch for the Arduino Nano to run the quantized trained model and learn how crucial the design of the pre-processing stage is for a memory-efficient deployment.

Fusing the pre-processing operators for efficient deployment

In this last recipe, we will develop a sketch to classify desk objects with the Arduino Nano. However, the ML deployment is not the only thing we must take care of. Indeed, a few additional operations must be implemented to supply the correct input image to the model.

Therefore, in this recipe, we will not just discuss model deployment but also delve into implementing a memory-efficient pre-processing pipeline, preparing the input for the model.

Getting ready

RAM usage is impacted by the variables allocated during the program execution, such as the input, output, and intermediate tensors of the ML model. However, the model is not solely responsible for memory utilization. In fact, the image acquired from the OV7670 camera needs to be pre-processed with the following operations to provide the appropriate input to the model:

- Image cropping to match the input shape aspect ratio of the model
- Image resizing to match the expected input shape of the model
- Color format conversion from YCbCr422 to RGB888
- Pixel scaling to bring the values from $[0, 255]$ to $[-1, 1]$
- Pixel quantization to convert the floating-point values to an 8-bit integer

Each of the preceding operations reads values from a buffer and produces the resulting computation in a new one, as shown in the following figure:

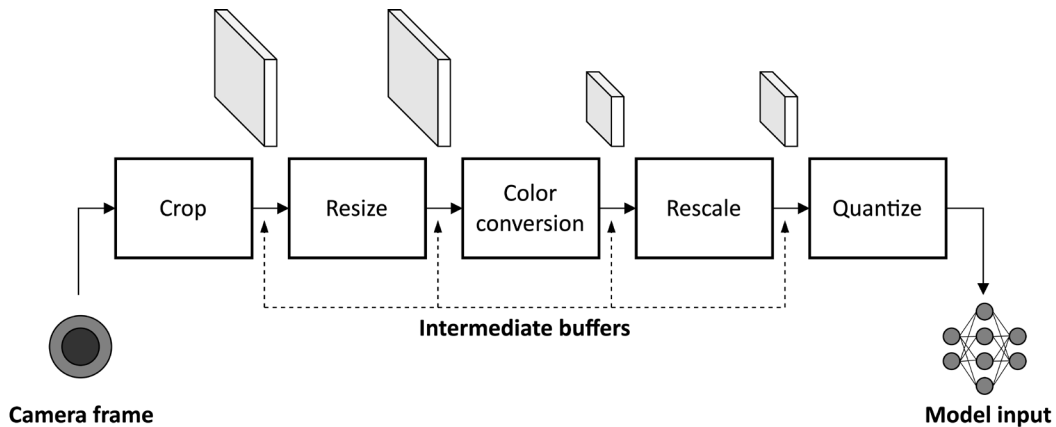


Figure 8.20: Pre-processing pipeline

An example of pseudo-C code to process the pipeline previously illustrated is as follows:

```
// Func(source, destination)

crop(camera_frame, buffer0);
resize(buffer0, buffer1);
color_convert(buffer1, buffer2);
rescale(buffer2, buffer3);
quantize(buffer3, model_input);
```

In the preceding code, each operator is applied sequentially to the entire output of the previous operator, resulting in multiple and large intermediate results being stored in memory. As you can guess, RAM usage is highly affected by the intermediate buffers passed from one operation to the next.

Operator fusion is the technique this recipe exploits to reduce RAM usage drastically. The idea is to combine all these five functions when processing each model input value. As a result, *these functions are called multiple times to process a small portion of the model input at a time*.

In order to combine these operators, it is necessary to know their implementation. The subsequent section will explain one such operator that has not yet been discussed: the image resizing operator with bilinear interpolation.

Image resizing with bilinear interpolation

Image resizing is an operator used to change an image's resolution (width and height), as shown in the following figure:

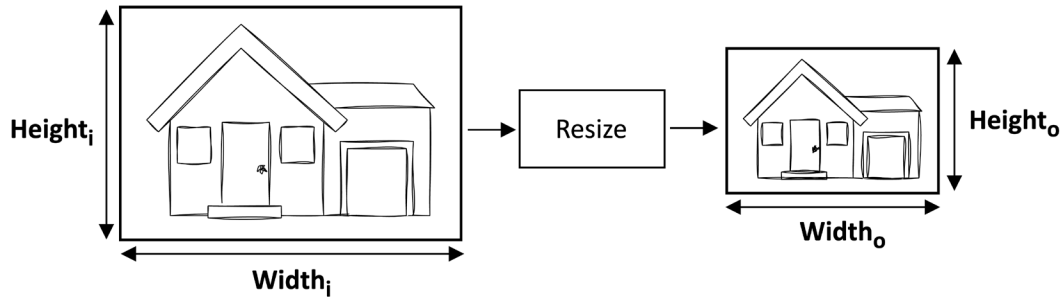


Figure 8.21: Image resizing

The resulting image is obtained from the pixels of the input image. Generally, the following formulas are applied to map the spatial coordinates of the output pixels with the corresponding input ones:

$$\begin{aligned} x_i &= x_o \cdot scale_x & scale_x &= \frac{width_i}{width_o} \\ y_i &= y_o \cdot scale_y & scale_y &= \frac{height_i}{height_o} \end{aligned}$$

From the previous two formulas:

- (x_i, y_i) are the spatial coordinates of the input pixel.
- (x_o, y_o) are the spatial coordinates of the output pixel.

- $(width_i, width_i)$ are the dimensions of the input image.
- $(width_o, height_o)$ are the dimensions of the output image.

As we know, a digital image consists of a grid of pixels located at integer coordinates. However, when applying the preceding two formulas, we may not always get an integer spatial coordinate, which means that the actual input sample doesn't always exist. This is one of the factors contributing to the deterioration of image quality whenever we alter an image's resolution. Some interpolation techniques exist to mitigate this problem, such as **nearest-neighbor**, **bilinear**, or **bicubic interpolation**. Bilinear interpolation is the technique adopted in this recipe.

As shown in *Figure 8.22*, this method takes the four pixels closest to the input sampling point in a 2x2 grid:

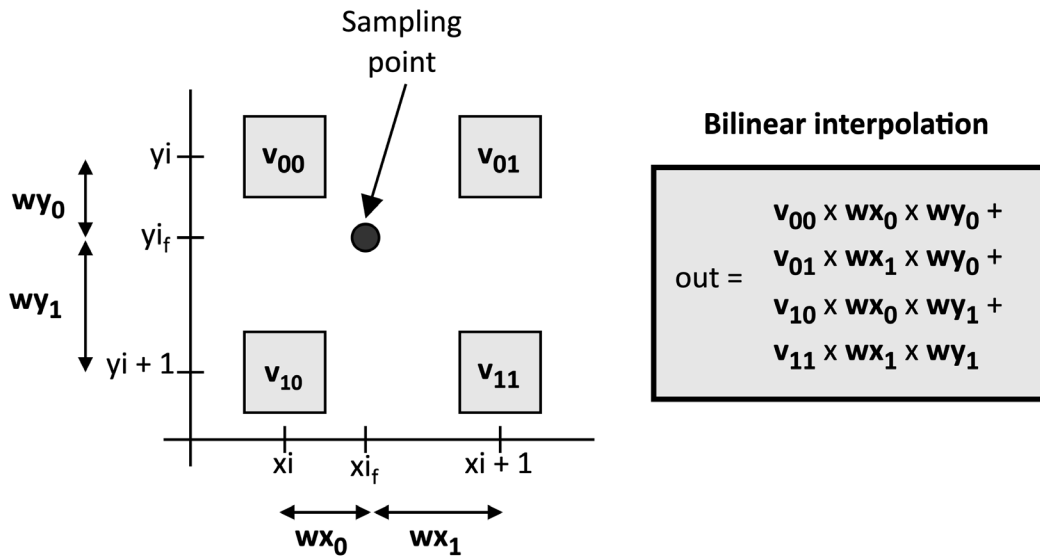


Figure 8.22: Bilinear interpolation

The bilinear interpolation function determines the value of the output pixel, with a weighted average of four pixels neighboring the sampling point, as described by the formula in *Figure 8.22*. The weights assigned to each pixel are based on their proximity to the sampling point, with closer pixels having a higher influence on the result.

In our case, we have shown an example of bilinear interpolation applied to a single-color component image. However, this method works regardless of the number of color components, since we can interpolate the values independently.

How to do it...

Disconnect the USB data cable from the Arduino Nano and remove the push-button from the breadboard. Next, open the Arduino IDE and create a new sketch.

After that, copy the contents of the previous sketch developed to convert QQVGA images from YCbCr422 to RGB888 in a new sketch, and remove the code within the `loop()` function and all the references to push-button usage.

Now, take the following steps to develop a sketch to classify desk objects with the Arduino Nano and the OV7670 camera module:

Step 1:

Download the Arduino TensorFlow Lite library from the **TinyML-Cookbook_2E** GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/ArduinoLibs/Arduino_TensorFlowLite.zip.

After downloading the ZIP file, import it into the Arduino IDE.

Step 2:

Import the header file containing the TensorFlow Lite model (`model.h`) into the Arduino project. As shown in the following screenshot, click on the tab button with the upside-down triangle and click on **Import File into Sketch**:

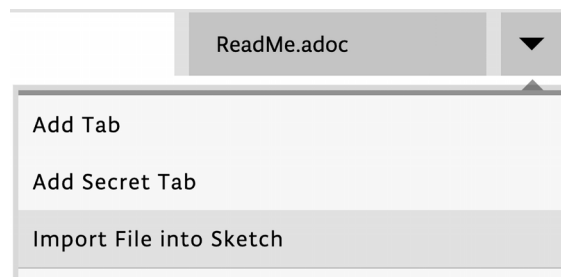


Figure 8.23: Importing the `model.h` file into the Arduino project

Once the file has been imported, include the `model.h` header file in the sketch:

```
#include "model.h"
```

Step 3:

Include the necessary header files for tflite-micro:

```
#include <TensorFlowLite.h>
#include <tensorflow/lite/micro/all_ops_resolver.h>
#include <tensorflow/lite/micro/micro_interpreter.h>
#include <tensorflow/lite/micro/micro_log.h>
#include <tensorflow/lite/micro/system_setup.h>
#include <tensorflow/lite/schema/schema_generated.h>
```

The header files are the same ones described in *Chapter 3, Building a Weather Station with TensorFlow Lite for Microcontrollers*.

Step 4:

Declare global variables for tflite-micro model and interpreter:

```
const tflite::Model* tflu_model          = nullptr;
tflite::MicroInterpreter* tflu_interpreter = nullptr;
```

Then, declare the TensorFlow Lite tensor objects (TfLiteTensor) to access the input and output tensors of the model:

```
TfLiteTensor* tflu_i_tensor          = nullptr;
TfLiteTensor* tflu_o_tensor          = nullptr;
```

The global variables declared in this step are those described in *Chapter 3, Building a Weather Station with TensorFlow Lite for Microcontrollers*.

Step 5:

Declare a global constant integer value to keep the desired size for the tensor arena. Then, declare a global `uint8_t*` value to point to the tensor arena-allocated memory space:

```
constexpr int tensor_arena_size = 128000;
uint8_t *tensor_arena = nullptr;
```

The tensor arena is the memory reserved for the tflite-micro runtime, for the intermediate tensors used during model inference. The tensor arena size must be large enough to accommodate all the intermediate tensors required for model inference but not too large to avoid wasting valuable memory resources.

In the case of the model designed with MobileNet v2, a tensor arena size of 128000 bytes should be adequate.

It is worth noting that the two lines mentioned earlier do not allocate any memory for the tensor arena. The tensor arena's memory space will be reserved later in the `setup()` function.

Step 6:

Declare global variables for the quantization parameters of the input tensor:

```
float tflu_scale = 0.0f;
int32_t tflu_zeropoint = 0;
```

Since the model's output is in a floating-point format, we will not need to retrieve the quantization parameters for the output tensor.

Step 7:

Declare global variables for the resolutions of the cropped camera frame and input model:

```
int32_t height_i = 120; int32_t width_i = height_i;
int32_t height_o = 48; int32_t width_o = 48;
```

Then, declare global variables for the scaling factors to resize the camera frame:

```
float scale_x = (float)width_i / (float)width_o;
float scale_y = scale_x;
```

Step 8:

Write the function to perform the bilinear interpolation for a single-color component pixel. The function should accept the four pixels neighboring the sampling point and the sampling point itself as input arguments:

```
uint8_t bilinear(uint8_t v00, uint8_t v01,
                 uint8_t v10, uint8_t v11,
                 float xi_f, float yi_f) {
    const float xi = (int32_t)std::floor(xi_f);
    const float yi = (int32_t)std::floor(yi_f);

    const float wx1 = (xi_f - xi);
    const float wx0 = (1.f - wx1);
```

```

const float wy1 = (yi_f - yi);
const float wy0 = (1.f - wy1);

float res = 0;
res += (v00 * wx0 * wy0);
res += (v01 * wx1 * wy0);
res += (v10 * wx0 * wy1);
res += (v11 * wx1 * wy1);

return clamp_0_255(res);
}

```

The implementation of the bilinear interpolation consists of the following four steps:

1. Calculate the coordinates of the top-left pixel (x_i and y_i) in the 2×2 grid by rounding down the sampling coordinates to the nearest integer values.
2. Calculate the weight values $wx0$, $wx1$, $wy0$, and $wy1$ for the four neighboring pixel values, based on their distances to the sampling point. The distance is calculated between the floating-point pixel location (x_{i_f} and y_{i_f}) and the corresponding rounded-down nearest integer pixel locations (x_i and y_i).
3. Compute the weighted sum of the four neighboring pixel values ($v00$, $v01$, $v10$, and $v11$).
4. Clamp the resulting value of the bilinear interpolation between 0 and 255 to return an 8-bit pixel value.

Step 9:

Write the function to rescale the pixel values from $[0, 255]$ to $[-1, 1]$:

```

float rescale(float x, float scale, float offset) {
    return (x * scale) - offset;
}

```

Then, write the function to quantize the input image:

```

int8_t quantize(float x,
                float scale,
                float zero_point) {
    return (x / scale) + zero_point;
}

```



Since rescaling and quantizing are executed one after the other, you can fuse them into a single function to make the implementation more efficient, in terms of the arithmetic instructions executed.

Step 10:

In the `setup()` function, dynamically allocate the memory for the tensor arena:

```
tensor_arena = new uint8_t[tensor_arena_size];
```

We allocate the tensor arena with the `new` operator to place the memory in the **heap**. As we know, the heap is the area of RAM related to the dynamic memory and can only be released explicitly by the user with the `delete[]` operator. The heap is opposed to the **stack** memory, where the data lifetime is limited to the scope.



The stack and heap memory sizes are defined in the startup code, executed by the microcontroller when the system resets.

Since the stack is typically much smaller than the heap, allocating the tflite-micro working space in the heap is preferable because the tensor arena takes a significant portion of RAM (144 KB).

Step 11:

In the `setup()` function, load the TensorFlow Lite model stored in the `model.h` header file:

```
tflu_model = tflite::GetModel(model_tflite);
```

Then, register all the DNN operations supported by tflite-micro, and initialize the tflite-micro interpreter:

```
tflite::AllOpsResolver tflu_ops_resolver;

static tflite::MicroInterpreter static_interpreter(
    tflu_model,
    tflu_ops_resolver,
```



```

        tensor_arena,
        tensor_arena_size);
    flru_interpreter = &static_interpreter;

```

Step 11:

In the `setup()` function, allocate the memory required for the model, and get the memory pointer of the input and output tensors:

```

    tflu_interpreter->AllocateTensors();
    tflu_i_tensor = tflu_interpreter->input(0);
    tflu_o_tensor = tflu_interpreter->output(0);

```

Step 12:

In the `setup()` function, get the quantization parameters for the input tensor:

```

    const auto* i_quant = reinterpret_
    cast<TfLiteAffineQuantization*>(tflu_i_tensor->quantization.params);

    tflu_scale = i_quant->scale->data[0];
    tflu_zero_point = i_quant->zero_point->data[0];

```

Step 13:

In the `loop()` function, read a camera frame and get the number of bytes per pixel:

```

void loop() {
    Camera.readFrame(data);
    int32_t bytes_per_pixel = Camera.bytesPerPixel();

```

Then, write a for loop to iterate over the spatial coordinates of the MobileNet v2 input shape (`width_o` and `height_o`). Then, calculate the corresponding input sampling point position for each coordinate (`xi_f` and `yi_f`). Finally, round down the sampling point to the nearest integer value (`xi` and `yi`):

```

    int32_t idx = 0;
    for (int32_t yo = 0; yo < height_o; yo++) {
        float yi_f = (yo * scale_y);
        int32_t yi = (int32_t)std::floor(yi_f);

        for(int32_t xo = 0; xo < width_o; xo++) {

```

```
float xi_f = (xo * scale_x);
int32_t xi = (int32_t)std::floor(xi_f);
```

As you can observe from the previous code, we iterate over the spatial coordinates of the MobileNet v2 input shape (48x48). For each *xo* and *yo*, we calculate the sampling coordinates (*xi_f* and *yi_f*) and the corresponding rounded-down nearest integer coordinates (*xi* and *yi*) to get the four neighboring pixels required for the resize operation.



In the previous code, the *idx* variable is used to keep track of the number of elements processed by the pre-processing pipeline. This variable will be used later when filling in the TensorFlow Lite input model.

Once you have the input coordinates, calculate the camera buffer offsets to read the Y component from the four YCbCr422 pixels needed for the bilinear interpolation:

```
int32_t x0 = xi;
int32_t y0 = yi;
int32_t x1 = std::min(xi + 1, width_i - 1);
int32_t y1 = std::min(yi + 1, height_i - 1);
int32_t width_c = Camera.width();

// stride_x/stride_y camera frame
int32_t stride_x = bytes_per_pixel;
int32_t stride_y = width_c * bytes_per_pixel;

int32_t off_Y00 = x0 * stride_x + y0 * stride_y;
int32_t off_Y01 = x1 * stride_x + y0 * stride_y;
int32_t off_Y10 = x0 * stride_x + y1 * stride_y;
int32_t off_Y11 = x1 * stride_x + y1 * stride_y;
```

In the previous code, *x0* and *y0* variables are used for the coordinates of the top-left pixel in a 2x2 grid, while *x1* and *y1* are for the coordinates of the bottom-right pixel.

To access the Y component from the four pixels in the camera frame, we calculate the offsets in a number of bytes (*off_Y00*, *off_Y01*, *off_Y10*, and *off_Y11*). The offset calculation needs two pieces of information:

- *Number of bytes per pixel*: This is the *stride_x*, which is the same as the *bytes_per_pixel* variable.

- *Number of bytes per row*: This is the `stride_y` variable, obtained by multiplying the number of pixels in a row by the number of bytes needed to represent each pixel.

Then, the offset is calculated as follows:

$$off_Y(y, x) = coord_x \cdot stride_x + coord_y \cdot stride_y$$

Step 14:

Read the **Y** component from the four pixels needed for the bilinear interpolation:

```
int32_t Y00 = data[off_Y00];
int32_t Y01 = data[off_Y01];
int32_t Y10 = data[off_Y10];
int32_t Y11 = data[off_Y11];
```

Step 15:

Read the red-difference components (**Cr**) from the four pixels needed for the bilinear interpolation:

```
int32_t adj_cr00 = xi % 2 == 0? 1 : -1;
int32_t adj_cr01 = (xi + 1) % 2 == 0? 1 : -1;

int32_t Cr00 = data[off_Y00 + adj_cr00];
int32_t Cr01 = data[off_Y01 + adj_cr01];
int32_t Cr10 = data[off_Y10 + adj_cr00];
int32_t Cr11 = data[off_Y11 + adj_cr01];
```

To read the **Cr** components, we can use the offsets to access the **Y** components with an adjustment (`adj_cr00` and `adj_cr01`), as shown in *Figure 8.24*:

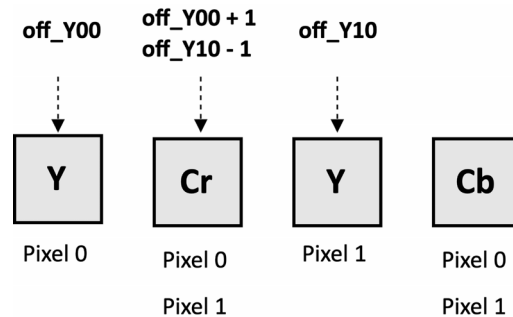


Figure 8.24: Offset adjustment to read the Cr component

As you can observe from the preceding illustration, when the pixel's integer X-coordinate (*xi*) is even, we should increment *off_Y00/off_Y10* by 1 and decrement *off_Y01/off_Y11* by 1. Conversely, when the pixel's integer X-coordinate is odd, we should decrement *off_Y00/off_Y10* by 1 and increment *off_Y01/off_Y11* by 1.

Step 16:

Read the blue-difference components (**Cb**) from the four pixels needed for the bilinear interpolation:

```
int32_t adj_cb00 = xi % 2 == 0? 3 : 1;
int32_t adj_cb01 = (xi + 1) % 2 == 0? 3 : 1;

int32_t Cb00 = data[off_Y00 + adj_cb00];
int32_t Cb01 = data[off_Y01 + adj_cb01];
int32_t Cb10 = data[off_Y10 + adj_cb00];
int32_t Cb11 = data[off_Y11 + adj_cb01];
```

To read the **Cb** components, we can proceed similarly to how we did with the **Cr** components by adjusting the offsets *off_Y00*, *off_Y01*, *off_Y10*, and *off_Y11*.

In this case, when the pixel's integer X-coordinate (*xi*) is even, we should increase *off_Y00/off_Y10* by 3 and *off_Y01/off_Y11* by 1. On the other hand, when the pixel's integer X-coordinate is odd, we should increase *off_Y00/off_Y10* by 1 and *off_Y01/off_Y11* by 3.

Step 17:

Convert the YCbCr422 pixels to RGB888:

```
uint8_t rgb00[3], rgb01[3], rgb10[3], rgb11[3];
ycbcr422_rgb888(Y00, Cb00, Cr00, rgb00);
ycbcr422_rgb888(Y01, Cb01, Cr01, rgb01);
ycbcr422_rgb888(Y10, Cb10, Cr10, rgb10);
ycbcr422_rgb888(Y11, Cb11, Cr11, rgb11);
```

In the previous code snippet, we used the function implemented in the *Converting YCbCr422 to RGB888* recipe to convert the image from YCbCr422 to RGB888.

Step 18:

Iterate over the channels of the RGB pixels and sequentially apply the bilinear interpolation, pixel rescaling, and pixel quantization. Then, store the resulting values in the input tensor of the TensorFlow Lite model:

```
uint8_t c_i; float c_f; int8_t c_q;
for(int32_t i = 0; i < 3; i++) {
    c_i = bilinear(rgb00[i], rgb01[i],
                  rgb10[i], rgb11[i],
                  xi_f, yi_f);
    c_f = rescale((float)c_i, 1.f/255.f, -1.f);
    c_q = quantize(c_f, tflu_scale, tflu_zeropoint);
    tflu_i_tensor->data.int8[idx++] = c_q;
}
```

Step 19:

Run the model inference and return the classification result over the serial:

```
    } // xo for loop
} // yo for loop

tflu_interpreter->Invoke();
size_t ix_max = 0;
float pb_max = 0;
for (size_t ix = 0; ix < 3; ix++) {
    if(tflu_o_tensor->data.f[ix] > pb_max) {
        ix_max = ix;
        pb_max = tflu_o_tensor->data.f[ix];
    }
}
const char *label[] = {"book",
                       "mug",
                       "unknown"};
Serial.println(label[ix_max]);
```

Compile and upload the sketch on the Arduino Nano. Then, move your camera and focus on a book or a mug. The application should now identify your objects and display the classification result in the serial monitor!

There's more...

In this recipe, we learned how to implement a memory-efficient image-processing pipeline using the fusion technique. Without this technique, developing this application would have been impossible because of the high memory usage required to perform the image-processing operations on the entire image.

At this point, after successfully developing this application, you may want to extend its capabilities to recognize additional desk objects. This can be easily achieved by performing three steps:

1. Collecting samples for the new classes
2. Retraining the model
3. Updating the label array in the previously deployed tflite-micro application

That's it! No further steps are required!

Summary

The recipes presented in this chapter demonstrated how to build an end-to-end image classification application with TensorFlow and an Arduino-compatible platform.

In the first part, we learned how to connect the OV7670 camera module to the Arduino Nano and acquire images, with a resolution and color format suitable for memory-constrained devices.

Then, we developed a Python script to create images from the pixels transmitted over the serial by the Arduino Nano. This script was then extended to upload the file images to Google Drive, laying the foundation to build the training dataset.

After the dataset preparation, we delved into the model design, where we leveraged transfer learning with TensorFlow to train a model to classify desk objects.

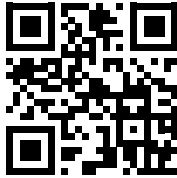
Ultimately, we quantized the trained model to 8-bit using the TensorFlow Lite converter and deployed it to the Arduino Nano. However, the development of the Arduino sketch went beyond mere model deployment. Crucially, we had to implement a memory-efficient image-processing pipeline using the fusion technique to make this application feasible on the Arduino Nano.

After this chapter, we can close the part related to tinyML with vision sensors on microcontrollers. So, what's next? In the upcoming chapter, we will delve into motion sensors and start discussing the principles to design ML models tailored for devices with limited memory.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



9

Building a Gesture-Based Interface for YouTube Playback with Edge Impulse and the Raspberry Pi Pico

Gesture recognition is the technology that enables people to interact with their devices without touching buttons or displays physically. By interpreting human gestures, this technology has found its space in various consumer electronics, including smartphones and game consoles. At its core, gesture recognition relies on two essential components: a sensor and a software algorithm.

In this chapter, we will show you how to use **accelerometer** measurements with **machine learning (ML)** to recognize three hand gestures with Raspberry Pi Pico. These recognized gestures will then be used to play/pause, mute/unmute, and change YouTube videos on our PC.

The development of this project will start by acquiring the accelerometer data to build the gesture recognition dataset. In this part, we will learn how to interface with the **I2C protocol** and use the **Edge Impulse data forwarder** tool. Next, we will focus on the Impulse design, where we will build a **spectral-features-based feedforward** neural network for gesture recognition. Finally, we will deploy the model on Raspberry Pi Pico and implement a Python script with the **PyAuto-GUI** library to build a touchless interface for YouTube video playback.

This chapter aims to help you develop an end-to-end gesture recognition application with Edge Impulse and Raspberry Pi Pico so that you can learn how to use the I2C peripheral, get acquainted with inertial sensors, write a multithreading program in **Arm Mbed OS**, and discover how to filter out redundant classification results during model inference.

In this chapter, we're going to cover the following recipes:

- Communicating with the MPU-6050 IMU through I2C
- Acquiring accelerometer data
- Building the dataset with the Edge Impulse data forwarder tool
- Designing and training the ML model
- Live classifications with the Edge Impulse data forwarder tool
- Developing a continuous gesture recognition application with Edge Impulse and Arm Mbed OS
- Building a gesture-based interface with PyAutoGUI

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- A Raspberry Pi Pico
- A SparkFun RedBoard Artemis Nano (optional)
- A micro-USB data cable
- A USB-C data cable (optional)
- 1 x half-size solderless breadboard
- 1 x MPU-6050 IMU
- 4 x jumper wires
- Laptop/PC with either Linux, macOS, or Windows
- Edge Impulse account



The source code and additional material are available in the Chapter09 folder in the GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Chapter09.

Communicating with the MPU-6050 IMU through I2C

Acquiring sensor data can be a challenging task in tinyML due to the need for direct interaction with hardware at a low level.

In this recipe, we will use the **MPU-6050 Inertial Measurement Unit (IMU)** to demonstrate the basic principles of a common communication protocol used with sensors known as the **Inter-Integrated Circuit (I2C)**. By the end of this recipe, we will have developed an Arduino sketch that can read out the MPU-6050 address.

Getting ready

The IMU sensor is an electronic device capable of measuring accelerations, angular rates, and, in some cases, body orientations through a combination of integrated sensors. This device is at the heart of many technologies in various industries, including automotive, aerospace, and consumer electronics, to estimate position and orientation. For example, IMU allows the screen of a smartphone to auto-rotate and enables **augmented reality/virtual reality (AR/VR)** use cases.

In this *Getting ready* section, we delve into the IMU sensor features and learn how to communicate with this device using the I2C communication protocol.

Introducing the features of the MPU-6050 IMU

MPU-6050 (<https://invensense.tdk.com/products/motion-tracking/6-axis/mpu-6050/>) is an IMU that combines a **three-axis accelerometer** and **three-axis gyroscope** sensors to measure accelerations and the angular rate of the body. This device has been on the market for several years, and due to its low cost and high performance, it is still a popular choice for DIY electronic projects based on motion sensors.

The MPU-6050 IMU can be found via various distributors, such as *Adafruit*, *Amazon*, *Pimoroni*, and *PiHut*, and it is available in different form factors. In this recipe, we opted for the compact breakout board that's offered by Adafruit (<https://learn.adafruit.com/mpu6050-6-dof-accelerometer-and-gyro/overview>), which can be powered with a 3.3V voltage and does not require connecting additional electronic components:

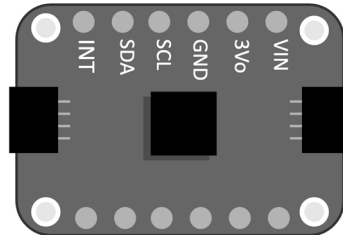


Figure 9.1: The MPU-6050 IMU module



Unfortunately, generally, this IMU module is supplied with unsoldered header strips. As a result, if you are unfamiliar with soldering, you might want to explore an alternative version of the module with pre-soldered pins. Alternatively, you can refer to the following tutorial for guidance on soldering the header strips yourself: <https://learn.adafruit.com/adafruit-agc-electret-microphone-amplifier-max9814/assembly>.

The **SDA** and **SCL** pins of the MPU-6050 IMU are used to communicate with the microcontroller through the I2C serial communication protocol. The following subsection describes some of the main features worth mentioning of I2C.

Basics of the I2C communication protocol

I2C is a communication protocol that relies on two wires: **SCL (clock signal)** and **SDA (data signal)**.

The protocol has been structured to allow communication between a **primary device** (for example, the microcontroller) and numerous **secondary devices** (for example, the sensors). Each secondary device is identified with a permanent 7-bit address.



The I2C protocol traditionally uses the terms master and slave to refer to devices involved in communication. In this book, we have adopted the alternative terms of primary and secondary to ensure a more inclusive language and remove unnecessary references to slavery.

The following diagram shows how the primary and secondary devices are connected:

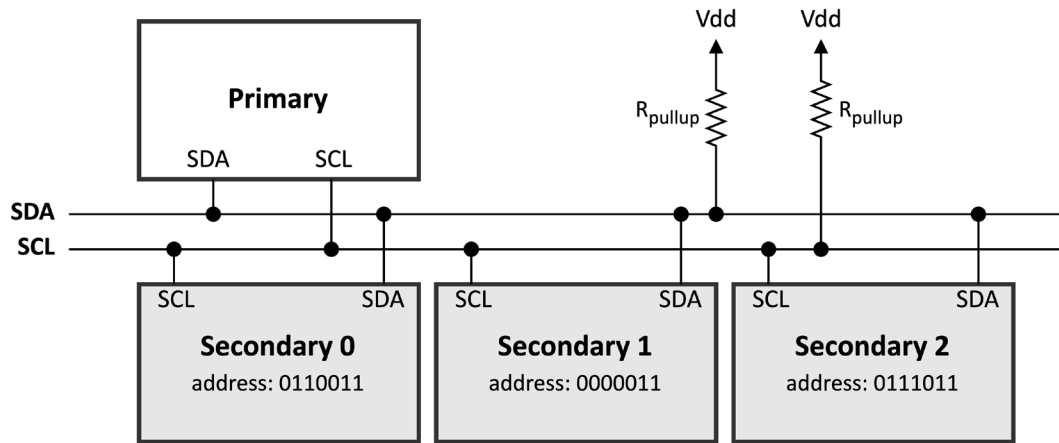


Figure 9.2: I2C communication

As we can see from the previous diagram, there are only two signals (**SCL** and **SDA**), regardless of the number of secondary devices. **SCL** is the clock signal produced by the primary device and is used by all I2C devices to sample the bits transmitted over the data signal. Both the primary and secondary devices can transmit data over the **SDA** bus.



The MPU-6050 supports a maximum SCL frequency of 400 kHz.

The pull-up resistors (**R_{pullup}**) are required because the I2C device can only drive the signal to the *LOW* level (logic level 0). In our case, the pull-up resistors are unnecessary because they are integrated into the MPU-6050 module.

From a communication protocol standpoint, *the primary device always starts the communication by transmitting the following bits:*

1. *Start bit:* SDA *HIGH* to *LOW* transition during SCL *HIGH*.
2. *7-bit address:* This is the 7-bit address of the target secondary device.
3. *Stop bit:* This is 1 bit that indicates the primary device's desire to either read data from or write data to the secondary device. When the stop bit is set to a logical *LOW*, it signifies the primary device's intention to write data to the secondary device (**write mode**).

Conversely, when the stop bit is set to a logical *HIGH*, it indicates the primary device's intention to read data from the secondary device (**read mode**).

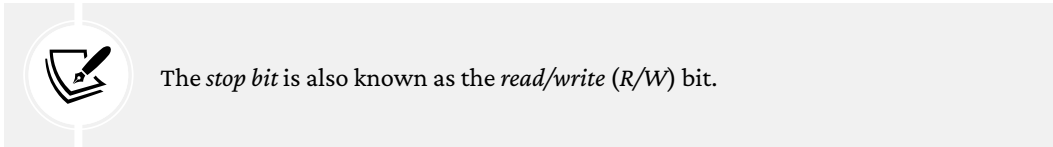


Figure 9.3 shows an example of a bit command sequence in the scenario where the primary device in Figure 9.2 wants to read data from the secondary device 0:

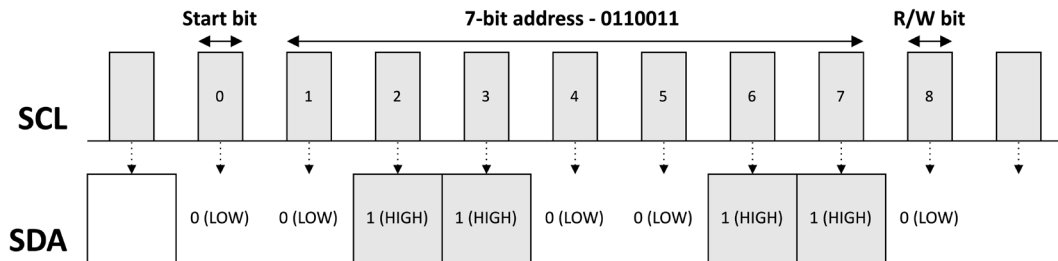
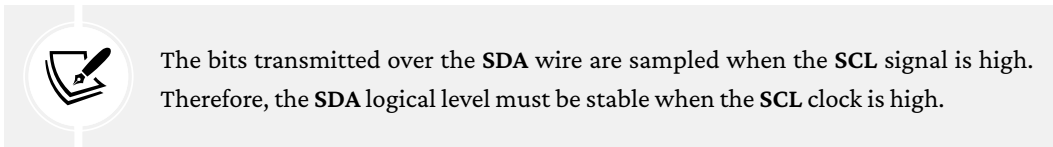


Figure 9.3: Bit command sequence transmitted by the primary device



The secondary device that matches the 7-bit address will respond with 1 bit at logical level *LOW* (**ACK**) over the SDA bus immediately after the stop bit.

If the secondary device responds with the ACK, the primary device can either transmit or read data in chunks of 8 bits accordingly with the R/W flag set.

In our context, *the microcontroller is the primary device*, and it can use the R/W flag to do the following operations:

- *Reading data from the sensor:* The microcontroller must request what it wants to read (**write mode**) before the MPU-6050 IMU can transmit the data (**read mode**).
- *Programming an internal feature of the IMU:* The microcontroller can use **write mode** to set an operating mode of MPU-6050 (for example, the sampling frequency of the sensors).

At this stage, you may have a question: *what do we read and write with the primary device?*

The primary device reads and writes registers on the secondary device. Therefore, the secondary device works like memory, where each register has a unique 8-bit address.



The register map for MPU-6050 is available at the following link: <https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf>.

Luckily, we will not need to deal with all these low-level details of the I2C protocol directly because we will employ the Mbed OS API, which provides a high level of abstraction to programmers.

Programming the I2C peripheral with Mbed OS

Before any data transmission over the I2C bus can start, the `mbed : : I2C` object must be initialized with the microcontroller's pins assigned to the SDA and SCL signals. Once the object is initialized, we can use its `read()` and `write()` methods for the respective read and write modes. The `read()` and `write()` methods share the same interface and accept the following arguments:

- The 8-bit variable storing the address of the secondary device. Therefore, the 7-bit address must be left-shifted by 1 bit to obtain the 8-bit representation.
- The `uint8_t` buffer containing the data to transmit.
- The number of elements contained in the data buffer.

Alongside the `read()` and `write()` methods, the `mbed : : I2C` class offers additional methods for configuring specific features of the I2C peripheral, such as the SCL clock frequency.

Having acquired knowledge of programming the I2C peripheral, let's now see what pins are dedicated to this peripheral on the Raspberry Pi Pico.

Accessing the I2C peripheral in Raspberry Pi Pico

The Raspberry Pi Pico provides two I2C peripherals (**I2C0** and **I2C1**), which can be accessed through the following pins:

I2C peripheral	SDA	SCL
I2C0	GP0 GP4 GP8 GP12 GP16 GP20	GP1 GP5 GP9 GP13 GP17 GP21
I2C1	GP2 GP6 GP10 GP14 GP18 GP26	GP3 GP7 GP11 GP15 GP19 GP27

Figure 9.4: Table reporting the pins to access I2C0 and I2C1 peripherals

The I2C peripheral can act as a primary and secondary device, with the primary device being the default one. The SCL clock frequency can work in three data transfer modes:

- **Standard mode:** SCL clock frequency up to 100 KHz
- **Fast mode:** SCL clock frequency up to 400 KHz
- **Fast Plus mode:** SCL clock frequency up to 1,000 KHz

Since the maximum SCL clock frequency for the MPU-6050 is 400 KHz, the I2C peripheral will need to work on either standard or fast mode.

How to do it...

Start this recipe by taking the breadboard and mounting Raspberry Pi Pico vertically among the left and right terminal strips.

Next, place the accelerometer sensor module at the bottom of the breadboard. Ensure that the breadboard's notch is in the middle of the two headers, as shown in the following diagram:

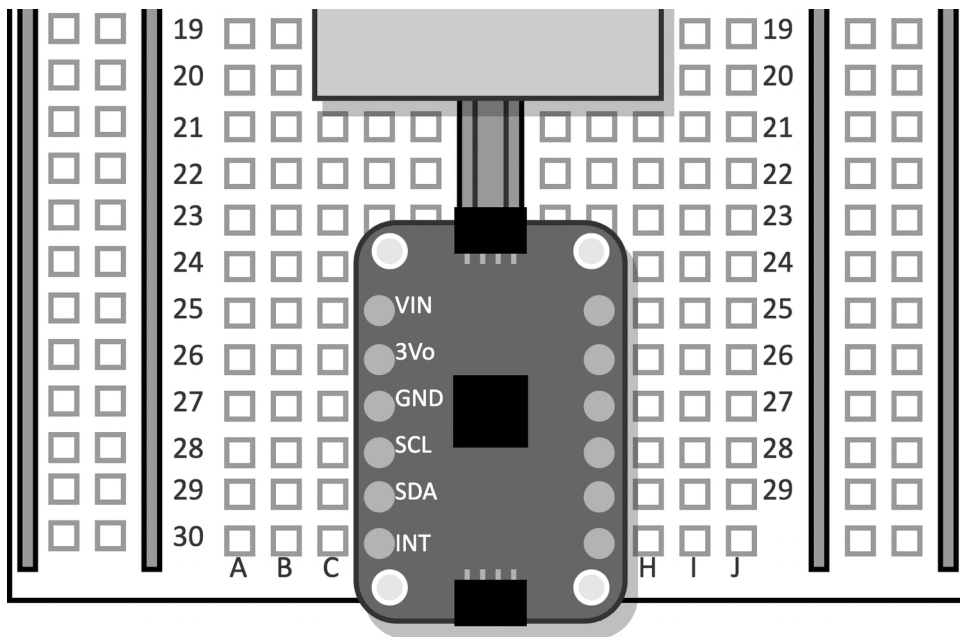


Figure 9.5: Place the MPU-6050 at the bottom of the breadboard

As you can see from the previous image, the I2C pins are located on the left terminal strips of the MPU-6050 module. The following steps will show you how to connect these pins with Raspberry Pi Pico and write a basic sketch to read the ID (address) of the MPU-6050 device.

Step 1:

Take four jumper wires and connect the MPU-6050 IMU to the I2C1 peripheral pins of Raspberry Pi Pico, as reported in the following table:

MPU-6050	VIN	GND	SCL	SDA
Raspberry Pi Pico	3V3	GND	GP7 (SCL1)	GP6 (SDA1)

Figure 9.6: Connections between the MPU-6050 IMU and Raspberry Pi Pico

Figure 9.7 will help you visualize how to do the wiring:

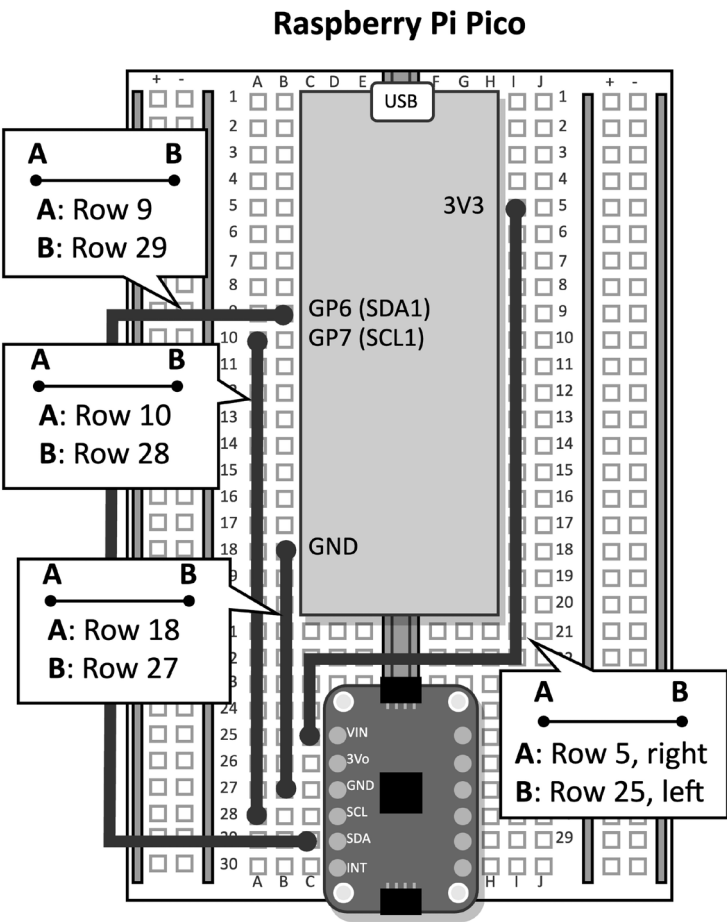


Figure 9.7: MPU-6050 is mounted at the bottom of the breadboard

In the previous circuit, we did not need to include pull-up resistors on the **SDA** and **SCL** wires because they had already been integrated into the IMU's breakout board.



The MPU-6050 comes in different forms and different pin names. Therefore, we recommend referring to the pin configuration of your module before connecting it with the microcontroller.

Step 2:

Open Arduino IDE and create a new sketch. In the file, include the `mbed.h` header file to access the Mbed OS functionalities:

```
#include "mbed.h"
```

Then, declare and initialize the `mbed::I2C` object to communicate with the MPU-6050 IMU:

```
#define I2C_SDA_PIN p6
#define I2C_SCL_PIN p7
mbed::I2C i2c(I2C_SDA_PIN, I2C_SCL_PIN);
```

The initialization of the I2C peripheral only requires the microcontroller's pins dedicated to the SDA (p6) and SCL (p7) buses.

Step 3:

Define a constant variable with the 7-bit address of the MPU-6050 IMU:

```
#define MPU6050_ADDR_7BIT 0x68
```

The `0x68` value is the 7-bit I2C address of the MPU-6050 IMU, as specified in its datasheet.

Then, left-shift the 7-bit address by 1 bit to obtain the 8-bit address needed by the `mbed::I2C` object.

```
#define MPU6050_ADDR_8BIT 0xD1
```

The `0xD1` value corresponds to the result obtained with the $(0x68 \ll 1)$ operation.

Step 4

Implement a utility function to read data from an MPU-6050 register:

```
void read_reg(int32_t addr_i2c,
              int32_t addr_reg,
```

```
        char *buf, int32_t length) {  
    char data = addr_reg;  
    i2c.write(addr_i2c, &data, 1);  
    i2c.read(addr_i2c, buf, length);  
    return;  
}
```

According to the I2C protocol, we must transmit the address of the MPU-6050 IMU and then send the register address to read. For this scope, we must use the `write()` method of the `mbd::I2C` class, which needs three input arguments as follows:

- The 8-bit address of the secondary device (`addr_i2c`)
- A char array containing the registered address (`char data = addr_reg`)
- The size of the data array (1 since we're only sending the registered address)

After sending the request to read the data from the register, we can use the `read()` method of the `mbd::I2C` class to retrieve the transmitted data. The `read()` method requires the following input arguments:

- The 8-bit address of the secondary device (`addr_i2c`)
- A char array to store the received data (`buf`)
- The size of the `buf` array (`length`)

The content of the MPU-6050 register will be stored in the `buf` array.

Step 5:

In the `setup()` function, initialize the serial peripheral with a baud rate of 115200:

```
void setup() {  
    Serial.begin(115200);  
    while (!Serial);  
}
```

Step 6:

In the `setup()` function, initialize the I2C clock frequency at the maximum speed that's supported by MPU-6050 (400 KHz):

```
i2c.frequency(400000);
```

Step 7:

In the `setup()` function, use the `read_reg()` function to read the **WHO_AM_I** register (0x75) of the MPU-6050 IMU:

```
#define MPU6050_WHO_AM_I 0x75
char id;
read_reg(MPU6050_ADDR_8BIT, MPU6050_WHO_AM_I, &id, 1);
```

The **WHO_AM_I** register contains the 7-bit I2C address of the MPU-6050 IMU (0x68). Therefore, verify whether the `id` variable matches the value of the `MPU6050_ADDR_7BIT` and communicate the result of this check over the serial peripheral:

```
if(id == MPU6050_ADDR_7BIT) {
    Serial.println("MPU6050 found");
} else {
    Serial.println("MPU6050 not found");
    while(1);
}
```

Compile and upload the sketch on Raspberry Pi Pico. Then, open the serial monitor in the Arduino IDE. If Raspberry Pi Pico can communicate with the MPU-6050 device, you will see the `MPU6050 found` string transmitted over the serial.

There's more...with the SparkFun Artemis Nano!

In this recipe, we learned the basics of the I2C communication protocol. Therefore, to validate our understanding, let's develop the same application for the SparkFun Artemis Nano microcontroller, which provides four I2C peripherals (I2C0, I2C2, I2C3, and I2C4).

On GitHub, we uploaded a SparkFun Artemis Nano pinout diagram (https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Chapter09/Extra/01_i2c_pins_sparkfun_artemis_nano.png) to quickly identify the SDA and SCL pins for each I2C peripheral.

As you will notice from the pinout diagram, one of the four I2C peripherals (**I2C2**) is exclusively accessible through SparkFun's **Qwiic** 4-pin connector. This connector allows you to connect the microcontroller board with the sensor without worrying about accidentally swapping the SDA, SCL, VIN, and GND wires on your breadboard.



To learn more about **SparkFun's Qwiic** connector, refer to its official page at the following link: <https://www.sparkfun.com/qwiic>.

As a result, there are two approaches for connecting the MPU-6050 module to the SparkFun Artemis Nano microcontroller, which are the following:

- Using the Qwiic connector (restricted to I2C2)
- Employing jumper wires (limited to I2C0, I2C3, and I2C4).

In our case, we have replicated this recipe on the SparkFun Artemis Nano by connecting the IMU using the jumper wires and employing the I2C3 peripheral.

The solution for this exercise is available in the Chapter09 folder on the GitHub repository.

Now that we know the basics of the I2C communication protocol, we are ready to acquire the accelerometer data.

In the upcoming recipe, we will delve into the underlying functionality of the accelerometer and show you how to acquire sensor data from the MPU-6050 module.

Acquiring accelerometer data

In this recipe, we will develop an application to read the accelerometer measurements from the MPU-6050 IMU with a frequency of 50 Hz and send them over the serial monitor.



The data transmitted serially will be acquired in the following recipe to build the training dataset.

Getting ready

The accelerometer is a sensor that measures accelerations on one, two, or three spatial axes, denoted as X, Y, and Z.

In this project, we will use the three-axis accelerometer integrated into the MPU-6050 IMU to recognize three hand gestures.

However, how does the accelerometer work, and how can we take the measurements from the sensor?

Let's answer these questions in the following subsection.

The basic principles of the accelerometer

Consider the following system, which consists of a mass attached to one end of a spring:

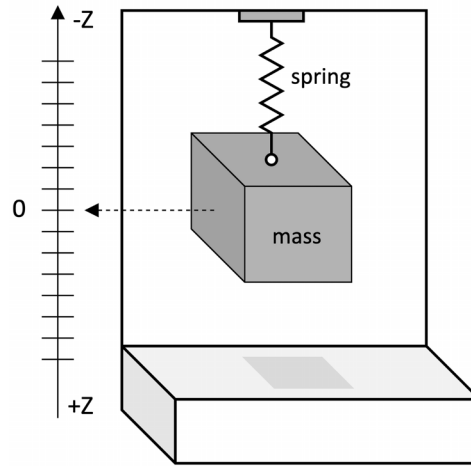


Figure 9.8: Mass-spring system

The preceding diagram models the physical principle of a **one-axis accelerometer** that works on a single spatial dimension (Z, in this case).

What happens if we place the accelerometer on a table?

In such a scenario, we can observe the mass going down due to the constant gravitational force. Consequently, the lower spring on the Z-axis experiences a displacement from its equilibrium position, as shown in the following diagram:

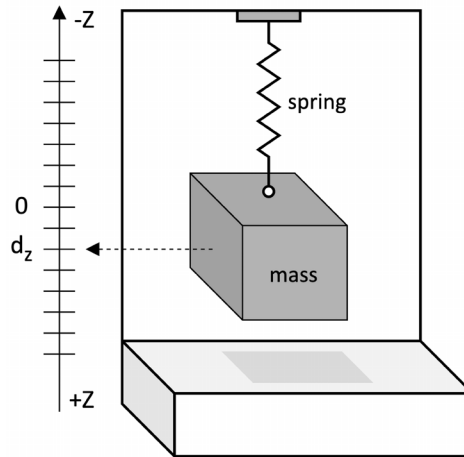


Figure 9.9: The mass-spring system under the influence of gravitational force

From physics class, we know that **Hooke's law** gives the **spring force (restoring force)**:

$$F = k \cdot d_z$$

Here, F is the force, k is the **elastic constant**, and d_z is the displacement.

From **Newton's second law**, we also know that the force that's applied to the mass is as follows:

$$F = m \cdot a$$

where m is the object's mass and a is the experienced acceleration.

When a one-axis accelerometer is placed on the table, the acceleration a corresponds to approximately 9.80665 m/s^2 , representing the object's acceleration as it falls due to the gravitational force.



The 9.80665 m/s^2 acceleration is commonly denoted by the **g** symbol ($9.80665 \text{ m/s}^2 = 1 \text{ g}$).

At this point, by equating the force expressions from Hooke's Law and Newton's Second Law, we have:

$$k \cdot d_z = m \cdot a$$

As a result, the spring displacement d_z is proportional to the acceleration:

$$d_z = \frac{m \cdot a}{k}$$

The spring displacement is the physical quantity the accelerometer acquires to measure acceleration.

Of course, we made some simplifications while explaining the device's functionality. Still, the core mechanism based on the mass-spring system is designed in silicon through the **micro-electromechanical systems (MEMS)** process technology.



Even in the case of accelerometers operating in two or three spatial dimensions, they can still be represented using a mass-spring system. For instance, a three-axis accelerometer can be modeled by employing three independent mass-spring systems, each responsible for measuring acceleration along a different axis.


Most accelerometers have a programmable measurement range (or scale) that can vary generally from $\pm 1 \text{ g}$ ($\pm 9.80665 \text{ m/s}^2$) to $\pm 16 \text{ g}$ ($\pm 156.9 \text{ m/s}^2$). This range is also proportional to the **sensitivity**, commonly expressed as the **least-significant bit over g (LSB/g)** and defined as the *minimum acceleration to cause a change in the numerical representation*. Therefore, the higher the sensitivity, the smaller the minimum detectable acceleration.

In the MPU-6050 IMU, we can program the measurement range through the **ACCEL_CONFIG** register at the **0x1C** address. The following table reports the supported measurement ranges with the corresponding sensitivity:

Measurement range (g)	$\pm 2\text{g}$	$\pm 4\text{g}$	$\pm 8\text{g}$	$\pm 16\text{g}$
Sensitivity (LSB/g)	16384	8192	4096	2048
ACCEL_CONFIG register value	0x00	0x01	0x02	0x03

Figure 9.10: Measurement range with corresponding sensitivity

As we can see from the table, the smaller the measurement range, the higher the sensitivity.



A ±2 g range is generally enough to acquire hand movement accelerations.

The MPU-6050 IMU provides measurements in a 16-bit integer format, which are stored across two 8-bit registers. The register names of these two registers are distinguished by the suffixes `_H` and `_L`, signifying the high and low bytes of the 16-bit variable, respectively. The following table reports the register names and corresponding addresses for the three-axis acceleration measurements:

Address	Register
3B	ACCEL_XOUT_H
3C	ACCEL_XOUT_L
3D	ACCEL_YOUT_H
3E	ACCEL_YOUT_L
3F	ACCEL_ZOUT_H
40	ACCEL_ZOUT_L

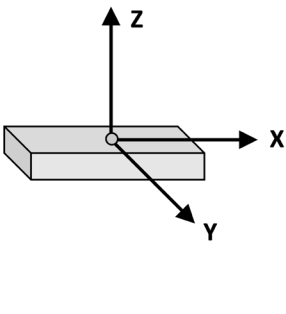


Figure 9.11: Registers to read the three-axis accelerometer measurements

As you can see from the table, the registers are placed at consecutive memory addresses, starting with `ACCEL_XOUT_H` at address `0x3B`. Therefore, reading 6 bytes starting from the `0x3B` address will be sufficient to obtain all accelerometer measurements.

How to do it...

Let’s keep working on the sketch from the previous recipe. The following steps will show you how to extend the program to read accelerometer data from the MPU-6050 IMU and transmit the measurements over the serial monitor.

Step 1:

Implement a function to write a single byte to a register in the MPU-6050:

```
void write_reg(int32_t addr_i2c,
```



```

        int32_t addr_reg,
        char v) {
    char data[2] = {addr_reg, v};
    i2c.write(addr_i2c, data, 2);
    return;
}

```

The `write_reg()` function allows for writing a single byte (`v`) into a specific register (`addr_reg`) of the MPU-6050 IMU with the specified I2C address (`addr_i2c`). This function will be crucial to initialize the MPU-6050 device.

Step 2:

Implement a function to read the accelerometer data from MPU-6050. To do so, create a function called `read_accelerometer()` that takes in three input arguments of type `float*`:

```

void read_accelerometer(float *x, float *y, float *z) {

```

The `x`, `y`, and `z` pointers will store the accelerometer reading for the X-axis, Y-axis, and Z-axis.

Within the `read_accelerometer()` function, read the accelerometer measurements from the MPU-6050 IMU:

```

#define MPU6050_ACCEL_XOUT_H 0x3B
char data[6];
read_reg(MPU6050_ADDR_8BIT,
        MPU6050_ACCEL_XOUT_H,
        data, 6);

```

The preceding code snippet will read 6 bytes of data starting from the high byte of the X-axis accelerometer (`MPU6050_ACCEL_XOUT_H`) from the MPU6050 IMU and store them in the `data` array.

Next, combine the low and high byte of each measurement to get the 16-bit data format representation:

```

int16_t ax_i16 = (int16_t)(data[0] << 8 | data[1]);
int16_t ay_i16 = (int16_t)(data[2] << 8 | data[3]);
int16_t az_i16 = (int16_t)(data[4] << 8 | data[5]);

```

After, convert the integer value into the floating-point acceleration value. To do so, divide the 16-bit integer value by the sensitivity assigned to the selected measurement range and multiply it by g (9.80665 m/s^2):

```
const float sensitivity = 16384.f;
const float k = (1.f / sensitivity) * 9.80665f;
const float ax_f32 = (float)ax_i16 * k;
const float ay_f32 = (float)ay_i16 * k;
const float az_f32 = (float)az_i16 * k;
```

The preceding code converts the raw data into an m/s^2 numerical value. The sensitivity is 16384 because we aim to program the measurement range to $\pm 2 g$.

Finally, assign the accelerometer measurements to the memory location pointed to by the x , y , and z input variables:

```
*x = ax_f32;
*y = ay_f32;
*z = az_f32;
return;
}
```

Step 3:

In the `setup()` function and after the MPU-6050 device discovery, clear the `PWR_MGMT_1` register to ensure that the MPU-6050 IMU is not in sleep mode:

```
if(id == MPU6050_ADDR_7BIT) {
    Serial.println("MPU6050 found");
} else {
    Serial.println("MPU6050 not found");
    while(1);
}

#define MPU6050_PWR_MGMT_1 0x6B
write_reg(MPU6050_ADDR_8BIT, MPU6050_PWR_MGMT_1, 0);
```

When the IMU is in sleep mode, the sensor does not return any measurements. Therefore, we must clear the sixth bit (*bit 6*) of the `PWR_MGMT_1` register to ensure the IMU is active. This operation can easily be done by directly clearing the `PWR_MGMT_1` register.



Most tinyML applications are powered by batteries. Therefore, efficient use of sleep modes is critical to reduce average power consumption.

Step 4:

In the `setup()` function, set the accelerometer measurement range of the MPU-6050 IMU to ± 2 g:

```
#define MPU6050_ACCEL_CONFIG 0x1C
write_reg(MPU6050_ADDR_8BIT, MPU6050_ACCEL_CONFIG, 0);
```

Step 5:

In the `loop()` function, acquire the accelerometer measurements with a frequency of 50 Hz (equivalent to 50 three-axis accelerometer samples per second) and transmit them over the serial monitor. Send the data with one line per accelerometer reading and the three-axis measurements (ax, ay, and az) comma-separated:

```
#define FREQUENCY_HZ 50
#define INTERVAL_MS (1000 / (FREQUENCY_HZ + 1))
#define INTERVAL_US (INTERVAL_MS * 1000)

void loop() {
    mbed::Timer timer;
    timer.start();
    float ax, ay, az;
    read_accelerometer(&ax, &ay, &az);
    Serial.print(ax);
    Serial.print(",");
    Serial.print(ay);
    Serial.print(",");
    Serial.println(az);
    timer.stop();

    using std::chrono::duration_cast;
```

```

using std::chrono::microseconds;
auto t0 = timer.elapsed_time();
auto t_diff = duration_cast<microseconds>(t0);
uint64_t t_wait_us = INTERVAL_US - t_diff.count();
int32_t t_wait_ms = (t_wait_us / 1000);
int32_t t_wait_leftover_us = (t_wait_us % 1000);
delay(t_wait_ms);
delayMicroseconds(t_wait_leftover_us);
}

```

In the preceding code, we did the following:

1. Started the `mbd::Timer` before reading the accelerometer measurements to take the time required to acquire the samples.
2. Read the accelerations with the `read_accelerometer()` function.
3. Stopped `mbd::Timer` and retrieved the elapsed time in microseconds (μ s).
4. Calculated the time the program needs to wait before the next accelerometer reading. This step is necessary to guarantee the 50 Hz sampling rate.

The program is paused with the `delay()` function, followed by `delayMicroseconds()`, due to the following reasons:

- The `delay()` function alone would be inaccurate since this timer needs the input argument in milliseconds (ms) and would be insufficient for achieving accurate timing at a μ s level.
- On the other hand, the `delayMicroseconds()` function can only handle delays up to 16,383 μ s, which is insufficient when targeting a sampling frequency of 50 Hz and requiring a delay of 2,000 μ s.

Therefore, a combination of both functions is necessary. To determine how long to wait in milliseconds, you must divide the `t_wait_us` variable by 1,000. Subsequently, the spare time in microseconds can be obtained with the remainder of the `t_wait_us / 1000` division (`t_wait_us % 1000`).

Sampling the accelerometer with a frequency of 50 Hz is not an arbitrary decision. For most human motion recognition tasks, *a sampling frequency of 20 to 100 Hz is commonly used*. For the task we aim to solve, a sampling frequency of 50 Hz should be enough to capture the hand movements adequately without unnecessary computational overhead and minimize the power consumption, which is critical in many tinyML applications.



The format used to send the accelerometer data over the serial (one line per reading with the three-axis measurements comma-separated) will be necessary to accomplish the task presented in the following recipe.

Now, compile and upload the sketch to Raspberry Pi Pico. Next, open the serial monitor and check whether the microcontroller transmits the accelerometer measurements. If so, lay the breadboard flat on the table. The expected acceleration for the Z-axis (third number of each row) should be roughly equal to the acceleration due to gravity (9.80665 m/s^2), while the accelerations for the other axes should be approximately close to zero, as shown in the following screenshot:

```
0.27, 0.24, 10.15  
0.24, 0.23, 10.12  
0.33, 0.25, 10.14  
0.28, 0.20, 10.17
```

Figure 9.12: Accelerations displayed in the Arduino serial monitor

Although accelerations may be influenced by offset and noise, the accuracy of the measurements is not a significant concern in our case. In fact, the ML model will recognize the intended gestures despite any minor inaccuracies in the data.

There's more...with the SparkFun Artemis Nano!

In this recipe, we learned how to obtain accelerometer data from the MPU-6050 sensor using the I2C interface at regular intervals. If you have successfully connected the sensor to the SparkFun Artemis Nano, replicating this recipe on this board will be straightforward since no modifications are required in the Arduino sketch. The same code can be used on any Arm Cortex-M microcontroller with Arm Mbed OS support. Therefore, all you need to do is compile and upload the program onto the microcontroller to view the accelerometer data through the serial connection. If you use the local Arduino IDE, you can graphically display the transmitted data with the **Serial Plotter** tool, as depicted in the following screenshot:

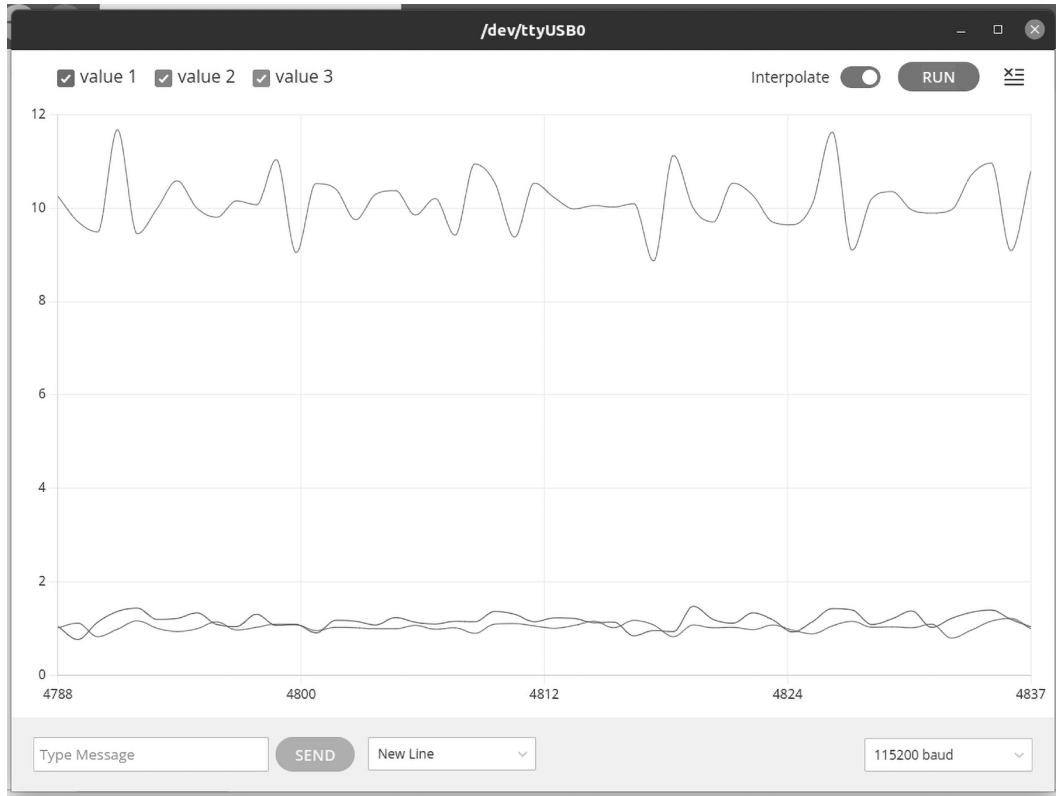


Figure 9.13: Serial Plotter tool available in the Arduino IDE

This tool, which you can access by clicking on the **Serial Plotter** option under **Tools**, provides a real-time graphical representation of the values received through the serial communication between the microcontroller and the computer.

Now that we can acquire accelerometer measurements from the sensor, it is time to start building the training dataset.

In the upcoming recipe, we will show you how to prepare it with Edge Impulse.

Building the dataset with the Edge Impulse data forwarder tool

Any ML algorithm needs a dataset, and for us, this means getting data samples from the accelerometer.

Recording accelerometer data is not as difficult as it may seem at first glance. This task can easily be carried out with Edge Impulse.

In this recipe, we will use the **Edge Impulse data forwarder** tool to take the accelerometer measurements when we make the following three movements with the breadboard:

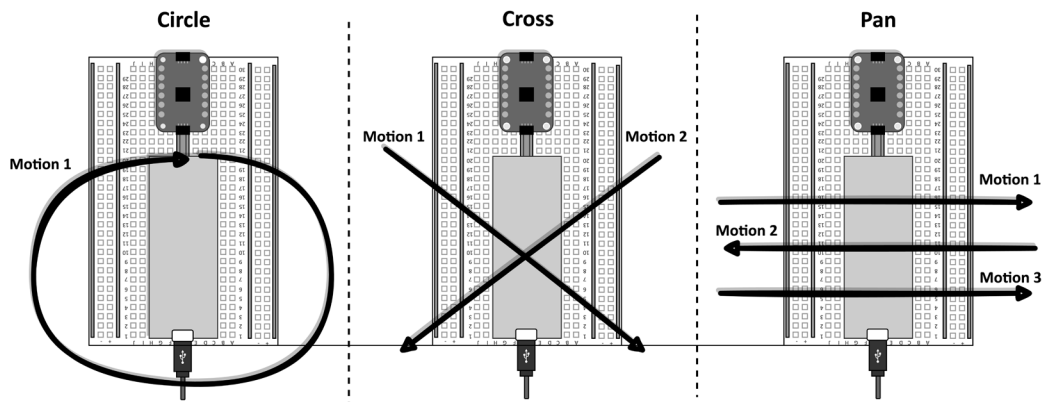


Figure 9.14: Hand gestures to recognize – circle, cross, and pan

When performing the motions indicated by the arrows in the previous figure, ensure that the breadboard is vertical and the Raspberry Pi Pico is in front of you.

Getting ready

The three gestures that we've considered for this project are as follows:

- *Circle:* Moving the board clockwise in a circular motion.
- *Cross:* Moving the board from the top left to the bottom right and then from the right top to the bottom left.
- *Pan:* Moving the board horizontally to the right, left, and right again.

An adequate dataset for gesture recognition requires at least 50 samples for each output class. The duration of the training sample can vary depending on the specific application. In our case, we have chosen 2.5 seconds. Therefore, we recommend completing each movement within approximately 2 seconds to align with the specified sample duration.

Although we have three output classes to identify, an additional one is required to cope with the *unknown* movements.

In this recipe, we will use the Edge Impulse data forwarder to build our dataset. This tool allows us to quickly acquire the accelerations from any device capable of transmitting data over the serial communication and import the samples directly in Edge Impulse.

The data forwarder will run on your computer, so the **Edge Impulse CLI** must be installed. If you haven't installed the Edge Impulse CLI yet, we recommend following the instructions in official documentation: <https://docs.edgeimpulse.com/docs/cli-installation>.

How to do it...

Compile and upload the sketch we developed in the previous recipe on your Raspberry Pi Pico. Then, ensure the Arduino serial monitor is closed because the serial peripheral on your computer can only communicate with one application.

Next, open Edge Impulse and create a new project. Edge Impulse will ask you to write the name of the project. In our case, we have named the project **gesture_recognition**.

Now, follow the following steps to build the dataset with the Edge Impulse data forwarder tool.

Step 1:

From the command line on your computer, run the **edge-impulse-data-forwarder** program with a **50 Hz** frequency and **115200** baud rate:

```
$ edge-impulse-data-forwarder --frequency 50 --baud-rate 115200
```

The data forwarder tool will ask you to authenticate on Edge Impulse, select the project you are working on, and give a name to your device. For example, you could call it **pico** since we are using the Raspberry Pi Pico.

Once you have configured the tool, the program will start parsing serial data. The data forwarder protocol expects one line per sensor reading with the three-axis accelerations either comma (,) or tab-separated, as shown in the following diagram:

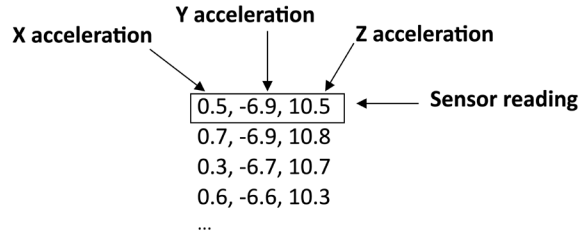


Figure 9.15: Data forwarder protocol

Since our Arduino sketch complies with the protocol we just described, the data forwarder will detect the three-axis measurements transmitted over the serial communication and ask you to assign a name. You can call them **ax**, **ay**, and **az**.

Step 2:

Launch the web browser and open Edge Impulse. Then, enter the **Data acquisition** section. In the **Record new data** area, you should see your Raspberry Pi Pico (**pico**) device:

The screenshot shows the 'Collect data' window in Edge Impulse. It features a header 'Collect data' with a microchip icon and a share icon. Below the header is a 'Device' dropdown menu with a question mark icon, currently set to 'pico'. There are four input fields: 'Label' (set to 'circle'), 'Sample length (ms.)' (set to '20000'), 'Sensor' (set to 'Sensor with 3 axes (ax, ay, az)'), and 'Frequency' (set to '50Hz'). A 'Start sampling' button is located at the bottom right.

Device	Label	Sample length (ms.)	Sensor	Frequency
pico	circle	20000	Sensor with 3 axes (ax, ay, az)	50Hz

Start sampling

Figure 9.16: The Collect data window in Edge Impulse

Ensure your Raspberry Pi Pico device is selected in the **Device** drop-down list and the **Frequency** field is set to **50Hz**.

To acquire the training data for each gesture, enter the label's name in the **Label** field (for example, **circle** for the circle gesture) and specify the recording duration in **Sample length (ms.)**. Once you are ready for recording, click on the **Start sampling** button. Since you will need to record at least 50 samples with a duration of 2.5 s, you can conveniently acquire 20 seconds of data where you repeat the same gestures multiple times, as shown in the following screenshot:

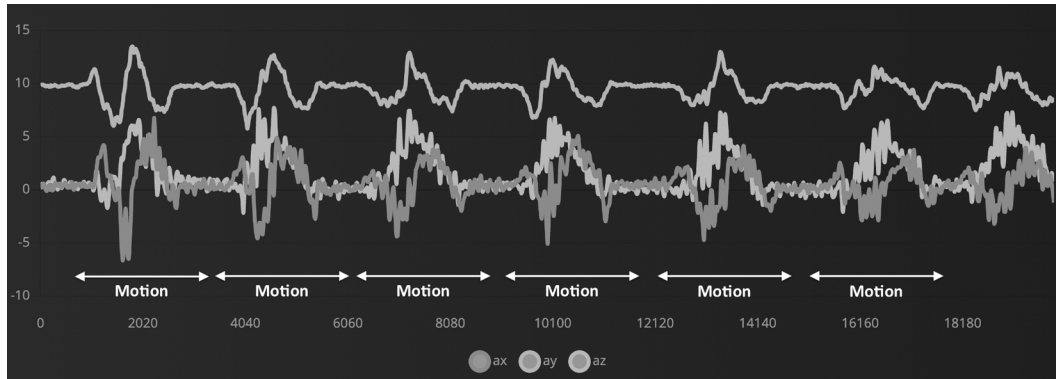


Figure 9.17: A single recording encompassing multiple repetitions of the same motion

We recommend waiting 1 or 2 seconds between movements to help Edge Impulse recognize and extract the motions in the following step.

Step 3:

Split the recording into samples of 2.5 seconds by clicking on the **:** button near the filename and then clicking **Split sample**, as shown in the following screenshot:

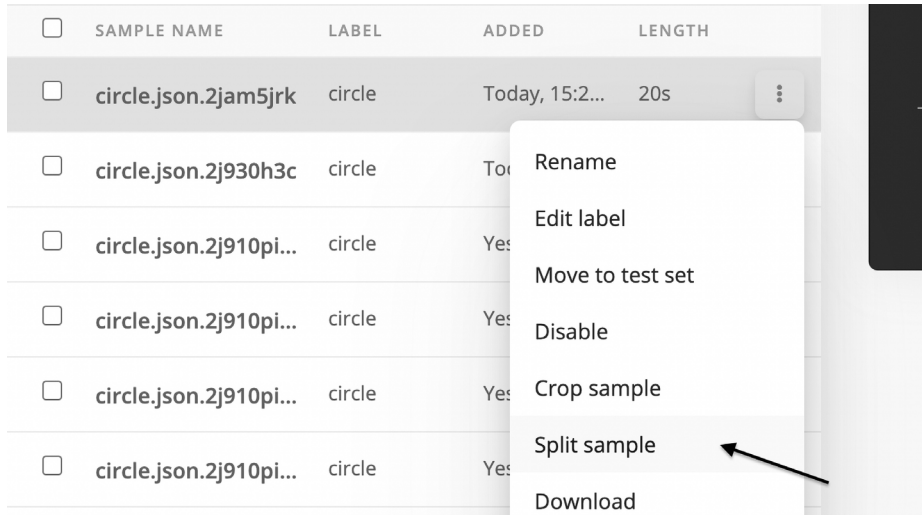


Figure 9.18: The Split sample option

In the new window, set **segment length (ms.)** to **2500** (2.5s) and click **Apply**. Edge Impulse will detect the motions and put a cutting window of 2.5 seconds on each one, as shown in the following screenshot:

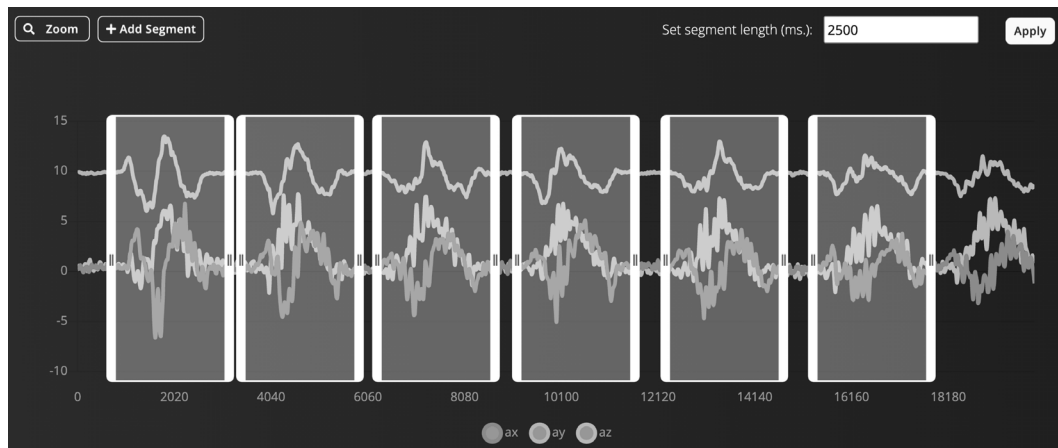


Figure 9.19: Samples are divided into windows of 2.5 seconds each

If Edge Impulse does not recognize a motion in the recording, you can always add the window manually by clicking the **Add Segment** button and clicking on the area you want to cut.

Once all the segments have been selected, click the **Split** button to extract the individual samples.

Step 4:

Record 50 random motions for the *unknown* class, employing a similar approach to what was done for the other hand gestures. To do so, acquire 40 seconds of accelerometer data where you move the breadboard randomly and lay it flat on the table.

Step 5:

Split the *unknown* recording into samples of 2.5 seconds by clicking the **:** button near the filename and selecting the **Split sample** option. In the new window, add 50 cutting windows and click on the **Split** button when you are done.

Step 6:

Split the samples between the training and test datasets by clicking on the **Perform train/test split** button in the **Danger zone** area of the Edge Impulse dashboard. Edge Impulse will ask you twice if you are sure that you want to perform this action. This is because the data shuffling operation is irreversible.

The dataset is now ready, with 80% of the samples assigned to the training/validation set and 20% to the test set.

There's more...

In this recipe, we learned how to acquire data samples from the accelerometer using the Edge Impulse data forwarder tool.

When building a dataset, the more diverse samples we have, the greater the likelihood of obtaining a generalized model. Having a generalized model is crucial for making the model effective across a broad spectrum of users. Therefore, why not consider expanding the dataset by including gestures performed by other people? For example, you may involve people in your household or contact friends to help augment your dataset. By doing so, you can account for variations in movement styles and increase the overall robustness of the dataset.

The dataset is now ready. As a result, we can proceed with the ML model training.

In the upcoming recipe, we will guide you through the model preparation for recognizing hand gestures using Edge Impulse.

Designing and training the ML model

The dataset is in our hands. Therefore, we can start designing the ML model.

In this recipe, we will develop the following architecture with Edge Impulse:

Spectral features

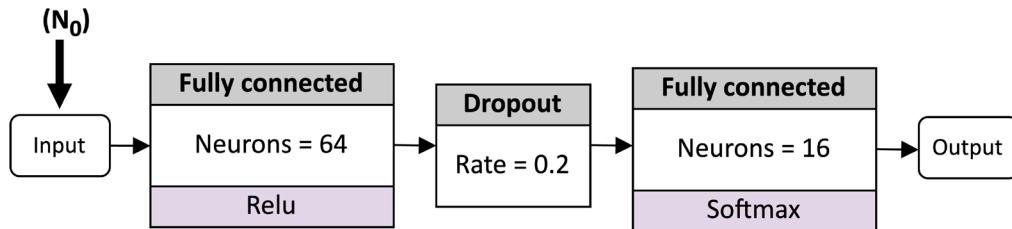


Figure 9.20: Fully connected neural network for recognizing hand gestures from spectral features

As you can see from the preceding figure, the model, which consists of two fully connected layers, takes N_0 spectral features as input. The following *Getting ready* section will explain the reasons behind this design.

Getting ready

In this recipe, we want to explain why the proposed **feedforward** neural network illustrated in Figure 9.20 is enough for recognizing gestures from accelerometer data.

When developing deep neural network architectures, we commonly feed the model with raw data, leaving the network to learn how to extract the features automatically.

This approach is effective and incredibly accurate in various applications, such as image classification. However, there are some applications where basic neural networks trained with hand-crafted engineering features offer similar accuracy results to deep learning while reducing the architecture's complexity. This is the case for gesture recognition, where we can extract features from the frequency domain.



If you are unfamiliar with frequency domain analysis, we recommend reading *Chapter 4, Using Edge Impulse and Arduino Nano to Control LEDs with Voice Commands*.

Further elaboration on the advantages of spectral features will be provided in the following subsection.

Using spectral analysis to recognize hand gestures

Spectral analysis allows us to discover characteristics of the signal that are not apparent in the time domain. For example, consider the following two signals:

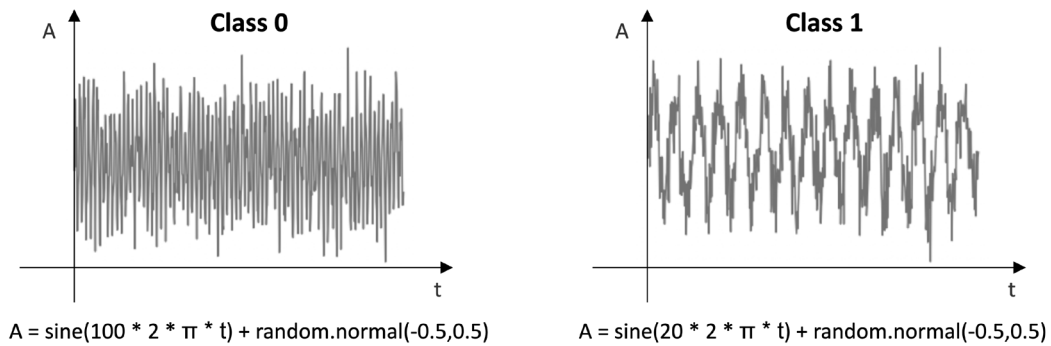


Figure 9.21: Two signals in the time domain

These two signals are assigned to two different classes: **class 0** and **class 1**.

What features would you use to discriminate class 0 from class 1 in the time domain?

Whatever set of features you may consider must be shift-invariant and robust to noise to be effective. Although there may be a set of features that distinguish **class 0** from **class 1**, the solution would be straightforward if we considered the problem in the frequency domain, as shown by their power spectrums in the following diagram:

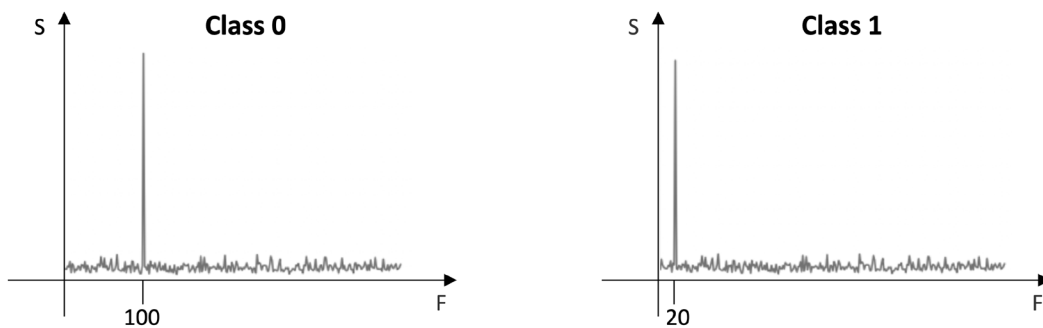


Figure 9.22: Frequency representations of the class 0 and class 1 signals

As we can see, **class 0** and **class 1** have different **dominant frequencies**, defined as the components with the highest magnitude. In other words, *the dominant frequencies are the components that carry more energy*.

Although signals from an accelerometer are not the same as **class 0** and **class 1**, they exhibit recurring patterns that make their frequency components well-suited for a classification task.

Another benefit of the frequency representation is the *possibility of obtaining a compressed representation of the original signal*.

For example, let's consider our training samples acquired with a three-axis accelerometer with a frequency of 50 Hz for 2.5 seconds. Each instance contains 375 data points (125 data points per axis). Now, let's apply the **Fast Fourier Transform (FFT)** with 128 output frequencies (**FFT length**) on each sample. This domain transformation produces 384 data points (128 data points per axis). Hence, FFT does not seem to be reducing the amount of data. However, as we saw in the previous example with **class 0** and **class 1**, not all frequencies provide meaningful information. Therefore, we could just extract the frequencies that bring the most energy (dominant frequencies) to reduce the amount of data and then facilitate the signal pattern recognition.

For gesture recognition, the spectral features are generally obtained by doing the following:

1. Taking half of the FFT output frequencies because the spectrum is symmetrical around 0 Hz.
2. Applying a low-pass filter in the frequency domain to remove the highest frequencies. This step makes feature extraction more robust against noise and reduces the number of spectral features by discarding the frequency bins above the cut-off frequency. Therefore, higher cut-off frequencies retain more frequency components. A reasonable cut-off frequency for hand gesture recognition is typically between 3 Hz and 10 Hz.
3. Extracting the frequency components with the highest magnitude. Commonly, we take the frequency with the highest peak.

From the preceding points, we can obtain a reduced number of spectral features, which usually ranges between 30 and 100, depending on the cut-off frequency. This data reduction from the original signal allows us to successfully train a tiny feedforward neural network like the one presented in *Figure 9.20*.

How to do it...

In Edge Impulse, click the **Create Impulse** tab from the left-hand menu. In the **Create Impulse** section, set the **Window size** to **2500ms** and the **Window increase** to **400ms**.

As we saw in *Chapter 4, Using Edge Impulse and Arduino Nano to Control LEDs with Voice Commands*, the **Window increase** parameter is required to run ML inference at regular intervals.

The following steps will show how to design the neural network presented at the beginning of this recipe.

Step 1:

Click the **Add a processing block** button and look for **Spectral Analysis**:

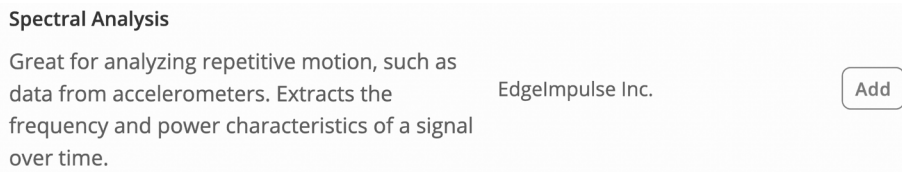


Figure 9.23: The Spectral Analysis processing block

Click the **Add** button to integrate the processing block into Impulse.

Step 2:

Click the **Add a learning block** button and select **Classification**.

The **Output features** block should report the four output classes (**circle**, **cross**, **pan**, and **unknown**), as shown in the following screenshot:

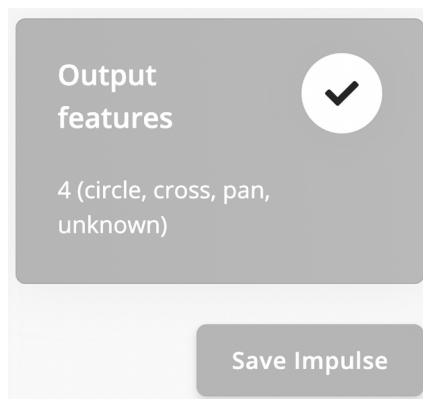


Figure 9.24: The Output features block

Click the **Save Impulse** button to save the Impulse.

Step 3:

Click the **Spectral features** button from the **Impulse design** category:

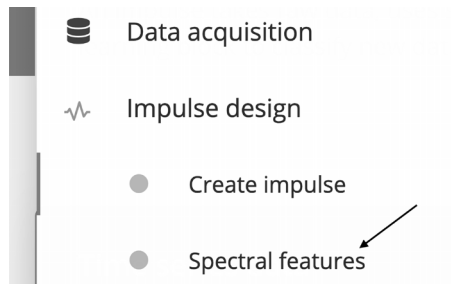


Figure 9.25: The spectral features button

In the new window, you can adjust the parameters related to feature extraction. For instance, you can choose the type of filter for the input signal in the **Filter** box and modify parameters that influence the FFT computation in the **Analysis** box.

In the **Filter** box, choose the **low** option from the **Type** drop-down menu to use a low-pass filter on the input signal. Next, adjust the **Cut-off frequency** to 10 Hz and the **Order** to 2. By setting these values, the Impulse will employ a 2nd-order **Butterworth** filter in the signal's frequency domain and remove frequencies above 10 Hz. You might now be curious: why did we opt for 10 Hz? The cut-off frequency plays a pivotal role in determining the quantity of spectral features. Thus, it must be set high enough to encompass all the predominant components that offer the most insight in distinguishing one gesture from another. Within Edge Impulse, this can be observed in the **Spectral power** chart. Alterations to the value in the **Cut-off frequency** field will lead to changes in the spectral power. In our tests, we found that 10 Hz adequately captures the highest peaks across all three gestures.

In the **Analysis** box, choose the **FFT** option from the **Type** drop-down to execute the FFT on the input signal. Next, configure the **FFT length** to 128 and ensure both **Take log of spectrum?** and **Overlap FFT frames?** options are deselected. Using these configurations, the Impulse will generate 128 output frequencies from the raw data presented in a linear scale.

Now, click the **Save parameters** button and then click the **Generate features** button to extract the spectral features from each training sample. Edge Impulse will return the **Job completed** message in the output log when the feature extraction process ends.

After this step, Edge Impulse will display the extracted features from each sample in the **Feature explorer** box. If you observe a clear separation among the four classes, it's an encouraging indication of the effectiveness of extracted features for our ML model.

Step 4:

Click the **Classifier** button under the Impulse design section and introduce a **Dropout layer** with a 0.2 ratio between the fully connected layers (referred to as the **Dense layer** in the subsequent image). Make sure the initial fully connected layer has **64** neurons while the other has **16** neurons, as shown in the following screenshot.

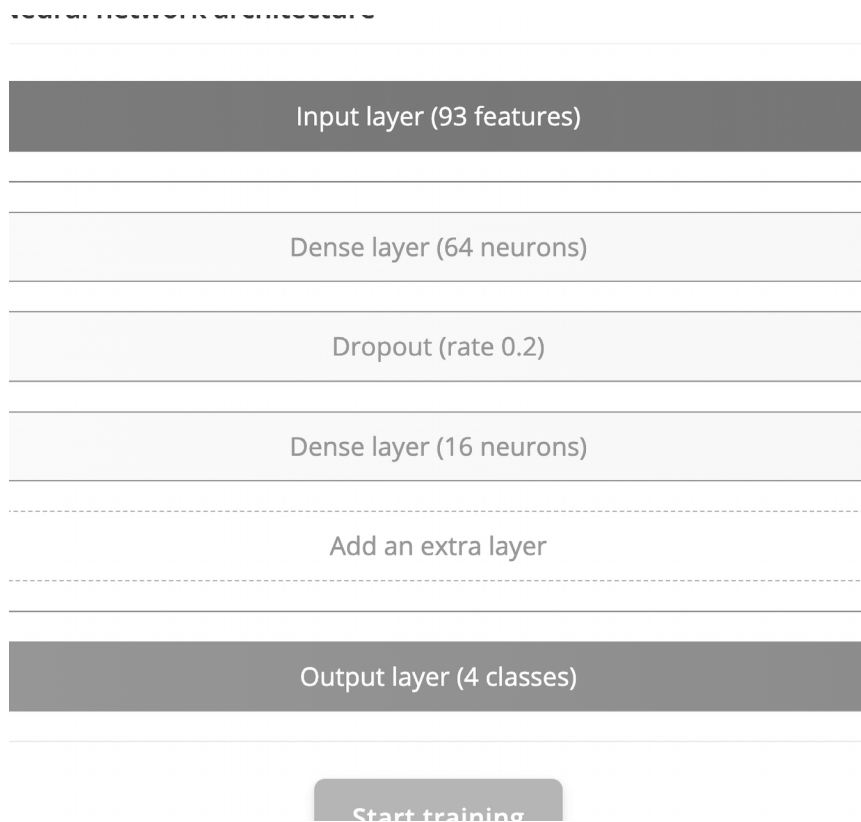


Figure 9.26: Neural network architecture

Set the **Number of training cycles** to **60** and launch the training by clicking the **Start training** button. The output console will report the accuracy and loss on the training and validation datasets during training after each epoch.

Step 5:

Evaluate the model's performance on the test dataset. To do so, click the **Model testing** button from the left panel. In the new window, click the **Classify all** button.

Edge Impulse will provide this progress in the **Model testing output** section and produce the confusion matrix once the process is completed:



Confusion matrix (validation set)

	CIRCLE	CROSS	PAN	UNKNOWN
CIRCLE	100%	0%	0%	0%
CROSS	0%	100%	0%	0%
PAN	0%	0%	100%	0%
UNKNOWN	10%	0%	0%	90%
F1 SCORE	0.97	1.00	1.00	0.95

Figure 9.27: The confusion matrix

As you can see from the previous screenshot, the tiny model can achieve an accuracy of **97.8 %** on our dataset.

There's more

In this recipe, we learned how to successfully train a model for gesture recognition using Edge Impulse.

The model we just trained demonstrated its effectiveness in recognizing our gestures. Nevertheless, it is worth exploring whether this solution is the best we can obtain regarding memory utilization and latency. Therefore, we recommend using the Edge Impulse EON Tuner to fine-tune the model and identify the optimal combination of the processing block and ML model.

Having trained the model, it is now the moment to evaluate its accuracy on unseen data before deploying it on the microcontroller.

In the upcoming recipe, we will use the Live classification tool of Edge Impulse to accomplish this task.

Live classifications with the Edge Impulse data forwarder tool

Deploying a model on microcontrollers is error-prone because the code may contain bugs, the integration could be incorrect, or the model could not work reliably in the field. Consequently, conducting model testing becomes essential to exclude at least the ML model from the source of failures. In this recipe, we will use the Live classification tool of Edge Impulse to carry out this investigation.

Getting ready

The most effective approach to assess the performance of an ML model is to evaluate its behavior directly on the target platform, and the Edge Impulse data forwarder tool provides a straightforward method for doing so.

In our specific case, since the dataset was built using the Raspberry Pi Pico, we have gained initial insights into the model's accuracy during the training phase.

However, there may be instances where the dataset may not be built on top of sensor data coming from the target device. In such cases, the deployed model on the microcontroller might exhibit behavior that differs from our expectations. This performance discrepancy is often attributed to variations in sensor specifications. Even though sensors may belong to the same type, they can have distinct characteristics such as offset, accuracy, range, sensitivity, and more. These differences can significantly impact the accuracy of the deployed model on the target platform.

How to do it...

Ensure your Raspberry Pi Pico device is still running the program we developed in the *Acquiring accelerometer data* recipe and that the **edge-impulse-data-forwarder** program is running on your computer. Next, click the **Live classification** tab and check whether the Raspberry Pi Pico device is being reported in the **Device** drop-down list, as shown in the following screenshot:



The screenshot shows two dropdown menus. The first is labeled 'Device' with a question mark icon, and it has 'pico' selected. The second is labeled 'Sensor', and it has 'Sensor with 3 axes (ax,' selected. Both dropdowns have a downward arrow icon on the right side.

Figure 9.28: The Device field must report the Raspberry Pi Pico microcontroller



If the device is not listed, follow the steps in the *How to do it...* section of the *Acquiring accelerometer data* recipe to pair your Raspberry Pi Pico with Edge Impulse again.

Now, follow the following steps to evaluate the model's performance with the live classification tool.

Step 1:

In the **Live classification** window, select **Sensor with 3 axes** from the **Sensor** drop-down list and set **Sample length (ms)** to **20000**. Keep **Frequency** at the default value of **50Hz**.

Step 2:

Position the Raspberry Pi Pico device in front of you and click the **Start sampling** button.

When the **Sampling...** message appears on the button, make the three movements the model can recognize (*circle*, *cross*, or *pan*). After the recording, the sample will be uploaded to Edge Impulse. Then, Edge Impulse will split the recording into samples of 2.5 seconds and run the model inference on each one. The classification results will be presented on the same page.

Step 3:

Check whether you can correctly classify the three gestures. To do so, click on the vertical ellipsis (three dots) of the test sample and then click on the **Show classification** option:

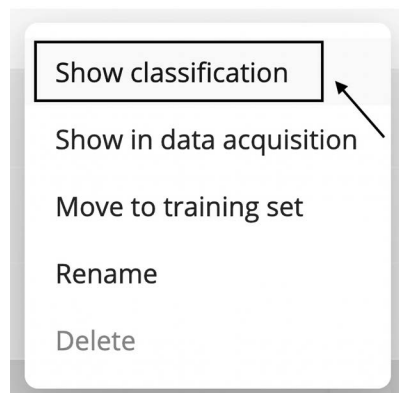


Figure 9.29: The Show classification option

If your model is well trained, you should observe numerous correct gesture detections with high accuracy (over 80%) around the time frame when you executed the hand motion.

There's more...with the SparkFun Artemis Nano

In this recipe, we learned how to use the Live classification tool of Edge Impulse to test the trained model on unseen data.

If you are interested in testing the model on the SparkFun Artemis Nano, the process is straightforward. Simply disconnect the Raspberry Pi Pico, connect the SparkFun Artemis Nano, and rerun the `edge-impulse-data-forwarder` tool.

After evaluating the model's effectiveness on unseen data, we can proceed with the model deployment.

In the upcoming recipe, we will build an application for the Raspberry Pi Pico to run the model's inference on a continuous data stream from the accelerometer and return the classification result through serial communication.

Developing a continuous gesture recognition application with Edge Impulse and Arm Mbed OS

Now that we have tested the model, we are ready to build the sketch in the Arduino IDE to recognize our three motion gestures. In this recipe, we will build a continuous gesture recognition application with the help of Edge Impulse, Arm Mbed OS, and an algorithm to filter out redundant or spurious classification results.

Getting ready

Our goal is to develop a continuous gesture recognition application, which means that the accelerometer data sampling and ML inference must be performed concurrently. This approach guarantees that we capture and process all the pieces of the input data stream so we don't miss any events.

The main ingredients to accomplish this task are as follows:

- Arm Mbed OS for writing a multithreading program
- An algorithm to filter out redundant classification results

Let's start by learning how to perform concurrent tasks easily with the help of **real-time operating system (RTOS)** APIs provided by Arm Mbed OS.

Managing concurrent tasks with Arm Mbed OS

All Arduino sketches developed for the Arduino Nano 33 BLE Sense, Raspberry Pi Pico, and Spark-Fun Artemis Nano are built on Arm Mbed OS, an open-source RTOS for Arm Cortex-M microcontrollers. So far, we have only used Mbed APIs to interface peripherals such as GPIO and I2C. Nevertheless, Arm Mbed OS extends beyond peripheral communication, offering functionalities typical of a canonical OS, such as the ability to manage concurrent tasks through **thread management**.



If you are interested in learning more about the functionalities of Arm Mbed OS, we recommend reading the official documentation, which can be found at the following link: <https://os.mbed.com/docs/mbed-os/v6.16/bare-metal/index.html>.

In a microcontroller, a thread refers to a piece of a program that operates independently on a single core. Since we may have multiple threads running simultaneously, the RTOS scheduler will be responsible for determining which one to execute and for how long.

Mbed OS employs a **preemptive scheduler** and uses a **round-robin** priority-based **scheduling algorithm** to ensure fair CPU utilization and prioritize time-sensitive tasks.

In our application, we will need two threads:

- **Sampling thread:** The thread that's responsible for acquiring the accelerations from the MPU-6050 IMU with a frequency of 50 Hz
- **Inference thread:** The thread that's responsible for running model inference after every 200 ms

In Mbed OS, a thread is created with the **rtos::Thread** which needs to be initialized with the intended thread priority.



The available priority values can be found at <https://os.mbed.com/docs/mbed-os/v6.16/apis/thread.html>.

It is important to note that the instantiation of the **rtos::Thread** does not begin the thread execution. The thread's execution starts when we call its **start()** method, which needs the function to be executed as an input argument.

As we mentioned at the beginning of this *Getting ready* section, developing a gesture recognition application also requires a filtering algorithm to filter out unnecessary and inaccurate predictions.

Let’s uncover this algorithm in the following subsection.

Filtering out redundant and spurious predictions

Our gesture recognition application employs a sliding window-based approach over a continuous data stream to determine whether we have a motion of interest. The idea behind this approach is to split the data stream into smaller windows of a fixed size and execute the ML inference on each one. As we already know, ML is a powerful tool for gathering robust classification results, even in the presence of temporal shifts within the input data. Therefore, we expect neighboring windows to have similar and high probability scores, leading to multiple and redundant detections.

In this recipe, we will adopt a **test and trace filtering algorithm** to make our application robust against spurious detections. This filtering algorithm considers the ML output class to be valid if the last *N* predictions (for example, the previous four) meet the following conditions:

- The output class remains the same and is not the *unknown* one
- The probability score is above a fixed **threshold** (for example, greater than 0.7)

To visually understand how this algorithm works, look at *Figure 9.30*:

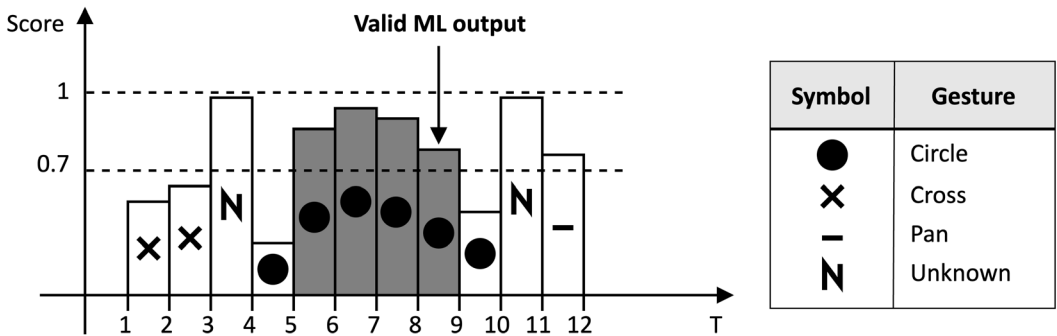


Figure 9.30: Example of a valid ML prediction

In the preceding diagram, each rectangular bar is the predicted class at a given time, where the following occurs:

- The symbol represents the output class
- The bar’s height is the probability score associated with the predicted class

Therefore, considering *N* as four and the *probability threshold* as 0.7, we can consider the ML output class valid only at **T=8**. In fact, the previous four classification results returned the *circle* motion and had probability scores greater than 0.7.

How to do it...

In Edge Impulse, click on **Deployment** from the left-hand side menu and select **Arduino Library** from the **Search deployment options**, as shown in the following screenshot:

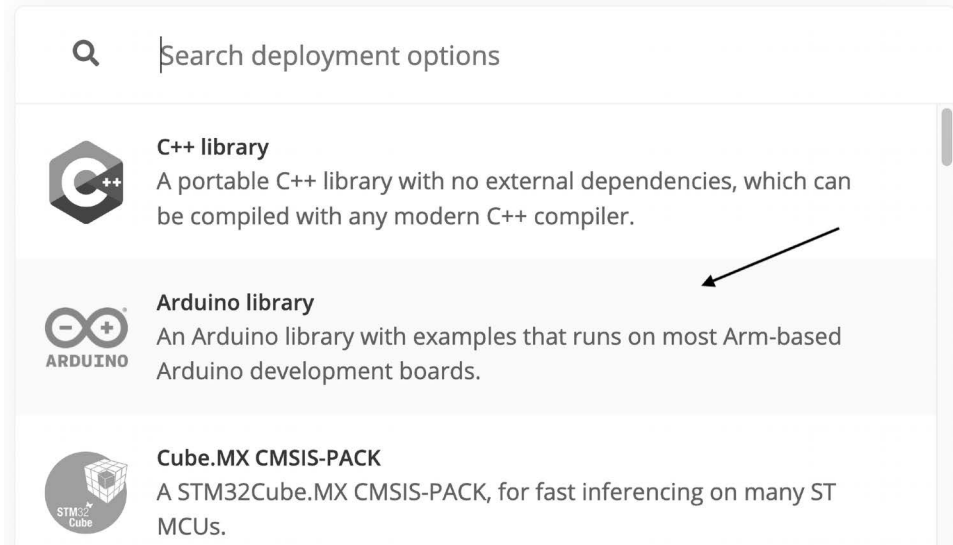
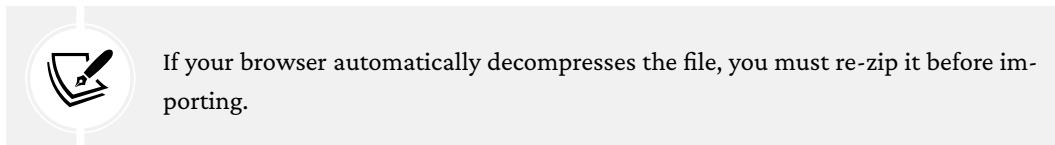


Figure 9.31: Edge Impulse deployment section

Then, click the **Build** button at the bottom of the page. Save the ZIP file on your computer.



Next, open Arduino IDE and import the library created with Edge Impulse. After, copy the sketch we developed in the *Acquiring accelerometer data* recipe in a new sketch. Now, follow the following steps to make the Raspberry Pi Pico capable of recognizing our three gestures.

Step 1:

Include the `<edge_impulse_project_name>_inferencing.h` header file in the sketch. For example, if the Edge Impulse project's name is `gesture_recognition`, you should include the following file:

```
#include <gesture_recognition_inferencing.h>
```

Including this header file is the only prerequisite for accessing the constants, functions, and C macros developed by Edge Impulse to facilitate the execution of the ML model.



The Arduino library built by Edge Impulse includes the header file `model_metadata.h`, which contains the Edge Impulse macro definitions for ML inference. These macros are marked with the `EI_` prefix and documented at the following link: https://docs.edgeimpulse.com/reference/model_metadadatah.

After including the Edge Impulse header file, include the `mbed.h` file to access the Mbed OS functionalities:

```
#include "mbed.h"
```

Step 2:

Declare two floating-point arrays, namely `buf_sampling` and `buf_inference`, both consisting of 375 elements:

```
#define INPUT_SIZE EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE
float buf_sampling[INPUT_SIZE] = { 0 };
float buf_inference[INPUT_SIZE];
```

In the preceding code, we used the Edge Impulse `EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE` C macro definition to get the number of input samples required for 2.5 seconds of accelerometer data (375).

The `buf_sampling` array will be used by the *sampling thread* to store the accelerometer data, while the `buf_inference` one will be used by the *inference thread* to feed the input to the model.

Step 3:

Declare a global thread object with a low-priority schedule for running the ML model:

```
rtos::Thread inference_thread(osPriorityLow);
```

The *inference thread* should have a lower priority (`osPriorityLow`) than the *sampling thread* because it has a longer execution time due to ML inference. Therefore, a low-priority schedule for the inference thread will guarantee we do not miss any accelerometer data samples.

Step 4:

Create a C++ class to implement the test and trace filtering algorithm. Make the filtering parameters (N and *probability threshold*) and the variables that are needed to trace the ML predictions (*counter* and the *last output valid class index*) as private members:

```
class TestAndTraceFilter {
private:
    int32_t      _n {0};
    float        _thr {0.0f};
    int32_t      _counter {0};
    int32_t      _last_idx_class {-1};
    const int32_t _num_classes {3};
```

The algorithm mainly needs two variables to trace the classification results. These variables are as follows:

- `_counter`: This variable is used to keep track of how many times we had the same classification with a probability score above the fixed threshold (`_thr`).
- `_last_idx_class`: This variable is used to find the output class index of the last inference.

In this recipe, we assign -1 to the `_last_idx_class` variable when the last inference returns either *unknown* or the probability score is below the fixed threshold (`_thr`).

Step 5:

In the `TestAndTraceFilter` class, declare an integer constant to hold the value for the invalid output index class (-1) as a public member:

```
public:
    static constexpr int32_t invalid_idx_class = -1;
```

Step 6:

In the `TestAndTraceFilter` class, implement the constructor. The constructor should accept the filtering parameters (N and *probability threshold*) as input arguments:

```
public:
    TestAndTraceFilter(int32_t n, float thr) {
        _thr = thr;
```

```
    _n = n;  
}
```

Step 7:

In the `TestAndTraceFilter` class, implement a public method to reset the internal variables (`_counter` and `_last_idx_class`) that are used to trace the ML predictions:

```
void reset() {  
    _counter = 0;  
    _last_idx_class = invalid_idx_class;  
}
```

Step 8:

In the `TestAndTraceFilter` class, implement a public method to update the filtering algorithm with the latest classification result:

```
void update(size_t idx_class, float prob) {  
    if(idx_class >= _num_classes || prob <= _thr) {  
        // Reset state  
        reset();  
    }  
    else {  
        if(prob > _thr) {  
            // Incremental state  
            if(idx_class != _last_idx_class) {  
                _last_idx_class = idx_class;  
                _counter = 0;  
            }  
            _counter += 1;  
        }  
        else {  
            // Reset state  
            reset();  
        }  
    }  
}  
} // void update(size_t idx_class, float prob) {
```

To explain the behavior of the preceding snippet code, consider the following flowchart:

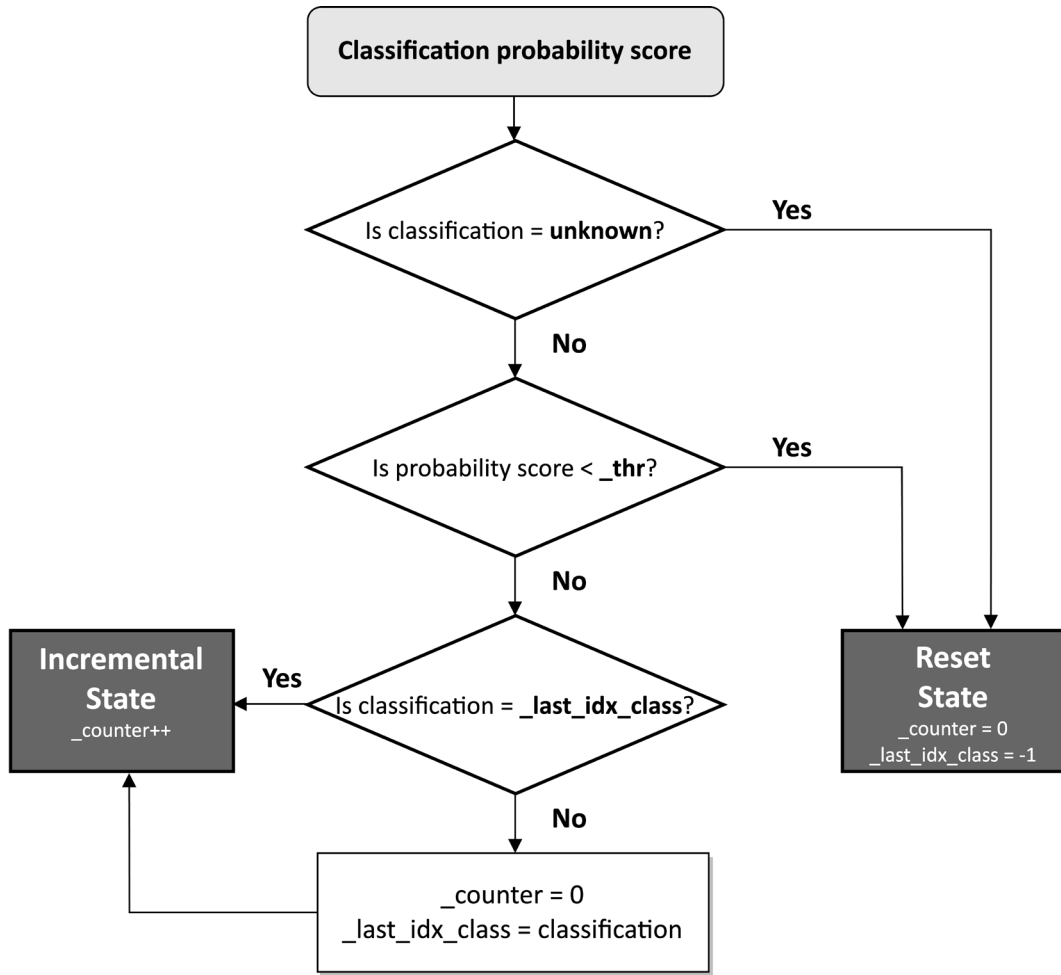


Figure 9.32: Test and trace filtering flowchart

As you can observe from the flowchart, the `TestAndTraceFilter` object works in two states – *incremental* and *reset*.

The *incremental* state occurs when the most recent classification is not *unknown* and the probability score is greater than the probability threshold (`_thr`). In all the other cases, we enter the *reset* state, setting the `_counter` variable to 0 and the `_last_idx_class` variable to -1.

In the *incremental* state, the `_counter` variable is incremented by one, and the `_last_idx_class` variable keeps the index of the output class.

Step 9:

In the `TestAndTraceFilter` class, implement a public method to return the filter's output. If the `_counter` variable is greater than or equal to `_n`, return the `_last_idx_class` and reset the test and trace filter function:

```
int32_t output() {
    if(_counter >= _n) {
        int32_t out = _last_idx_class;
        reset();
        return out;
    }
    else {
        return invalid_idx_class;
    }
}
}; // class TestAndTraceFilter
```

Step 10:

Write the function to be executed by the RTOS inference thread (`inference_thread`) to run the ML inference whenever the sampling buffer is full. To do so, define the function without any input arguments:

```
void inference_func() {
```

Within the function, initialize the test and trace filter object. Set N and *probability threshold* to 3 and 0.7f, respectively:

```
TestAndTraceFilter filter(3, 0.7f);
```

Then, wait for the first sampling buffer to become full:

```
delay((EI_CLASSIFIER_INTERVAL_MS * EI_CLASSIFIER_RAW_SAMPLE_COUNT) +
100);
```

The `delay()` function ensures that the required accelerometer samples for the first ML inference are available in the sampling buffer. The delay duration is determined by multiplying the interval (in milliseconds) between each sample acquisition (`EI_CLASSIFIER_INTERVAL_MS`) by the number of accelerometer samples (`EI_CLASSIFIER_RAW_SAMPLE_COUNT`) and adding 100 milliseconds for extra assurance.

Once the first sampling buffer is ready, enter an infinite while loop where you run ML inference:

```
while (1) {
    memcpy(buf_inference,
           buf_sampling,
           INPUT_SIZE * sizeof(float));
    signal_t signal;
    numpy::signal_from_buffer(buf_inference,
                              INPUT_SIZE,
                              &signal);
    ei_impulse_result_t result = { 0 };
    run_classifier(&signal, &result, false);
```

As you can see from the preceding snippet code, the data is initially transferred from the buf_sampling array to the buf_inference one. Following that, we initialize the Edge Impulse signal_t object with the buf_inference buffer, which is needed to run the ML inference with the run_classifier() function.

After ML inference, get the output class with the highest probability and update the TestAndTraceFilter object with the latest classification result:

```
size_t ix_max = 0; float pb_max = 0;
#define NUM_CLASSES EI_CLASSIFIER_LABEL_COUNT

size_t ix = 0;
for (; ix < NUM_CLASSES; ix++) {
    if(result.classification[ix].value > pb_max) {
        ix_max = ix;
        pb_max = result.classification[ix].value;
    }
}
filter.update(ix_max, pb_max);
```

Then, read the output of the TestAndTraceFilter object. If the output is not -1 (invalid output), send the label that was assigned to the predicted gesture over the serial port and wait for two seconds before starting a new inference:

```
int32_t out = filter.output();
if(out != filter.invalid_idx_class) {
    Serial.println(result.classification[out].label);
```

```
    delay(2000);
}
```

If the output is invalid, wait for 400 ms (**window increase** set in the Edge Impulse project) before running the subsequent inference:

```
    else {
        delay(400);
    }
} // while(1)
} // end function
```

The `delay()` function puts the current thread in a waiting state. As a rule of thumb, *we should always put a thread in a waiting state when it does not perform computation for a long time*. This approach guarantees that we don't waste computational resources and that other threads can run in the meantime.

Step 11:

In the `setup()` function, start the RTOS inference thread (`inference_thread`):

```
auto func_mbed = mbed::callback(&inference_func);
inference_thread.start(func_mbed);
```

In the `loop()` function, replace the print statements used to transmit the accelerometer measurements over the serial with the code that's required to fill the `buf_sampling` array:

```
float ax, ay, az;
read_accelerometer(&ax, &ay, &az);
numpy::roll(buf_sampling, INPUT_SIZE, -3);
buf_sampling[INPUT_SIZE - 3] = ax;
buf_sampling[INPUT_SIZE - 2] = ay;
buf_sampling[INPUT_SIZE - 1] = az;
```

Since the Arduino `loop()` function is an RTOS thread with high priority, we don't need to create an additional thread for sampling the accelerometer measurements.

The `buf_sampling` array is filled as follows:

1. Shifting the data in the `buf_sampling` array by three positions using the `numpy::roll()` function. The `numpy::roll()` function is provided by the Edge Impulse library, and it works similarly to its Python version.

2. Storing the three-axis accelerometer measurements (ax, ay, and az) in the last three positions of the `buf_sampling` array.

By adopting this approach, we ensure that the latest accelerometer measurements are always in the last three positions of the `buf_sampling` array. Therefore, the inference thread can copy this buffer's content into the `buf_inference` array and feed the ML model directly without performing data reshuffling.

Step 12:

If you haven't already, stop the `edge-impulse-data-forwarder` tool for releasing the serial port before programming the microcontroller. Then, compile and upload the sketch on the Raspberry Pi Pico. Now, if you make any of the three movements the ML model can recognize (*circle*, *cross*, or *pan*), you will see the gestures recognized in the Arduino serial monitor.

There's more...with the SparkFun Artemis Nano

In this recipe, we learned how to develop a continuous gesture recognition application for the Raspberry Pi Pico using the Edge Impulse Inferencing SDK and Arm MbedOS.

The exciting news is that this sketch is compatible with any Arduino compatible board with an Arm Cortex-M-based microcontroller.

Therefore, what do you think about testing the application on the SparkFun Artemis Nano microcontroller?

Unfortunately, at the time of writing, there is a known issue when compiling the project for this platform due to a function name collision, which throws the following error message: *error: macro "F" passed 2 arguments, but takes just 1.*

However, the discussion around this problem, as well as the simple workaround, can be found in the Edge Impulse forum at the following link: <https://forum.edgeimpulse.com/t/macro-f-passed-2-arguments-but-takes-just-1/3026>. This function name collision can be easily solved by changing the template argument `T` in the `src/edge-impulse-sdk/tensorflow/lite/kernels/internal/reference/comparisons.h` file within the Arduino Edge Impulse library to a different name (for example, `TFLiteFunc`). To do so:

1. Unzip the Arduino Edge Impulse library downloaded from Edge Impulse.
2. Open the `src/edge-impulse-sdk/tensorflow/lite/kernels/internal/reference/comparisons.h` file with a text editor.

3. Inside the `comparisons.h` file, replace `F>` with `TFLiteFunc>`.
4. Inside the `comparisons.h` file, replace `F(` with `TFLiteFunc(`.
5. Compress the Arduino Edge Impulse library again with Zip and import the compressed file into the Arduino IDE.

Now that we have successfully deployed the model on the microcontroller, the remaining step is controlling the YouTube playback with gestures.

In the upcoming final recipe, we will tackle this task by implementing a Python script to read the recognized gestures transmitted over the serial port and simulate keyboard input with the `PyAutoGUI` library.

Building a gesture-based interface with PyAutoGUI

Now that we can recognize the hand gestures with the Raspberry Pi Pico, our final objective is to create a touchless interface for controlling YouTube video playback.

In this recipe, we will develop a Python script to read the recognized motion transmitted over the serial port and use the `PyAutoGUI` library to build a gesture-based interface to play, pause, mute, unmute, and change YouTube videos.

Getting ready

The Python script that we are going to develop in this recipe relies primarily on two Python libraries:

- `pySerial`, which allows us to read the data transmitted serially
- `PyAutoGUI` (<https://pyautogui.readthedocs.io/>), which allows controlling the mouse and simulating keyboard input

The `PyAutoGUI` library can be installed with the following `pip` command:

```
$ pip install pyautogui
```

The API of this `PyAutoGUI` has been designed to be very simple to use. For example, we can easily simulate the keyboard key pressing using the `press()` method provided by this library. This method only requires a string as an input argument specifying the button to press. Additionally, the library offers the `hotkey()` method to simulate the creation of key combinations, also known as **hotkeys**, by specifying multiple keys together.

At this stage, the final step is to decide the actions to assign to each hand motion and familiarize yourself with the keyboard shortcuts for YouTube video playback.



You can find the list of keyboard shortcuts for YouTube video playback at the following link: <https://support.google.com/youtube/answer/7631406>.

The following table reports our mapping choices with their corresponding keyboard shortcuts:

Gesture	Playback action	Keyboard shortcut
Circle	Mute/Unmute	m
Cross	Play/Pause	k
Pan	Move to the next video	Shift + N

Figure 9.33: Table reporting the function and keyboard shortcut associated with each hand motion

To illustrate with an example, suppose the microcontroller transmits the gesture **circle** over the serial connection. In this case, we must simulate the key press of **m** to mute/unmute the YouTube video playback.

How to do it...

Create a new Python script on your computer and follow the following steps to build a gesture-based interface with PyAutoGUI.

Step 1:

Import the necessary Python libraries:

```
import pyautogui
import serial
```

Then, initialize the serial communication with the port and baud rate (115200) used by the microcontroller:

```
ser = serial.Serial()
ser.port = '/dev/ttyACM0'
ser.baudrate = 115200
```

The serial port name (`ser.port`) is system-dependent. Therefore, ensure you use the same port name reported in the Arduino IDE for the attached microcontroller.

After the initialization of the serial communication, open the serial port and discard the content in the input buffer:

```
ser.open()  
ser.reset_input_buffer()
```

Once the input buffer is cleared, the program can start receiving the data transmitted over the serial.

Step 2:

Open a while loop that runs indefinitely until the program ends. Within the loop statement, read data from the serial port:

```
def serial_readline(obj):  
    data = obj.readline()  
    return data.decode("utf-8").strip()  
  
while True:  
    data_str = serial_readline(ser)
```

If the microcontroller has transmitted the circle string over the serial port, press the *m* key to *mute/unmute*:

```
if str(data_str) == "circle":  
    pyautogui.press('m')
```

Else, if the microcontroller has transmitted the cross string over the serial port, press the *k* key to *play/pause*:

```
if str(data_str) == "cross":  
    pyautogui.press('k')
```

Else, if the microcontroller has transmitted the pan string over the serial port, press the *Shift + N* hot-key to *move to the following video*:

```
if str(data_str) == "pan":  
    pyautogui.hotkey('shift', 'n')
```

Now, we are ready to control YouTube video playback with our gestures.

To begin, launch the Python script while ensuring your Raspberry Pi Pico is running the sketch we developed in the previous recipe.

Then, open YouTube from your web browser, play a video, and have your Raspberry Pi Pico in front of you. Now, whenever you perform any of the three movements learned by the ML model (*circle*, *cross*, or *pan*), you can control the YouTube video playback using gestures!

There's more

In this recipe, we learned how to convert the recognized motion transmitted over the serial port into keyboard inputs to play, pause, mute, unmute, and change YouTube videos.

The application we have built is just one of the numerous possibilities that can be unlocked through gesture recognition. For example, you may use these three gestures to develop a game like rock, paper, scissors, or to launch a specific computer application.

Summary

The recipes presented in this chapter demonstrated how to build an end-to-end gesture recognition application with Edge Impulse and the Raspberry Pi Pico.

Initially, we learned how to connect an IMU sensor with the microcontroller using the I2C communication protocol and leverage the Mbed OS API to read the accelerometer measurements.

Afterward, we delved into the dataset preparation, model design, and model training using Edge Impulse. Here, we introduced the spectral feature ingredient necessary to obtain a compact model for gesture recognition. Then, we discovered how to implement a multithreading program using Arm Mbed OS to run the model inference concurrently with the accelerometer data acquisition.

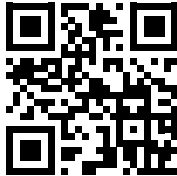
Finally, we concluded the project by developing a Python script that leverages the PyAutoGUI library to simulate keyboard input, thereby facilitating the control of YouTube playback.

In this chapter, we began discussing how to build compact ML models for microcontrollers. With the next project, we will raise the bar to design an image recognition model that can run successfully on a microcontroller with only 64 KB of SRAM.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



10

Deploying a CIFAR-10 Model for Memory-Constrained Devices with the Zephyr OS on QEMU

Prototyping a tinyML application on a physical device is really fun because we can instantly transform our ideas into something that looks and feels like a real thing. However, before any application comes to life, we must ensure that the models work as expected and, possibly, on different devices. Testing and debugging applications directly on microcontroller boards often require a lot of development time. The main reason for this is the necessity to upload a program onto a device for every code change. However, virtual platforms can come in handy to make testing more straightforward and faster.

In this chapter, we will build an image classification application with **TensorFlow Lite for Microcontrollers (tflite-micro)** for an emulated Arm Cortex-M3 microcontroller. To accomplish our task, we will start by installing the **Zephyr OS**, the primary framework used in this chapter. Next, we will design a tiny quantized **CIFAR-10** model with **TensorFlow**. This model will be capable of running on a microcontroller with only 256 KB of program memory and 64 KB of RAM. Ultimately, we will deploy an image classification application on an emulated Arm Cortex-M3 microcontroller through **Quick Emulator (QEMU)**.

The aim of this chapter is to learn how to build and run a TensorFlow-based application with the Zephyr OS on a virtual platform and provide practical advice on designing an image classification model for memory-constrained microcontrollers.

In this chapter, we're going to cover the following recipes:

- Getting started with the Zephyr OS
- Designing and training a CIFAR-10 model for memory-constrained devices
- Evaluating the accuracy of the quantized model
- Converting a NumPy image into a C-byte array
- Preparing the Zephyr Project structure
- Deploying the TensorFlow Lite for Microcontrollers application on QEMU

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- A laptop/PC with either Linux or macOS



The source code and additional material are available in the Chapter10 folder of the GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Chapter10

Getting started with the Zephyr OS

In this recipe, we will install the Zephyr Project, the framework that will allow us to build and run the TensorFlow application on the emulated Arm Cortex-M3 microcontroller. By the end of this recipe, we will have the environment ready on our laptop/PC to run a sample application on the virtual platform considered for our project.

Getting ready

Zephyr (<https://zephyrproject.org/>) is an open-source Apache 2.0 project that provides a small-footprint **Real-Time Operating System (RTOS)** for various hardware platforms based on multiple architectures, including Arm Cortex-M, Intel x86, ARC, Nios II, and RISC-V. The RTOS has been designed for memory-constrained devices with security in mind.

Zephyr does not provide just an RTOS, though. It also offers a **Software Development Kit (SDK)** with a collection of ready-to-use examples and tools to build Zephyr-based applications for a wide range of devices, including virtual platforms through QEMU.

QEMU (<https://www.qemu.org/>) is an open-source machine emulator that allows us to test programs without using real hardware.

The Zephyr SDK supports two QEMU Arm Cortex-M-based microcontrollers, which are as follows:

- The *BBC micro:bit v1* with the Arm Cortex-M0: https://docs.zephyrproject.org/2.7.5/boards/arm/qemu_cortex_m0/doc/index.html
- *Texas Instruments' LM3S6965* with the Arm Cortex-M3: https://docs.zephyrproject.org/2.7.5/boards/arm/qemu_cortex_m3/doc/index.html

From the preceding two QEMU platforms, we will use the LM3S6965. The primary reason for choosing this platform is the bigger RAM capacity than the BBC micro:bit. In fact, although the devices have the same program memory size (256 KB), LM3S6965 has 64 KB of RAM. Unfortunately, the BBC micro:bit v1 has only 16 KB of RAM – not enough for running the model we aim to deploy in this chapter.

How to do it...

The Zephyr installation consists of the following steps:

1. Installing Zephyr prerequisites
2. Getting Zephyr source code and related Python dependencies
3. Installing the Zephyr SDK

Before you start, make sure you have installed the **Python Virtual Environment (virtualenv)** tool to create an isolated Python environment.

Then, take the following steps to prepare the Zephyr environment and run a simple application on the virtual Arm Cortex-M3-based microcontroller:

Step 1:

Follow the instructions in the Zephyr *Getting Started Guide* (https://docs.zephyrproject.org/2.7.5/getting_started/index.html) up to and including the **Install a Toolchain** section.

After the Zephyr installation, the Zephyr modules will be available in the `~/zephyrproject` directory.

Step 2:

Check out the Zephyr 2.7.5 **long-term support (LTS)** release, recommended for reproducing the steps in this chapter:

```
$ cd ~/zephyrproject/zephyr
```

```
$ git checkout v2.7.5
$ west update
```

Step 3:

Navigate into the Zephyr source code directory and enter the `samples/synchronization` folder:

```
$ cd ~/zephyrproject/zephyr/samples/synchronization
```

Zephyr provides ready-to-use applications in the `samples/` folder to demonstrate the usage of RTOS features. Since our goal is to run an application on a virtual platform, we consider the synchronization sample because it does not require interfacing with external components (for example, LEDs).

Step 4:

Build the pre-built synchronization sample for `qemu_cortex_m3` using the following command with the `west` tool:

```
$ west build -b qemu_cortex_m3 .
```

The `west` tool, documented in the Zephyr Project's guide (<https://docs.zephyrproject.org/2.7.5/guides/west/index.html>), is a versatile tool developed by Zephyr to manage multiple repositories with a few command lines. However, `west` is more than a repository manager. In fact, this tool can also plug in additional functionalities through extensions. Zephyr uses this pluggable mechanism for compiling, flashing, and debugging applications.



You can learn more about the `west` tool at the following link: <https://docs.zephyrproject.org/2.7.5/guides/west/build-flash-debug.html>.

The syntax for using the `west` command to compile the application is as follows:

```
$ west build -b <BOARD> <EXAMPLE-TO-BUILD>
```

Let's break down the preceding command:

- `<BOARD>`: This is the name of the target platform. In our case, it is the QEMU Arm Cortex-M3 platform (`qemu_cortex_m3`).
- `<EXAMPLE-TO-BUILD>`: This is the path to the sample test to compile.

Once the application has been successfully built, we can run it on the target device.

Step 5:

Run the synchronization example on the LM3S6965 virtual platform with the following `west` command:

```
$ west build -t run
```

To run the application, we just need to use the `west build` command, followed by the build system target (`-t`) as a command-line argument. Since we specified the target platform when we built the application, we can simply pass the `run` option to upload and run the program on the device.

If Zephyr is installed correctly, the synchronization sample will run on the virtual Arm Cortex-M3 platform and print the following output:

```
thread_a: Hello World from cpu 0 on qemu_cortex_m3!
thread_b: Hello World from cpu 0 on qemu_cortex_m3!
thread_a: Hello World from cpu 0 on qemu_cortex_m3!
thread_b: Hello World from cpu 0 on qemu_cortex_m3!
```

You can now close QEMU by pressing `Ctrl + A`.

There's more...

In this recipe, we learned how to run an application on a virtual platform through QEMU using Zephyr.

As mentioned in the *Getting ready* section, the Zephyr Project also provides a virtual platform for the BBC micro:bit v1. Therefore, you may consider testing the application on this device by specifying `qemu_cortex_m0` as the target platform when compiling the synchronization example using the `west` command:

```
$ west build -b qemu_cortex_m0 . --pristine
```

In the preceding command, we employed the `--pristine` option to remove previous builds from the directory since we use the same output directory (`.`) of the `qemu_cortex_m3` device. Once you have built the application, you can run it with the command you previously used for the `qemu_cortex_m3` virtual platform.

Having installed the Zephyr Project, we can start digging into the ML model design.

In the upcoming recipe, we will discover how to design a CIFAR-10 model tailored for memory-constrained devices.

Designing and training a CIFAR-10 model for memory-constrained devices

The tight memory constraint on LM3S6965 forces us to develop a model with extremely low memory utilization. In fact, the target microcontroller has four times less memory capacity than the Arduino Nano.

Despite this challenging constraint, in this recipe, we will demonstrate the effective deployment of CIFAR-10 image classification on this microcontroller by employing the following **convolutional neural network (CNN)** with TensorFlow:

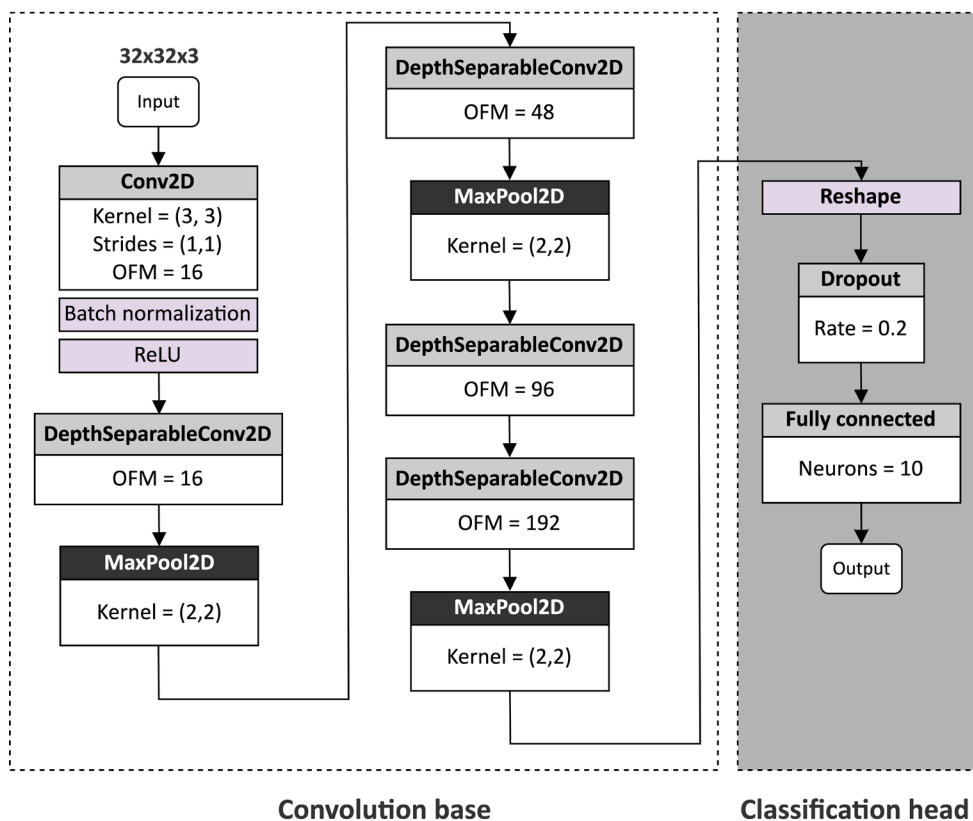


Figure 10.1: The model tailored for CIFAR-10 dataset image classification

As you can see from the preceding model architecture, the depthwise separable convolution (**DepthSeparableConv2D**) layer is the leading operator of the model. This operator, discussed in the upcoming *Getting ready* section, will make the model compact and accurate.

Getting ready

The network tailored in this recipe takes inspiration from the success of the MobileNet v1 model (<https://arxiv.org/abs/1704.04861>) on the **ImageNet** dataset classification and aims to classify the 10 classes of the CIFAR-10 dataset: *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, and *truck*.



The CIFAR-10 dataset is available at <https://www.cs.toronto.edu/~kriz/cifar.html> and consists of 60,000 RGB images with 32x32 resolution.

To understand why the proposed model can run successfully on LM3S6965, we want to outline the architectural design choices that make this network suitable for our target device.

As shown in the model architecture illustrated in *Figure 10.1*, the model has a **convolution base**, which acts as a feature extractor, and a **classification head**, which takes the learned features to perform the classification.

Early layers have large spatial dimensions and produce few **output feature maps (OFMs)** to learn simple features (for example, simple lines). Deeper layers, instead, have small spatial dimensions and generate more OFMs to learn complex features (for example, shapes).



In the context of convolutional neural networks, the feature maps refer to the number of output images generated by the convolution layer.

The model uses pooling layers to halve the spatial dimensionality of the tensors and reduce the risk of overfitting when increasing the OFMs. Generally, we want several feature maps for deep layers to combine as many complex features as possible. Therefore, the idea is to get smaller spatial dimensions to afford more OFMs.

In the following subsection, we will explain the primary operator that helps us to reduce the model size significantly: the **depthwise separable convolution (DWSC)** layer.

Replacing convolution with depthwise separable convolution

DWSC is the layer that made the MobileNet v1 model successful on the ImageNet dataset and the heart of our proposed convolution-based architecture.

This operator took the lead in MobileNet v1 to produce an accurate model that can run on a device with limited memory and computational resources.

The DWSC operator comprises a depthwise convolution, followed by a convolution layer with a 1x1 kernel size, also known as **pointwise convolution**:

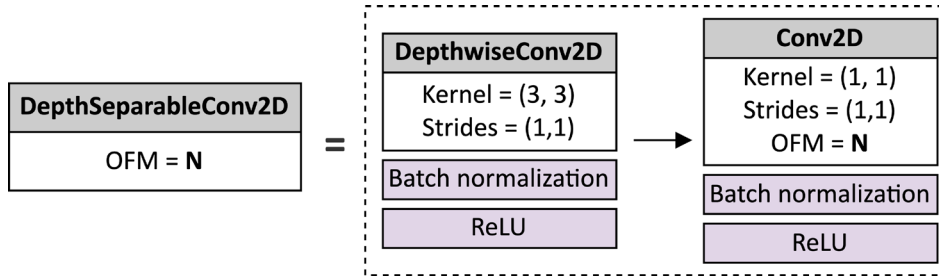


Figure 10.2: The depthwise separable convolution

To demonstrate the efficiency of this operator in contrast to the standard convolution operation, let's consider the first use of DWSC in the network we intend to deploy. As shown in the following diagram, the input tensor has a size of 32x32x16, while the output tensor has a size of 32x32x16:

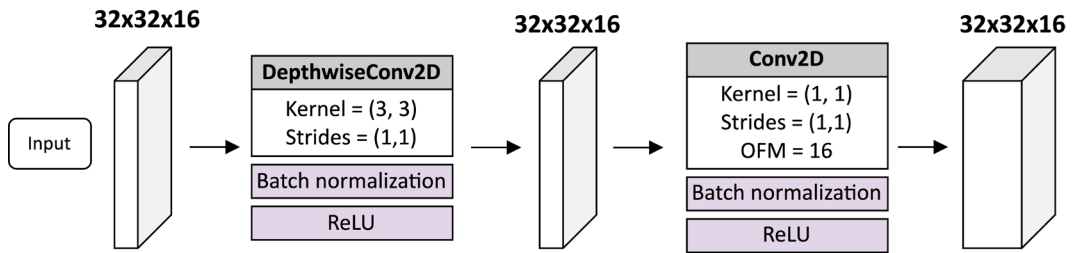


Figure 10.3: The first use of DWSC in our model

In depthwise convolution (**DepthwiseConv2d**), each input channel is convolved independently with its corresponding filter, resulting in a set of feature maps that matches the number of channels in the input (16). This algorithm differs from standard convolution, where all input channels are convolved using a shared filter set. Therefore, the preceding DWSC layer needs 560 trainable parameters distributed as follows:

- 144 weights and 16 biases for the depthwise convolution layer with a 3x3 filter size (**DepthwiseConv2D**)
- 256 weights and 16 biases for the pointwise convolution (**Conv2D**)

If we replaced the DWSC with a regular convolution with a 3x3 filter, we would need 3,480 trainable parameters. Among these parameters, 2,304 are weights (3x3x16x16), and 16 are biases.

As a result, in this particular case, the DWSC layer yields roughly five times fewer trainable parameters than a regular convolution 2D layer.

The reduction in model size is not the only benefit this layer offers. The other advantage of using the DWSC is the reduction of the arithmetic operations. While both layers involve **Multiply-Accumulate (MAC)** operations, the DWSC requires significantly fewer MAC operations than 2D convolution. This fact can be demonstrated by examining the equations used to calculate the MAC operations required for standard convolution and the DWSC:

$$MACs_{conv2d} = F_{size} \cdot W_{out} \cdot H_{out} \cdot C_{out} \cdot C_{in}$$

$$MACs_{DWSC} = (F_{size} \cdot W_{out} \cdot H_{out} \cdot C_{in}) + (W_{out} \cdot H_{out} \cdot C_{out} \cdot C_{in})$$

The terms in the preceding two formulas are as follows:

- $MACs_{conv2d}$: The MAC operations required by the standard convolution
- $MACs_{DWSC}$: The MAC operations required by DWSC
- F_{size} : The filter size
- W_{out}, H_{out} : The output spatial dimensions (width and height)
- C_{in}, C_{out} : The number of input and output feature maps

By observing the equation used to determine the MAC operations for the DWSC, it becomes evident that it comprises two distinct components:

- $(F_{size} \cdot W_{out} \cdot H_{out} \cdot C_{in})$: The MAC operations required by the depthwise convolution
- $(W_{out} \cdot H_{out} \cdot C_{out} \cdot C_{in})$: The MAC operations required by the pointwise convolution

When applying the formulas to the scenario depicted in *Figure 10.3*, we find that standard convolution requires 2,359,296 MACs, whereas DWSC only necessitates 409,600. Therefore, *there is a computational complexity reduction of over five times with DWSC.*

Hence, the efficiency of the DWSC layer is twofold since it decreases the trainable parameters and arithmetic operations involved.

Now that we know the benefits of this layer, let's discuss how we can estimate the program and data memory usage for our model.

Keeping the model memory requirement under control

Our goal is to produce a model that can fit in 256 KB of program memory and run with 64 KB of RAM. The program memory usage can be obtained directly from the .tflite model generated. Alternatively, you can check the `Total params` value returned by the `Keras.summary()` method (<https://keras.io/api/models/model/#summary-method>) to gain an estimate of the model's size. The **Total params** value represents the number of trainable parameters affected mainly by the OFM produced by each layer. In our case, the convolution base has five trainable layers with a maximum of 192 feature maps. This choice will make our model utilize at least 30% of the total program memory.

The estimation of the RAM utilization is a bit more complicated, though. Generally, all the non-constant variables, such as the network input, output, and intermediate tensors, stay in RAM. Since the network needs several temporary tensors for the model's inference, tflite-micro integrates a **memory manager** to minimize RAM usage. When a temporary tensor is required for the computation, the tflite-micro runtime takes the chunk of memory from a pre-allocated buffer called a **scratch buffer**. Therefore, the RAM utilization depends mainly on the scratch buffer size, which depends on the model's architecture.

For a sequential model like ours, where each layer consists of one input and one output tensor, a ballpark figure for the RAM usage can be obtained by summing the following:

- The memory required for the model input and output tensors
- The two largest intermediate tensors

As shown in the following figure, the first DWSC layers in our network are responsible for the largest intermediate tensor, which comprises 16,384 elements ($32 \times 32 \times 16$):

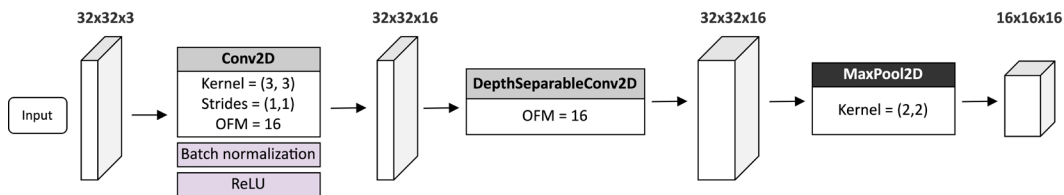


Figure 10.4: The first DWSC produces the biggest intermediate tensor

As shown in Figure 10.4, the first DWSC returns a tensor with 16 OFMs, which we found to be a good compromise between accuracy and RAM utilization. However, you may reduce this further to make the model even smaller and more performant.

How to do it...

Create a new Colab notebook and take the following steps to design and train a quantized CIFAR-10 model with TensorFlow Lite:

Step 1:

Download the CIFAR-10 dataset:

```
from tensorflow.keras import datasets
(train_imgs, trainlbls), (val_imgs, vallbls) = datasets.cifar10.load_data()
```

Step 2:

Normalize the pixel values between 0 and 1 by dividing the pixel values by 255:

```
train_imgs = train_imgs / 255.0
val_imgs = val_imgs / 255.0
```

This step ensures that all data is standardized and within the same scale.

Step 3:

Write a function to implement the DWSC:

```
def separable_conv(i, ch):
    x = layers.DepthwiseConv2D((3,3), padding="same")(i)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    x = layers.Conv2D(ch, (1,1), padding="same")(x)
    x = layers.BatchNormalization()(x)
    return layers.Activation("relu")(x)
```

The `separable_conv()` function accepts the following input arguments:

- `i`: The input to feed to the depthwise convolution
- `ch`: The desired number of OFMs to produce

The batch normalization layer (`layers.BatchNormalization()`) standardizes the input to a layer, resulting in faster and more stable model training.

Step 4:

Design the convolution base:

```
from tensorflow.keras import layers
input = layers.Input((32,32,3))
x = layers.Conv2D(16, (3, 3), padding='same')(input)
x = layers.BatchNormalization()(x)
x = layers.Activation("relu")(x)
x = separable_conv(x, 16)
x = layers.MaxPooling2D((2, 2))(x)
x = separable_conv(x, 48)
x = layers.MaxPooling2D((2, 2))(x)
x = separable_conv(x, 96)
x = separable_conv(x, 192)
x = layers.MaxPooling2D((2, 2))(x)
```

The pooling layers reduce the spatial dimensionality of the feature maps through the network, preventing overfitting. Although we could use DWSC with non-unit strides to accomplish a similar sub-sampling task, we opted for pooling layers to maintain a smaller number of trainable parameters and to enhance model accuracy.

Step 5:

Design the classification head:

```
x = layers.Flatten()(x)
x = layers.Dropout(0.2)(x)
x = layers.Dense(10)(x)
```

Step 6:

Generate the model and print its summary:

```
from keras.models import Model
model = Model(input, x)
model.summary()
```

The model’s summary should report a total of 59,530 trainable parameters, as shown in *Figure 10.5*:

```
=====
Total params: 59530 (232.54 KB)
Trainable params: 58442 (228.29 KB)
Non-trainable params: 1088 (4.25 KB)
```

Figure 10.5: Expected model’s summary

Regarding 8-bit quantization, 59,530 Total params correspond to 59,530 8-bit integer values. Therefore, the weights contribute roughly 60 KB to the model size, well away from the 256 KB maximum target. However, it is essential to note that this value should not be considered the actual model size. The deployed model on a microcontroller consists of the TensorFlow Lite file, which also contains the network architecture and the quantization parameters.

An approximate estimation of the RAM usage can be obtained by examining the tensor sizes of each intermediate tensor within the network. This information can be extracted from the output of the `model.summary()` function. In our network, the first DWSC layer is expected to contribute to the largest tensor, as confirmed by inspecting the tensor shapes reported for each layer in the model’s summary. The provided screenshot, obtained by executing the Colab notebook uploaded on GitHub, displays a portion of the complete model’s summary, specifically showcasing the tensor shapes of the first DWSC layer:

act1 (Activation)	(None, 32, 32, 16)	0	
dwc0_dwsc2 (DepthwiseConv2D)	(None, 32, 32, 16)	160	
bn0_dwsc2 (BatchNormalization)	(None, 32, 32, 16)	64	
act0_dwsc2 (Activation)	(None, 32, 32, 16)	0	
conv0_dwsc2 (Conv2D)	(None, 32, 32, 16)	272	DWSC
bn1_dwsc2 (BatchNormalization)	(None, 32, 32, 16)	64	
act1_dwsc2 (Activation)	(None, 32, 32, 16)	0	
pool1 (MaxPooling2D)	(None, 16, 16, 16)	0	

Figure 10.6: The CIFAR-10 model’s summary (the first DWSC) obtained by executing the Colab notebook uploaded on GitHub

As you can see from the **DWSC** area marked in *Figure 10.6*, the two tensors with the largest number of elements are as follows:

- The output of **act0_dwsc2**: (None, 32, 32, 16)
- The output of **conv0_dwsc2**: (None, 32, 32, 16)

As a result, the expected memory utilization for the intermediate tensor should be approximately 32 KB. To obtain a more accurate estimate of the RAM usage, this value should be combined with the memory required for the input and output tensors. The input and output tensors occupy 3,082 bytes, with 3,072 bytes allocated for the input (32x32x3) and 10 bytes for the output. In total, we can anticipate a RAM usage of roughly 35 KB during model inference, which falls below the target of 64 KB.



The preceding figure has three layers with the (None, 32, 32, 16) output shape: **conv0_dwsc2**, **bn1_dwsc2**, and **act1_dwsc2**. However, only the pointwise convolution layer (**conv0_dwsc2**) counts for memory utilization of the intermediate tensors because batch normalization (**bn1_dwsc2**) and activation (**act1_dwsc2**) will be fused into the convolution (**conv0_dwsc2**) by the TensorFlow Lite converter.

Step 7:

Compile and train the model with 10 epochs and a batch size of 32:

```
import tensorflow as tf
loss_f = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam',
              loss=loss_f,
              metrics=['accuracy'])

history = model.fit(
    train_imgs,
    train_lbls,
    epochs=10,
    batch_size=32,
    validation_data=(val_imgs, val_lbls))
```

After 10 epochs, we have obtained an accuracy of 71.9% on the validation dataset.

Step 8:

Save the TensorFlow model:

```
model.save("cifar10")
```

Our CIFAR-10 model is now ready to be quantized with the TensorFlow Lite converter.

There's more...

In this recipe, we learned how to design a convolutional neural network for memory-constrained devices and estimate its program (Flash) and data (SRAM) memory usage. Automating the process of analyzing tensor shape sizes is crucial, eliminating the need for visual inspection of the model's summary generated by Keras. In Colab's notebook uploaded on GitHub, we have provided the code to display the memory usage for each intermediate tensor. This code can be adapted according to your needs and keep the memory usage always under control during the model design.

Now that we have trained the model, we can proceed with the model quantization using the TensorFlow Lite converter.

In the upcoming recipe, we will quantize the model to 8-bit and assess its accuracy using the validation dataset to ensure that we do not have any accuracy drop.

Evaluating the accuracy of the quantized model

The trained model can classify the 10 classes of CIFAR-10 with an accuracy of 71.9%. However, before deploying the model on a microcontroller, it must be quantized with TensorFlow Lite, which may reduce the accuracy.

In this recipe, we will demonstrate the quantization process and perform an accuracy evaluation on the validation dataset using the TensorFlow Lite Python interpreter. The reason for using the validation rather than the test dataset is to assess how much the 8-bit quantization alters the accuracy observed during model training. Following the accuracy evaluation, we will finalize the recipe by converting the TensorFlow Lite model into a C-byte array.

Getting ready

As we know, the trained model must be converted into a more compact and lightweight representation before being deployed on a resource-constrained device such as a microcontroller.

Quantization is the essential part of this step to make the model small and improve the inference performance. However, post-training quantization may change the model accuracy because of the arithmetic operations at a lower precision. Therefore, it is crucial to check whether the accuracy of the generated .tflite model is within an acceptable range before deploying it into the target device.

Unfortunately, TensorFlow Lite does not provide a Python tool for the model accuracy evaluation. Hence, we will use the TensorFlow Lite Python interpreter to accomplish this task. The interpreter will allow us to feed the input data to the network and read the classification result. The accuracy will be reported as the fraction of samples correctly classified from the test dataset.

How to do it...

Take the following steps to evaluate the accuracy of the quantized CIFAR-10 model on the test dataset:

Step 1:

Take a few hundred samples from the training dataset to calibrate the quantization parameters:

```
cifar_ds = tf.data.Dataset.from_tensor_slices(train_imgs)
def representative_data_gen():
    for i_value in cifar_ds.batch(1).take(100):
        i_value_f32 = tf.dtypes.cast(i_value, tf.float32)
        yield [i_value_f32]
```

The TensorFlow Lite converter will use the representative dataset to estimate the quantization parameters.

Step 2:

Initialize the TensorFlow Lite converter to perform the 8-bit quantization:

```
tfl_conv = tf.lite.TFLiteConverter.from_saved_model("cifar10")
tfl_conv.representative_dataset = tf.lite.
RepresentativeDataset(representative_data_gen)
tfl_conv.optimizations = [tf.lite.Optimize.DEFAULT]
tfl_conv.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
tfl_conv.inference_input_type = tf.int8
tfl_conv.inference_output_type = tf.int8
```

The preceding snippet of code converts the TensorFlow model (cifar10) into a quantized TensorFlow Lite model, enforcing full integer quantization.

Step 3:

Convert the model into the TensorFlow Lite file format:

```
tfl_model = tfl_conv.convert()
```

Step 4:

Estimate the program memory utilization by evaluating the TensorFlow Lite model size:

```
print(len(tfl_model))
```

The expected model size should be approximately 81,208 bytes. Therefore, the model can comfortably fit within the 256 KB program memory constraint.

Step 5:

Evaluate the accuracy of the quantized model using the test dataset. To do so, initialize the TensorFlow Lite Python interpreter and allocate the tensors:

```
tfl_inter = tf.lite.Interpreter(model_content=tfl_model)
tfl_inter.allocate_tensors()
```

After, get the quantization parameters of the input and output nodes:

```
i_details = tfl_inter.get_input_details()[0]
o_details = tfl_inter.get_output_details()[0]
i_quant = i_details["quantization_parameters"]
o_quant = o_details["quantization_parameters"]
i_scale = i_quant['scales'][0]
i_zero_point = i_quant['zero_points'][0]
o_scale = o_quant['scales'][0]
o_zero_point = o_quant['zero_points'][0]
```

Then, write a function called `classify()` to perform the model inference using the Python TensorFlow Lite interpreter:

```
def classify(i_data, o_value):
    input_data = i_value.reshape((1, 32, 32, 3))
    i_value_f32 = tf.dtypes.cast(input_data, tf.float32)
```



```

i_value_f32 = i_value_f32 / i_scale + i_zero_point
i_value_s8 = tf.cast(i_value_f32, dtype=tf.int8)

tfl_inter.set_tensor(i_details["index"], i_value_s8)
tfl_inter.invoke()
o_pred = tfl_inter.get_tensor(o_details["index"])[0]

return (o_pred - o_zero_point) * o_scale

```

After, declare a variable to keep track of the number of samples correctly classified:

```
num_correct_samples = 0
```

Finally, iterate through all the samples in the validation dataset, and for each one, perform the model inference and check whether the prediction matches the expected output class. If so, increment the `num_correct_samples` variable:

```

import numpy as np
for i_value, o_value in zip(val_imgs, val_lbls):
    o_pred_f32 = classify(i_value, o_value)
    if np.argmax(o_pred_f32) == o_value:
        num_correct_samples += 1

```

Step 6:

Print the accuracy of the quantized TensorFlow Lite model:

```

total_samples = len(list(val_imgs))
print("Accuracy:", num_correct_samples/total_samples)

```

After a few minutes, the accuracy result will be printed in the output log.

In our case, we found an accuracy of 71.1%, which is only marginally less than what was achieved with the floating-point model. As such, we can conclude that the quantization does not substantially reduce our model's accuracy.

Step 7:

Convert the TensorFlow Lite model into a C-byte array with the `xxd` tool:

```

open("cifar10.tflite", "wb").write(tfl_model)
!apt-get update && apt-get -qq install xxd

```

```
!xxd -i cifar10.tflite > model.h  
!sed -i 's/unsigned char/const unsigned char/g' model.h  
!sed -i 's/const/alignas(8) const/g' model.h
```

The command generates a C header file containing the TensorFlow Lite model as a constant unsigned char array.



Remember the final two lines where we employ the sed command to make the array containing the model an `alignas(8) const`. These changes guarantee that the model resides in the program memory (Flash) and that the array is aligned to the 8-byte boundary.

You can now download the `model.h` and `cifar10.tflite` files from Colab's left pane. The `cifar10.tflite` file will be needed later in the chapter to identify the operators the model utilizes. This knowledge will help us minimize the program memory consumption of the `tflite-micro` application on the microcontroller.

There's more...

In this recipe, we learned how to use the TensorFlow Lite Python interpreter to carry out the model's inference and assess the accuracy of the quantized model.

Nonetheless, testing the model on images not part of the training set is beneficial. Therefore, consider building a test dataset with images sourced from the web. For instance, you can obtain images from **Pixabay** (<https://pixabay.com/>), a royalty-free photo platform.

In the Colab notebook shared on GitHub, we have provided a code block titled `Testing model's accuracy with unseen data` that evaluates the model's accuracy using image files with the TensorFlow Lite Python interpreter.

The model is now ready to be deployed on the virtual platform. However, we must provide an input image stored in a C-byte array to validate the model inference on the target device.

In the upcoming recipe, we will discover how you can generate this input image using a sample of the validation dataset.

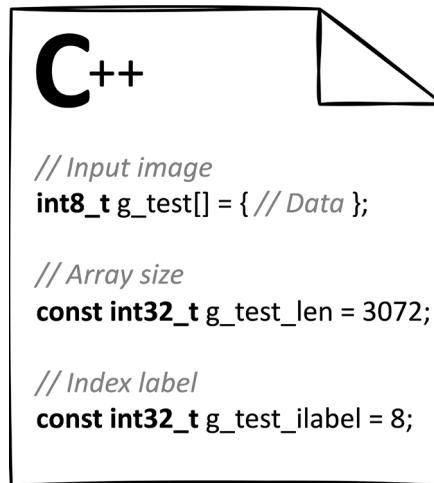
Converting a NumPy image into a C-byte array

Our application will be running on a virtual platform with no access to a camera module. Therefore, we must provide a valid test input image for our application to check whether the model works as expected.

In this recipe, we will get an image from the validation dataset belonging to the **ship** class. The sample will then be converted into an `int8_t` C array and saved as an `input.h` file.

Getting ready

To prepare this recipe, we must know how to structure the C file containing the input test image. The structure of this file is quite simple and illustrated in *Figure 10.7*:



```
C++  
  
// Input image  
int8_t g_test[] = { // Data };  
  
// Array size  
const int32_t g_test_len = 3072;  
  
// Index label  
const int32_t g_test_ilabel = 8;
```

Figure 10.7: The C header file structure for the input test image

As you can observe from the file structure, we only need an array and two variables to describe our input test sample. These variables are as follows:

- `g_test`: An `int8_t` array containing a ship image with the normalized and quantized pixel values. The pixel values (`// Data`) should be stored in the array as comma-separated integer values.
- `g_test_len`: An integer variable for the array size. Since the input model is an RGB image with a 32x32 resolution, we expect an array with 3,072 `int8_t` elements.
- `g_test_ilabel`: An integer variable for the class index of the input test image. Since we have a ship image, the expected class index is 8.

Since the input image will be obtained from the validation dataset, we need to implement a function in Python to convert the image stored in NumPy format into a C array.

How to do it...

Take the following steps to generate a C header file containing an image from the test dataset that can be correctly classified as a ship by the TensorFlow Lite interpreter:

Step 1:

Implement a function to convert a 1D NumPy array of `np.int8` values into a single string of comma-separated integer values:

```
def array_to_str(data):
    NUM_COLS = 12
    val_string = ''
    for i, val in enumerate(data):
        val_string += str(val)
        if (i + 1) < len(data):
            val_string += ','
        if (i + 1) % NUM_COLS == 0:
            val_string += '\n'
    return val_string
```

In the preceding code, the `NUM_COLS` variable limits the number of values on a single row. In our case, `NUM_COLS` is set to 12 to add a newline (`\n`) character after every 12 values.

Step 2:

Implement a function to generate a C header file as a string containing the input test image stored in an `int8_t` array, the array size, and the index class of the test image:

```
def gen_h_file(size, data, ilabel):
    str_out = f'int8_t g_test[] = '
    str_out += "\n{\n"
    str_out += f'{data}'
    str_out += '};\n'
    str_out += f"const int g_test_len = {size};\n"
    str_out += f"const int g_test_ilabel = {ilabel};\n"
    return str_out
```

In the preceding code snippet, the `gen_h_file` function accepts the following input arguments:

- The array size (`size`)
- The data containing the comma-separated integer values representing the pixel data of the image as a string (`data`)
- The index of the class assigned to the input image (`ilabel`)

Step 3:

Retrieve an image from the test dataset that the model can correctly classify as a ship. To do so, create a pandas DataFrame from the CIFAR-10 test dataset:

```
import pandas as pd
imgs = list(zip(val_imgs, val_lbls))
cols = ['Image', 'Label']
df = pd.DataFrame(imgs, columns = cols)
```

Then, obtain a pandas DataFrame with only images belonging to the **ship** class:

```
cond = df['Label'] == 8
ship_samples = df[cond]
```

In the preceding code, 8 is the index for the **ship** class.

Next, iterate over all ship images and run the inference with the TensorFlow Lite Python interpreter:

```
c_code = ""
for index, row in ship_samples.iterrows():
    i_value = np.asarray(row['Image']).tolist()
    o_value = np.asarray(row['Label']).tolist()
    o_pred_f32 = classify(i_value, o_value)
```

Finally, check whether the image has been classified as a **ship**. If so, convert the input image into a C-byte array and exit the loop:

```
if np.argmax(o_pred_f32) == o_value:
    i_value_f32 = i_value / i_scale + i_zero_point
    i_value_s8 = i_value_f32.astype(dtype=np.int8)
    i_value_s8 = i_value_s8.ravel()
```

Generate a string from the NumPy array

```
val_string = array_to_str(i_value_s8)

# Generate the C header file
c_code = gen_h_file(i_value_s8.size,
                   val_string, "8")

break
```

Step 4:

Save the `c_code` string as an `input.h` file:

```
with open("input.h", 'w') as file:
    file.write(c_code)
```

You can download the `input.h` file containing the input test image from Colab's left pane.

There's more...

In this recipe, we learned how to convert an image in a NumPy array into a C-byte array using just a few lines of Python code.

It is worth noting that the input test image stored in the `input.h` file does not include the `const` prefix.

As we know, the `const` prefix is typically utilized to inform the compiler that the content of the array does not change at runtime and can be placed in program memory to reduce the SRAM utilization. However, in the case of the input test image, we need to consider that it is likely to be produced from a live camera. Therefore, its content changes.

Given this consideration, we should not use the `const` prefix for the input test image to ensure a more accurate evaluation of the program (Flash) and data (SRAM) memory usage when testing the application.

After generating the input test image, it is time to prepare the application to run on the virtual platform. However, what do we need to build a `tf-lite-micro` application with the Zephyr OS?

In the upcoming recipe, we will show you how to prepare the application using a pre-built `tf-lite-micro` example in the Zephyr SDK.

Preparing the Zephyr Project structure

Only a few steps are separating us from the completion of this project. Now that we have the model and the input test image, we can leave Colab's environment and focus on the application development with the Zephyr OS.

In this recipe, we will prepare the skeleton of the tflite-micro application from the pre-built `hello_world` sample available in the Zephyr SDK.

Getting ready

The easiest way to start a new tflite-micro project is to copy and edit one of the pre-built samples provided by the Zephyr SDK, available in the `~/zephyrproject/zephyr/samples/modules/tflite-micro/` folder. At the time of writing, there are two ready-to-use examples:

- `hello_world` – A sample showing the basics to replicate a sine function with a TensorFlow Lite model: https://docs.zephyrproject.org/2.7.5/samples/modules/tflite-micro/hello_world/README.html
- `magic_wand` – A sample application showing how to recognize gestures with accelerometer data: https://docs.zephyrproject.org/2.7.5/samples/modules/tflite-micro/magic_wand/README.html

In this recipe, we will base our application on the `hello_world` sample, and the following screenshot shows what you should find in its corresponding directory:

```
(.venv): ls ~/zephyrproject/zephyr/samples/modules/tflite-micro/hello_world
CMakeLists.txt  images  prj.conf  README.rst  sample.yaml  src  train
```

Figure 10.8: Content of the hello_world tflite-micro pre-built sample

Among the three subfolders within the `hello_world/` directory (`images/`, `src/`, and `train/`), we only require the `src/` folder as it houses the application's source code, as shown in the following screenshot:

```
(.venv): ls ~/zephyrproject/zephyr/samples/modules/tflite-micro/hello_world/src
assert.cc  constants.c  constants.h  main.c  main_functions.cc  main_functions.h
model.cc  model.h  output_handler.cc  output_handler.h
```

Figure 10.9: Files in the src/ directory of the hello_world tflite-micro pre-built sample

In the `src/` directory, not all files are relevant to our project. For example, `assert.cc`, `constants.c`, `constants.h`, `model.cc`, `model.h`, `output_handler.cc`, and `output_handler.h` are exclusively necessary for the sine wave sample application. Therefore, the essential C files required for our application are as follows:

- `main.c`: This file contains the standard C/C++ `main()` function responsible for starting and terminating the program execution. The `main()` function comprises a `setup()` function called once and a `loop()` function executed 50 times. As a result, the body of the `main()` function resembles, to a large extent, what we find in an Arduino sketch.
- `main_functions.h` and `main_functions.cc`: These files contain the declaration and definition of the `setup()` and `loop()` functions.

In addition to the C files, we will need the `CMakeList.txt` and `prj.conf` files. These files, located in the `hello_world/` directory, are necessary to compile the application.

How to do it...

Open Terminal and take the following steps to create a new `tf-lite-micro` project with the Zephyr OS:

Step 1:

Navigate into the `~/zephyrproject/zephyr/samples/modules/tf-lite-micro/` directory and create a new folder named `cifar10`:

```
$ cd ~/zephyrproject/zephyr/samples/modules/tf-lite-micro/  
$ mkdir cifar10
```

Step 2:

Copy the content of the `hello_world/` directory into the `cifar10/` directory:

```
$ cp -r hello_world/* cifar10
```

Step 3:

Navigate into the `cifar10/` directory and remove the following files from the `src/` directory – `constants.h`, `constants.c`, `model.c`, `model.h`, `output_handler.cc`, `output_handler.h`, and `assert.cc`:

```
$ cd cifar10/src  
$ rm constants.h constants.c model.cc model.h \  
output_handler.cc output_handler.h assert.cc
```


Step 4:

Copy the `model.h` and `input.h` files generated in the previous two recipes into the `cifar10/src/` folder. Once you have copied the files, the `cifar10/src` folder should contain the following files:

```
(.venv): ls ~/zephyrproject/zephyr/samples/modules/tflite-micro/cifar10/src
input.h  main.c  main_functions.cc  main_functions.h  model.h
```

Figure 10.10: Files in the `src/` directory of the `cifar10 tflite-micro` project

Now, open your default C editor (for example, **Vim**) to make code changes in the `main.c` and `main_functions.cc` files.

Step 5:

Open the `main.c` file and replace `for (int i = 0; i < NUM_LOOPS; i++)` with `while(1)`. After this change, the `main.c` file will look like the following:

```
int main(int argc, char *argv[]) {
    setup();
    while(1) {
        loop();
    }
    return 0;
}
```

This preceding code precisely replicates the behavior of an Arduino sketch, with the `setup()` function being called once and the `loop()` function being repeated indefinitely. Hence, we only need to implement the `setup()` and `loop()` functions in the `main_functions.cc` file similar to what we have done in previous chapters.

Step 6:

Open the `main_functions.cc` file and remove the following:

- `constants.h`, `model.h`, and `output_handler.h` from the list of header files.
- The `inference_count` variable and all its usages. This variable will not be required in our application.
- The code within the `loop()` function.

The project structure is now ready, and we can finally proceed to the following recipe to build our application.

There's more...

In this recipe, we learned how to leverage the pre-built tflite-micro sample provided by the Zephyr SDK to prepare the project structure for our application.

Although there may be similarities in terms of syntax, the application we are implementing is distinct from an Arduino sketch. The main difference lies in the underlying RTOS, which offers its functions and mechanisms to interact with microcontroller peripherals that may differ from those commonly employed in Arduino sketches. To learn more about the foundational building blocks provided by the Zephyr OS, we recommend referring to the GitHub repository of the Zephyr Project (<https://github.com/zephyrproject-rtos/zephyr>). There, you can find the documentation and examples to familiarize yourself with the unique features and capabilities of the Zephyr OS.

Having prepared the project structure for our tflite-micro application, we are set to complete our project.

In the upcoming recipe, we will finalize the application based on the Zephyr OS to deploy the trained CIFAR-10 model on the virtual Arm-based microcontroller.

Deploying the TensorFlow Lite for Microcontrollers program on QEMU

The skeleton of our Zephyr Project is ready, so we just need to finalize our application to classify our input test image.

In this recipe, we will see how to implement the tflite-micro application and run the model on the emulated Arm Cortex-M3-based microcontroller.

Getting ready

Most of the ingredients required for this recipe are related to tflite-micro and have already been covered in earlier chapters, such as *Chapter 3, Building a Weather Station with TensorFlow Lite for Microcontrollers*. Nevertheless, there is one useful feature of this framework that has not been introduced yet, but it can come in handy when we want to reduce the program memory usage of our application drastically. This feature is the `tflite::MicroMutableOpResolver`, which *enables the compilation of a subset of all operators available in tflite-micro*.

As shown in the following ML model example, the model is composed of a series of operations that define the computation to be performed:

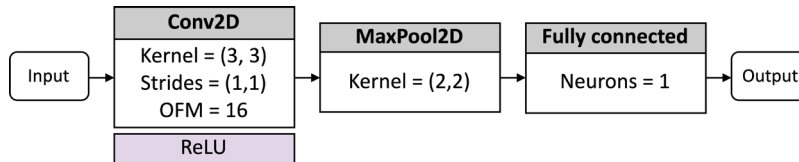


Figure 10.11: An example of an ML model

Each operation reported in the previous figure, including convolution (**Conv2D**), activation (**ReLU**), max pooling (**MaxPool2D**), and fully connected (**Fully connected**), is implemented as a function in TensorFlow Lite.

When we initialize the TensorFlow Lite interpreter with the model to run, the interpreter must know each operator's function pointer. Until now, we have been using the `tflite::AllOpsResolver` interface, which provides this information for all operators supported in `tflite-micro`. However, this interface might demand significant program memory, making deploying the application on memory-constrained devices challenging.

Thankfully, `tflite-micro` offers an alternative and more efficient interface to load only the operators required by the model. This interface is `tflite::MicroMutableOpResolver`, where its use is shown in the following code snippet, which loads only the operators needed for the model illustrated in *Figure 10.11*:

```
static tflite::MicroMutableOpResolver resolver;

// Add the operators to compile here
resolver.AddConv2D();
resolver.AddRelu();
resolver.AddMaxPool2D();
resolver.AddFullyConnected();
```

To determine which operators your model requires, you can visualize the graph representation of the TensorFlow Lite model stored in the `.tflite` file using the **Netron** web application (<https://netron.app/>).

How to do it...

Let's start this recipe by visualizing the architecture of our TensorFlow Lite CIFAR-10 model file (`cifar10.tflite`) with Netron to determine the operators required by your model.

Figure 10.12 shows a slice of our model visualized with this tool:

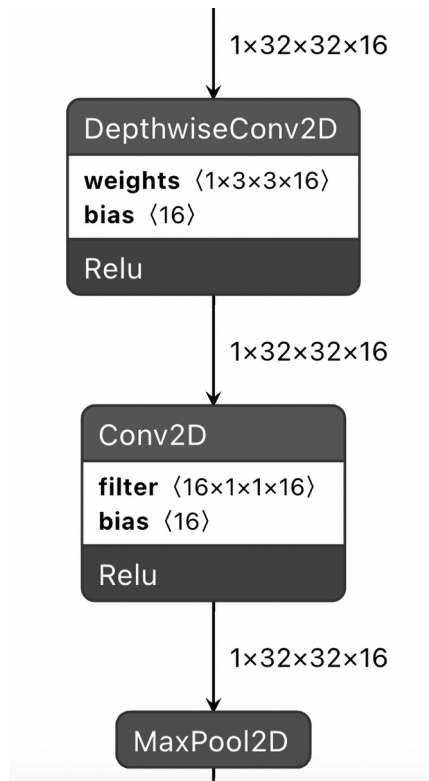


Figure 10.12: A slice of the CIFAR-10 model visualized with the Netron tool (courtesy of netron.app)

By inspecting the model with Netron, we can see that the model only uses six operators: **Conv2D**, **Relu**, **DepthwiseConv2D**, **MaxPool2D**, **Reshape**, and **FullyConnected**.

Now, open your default C editor and then open the `main_functions.cc` file and follow these steps to build the `tf-lite-micro` application:

Step 1:

Use the `#include` directive to add the header files with the input test image (`input.h`) and the TensorFlow Lite model (`model.h`):

```
#include "input.h"
#include "model.h"
```

Step 2:

Increase the arena size (tensor_arena_size) to 44,000:

```
constexpr int tensor_arena_size = 44000;
```



The original variable name for the tensor arena is kTensorArenaSize. To keep consistency with the lower_case naming convention used in the book, we have renamed this variable to tensor_arena_size.

The tflite-micro tensor arena is the portion of memory allocated by the user to accommodate the network input, output, intermediate tensors, and other data structures required by the framework. As we have seen from the design of the CIFAR-10 model, the estimated RAM usage for the model inference is in the order of 44 KB. Hence, we adjust the tensor arena size to match this figure.

Step 3:

Declare two global variables for the output quantization parameters:

```
float o_scale = 0.0f;
int32_t o_zero_point = 0;
```

As the input test image is already quantized, there is no need for the input quantization parameters.

Step 4:

In the setup() function, replace g_model with the corresponding array name from model.h in the tflite::GetModel() function. For example, if the array is named cifar10_tflite, the updated code would be as follows:

```
model = tflite::GetModel(cifar10_tflite);
```

Step 5:

In the setup() function, replace the instantiation of tflite::AllOpsResolver with tflite::MicroMutableOpResolver<6>. Then, add the operators used by the model into the tflite::MicroMutableOpResolver object before the initialization of the TensorFlow Lite interpreter:

```
static tflite::MicroMutableOpResolver<6> resolver;
resolver.AddConv2D();
resolver.AddRelu();
```

```

resolver.AddDepthwiseConv2D();
resolver.AddMaxPool2D();
resolver.AddReshape();
resolver.AddFullyConnected();
static tf::MicroInterpreter static_interpreter(
    model,
    resolver,
    tensor_arena,
    tensor_arena_size,
    error_reporter);
interpreter = &static_interpreter;

```

Step 6:

In the `setup()` function, and after obtaining the model's input and output tensors, get the output quantization parameters from the output tensor:

```

const auto* o_quantization = reinterpret_cast<TfLiteAffineQuantization*>
(output->quantization.params);
o_scale      = o_quantization->scale->data[0];
o_zero_point = o_quantization->zero_point->data[0];

```

Step 7:

In the `loop()` function, initialize the input tensor with the content of the input test image and run the inference:

```

for(int32_t i = 0; i < g_test_len; i++) {
    input->data.int8[i] = g_test[i];
}

interpreter->Invoke();

```

Step 8:

After the model inference, return the output class with the highest score:

```

int32_t ix_max = 0;
float   pb_max = 0;

for(int32_t ix = 0; ix < 10; ++ix) {

```

```

    int8_t out_val = output->data.int8[ix];
    float pb = ((float)out_val - o_zero_point);
    pb *= o_scale;

    if(pb > pb_max) {
        ix_max = ix;
        pb_max = pb;
    }
}

```

The preceding code iterates over the quantized output values and returns the class index with the highest score in the `ix_max` variable.

Step 9:

Check whether the classification result (`ix_max`) equals the label index assigned to the input test image (`g_test_label`). If so, send the `PASSED!` string over the serial with the `TF_LITE_REPORT_ERROR()` function:

```

if(ix_max == g_test_label) {
    TF_LITE_REPORT_ERROR(error_reporter, "PASSED!\n");
}

```

Otherwise, send the `FAILED!` string over the serial with the `TF_LITE_REPORT_ERROR()` function:

```

else {
    TF_LITE_REPORT_ERROR(error_reporter, "FAILED!\n");
}

```

Finalize the `loop()` function by adding an infinite `while()` loop to avoid repeating the model inference:

```

    while(1);
} // loop()

```

Step 10:

Open Terminal, enter the `cifar10/` folder in the Zephyr SDK, and use the `west` tool to build the project for `qemu_cortex_m3`:

```

$ cd ~/zephyrproject/zephyr/samples/modules/tflite-micro/cifar10
$ west build -b qemu_cortex_m3 .

```

After a few seconds, the west tool should display the program memory usage in Terminal, confirming that the program has been successfully compiled:

Memory region	Used Size	Region Size	%age Used
FLASH:	138588 B	256 KB	52.87%
SRAM:	51640 B	64 KB	78.80%
IDT_LIST:	0 GB	2 KB	0.00%

Figure 10.13: The program memory usage summary

From the summary generated by the west tool, you can see that our CIFAR-10-based application uses **52.87%** of program memory (**FLASH**) and **78.80%** of data memory (**SRAM**). Based on this summary, it is evident that only a small amount of data memory remains available.

Step 11:

Run the application on the virtual platform with the west tool:

```
$ west build -t run
```

The west tool will boot the virtual device and return the following output, confirming that the model correctly classified the input image:

```
(.venv): west build -t run
-- west build: running target run
[0/1] To exit from QEMU enter: 'CTRL+a, x' [QEMU] CPU: cortex-m3
qemu-system-arm: warning: nic stellaris_enet.0 has no peer
Timer with period zero, disabling
*** Booting Zephyr OS build v2.7.5 ***
PASSED
```

Figure 10.14: The expected output after the model inference

As you can see from the preceding screenshot, the virtual device returns the **PASSED** message on the output log, confirming the successful execution of our tiny CIFAR-10 model!

There's more...

In this final recipe, we learned how to compile and run a tflite-micro application using the Zephyr SDK.

The `tflite::MicroMutableOpResolver` interface was essential to reduce the program memory utilization, allowing us to deploy the model on the virtual Arm Cortex-M3 device. Without the strategic use of the `tflite::MicroMutableOpResolver` interface, the memory requirements of our application would exceed what is available on the virtual device.

An important note is that Zephyr is versatile in its device support. The application, tailored for the virtual platform, can be compiled for other devices supported by Zephyr. For example, by replacing `qemu_cortex_m3` with `arduino_nano_33_ble` in the `west build` command, you can compile the Zephyr Project for the Arduino Nano. To learn how to upload the program on the microcontroller, you can refer to the official Zephyr documentation for the Arduino Nano 33 BLE sense board, available at the following link: https://docs.zephyrproject.org/2.7.5/boards/arm/arduino_nano_33_ble/doc/index.html.

Summary

In this chapter, our focus has been on tailoring an ML model for image classification on a memory-constrained device with just 64 KB of SRAM.

In the first part, we prepared the Zephyr development environment by installing the components required to build and run a Zephyr application on virtual devices with QEMU.

Following the Zephyr installation, our attention shifted to model design. Here, we designed and trained a CNN based on the depthwise separable convolution layer, allowing us to reduce the training parameters and computational demand drastically.

Once the model was trained, we quantized it to 8-bit using the TensorFlow Lite converter and assessed its accuracy on the validation dataset. The evaluation proved that quantizing to 8-bit only marginally reduces the model's accuracy.

Finally, we developed the Zephyr application to deploy the TensorFlow Lite quantized model and run it on the virtual device.

TensorFlow Lite for Microcontrollers has been the backbone of our projects up to this point. However, the open-source community provides alternative solutions for model deployment on memory-constrained devices, some with very peculiar features.

In the upcoming chapter, we will explore a framework gaining popularity in the field of ML model inference at the edge because of its unique capability to convert ML models into code via compiler technology: Apache TVM.

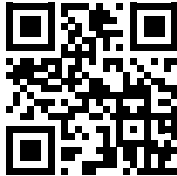
References

- The images in this chapter are from *Learning Multiple Layers of Features from Tiny Images*, Alex Krizhevsky, 2009: <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>. They are part of the CIFAR-10 dataset (toronto.edu): <https://www.cs.toronto.edu/~kriz/cifar.html>.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



11

Running ML Models on Arduino and the Arm Ethos-U55 microNPU Using Apache TVM

In all our projects developed so far, we have relied on **TensorFlow Lite for Microcontrollers** (**tf-lite-micro**) as a software stack to deploy **machine learning (ML)** models on the Arduino Nano, Raspberry Pi Pico, and SparkFun Artemis Nano. However, other frameworks are available within the open-source community for this scope. Among these alternatives, **Apache TVM** (or simply **TVM**) has gained considerable attraction due to its ability to generate optimized code tailored to the desired target platform.

In this chapter, we will explore how to leverage this technology to deploy a quantized **CIFAR-10** TensorFlow Lite model in various scenarios.

The chapter will start by giving an overview of **Arduino Command Line Interface (CLI)**, an indispensable tool to compile and run the code generated by TVM on any Arduino-compatible platform.

After introducing Arduino CLI, we will present TVM by showing how to generate C code from an ML model and how to run it on a machine hosting the Colab environment. In this part, we will also discuss the **ahead-of-time (AoT)** executor, a crucial feature of TVM that can help reduce the program memory usage of the final application.

Then, we will delve into running the model on the Arduino Nano and Raspberry Pi Pico, and we will discuss how to build and compile a sketch from the code generated by TVM.

Finally, we will explore the model deployment on a new, advanced processor designed to extend the computational power and energy efficiency of ML workloads on microcontrollers. This processor is the **Micro-Neural Processing Unit (microNPU)**, which promises to unlock new use cases and make the “things” around us even more intelligent.

Specifically, we will uncover how to run the quantized CIFAR-10 model on a virtual **Arm Ethos-U55** microNPU with the assistance of the **Arm Corstone-300 Fixed Virtual Platform (Corstone-300 FVP)**.

The purpose of this chapter is to get familiar with the model’s deployment through the TVM framework and learn how to leverage the computation on the Arm Ethos-U55 microNPU.

In this chapter, we’re going to cover the following recipes:

- Getting familiar with Arduino CLI
- Downloading a pre-trained CIFAR-10 model and input test image
- Deploying the model with TVM using the AoT executor on the host machine
- Deploying the model on the Arduino Nano
- Deploying the model on the Raspberry Pi Pico
- Installing the FVP for the Arm Corstone-300
- Code generation with TVMC for Arm Ethos-U55
- Installing the software dependencies to build an application for the Arm Ethos-U microNPU
- Running the CIFAR-10 model inference on the Arm Ethos-U55 microNPU



Since TVM is under heavy development, it is strongly advised to use the specific version reported for each software package installed.

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- An Arduino Nano 33 BLE Sense
- A Raspberry Pi Pico
- A SparkFun RedBoard Artemis Nano (optional)
- A micro-USB data cable

- A USB-C data cable (optional)
- A laptop/PC with either Linux, macOS, or Windows
- A Google account



The source code and additional material are available in the Chapter11 folder in the GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Chapter11.

Getting familiar with Arduino CLI

In this first recipe, we will install Arduino CLI on our local machine, to be ready to compile and run the code generated by TVM on any Arduino-compatible microcontroller. This step is essential to learn how to use this tool to create sketches for Arduino Nano and Raspberry Pi Pico.

Getting ready

Arduino CLI (<https://arduino.github.io/arduino-cli>) is a software package that exposes almost all functionalities bundled in the Arduino IDE into a CLI tool. Therefore, using shell commands, you can use Arduino CLI to create new sketches, download libraries, compile and upload programs on Arduino-compatible microcontrollers, and much more!



Arduino CLI is also available as a Python package (**pyduinocli**), a wrapper library around Arduino CLI. You can discover more about the Python Arduino CLI at the following link: <https://pypi.org/project/pyduinocli/>.

To install Arduino CLI, you can follow the instructions reported at the following link: <https://arduino.github.io/arduino-cli/latest/installation/>.

How to do it...

Open your local virtual Python environment and follow the steps to install TVM, Arduino CLI, and build a simple sketch from the CLI:

Step 1:

Install Arduino CLI following the instructions in the official Arduino documentation: <https://arduino.github.io/arduino-cli/0.34/installation/>.

If you are working on a Linux environment, you can download Arduino CLI 0.34.0 using the following command:

```
$ curl -fsSL https://raw.githubusercontent.com/arduino/arduino-cli/0.34.0/install.sh | sh
```

The preceding command will install the Arduino CLI binaries on your local system.

If the output log displays the message *arduino-cli not found. You might want to add "/content/bin" to your \$PATH*, you can include the path reported in the message (in this case, /content/bin) to the \$PATH environment variable using the following command:

```
$ export PATH=/content/bin:$PATH
```

To verify the successful installation of Arduino CLI, you can enter the following command:

```
$ arduino-cli version
```

The previous command will display the version of Arduino CLI installed on your system.

Step 2:

Use Arduino CLI to install the software packages to build sketches for the Arduino Nano and Raspberry Pi Pico, with Mbed OS support.

To do so, update the index of the **Arduino core** to the latest version:

```
$ arduino-cli core update-index
```

The Arduino core is the set of libraries, tools, and configuration files to build and upload sketches to the board. Each Arduino-compatible device has its own core.

After updating the index, install the Arduino core for the Arduino Nano with Mbed OS support:

```
$ arduino-cli core install arduino:mbed_nano
```

Then, install the Arduino core for the Raspberry Pi Pico with Mbed OS support:

```
$ arduino-cli core install arduino:mbed_rp2040
```

Finally, list the Arduino cores installed with the following command:

```
$ arduino-cli core list
```

The expected output should be as follows:

ID	Installed	Latest	Name
arduino:mbed_nano	4.0.2	4.0.2	Arduino Mbed OS Nano Boards
arduino:mbed_rp2040	4.0.2	4.0.2	Arduino Mbed OS RP2040 Boards

Figure 11.1: Expected Arduino cores installed

The previous screenshot confirms that the Arduino cores for the Arduino Nano and Raspberry Pi Pico have been installed.

Step 3:

Create a new sketch called `blink` with Arduino CLI:

```
$ arduino-cli sketch new blink
```

After running the previous command, a new file named `blink.ino`, containing the standard Arduino `setup()` and `loop()` functions, will be created in a new folder called `blink`. The path to the `blink/` folder will be reported on the output log after the command execution.

Step 4:

Enter the `blink/` folder created in the previous step, and open the `blink.ino` file with your favorite IDE. Then, implement the sketch to blink the built-in LED on the Arduino Nano and Raspberry Pi Pico, using the Mbed OS API:

```
#include "mbed.h"

mbed::DigitalOut led(LED1);

void setup() {
}

void loop() {
    led = 1;
    delay(1000);
    led = 0;
    delay(1000);
}
```


In the preceding code block, the LED1 definition refers to the PinName of the built-in LED on the Arduino Nano and Raspberry Pi Pico.

Step 5:

Compile the sketch for the Arduino Nano:

```
$ arduino-cli compile \  
-b arduino:mbed_nano:nano33ble \  
blink
```

Then, compile the sketch for the Raspberry Pi Pico:

```
$ arduino-cli compile \  
-b arduino:mbed_rp2040:pico \  
blink
```

The commands previously used to compile the sketch for Arduino Nano and Raspberry Pi Pico have the following structure:

```
$ arduino-cli compile -b <FQBN> <sketch_name>
```

The `-b` flag is used to specify the target Arduino **Fully Qualified Board Name (FQBN)**. Please note that the Arduino board does not correspond to the core name. The reason is that a single Arduino core may support multiple devices. To list the names of the Arduino boards installed in Arduino CLI, you may consider using the following command:

```
$ arduino-cli board listall
```

In our case, the preceding command should return the following output:

Board Name	FQBN
Arduino Nano 33 BLE	arduino:mbed_nano:nano33ble
Arduino Nano RP2040 Connect	arduino:mbed_nano:nanorp2040connect
Raspberry Pi Pico	arduino:mbed_rp2040:pico

Figure 11.2: List of Arduino boards installed

As you can see from *Figure 11.2*, the **FQBN** for Arduino Nano (**Arduino Nano 33 BLE**) is `arduino:mbed_nano:nano33ble`, while for the Raspberry Pi Pico, the name is `arduino:mbed_rp2040:pico`.

Step 6:

Connect either the Arduino Nano or Raspberry Pi Pico to your laptop/PC. Then, use Arduino CLI to determine the port name of the connected device:

```
$ arduino-cli board list
```

The preceding command will return an output similar to what is shown in the following screenshot:

```
$ arduino-cli board list
Port      Protocol Type      Board Name      FQBN      Core
/dev/ttyACM0 serial  Serial Port (USB) Arduino Nano 33 BLE arduino:mbed_nano:nano33ble arduino:mbed_nano
/dev/ttyS4 serial  Serial Port      Unknown
```

Figure 11.3: List of Arduino boards connected

Take note of the serial port of the connected device. For example, in *Figure 11.3*, we have an Arduino Nano 33 BLE Sense board connected to the `/dev/ttyACM0` serial port.

Step 7:

Use Arduino CLI to upload the compiled sketch on the microcontroller. The command structure to upload the sketch is as follows:

```
$ arduino-cli upload \
  -p <port_name> \
  -b <FQBN> \
  <sketch_name>
```

The only difference from the compile command is the `<port_name>`, which, in our case, is `/dev/ttyACM0` for the `arduino:mbed_nano:nano33ble` board.

Upon the command execution, you should see the built-in LED blinking on the microcontroller, confirming that your program has been correctly uploaded.

There's more...with the SparkFun Artemis Nano!

In this recipe, we learned how to use Arduino CLI to compile and upload sketches on Arduino-compatible platforms.

In Arduino CLI, you can also install additional boards, including the SparkFun Artemis Nano. To install the SparkFun Artemis Nano in Arduino CLI, you need to do the following:

Step 1:

Create an Arduino CLI configuration file with the SparkFun Artemis Nano package URL (https://raw.githubusercontent.com/sparkfun/Arduino_Apollo3/main/package_sparkfun_apollo3_index.json):

```
$ arduino-cli config init --additional-urls https://raw.githubusercontent.com/sparkfun/Arduino_Apollo3/main/package_sparkfun_apollo3_index.json
```

After the command execution, Arduino CLI returns the configuration file's location.

Step 2:

Update the index of the **Arduino core**:

```
$ arduino-cli core update-index
```

The preceding Arduino CLI command downloads the package index file for the SparkFun boards compatible with Arduino.

Step 3:

Install the SparkFun Apollo3 2.2.1 Arduino core release, recommended to replicate the recipes detailed in this chapter for the SparkFun Artemis Nano:

```
$ arduino-cli core install SparkFun:apollo3@2.2.1
```

The preceding command installs the 2.2.1 version of the SparkFun:apollo3 core. Please note that if you have previously installed the SparkFun Apollo3 2.2.1 Arduino core by following the **Setting up the SparkFun Artemis Nano board in the Arduino IDE** guide (https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Docs/setup_sparkfun_artemis_nano.md), this step will not make any changes, since it is already in place.

Step 4:

List the name of the Arduino boards installed in Arduino CLI to discover the FQBN value for the SparkFun Artemis Nano:

```
$ arduino-cli board listall
```

In the output log returned by the preceding command, you should see that the FQBN value for the SparkFun Artemis Nano is `SparkFun:apollo3:sfe_artemis_nano`.

Now, plug in the board to your laptop/PC, and determine the serial port name that it is connected to:

```
$ arduino-cli board list
```

After identifying the FQBN value and the serial port name of the SparkFun Artemis Nano, you are set to compile and upload the Arduino sketch on this board.

Arduino CLI will be incredibly helpful to build our applications with just a few command lines. As we have seen, this tool must be installed on our local machine because Colab cannot access devices connected to our computer. However, in what development environment should we install TVM? Since TVM will not require interaction with the microcontrollers in our experiments, we will use this framework in the Colab environment to provide a cross-platform solution.

Therefore, in the upcoming recipe, we will leave our local setup and start preparing the application in Colab.

Downloading a pre-trained CIFAR-10 model and input test image

In this project, we will not develop an ML model from scratch; instead, we will use a pre-trained model to dedicate more space to the model deployment with TVM. In this recipe, we will provide information on where to download the CIFAR-10 model and a header file containing an input test image as a C-byte array, required to validate the output classification.

Getting ready

The focus of this chapter will be primarily on the use of TVM to generate code for multiple target devices. To keep the problem as simple as possible and only focus on the model deployment, we will use a pre-trained CIFAR-10 quantized model and a constant input with a known output class to validate the inference process.

How to do it...

Open the web browser and create a new Colab notebook. Then, follow the following steps to download the pre-trained CIFAR-10 model and a C header file containing the input data, with an image that can be correctly classified as a ship:

Step 1:

Download the pre-trained CIFAR-10 quantized model from the TinyML-Cookbook_2E GitHub repository:

```
!wget https://github.com/PacktPublishing/TinyML-Cookbook_2E/raw/main/Chapter11/Assets/cifar10.tflite
```

This model just downloaded is what you would have obtained by following *Chapter 10, Deploying a CIFAR-10 Model for Memory-Constrained Devices with the Zephyr OS on QEMU*. Therefore, the model's input and output are quantized (int8).

Step 2:

Download the input.h C header file from the TinyML-Cookbook_2E GitHub repository:

```
!wget https://raw.githubusercontent.com/PacktPublishing/TinyML-Cookbook_2E/main/Chapter11/Assets/input.h
```

Within the input.h header file, you can find an int8_t array that holds the input test image of a ship (g_test), along with the array size (g_test_len) and the index class of the test image (g_test_label). The input test has already been normalized and quantized, making it ready for use with the quantized CIFAR-10 model.

There's more...

In this recipe, we downloaded the pre-trained model and the C header file containing the input test image, required to build our application with TVM.

As you may have noticed, the two files are from *Chapter 10, Deploying a CIFAR-10 Model for Memory-Constrained Devices with the Zephyr OS on QEMU*. Therefore, if you have completed that chapter, you can seamlessly upload your trained model and input test image.

At this point, having these two files, we can proceed with the code generation with TVM.

In the upcoming recipe, we will use TVM to deploy the pre-trained CIFAR-10 model on the machine hosting the Colab notebook.

Deploying the model with TVM using the AoT executor on the host machine

In this recipe, we will employ TVM for the first time to convert the pre-trained CIFAR-10 model into generic C code. Instead of deploying this code on a microcontroller, we will leverage the host-driven executor through Python. This simple approach will allow us to focus more on the essential TVM components needed for generating code.

Getting ready

The main advantage we have found in all projects developed with tflite-micro is certainly **code portability**. Regardless of the target device, the model inference can be accelerated on various devices using almost the same application code, which can be exemplified with the following pseudocode:

```
model = load_model(tflite_model)
model.allocate_memory()
model.invoke();
```

In the preceding code snippet, we do the following:

1. Load the model at runtime with `load_model()`
2. Allocate the memory required for the model inference with `allocate_memory()`
3. Invoke the model inference with `invoke()`

When writing the tflite-micro application code, it is not strictly necessary to have prior knowledge of the target microcontroller because *the software stack takes advantage of vendor-specific optimized operator libraries (performance libraries)* to execute the model efficiently. As a result, the selection of the appropriate set of optimized operators happens during the compilation of the application. For example, suppose we are compiling for an Arm-based microcontroller. If so, tflite-micro delegates the computation to the **CMSIS-NN** library, an open-source software library developed by Arm that provides superior performance on Arm Cortex-M processors.



You can learn more about CMSIS-NN at the following link: <https://arm-software.github.io/CMSIS-NN/main/index.html>

As we can guess, the primary way to leverage an efficient computation on the target device is by providing highly optimized operators in the performance library.

However, the computational efficiency of an ML model does not stop at the **operator-level**, though. In fact, some crucial optimizations can also be applied at the **graph-level**. One common optimization is **operator fusion**, where *two or more operators are combined to reduce the data movement and arithmetic operations involved*. The following figure visually illustrates this optimization:

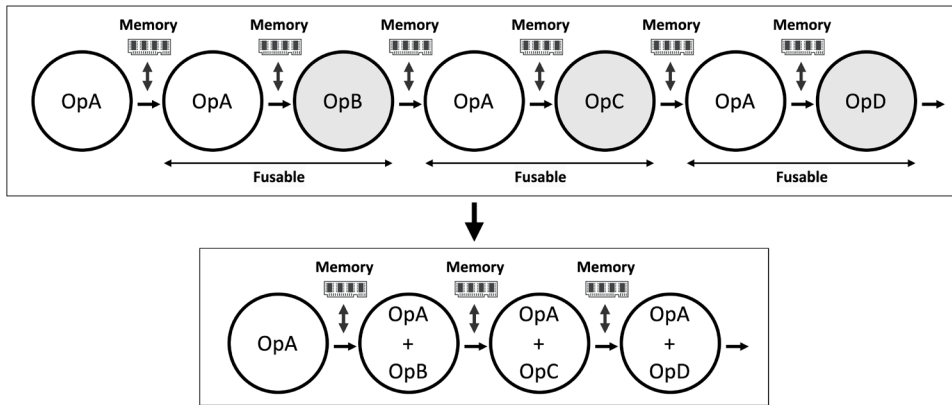


Figure 11.4: Operator fusion

In tflite-micro, operator fusion can only be done if the performance library provides operators with fusion capabilities. The example depicted in Figure 11.4 implies that the performance library makes available the following functions: **OpA**, **OpA + OpB**, **OpA + OpC**, and **OpA + OpD**.

From this simple example, it becomes evident that designing a highly optimized performance library may require an enormous engineering effort, as the possible fusion patterns are endless.

TVM (<https://tvm.apache.org/>) promises to overcome this limitation by providing a *framework capable of generating code for the operators*. As a result, if previously we went directly from the TensorFlow Lite model to the deployment, we now need an intermediate step to produce code for the device, as shown in the following figure:

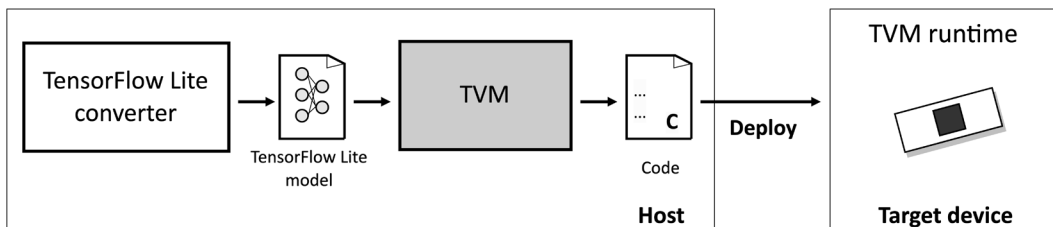


Figure 11.5: TVM can generate code from the TensorFlow Lite model

As we can guess, TVM must know *the target device to generate optimal code* from the TensorFlow Lite model.

Due to its nature of generating code, TVM falls into the category of **deep learning (DL) compilers**, and the following subsection will highlight its key features.

Behind the TVM compiler

The ambitious goal of TVM is to *translate any pre-trained ML model to optimized code for any device*:

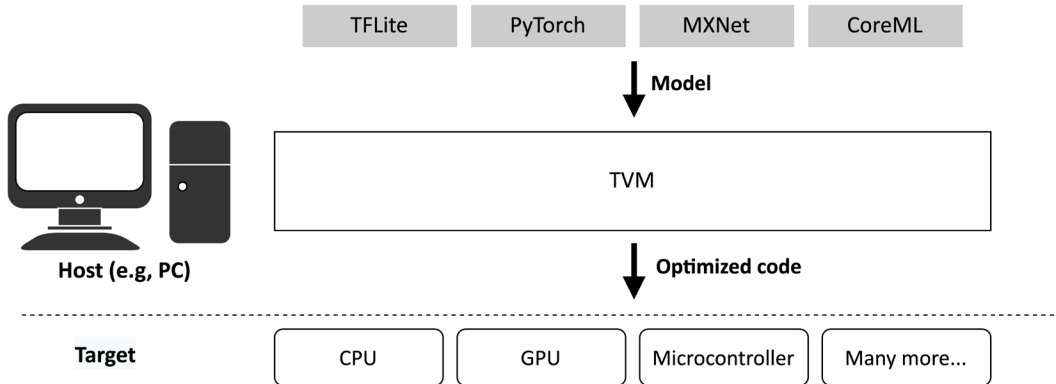


Figure 11.6: TVM generates optimized code from pre-trained models

TVM accepts a pre-trained model in various formats (for example, TensorFlow Lite or ONNX) and performs the code optimizations in two main steps, as shown in the following diagram:

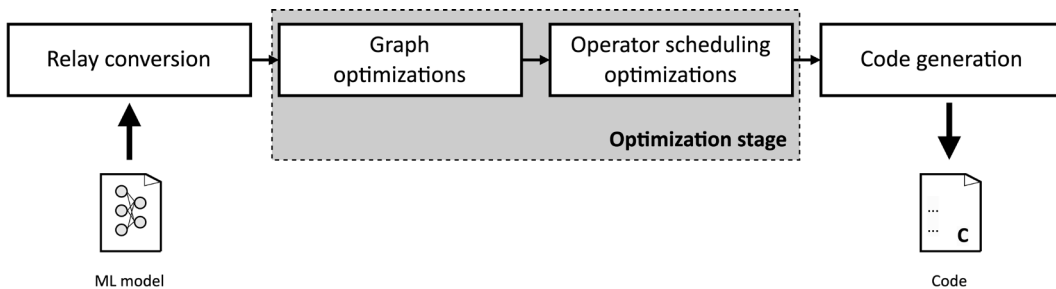


Figure 11.7: Main optimization stages in TVM

The previous diagram illustrates the process employed by TVM to generate optimized code. First, the input model is transformed into an internal high-level neural network language called **Relay**. Then, the compiler does the first optimization step at the model level (**graph optimizations**). The operator fusion optimization is performed at this level.

When TVM spots fusion patterns, it transforms the model by replacing the original operators with the new fused ones, as shown in the following example:

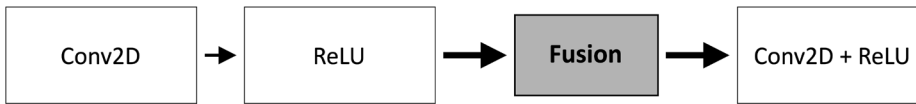


Figure 11.8: Conv2D + ReLU fusion

In the preceding example, fusion aims to create a single operator for **Conv2D (Convolution 2D)** and **ReLU** activation instead of having two separate ones as in the original model.

When fusion happens, generally, the computation time decreases because the code has fewer arithmetic instructions and memory transfers from/to memory.

The second optimization step performed by TVM occurs at the operator level (**operator scheduling**), which aims to find the most efficient way to execute each operator on the target device. This optimization is at the code level and affects adopting computing strategies such as tiling, unrolling, and vectorization. As we can imagine, the best computing method will depend on the target platform.

In the upcoming subsection, we will delve in more detail into what TVM generates from an ML model.

Code generation with TVM

TVM offers a Python interface to facilitate the code generation immediately after the TensorFlow Lite conversion. As shown in the following diagram, TVM produces mainly two software components: a library containing optimized operators for the target device (**TVM Lib codegen**) and an interface (**TVM runtime**) that facilitates invoking the model inference:

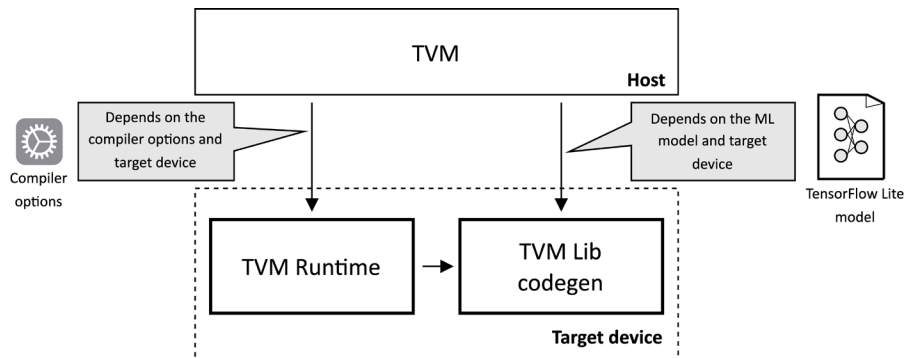


Figure 11.9: TVM runtime versus TVM Lib codegen

The **TVM runtime** and **TVM Lib codegen** depend on the target device. However, the **TVM Lib codegen** relies on the specific ML model; the **TVM runtime**, on the other hand, depends primarily on the chosen TVM compilation options.

To generate the code from the ML model, the TVM Python interface requires the following information:

- *Input model*: In our case, it will be the `cifar10.tflite` file.
- *Target device*: Since TVM is a compiler, it needs to know the target device to produce the optimal code.
- *Runtime language*: As we know, TVM can translate pre-trained ML models to optimized code for many devices and programming languages, such as Javascript, Java, Python, and C++. As a result, the runtime language is required to *provide the appropriate interface to invoke the model inference* into the application. For instance, if we develop a Python application, the runtime interface will be in Python. On the other hand, if we are targeting a microcontroller, the runtime interface will likely be in C.
- *Executor type*: The executor type defines how a model should be executed on the target device, and *it has implications on memory usage and the ability to update the model*. There are two types of executors to run a model with TVM: the graph executor and the AoT executor.

The following subsection will explain the distinction between the two types of executors.

Graph executor versus AoT executor

The **graph executor** is a runtime execution approach that dynamically builds the ML compute graph during application startup. Therefore, when the application on the device initializes, the model is loaded, possibly from a file, and the list of operations to be scheduled is prepared on the fly. Tflite-micro exclusively adopts this execution type, which provides the following benefits:

- Accommodating dynamic changes such as model updates or modifications in input dimensions
- Ease of deployment because the application is almost the same for all target devices

As we can guess, the *graph executor is suitable in scenarios where runtime flexibility is necessary*.

On the other hand, the **AoT executor** takes a different approach by pre-building the ML compute graph before deployment on the target device.

This execution type can only be adopted when the target device is known, and it brings several advantages, such as the following:

- *Faster startup times*: The application can start more quickly, since the compute graph is already prebuilt.
- *More efficient execution*: Since there is no need for an advanced runtime to call operations to execute, the scheduling overhead is typically reduced.
- *Enhanced memory resource efficiency*: Since we do not need the code to load the model at runtime, program memory occupancy can be minimized.

Therefore, although it may reduce runtime flexibility, this approach offers significant benefits for resource-constrained devices or time-critical applications, which are standard requirements in many tinyML applications on microcontrollers.

Before proceeding with the steps required to generate code with TVM, one aspect must be discussed: the output code format produced by the compiler for microcontrollers.

Introducing microTVM for microcontroller deployment

As mentioned earlier, TVM generates code from the ML model, and traditionally, it produces the following output files:

- `.so`: A C++ dynamic library containing the optimized operators to execute the model
- `.json`: A JSON file containing the computation graph and weights
- `.params`: A file containing the parameters of the pre-trained model

Unfortunately, the preceding three files are not suitable for microcontroller deployment for the following reasons:

- Microcontrollers do not have a **memory management unit (MMU)**, so we cannot load dynamic libraries at runtime.
- The weights are stored in an external file (`.json`), which is not ideal for microcontrollers for two reasons. The first is that we may not have an OS that provides an API to read external files. The second is that weights loaded from an external file go into SRAM, generally smaller than the program memory.

For the preceding reasons, an extension to TVM was proposed to produce a suitable output for microcontrollers. This extension is **microTVM** (<https://tvm.apache.org/docs/topic/microtvm/index.html>), which aims to *produce C source code that does not require OS and dynamic memory allocation*.



Devices without an OS are commonly called bare-metal.

The output of microTVM is a TAR package containing the C code for the TVM runtime and TVM Lib. This package is commonly called a **model library format (MLF)**.

How to do it...

Continue working in Colab, and follow the following steps to generate the code from a pre-trained CIFAR-10 model, using the TVM Python interface:

Step 1:

Install the 0.11.1 release version of TVM, recommended to reproduce the recipes presented in this chapter:

```
!pip install apache-tvm==0.11.1
```

Step 2:

Install the 2.10.0 release version of the TensorFlow Lite Python package (**tf lite**), required by the 0.11.1 release of TVM:

```
!pip install tf lite==2.10.0
```

The **tf lite** Python package (<https://pypi.org/project/tf lite/>) is a library for parsing TensorFlow Lite models built by the TensorFlow Lite converter. This package is required to load the CIFAR-10 model in the next step.

Step 3:

Load the CIFAR-10 model from the disk, and parse it with the **tf lite** Python package:

```
import tf lite
tfl_file = open("cifar10.tflite", "rb").read()
tfl_model = tf lite.Model.GetRootAsModel(tfl_file, 0)
```

The `tf lite.Model.GetRootAsModel` will import the CIFAR-10 TensorFlow Lite model into the `tfl_model` object.

Step 4:

Import the TensorFlow Lite model into TVM:

```
import tvm
mod, params = tvm.relay.frontend.from_tflite(tfl_model)
```

The preceding function converts the TensorFlow Lite model (`tfl_model`) into a relay module object (`mod`). The relay module is what we will provide to the TVM compiler, along with the target device and compiler options, to generate the code.

Step 5:

Define the host machine as the target device:

```
target = tvm.target.target.micro("host")
```

In the preceding code snippet, we select the host machine (`host`) as a target device.

To know the list of supported microcontroller boards by TVM, you can print the dictionary returned by `tvm.target.target.MICRO_SUPPORTED_MODELS`, as shown in the following code block:

```
print(tvm.target.target.MICRO_SUPPORTED_MODELS)
```

As shown in *Figure 11.10*, the dictionary contains the name of each supported target device with its corresponding CPU compiler flag:

```
{'host': [], 'atsamd51': ['-mcpu=cortex-m4'], 'cxd5602gg': ['-mcpu=cortex-m4']},
```

Figure 11.10: A portion of the MICRO_SUPPORTED_MODELS dictionary

For example, looking at the list of target devices returned by `tvm.target.target.MICRO_SUPPORTED_MODELS`, we can find:

- `'host': []`: This is the host machine, which has an empty list of CPU compiler flags, indicating that the generated code will be generic and not optimized for a specific architecture.
- `'atsamd51': ['-mcpu=cortex-m4']`: This is the Microchip ATSAM51 microcontroller, which has the compiler flag `-mcpu=cortex-m4`, indicating that the generated code will be optimized for the Arm Cortex-M4 CPU.

This approach to selecting the target platform using the board's name is the easiest and most recommended, as we do not need to worry about the processor architecture name and programming output language.

If your board is not included in the supported devices, an alternative approach allows you to specify the CPU architecture and output language directly. The following code snippet is an example of generating code for an Arm Cortex-M0-based microcontroller:

```
target_cpu = "c -keys=arm_cpu,cpu -mcpu=cortex-m0"
target = tvm.target.Target(target_cpu)
```

The preceding example indicates our preference to generate C code for an Arm CPU (`arm_cpu,cpu`) with an Arm Cortex-M0 processor (`-mcpu=cortex-m0`).

Step 6:

Define the C runtime as the TVM runtime type:

```
crt = tvm.relay.backend.Runtime("crt",
                               {"system-lib" : True})
```

In the previous code snippet, we requested TVM to generate a C runtime, as most microcontrollers can only run C code. In addition to this request, we enabled static linking by setting `system-lib` to `True` because it is required to build the host target application.

Step 7:

Define the AoT executor as the TVM executor type:

```
aot = tvm.relay.backend.Executor("aot")
```

As mentioned in the earlier *Getting ready* section, the AoT executor is recommended to minimize the model's memory footprint of the application.

Step 8:

Compile the model for the host machine:

```
compiler_opts = {"tir.disable_vectorize": True}
with tvm.transform.PassContext(opt_level=3,
                               config=compiler_opts):
    module = tvm.relay.build(mod,
                             target,
                             runtime=crt,
                             executor=aot,
                             params=params)
```

The compilation step with TVM consists of two sequential operations:

1. Defining the compiler options through the `tvm.transform.PassContext()` function. In our case, we turn off the vectorization (`{"tir.disable_vectorize": True}`) to ensure the generated C code remains generic and can function on any device. Additionally, we set the optimization level of the compilation to 3 to enable operator fusion. For details on the optimizations enabled by level 3, you can refer to the official documentation available at the following link: <https://tvm.apache.org/docs/reference/api/python/relay/index.html>.
2. Generating the code with the `tvm.relay.build()` function. This function needs at least the relay module (`mod`), the target device (`target`), and the executor type (`aot`) to compile the model.

After this step, the C code for the target device will be ready to be integrated into the application.

Step 9:

Create the project from the code generated in the previous step:

```
import os
import shutil

base_dir = "/tmp/tvm/"
build_dir = base_dir + "host"
is_exist = os.path.exists(base_dir)

if is_exist:
    shutil.rmtree(base_dir)

os.mkdir(base_dir)

project = tvm.micro.generate_project(
    tvm.micro.get_microtvm_template_projects("crt"),
    module,
    build_dir
)
```

Once we have generated the code with the `tvm.relay.build()` function, we can create the application. For this scope, TVM provides the `tvm.micro.generate_project()` method to *generate a project from a template*.



A template project is a directory, with a pre-designed folder structure and pre-generated files, that can be used as a starting point to create a new project of the same type.

The `tvm.micro.generate_project()` function requires the following mandatory arguments:

- The path to the directory containing the template project
- The output generated by the `tvm.relay.build()` function
- The path to the directory containing the output project

As we have seen in previous chapters, the project structure can vary, depending on the target platform and operating system. As a result, the path to the directory containing the template project may differ, depending on the project we intend to build for. In this case, we have built the project to run the model on the host machine using the host-driven interface. The path to the directory containing the template project is obtained using `tvm.micro.get_microtvm_template_projects("crt")`.

Step 10:

Compile (build) and upload (flash) the project:

```
project.build()
project.flash()
```

The `build()` and `flash()` functions are used in microTVM to compile and upload the project on the target device. However, since we are working in a cloud environment, these two functions will not actually deploy the application on the physical microcontroller at this point.



Later in the chapter, we will demonstrate how you can compile and upload the code generated by TVM using Arduino CLI.

Step 11:

Get the model's input and output quantization parameters using the tflite Python package.

To do so, retrieve the graph from the TensorFlow Lite model:

```
graph = tfl_model.Subgraphs(0)
```

Then, get the tensor index for the first (model's input) and last tensor (model's output):

```
i_idx = 0  
o_idx = graph.TensorsLength() - 1
```

After that, get the input and output tensors:

```
i_tensor = graph.Tensors(i_idx)  
o_tensor = graph.Tensors(o_idx)
```

Finally, get the scale and zero point quantization parameters from the input and output tensor:

```
i_quant = i_tensor.Quantization()  
o_quant = o_tensor.Quantization()  
  
i_scale = i_quant.Scale(0)  
i_zero_point = i_quant.ZeroPoint(0)  
  
o_scale = o_quant.Scale(0)  
o_zero_point = o_quant.ZeroPoint(0)  
  
print(o_scale)  
print(o_zero_point)
```

In the preceding code snippet, we print the output quantization parameters (`o_scale` and `o_zero_point`) because they are required to implement the Arduino sketch in the following recipe. In our case, the `o_scale` and `o_zero_point` variables are equal to 0.10305877029895782 and 20, respectively.

Step 12:

Create a NumPy input sample image to test the CIFAR-10 model inference.

To do so, download the image from the TinyML-Cookbook_2E GitHub repository:

```
!wget -O ship.jpg https://github.com/PacktPublishing/TinyML-Cookbook_2E/
blob/main/Chapter11/Assets/ship.jpg?raw=true
```

Then, load the image with the **Pillow** (<https://pypi.org/project/Pillow>) library and resize it to 32x32, as required by the CIFAR-10 input model:

```
from numpy import asarray
from PIL import Image
img_name = 'ship.jpg'
image = Image.open(img_name)
image = image.resize((32,32))
```

After, convert the image to a NumPy array:

```
import numpy as np
sample = asarray(image)[np.newaxis, :]
```

Finally, normalize the image pixels by dividing them by 255, and apply quantization using the input quantization parameters of the CIFAR-10 model:

```
sample = sample / 255.0
sample = (sample / i_scale) + i_zero_point
```

Step 13:

Run the CIFAR-10 model using the Python host-driven interface:

```
import numpy as np

with tvm.micro.Session(project.transport()) as session:
    x = session.create_aot_executor()
    exec = tvm.runtime.executor.aot_executor.AotModule(x)
    exec.get_input(0).copyfrom(sample)
    exec.run()
    result = exec.get_output(0).numpy()
```

As you can see from the previous code snippet, the model inference requires four key actions:

- *Initializing the TVM executor* through the `session.create_aot_executor()` and `tvm.runtime.executor.AotModule()` methods

- *Writing the input sample into the input tensor* using the `get_input(0).copyfrom()` method
- *Invoking the model inference* using the `run()` method
- *Reading the output tensor* using `get_output(0)`

After the model inference, dequantize the output tensor and return the classification result on the output log:

```
result = o_scale * (result - o_zero_point)
labels = [
    "airplane", "automobile", "bird",
    "cat", "deer", "dog",
    "frog", "horse", "ship", "truck"
]

print(f"Result: '{labels[np.argmax(result)]}'")
```

After executing the preceding code block, you should see the `Result: 'ship'` message in the output log, confirming that the input sample has been correctly classified as a ship.

There's more...

In this recipe, we discovered the features of TVM and learned how to use its AoT executor to deploy a TensorFlow Lite model on the machine hosting the Colab environment.

In the *Getting ready* section of this recipe, we have illustrated just the main points of the TVM architecture to give you the big picture of how this compiler technology works. For more information about the TVM architecture, you can refer to the TVM introduction guide, which provides a step-by-step explanation of the model optimizations: <https://tvm.apache.org/docs/tutorial/introduction.html#sphx-glr-tutorial-introduction-py>.

Now that we have familiarized ourselves with this technology, we can leverage it to deploy ML models on actual microcontrollers.

In the upcoming recipe, we will use TVM to run the CIFAR-10 model inference on the Arduino Nano.

Deploying the model on the Arduino Nano

Now that we have acquired the knowledge of code generation with TVM, we are prepared to shift our attention toward deploying an actual model on physical microcontrollers.

Thus, in this recipe, we aim to deploy the quantized CIFAR-10 model on the Arduino Nano.

Getting ready

To get ready with this recipe, we need to know how to generate and structure an Arduino project.

In the previous recipe, we executed the CIFAR-10 model on the host machine through the Python host-driven interface. However, we haven't seen any actual code generated apart from a few Python objects returned by TVM.

As mentioned earlier, when dealing with microcontrollers, the output of TVM is a TAR package that contains the C code for the TVM runtime and TVM Lib, known as MLF. The TAR file is created when we call the `tvm.micro.generate_project()` function and is automatically decompressed, integrating only the necessary files into the target template project.

Inside the MLF file named `model.tar`, located in the output directory provided to the `tvm.micro.generate_project()` function, the main folders of interest are:

- `codegen`: This is the TVM LIB, which includes the C code generated for the operators required by your model.
- `runtime`: This is the TVM runtime, which includes the necessary functions to execute the model on the target device.
- `templates`: This includes template files that can be edited to build an application with TVM without starting from scratch.

The MLF file also contains the `metadata.json` file, which includes information mainly about the model operators, the type of executor, and memory requirements. The information about the memory requirement is available in the memory section. In particular, the program memory and SRAM usage are reported in the `constants_size_bytes` and `workspace_size_bytes` fields, respectively. Additionally, in the same memory section, you can find the memory requirements for both input and output because *these tensors need to be allocated by us*.

The generation of the MLF file depends on the target device. As a result, the project must be generated for each target device.

Building a project for Arduino-compatible boards is straightforward, as we can still do it with the `tvm.micro.generate_project()` function. However, in contrast to what we have seen previously with the host target, the function needs a different path to the TVM template project and one additional argument, as shown in the following code snippet:

```
opts = {  
    "board": "nano33ble",  
    "project_type": "example_project",  
}
```

```
prj = tvm.micro.generate_project(  
    tvm.micro.get_microtvm_template_projects("arduino"),  
    mod,  
    build_dir,  
    opts)
```

As you can see from the previous code block, which builds a project for Arduino Nano, the main differences from the project generated for the host machine are due to the following:

- The path to the directory containing the Arduino template project
- The options necessary to build the project for the Arduino board with TVM

The options passed to the `tvm.micro.generate_project()` function are required mainly to provide the target Arduino board ("board").



Important

To generate the Arduino project within TVM, it is essential to have Arduino CLI installed in your working environment, which, in our case, is Colab. Therefore, you must have installed Arduino CLI before invoking the `tvm.micro.generate_project()` function.

The list of Arduino boards supported in TVM can be found in the `boards.json` file within the directory where the Arduino template project is located. To view the content of this file, you can conveniently make use of the following `cat` command:

```
import tvm  
x = tvm.micro.get_microtvm_template_projects("arduino")  
file_path = x + "/boards.json"  
  
!cat $file_path
```

Within the file, you should find the board name for the Arduino Nano microcontroller, which is `nano33ble`.



The `boards.json` file also contains the target for the Raspberry Pi Pico. Unfortunately, this target is not for Mbed OS and cannot be used for our current purpose. However, there's no need to worry. In the upcoming recipe, you will learn how to deploy the ML model with TVM on the Raspberry Pi Pico, or any Arduino-compatible board.

At this point, you might wonder, how do we invoke the model inference with the TVM runtime in an Arduino sketch?

Running the model inference using the TVM runtime

As we will see in the upcoming *How to do it...* section, running an ML model on Arduino requires calling only the following two functions:

- `TVMInitialize()`: This function should be called in the `setup()` routine to allocate the temporary memory and perform the initialization tasks.
- `TVMExecute()`: This function invokes the model inference and should be called in the `loop()` routine. The pointers to the memory regions of the inputs and output are the input arguments of the `TVMExecute()` function.

To use these functions, you only need to include the `model.h` C header file in your sketch, generated by TVM, when calling the `tvm.micro.generate_project()` function.



The `model.h` file is in the `src/` directory of the Arduino project created by TVM.

Now that we know how the TVM runtime works, the only remaining thing to be discussed is the preparation of an Arduino sketch with the code generated by TVM.

Building an Arduino sketch with the code generated by TVM

When programming with Arduino, the program on the Arduino board is commonly called a sketch. Traditionally, we consider the sketch to be the `.ino` file. However, technically, the sketch is the folder containing the `.ino` file and other necessary source code files for the application.

When creating a new sketch using the Arduino CLI command, the tool generates a new folder and, within it, creates a `.ino` file with a name identical to the folder's name, as shown in the following file hierarchy:

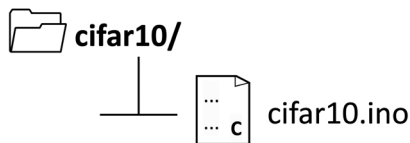


Figure 11.11: The sketch is the folder containing the `.ino` file



The initial `.ino` file has empty `setup()` and `loop()` functions.

Each sketch must consist of only one `.ino` file, and the filename must match the name of the directory in which it is located. However, as we can guess, large applications like the one we are deploying with TVM may need additional source code files. These files, which can have extensions of standard C and C++ files (for example, `.c`, `.cpp`, `.h`, and so on), can be placed in two locations:

- Into the root directory of the sketch
- Into the `src/` subfolder



The contents of the `src/` subfolder are compiled recursively.

Therefore, a sketch with multiple source code files may take the following form:

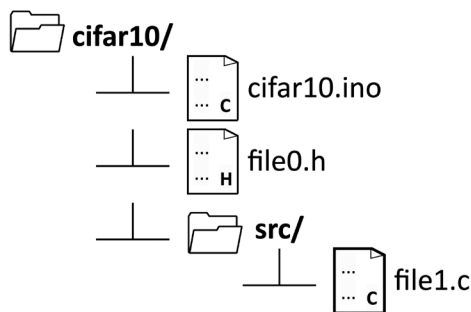


Figure 11.12: Example of a sketch with multiple source code files



To find more details about the sketch specification, you can refer to the official documentation available at the following link: <https://arduino.github.io/arduino-cli/latest/sketch-specification/>.

The `tvm.micro.generate_project()` function is responsible for creating the Arduino project with the required files and folder structure for the sketch. In particular, this function generates the following:

- The `src/` folder containing the TVM runtime and TVM Lib

- An Arduino reference sketch to show how to invoke the ML model inference with the TVM runtime

Therefore, once we have generated the Arduino project using TVM, we just need to do the following to build the Arduino sketch using Arduino CLI:

1. Remove the Arduino reference sketch created by TVM from the Arduino project
2. Create a new sketch with Arduino CLI
3. Copy the content of the Arduino project created by TVM into the sketch.
4. Write an Arduino sketch to invoke the ML model inference using the TVM runtime

The preceding steps will allow us to deploy the application on the microcontroller.

How to do it...

Reopen the Colab notebook, and follow the following steps to deploy the CIFAR-10 model on the Arduino Nano using TVM:

Step 1:

Install Arduino CLI 0.34.0, required by the 0.11.1 release of TVM to create the Arduino project:

```
!curl -fsSL https://raw.githubusercontent.com/arduino/arduino-cli/0.34.0/
install.sh | sh

import os
os.environ['PATH'] += ':/content/bin'
```

Step 2:

Define the Arduino Nano as the target device:

```
nano33 = tvn.target.target.micro("nrf52840")
```

The Arduino Nano target is selected by specifying its respective microcontroller name: nrf52840.

Step 3:

Define the C runtime and AoT executor as the TVM runtime type and executor type, respectively:

```
crt = tvn.relay.backend.Runtime("crt")
opts_exec = {"unpacked-api": True}
aot = tvn.relay.backend.Executor("aot", opts_exec)
```


In contrast to the model deployment on the machine hosting the Colab notebook, we do not need to enable static linking by setting `system-lib` to `True` when deploying the application on a physical microcontroller. However, you need to pass `unpacked-api = True` to the `Executor()` function to replace TVM-specific structs with native types (<https://discuss.tvm.apache.org/t/rfc-utvm-aot-optimisations-for-embedded-targets/9849>).

Step 4:

Write a helper function to compile the model for a target device:

```
def compile_model(device):
    compiler_opts = {"tir.disable_vectorize": True}
    with tvm.transform.PassContext(opt_level=3,
                                    config=compiler_opts):
        return tvm.relay.build(mod,
                                device,
                                runtime=crt,
                                executor=aot,
                                params=params)
```

Since we aim to use the same TVM runtime and executor type in all Arduino projects, it is convenient to encapsulate the model compilation process within a helper function. The function takes the target device (`device`) as the input argument and compiles the model accordingly, with the `tvm.relay.build()` function.

Step 5:

Use the `compile_model()` helper function to compile the model for the Arduino Nano:

```
lib_nano33 = compile_model(nano33)
```

Step 6:

Write a helper function to create an Arduino project from the library, generated by the `tvm.relay.build()` function:

```
def build_arduino_prj(board, lib):
    base_dir = "/tmp/tvm/"
    is_exist = os.path.exists(base_dir)
```

```

if is_exist:
    shutil.rmtree(base_dir)

os.mkdir(base_dir)

build_dir = base_dir + board

shutil.rmtree(build_dir, ignore_errors=True)

return tvn.micro.generate_project(
    tvn.micro.get_microtnv_template_projects("arduino"),
    lib,
    build_dir,
    {
        "board": board,
        "project_type": "example_project",
    },
)

```

This helper function takes the target Arduino board (board) and the library obtained with the `compile_model()` function (lib) as input arguments to generate the Arduino project using the `tnv.micro.generate_project()` function.

Step 7:

Use the `build_arduino_prj()` helper function to create the Arduino project:

```
prj_nano = build_arduino_prj("nano33ble", lib_nano33)
```

The `build_arduino_prj()` function creates the Arduino project within the `/tmp/tnv/nano33ble` directory, which contains the following files and directories:

```

!ls /tmp/tnv/nano33ble
boards.json include Makefile microtnv_api_server.py nano33ble.ino src

```

From the list files in the `/tmp/tnv/nano33ble` directory, you can see an Arduino sketch called `nano33ble.ino`. This sketch is a reference implementation generated by TVM to show how to invoke the ML model inference with the TVM runtime.

Step 8:

In the `/tmp/tvm/nano33ble` directory, delete the `boards.json`, `Makefile`, `microtvm_api_server.py`, and `nano33ble.ino` files because they are not required for the application:

```
!rm /tmp/tvm/nano33ble/boards.json
!rm /tmp/tvm/nano33ble/Makefile
!rm /tmp/tvm/nano33ble/microtvm_api_server.py
!rm /tmp/tvm/nano33ble/nano33ble.ino
```

After that, remove the inclusion of the `Arduino.h` file in the `/tmp/tvm/nano33ble/model.c` file, using the following `sed` command:

```
!sed -i 's/#include "Arduino.h"//g' \
/tmp/tvm/nano33ble/src/model.c
```

The reason for removing the inclusion of this file is that it could cause compilation issues when using Arduino CLI.

Step 9:

Copy the `input.h` C header file, containing the normalized and quantized input test image for the CIFAR-10 model, into the `/tmp/tvm/nano33ble/src` directory:

```
!cp input.h /tmp/tvm/nano33ble/src
```

Step 10:

Zip the content of the `/tmp/tvm/nano33ble` directory to export the code generated by TVM to our local machine:

```
!cd /tmp/tvm/nano33ble; zip -r micro_tvm_code.zip .
!mv /tmp/tvm/nano33ble/micro_tvm_code.zip .
```

Now, download the `micro_tvm_code.zip` file from Colab's left pane and leave the Colab environment to finalize the Arduino application on the local environment.

Step 11:

Open Terminal, and create a new Arduino sketch called `micro_tvm`, using Arduino CLI:

```
$ arduino-cli sketch new micro_tvm
```

Take note of the folder location returned after executing the preceding command because you will need to unzip the `micro_tvm_code.zip` file within it.

Step 12:

Unzip the `micro_tvm_code.zip` file into the Arduino sketch folder created in the previous step. Once you have unzipped the file, you should have the following files inside the `micro_tvm/` sketch:

```
$ ls /home/user/micro_tvm
include  micro_tvm.ino  src
```

Figure 11.13: Content in Arduino sketch's folder after unzipping the `micro_tvm_code.zip` file

Ensure the `src/` and `include/` directories are at the same level as the `micro_tvm.ino` file.

Step 13:

In the Arduino sketch's folder, open the `micro_tvm.ino` sketch file with your favorite IDE.

Then, write an application to turn on the built-in LED of the Arduino Nano if the model can correctly classify the input test image.

To implement the sketch, include the following header files:

- `src/model.h`: To employ the TVM runtime
- `src/input.h`: To access the array containing the input test image with its corresponding class index
- `mbed.h`: To drive the LED with the Mbed OS API

```
#include "src/model.h"
#include "src/input.h"
#include "mbed.h"
```

Once you have included the header files, define the output quantization parameters as global variables, and implement a function to dequantize the output model:

```
float out_scale = 0.10305877029895782;
int32_t out_zero_point = 20;

void dequantize(int8_t* src, float* dst, int32_t len) {
    for(int32_t i = 0; i < len; ++i) {
        dst[i] = out_scale * (src[i] - out_zero_point);
    }
}
```

The quantization parameters (`out_scale` and `out_zero_point`) are initialized with the quantization parameters of the model's output, retrieved in the preceding recipe.

After implementing the dequantization function, write a function to return the index of the output class with the highest score (`argmax`):

```
int32_t argmax(float* src, int32_t len) {
    int32_t max_idx = 0;
    float max_score = src[0];

    for(int32_t i = 1; i < len; ++i) {
        if(src[i] > max_score) {
            max_score = src[i];
            max_idx = i;
        }
    }
    return max_idx;
}
```

Finally, initialize the TVM runtime, invoke the model inference, and turn the built-in LED on if the output classification is correct:

```
mbed::DigitalOut led(LED1);

void setup() {
    TVMInitialize();
    led = 0;
}

void loop() {
    int8_t out_q8[10];
    float out_f32[10];
    TVMExecute(g_test, out_q8);

    dequantize(out_q8, out_f32, 10);

    int32_t max_idx = argmax(out_f32, 10);

    if(max_idx == g_test_ilabel) {
```

```
    led = 1;
  }
  while(1);
}
```

From the preceding code block, pay attention to the allocation of the output tensor (`int8_t out_q8[10]`). Unlike in `tflite-micro`, where the output allocation is managed internally, we must allocate the output tensor ourselves. As a result, the output should be of the same type (`int8_t`) and size (10) as the model output.

The same principle applies to the input, which has already been pre-generated in the `input.h` file.

Step 14:

Connect the Arduino Nano to your computer. Then, compile and upload the sketch on the Arduino Nano, using Arduino CLI:

```
$ arduino-cli compile \
-b arduino:mbed_nano:nano33ble \
micro_tvm

$ arduino-cli upload \
-p <port-name> \
-b arduino:mbed_nano:nano33ble \
micro_tvm
```

In the previous command, you should replace `<port-name>` with the port name of the Arduino Nano connected to your computer, which you can easily find using Arduino CLI:

```
$ arduino-cli board list
```

Once the sketch has been uploaded on the microcontroller, you should see the built-in LED switch on, confirming the correct classification of the input test image.

There's more...

In this recipe, we learned how to generate code with TVM to run the CIFAR-10 model inference on an Arduino Nano and compile it with Arduino CLI.

TVM offers multiple options to control the code generation flow, improving latency performance and memory usage.

Among these compilation options, **Unified Static Memory Planning (USMP)** is critical for deployment on microcontrollers because it performs memory planning of all tensors, leading to optimal memory utilization.

To use the USMP compilation option, you just need to include `"tir.usmp.enable" = True` in the compiler options. As a result, the new `compile_model()` helper function should be like the following:

```
def compile_model(device):
    compiler_opts = {"tir.disable_vectorize": True}
    compiler_opts["tir.usmp.enable"] = True
    with tvm.transform.PassContext(
        opt_level=3,
        config=compiler_opts):
        return tvm.relay.build(mod,
                                device,
                                runtime=crt,
                                executor=aot,
                                params=params)
```

The only change made to the `compile_model(device)` function is adding the line in bold, which enables the USMP optimization.

This recipe taught us how to generate code for the Arduino Nano. However, how can we generate code for the Raspberry Pi Pico or any Arduino-compatible microcontrollers?

In the upcoming recipe, we will answer this question by demonstrating how to generate code with TVM, to run the CIFAR-10 model inference on the Raspberry Pi Pico.

Deploying the model on the Raspberry Pi Pico

The deployment of the quantized CIFAR-10 model on the Arduino Nano showcased TVM's capability to generate code to run the model inference on this specific platform.

In this recipe, we will discover how we can use TVM to generate code for the Raspberry Pi Pico.

Getting ready

As we have seen in the previous chapter, the `tvm.micro.generate_project()` function is responsible for generating the Arduino project. Among the list of input arguments, this function requires the Arduino board name. However, what is the purpose of this information?

The board name is not used during the code generation phase because the code is already generated when calling the `tvm.micro.generate_project()` function. Instead, this information is required because TVM offers commands that allow building and flashing the application directly on the Arduino board from Python. Since we are not employing these commands to compile and upload the Arduino application on the microcontroller, providing `nano33ble` as a board name of any Arduino-compatible platform to the `tvm.micro.generate_project()` function will not affect the structure of the Arduino project. However, the only inconvenience is that the directory containing the Arduino project will always be called `nano33ble`, regardless of the actual target device.

How to do it...

Reopen the Colab notebook and follow the following steps to deploy the CIFAR-10 model on the Raspberry Pi Pico, using TVM:

Step 1:

Define the Raspberry Pi Pico as the target device:

```
rp2040 = tvm.target.target.micro("rp2040")
```

The Raspberry Pi Pico target is selected in the previous code snippet by specifying its respective microcontroller name: `rp2040`.

Step 2:

Define the C runtime and AoT executor as the TVM runtime type and executor type, respectively:

```
crt = tvm.relay.backend.Runtime("crt")
opts_exec = {"unpacked-api": True}
aot = tvm.relay.backend.Executor("aot", opts_exec)
```

Step 3:

Use the `compile_model()` helper function to compile the model for the Raspberry Pi Pico:

```
lib_rp2040 = compile_model(rp2040)
```

Step 4:

Use the `build_arduino_prj()` helper function to create the Arduino project:

```
prj_pico = build_arduino_prj("nano33ble", lib_rp2040)
```


As you can see from the input arguments of the `build_arduino_prj()` helper function, the name of the Arduino board ("nano33ble") is the same as the one used to generate the project for Arduino Nano. However, there is no reason to be concerned about the code generation. The code, in fact, relies on the `lib_rp2040` module, which is obtained by specifying the Raspberry Pi Pico as the target device.

Step 5:

In the `/tmp/tvm/nano33ble` directory, delete the `boards.json`, `Makefile`, `microtvm_api_server.py`, and `nano33ble.ino` files because they are not required for the application:

```
!rm /tmp/tvm/nano33ble/boards.json
!rm /tmp/tvm/nano33ble/Makefile
!rm /tmp/tvm/nano33ble/microtvm_api_server.py
!rm /tmp/tvm/nano33ble/nano33ble.ino
```

After that, remove the inclusion of the `Arduino.h` file in the `tmp/tvm/nano33ble/model.c` file using the following `sed` command:

```
!sed -i 's/#include "Arduino.h"//g' \
/tmp/tvm/nano33ble/src/model.c
```

Step 6:

Copy the `input.h` C header file, containing the normalized and quantized input test image for the CIFAR-10 model, into the `/tmp/tvm/nano33ble/src` directory:

```
!cp input.h /tmp/tvm/nano33ble/src
```

Step 7:

Zip the content of the `/tmp/tvm/nano33ble` directory to export the code generated by TVM to our local machine:

```
!cd /tmp/tvm/nano33ble; zip -r micro_tvm_code.zip .
!mv /tmp/tvm/nano33ble/micro_tvm_code.zip .
```

Now, download the `micro_tvm_code.zip` file from Colab's left pane, and leave the Colab environment to finalize the Arduino application on the local environment.

Step 8:

Open Terminal, and create a new Arduino sketch called `micro_tvm_pico`, using Arduino CLI:

```
$ arduino-cli sketch new micro_tvm_pico
```

Take note of the folder location returned after executing the preceding command because you will need to unzip the `micro_tvm_code.zip` file within it.

Step 9:

Unzip the `micro_tvm_code.zip` file into the Arduino sketch folder created in the previous step, and ensure that the `src/` and `include/` directories are at the same level as the `micro_tvm_pico.ino` file.

Step 10:

In the Arduino sketch folder, open the `micro_tvm_pico.ino` sketch file with your favorite IDE. Then, implement a sketch to turn on the built-in LED of the Raspberry Pi Pico if the model can correctly classify the input test image. The sketch should follow the exact implementation of the one developed for the Arduino Nano. For a detailed explanation of the sketch, you can refer to the previous recipe, which provides all the steps to use the TVM runtime to invoke the model inference.

Step 11:

Connect the Raspberry Pi Pico to your computer. Then, compile and upload the sketch on the Raspberry Pi Pico, using Arduino CLI:

```
$ arduino-cli compile \  
-b arduino:mbed_rp2040:pico \  
micro_tvm_pico  
  
$ arduino-cli upload \  
-p <port-name> \  
-b arduino:mbed_rp2040:pico \  
micro_tvm_pico
```

In the previous command, you should replace `<port-name>` with the port name of the Raspberry Pi Pico connected to your computer, which you can easily find using Arduino CLI:

```
$ arduino-cli board list
```

Once the sketch has been uploaded on the microcontroller, you should see the built-in LED switch on, confirming the correct classification of the input test image.

There's more...with the SparkFun Artemis Nano!

In this recipe, we learned how to generate code with TVM to run the CIFAR-10 model inference on the Raspberry Pi Pico and use Arduino CLI to compile and upload it on the microcontroller.

Unfortunately, at the time of writing, TVM does not support the SparkFun Artemis Nano's microcontroller (Ambiq Apollo3) as a target device. However, you can still generate the code for this platform by providing the CPU architecture of the Ambiq Apollo3 microcontroller (Arm Cortex-M4) and the desired output language (C), as shown in the following code snippet:

```
t = "c -keys=arm_cpu,cpu -mcpu=cmsis-nn,cortex-m4"
artemis_nano = tvm.target.Target(t)
```

Once you have specified the target device, you can build the Arduino project for the SparkFun Artemis Nano using TVM by following the steps presented for Arduino Nano and Raspberry Pi Pico. In the Colab notebook located in the Chapter11 folder in the GitHub repository, we have provided the code block to generate the Arduino project for this platform. Specifically, the code block is under the section titled *Generate code with TVM for the SparkFun Artemis Nano*.

This recipe concludes our experiments with TVM on Arduino-compatible platforms. Yet, our experiments will continue in the following recipes, focusing on a new, advanced processor designed explicitly for ML workloads on microcontrollers. This processor is the **Micro-Neural Processing Unit (microNPU)**.

In the upcoming recipe, we will introduce this processor and install a virtual device in Colab to play with it!

Installing the Fixed Virtual Platform (FVP) for the Arm Corstone-300

So far, our focus has been primarily on Arduino boards. However, TVM can generate code for various platforms, including those with the **Arm Ethos-U55** processor, the first microNPU designed by Arm to extend the ML capabilities of Cortex-M-based devices.

In this recipe, we will give more details on the computational capabilities of the Arm Ethos-U55 microNPU and install the FVP model for the **Arm Corstone-300** platform. This virtual device will allow us to see this new processor in action without needing a physical device.

Getting ready

The FVP model for the Arm Corstone-300 platform (<https://developer.arm.com/tools-and-software/open-source-software/arm-platforms-software/arm-ecosystem-fvps>) is a free-of-charge virtual platform, based on an **Arm Cortex-M55** CPU and **Ethos-U55** microNPU.

Arm Ethos-U55 (<https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-u55>) is a specialized processor for ML inference that works in conjunction with a Cortex-M CPU, as shown in *Figure 11.14*:

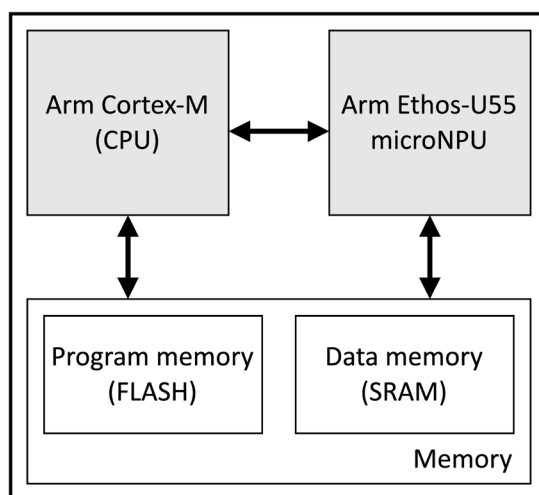


Figure 11.14: Microcontroller with an Arm Cortex-M CPU and Ethos-U55 microNPU

Arm Ethos-U55 has been designed to efficiently compute most of the elementary operations that we may find in quantized 8-bit/16-bit deep neural networks, such as the **multiply and accumulate (MAC)** at the heart of convolution, fully connected, and depthwise convolution layers.

When using the microNPU to execute the ML inference, the CPU is responsible for driving the ML workload. In addition, if the microNPU encounters an unsupported operator that it cannot perform, the CPU will take over and handle the computation as a fallback mechanism.

Figure 11.15 lists some of the operators supported by Arm Ethos-U55:

Add/Sub/Mul	Average pooling	Convolution 2D	De-convolution
Deptwise convolution 2D	Fully connected	LSTM/GRU	Max pooling
ReLu/ReLu6/TanH Sigmoid	Reshape	Softmax	Many more...

Figure 11.15: Table listing some of the operators supported by the Arm Ethos-U55 microNPU

From the perspective of microcontroller programming, the process of providing the model as a C/C++ program remains unchanged. The weights, biases, and quantization parameters can still be stored in program memory, while the input and output tensors can be kept in SRAM. Therefore, nothing changes from what we have seen in the previous chapters regarding memory locations for the ML parameters and the input/output tensors.

Once the program has been uploaded into the microcontroller, the driver offloads the computation to the microNPU by dispatching a sequence of commands to it, also known as a **command stream**. These commands instruct the microNPU on which operations to execute and the memory locations to read and write data.

Once all commands are executed and the output is stored in the user-defined memory region, the microNPU sends an interrupt to the CPU, signaling the completion of the execution.

How to do it...

Reopen the Colab notebook and follow the following steps to install the FVP model for the Arm Corstone-300 platform:

Step 1:

Download the FVP model for the Arm Corstone-300 platform, using the following `wget` command:

```
!wget https://developer.arm.com/-/media/Arm%20Developer%20Community/
Downloads/OSS/FVP/Corstone-300/FVP_Corstone_SSE-300_11.22_20_Linux64.
tgz?rev=018659bd574f4e7b95fa647e7836ccf4&hash=22A79103C6FA5FFA7AFF3B
E0447F3FF9
```

The FVP model for the Arm Corstone-300 platform can also be downloaded manually from the **Arm Ecosystem FVPs** web page (<https://developer.arm.com/tools-and-software/open-source-software/arm-platforms-software/arm-ecosystem-fvps>).

Particularly, on the Arm Ecosystem FVPs web page, you should click on **Corstone-300 Ecosystem FVPs** and then the **Download Linux** button, as shown in *Figure 11.16*:

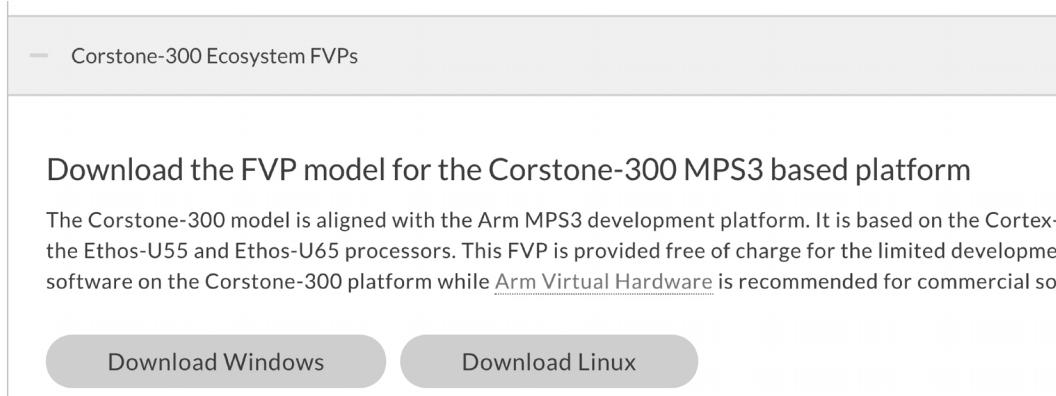
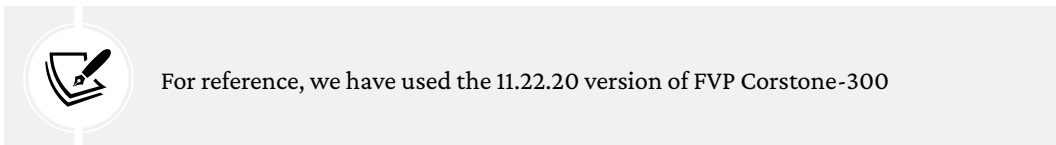


Figure 11.16: The Corstone-300 FVP section

Once the download of the .tgz file is finished, you should import it into Colab.



Step 2:

Decompress the .tgz file, and make the FVP_Corstone_SSE-300.sh executable with the following shell commands:

```
!tar -xvzf FVP_Corstone_SSE-300_11.22_20_Linux64.tgz?rev=018659bd574f4e7b95fa647e7836ccf4
!chmod +x FVP_Corstone_SSE-300.sh
```

Step 3:

Run the `FVP_Corstone_SSE-300.sh` script, using the following command:

```
!./FVP_Corstone_SSE-300.sh \  
  --i-agree-to-the-contained-eula \  
  --no-interactive
```

After executing the preceding command, the Corstone-300 FVP binaries will be placed in the `/usr/local/FVP_Corstone_SSE-300` directory.

Once the installation is completed, you will see the `Installation completed successfully` message in the output log.

Step 4:

Add the path of the Corstone-300 binaries to the `$PATH` environment variable:

```
os.environ['PATH'] += ':/usr/local/FVP_Corstone_SSE-300/models/Linux64_  
GCC-9.3'
```

Step 5:

Check whether the Corstone-300 binaries are installed, by printing the version info of the `FVP_Corstone_SSE_Ethos-U55` virtual platform with the Arm Ethos-U55 on the output log:

```
!FVP_Corstone_SSE-300_Ethos-U55 --version
```

If the `$PATH` environment variable has been correctly updated, the preceding command should return the following message in the output log:

```
Fast Models [11.22.20 (Jul 13 2023)]  
Copyright 2000–2023 ARM Limited.  
All Rights Reserved.
```

Figure 11.17: FVP Corstone-300 version

If you can see a similar message on the log, it indicates that the virtual hardware with Arm Cortex-M55 CPU and Ethos-U55 microNPU is installed and ready to be used.

There's more...

In this recipe, we explored the capabilities of the Arm Ethos-U55 microNPU and learned how to install the FVP model for the Arm Corstone-300 platform, featuring this processor.

However, what microcontrollers feature the Arm Ethos-U55 microNPU apart from this virtual device? One of the microcontrollers available in the market with this microNPU is the **Alif Semiconductor Ensemble** (<https://alifsemi.com/products/ensemble/>), capable of unlocking real-time tasks for speech recognition and face detection, considered unattainable just a few years ago on microcontrollers.

Having installed the FVP model for the Arm Corstone-300 to run our experiments on the Arm Ethos-U55, let's see how we can generate code for this processor using TVM.

In the upcoming recipe, we will discover how we can accomplish this task by employing an alternative tool provided by TVM: TVMC.

Code generation with TVMC for Arm Ethos-U55

Until now, we have generated code with TVM using its native Python interface. However, TVM provides an alternative approach through the **TVMC** tool, which allows us to execute the same actions as the Python interface but from the command line.

In this recipe, we will show how you can use this tool in Colab to produce the MLF model, containing the generated code to run the CIFAR-10 model inference on the Arm Ethos-U55.

Getting ready

The native TVM Python interface proved to be convenient to use to generate code for a desired target. However, the framework also offers an additional tool to simplify code generation. This tool is TVMC, a command-line driver that exposes the same features of the Python API in a single command line.

At this point, you may wonder: *where can we find the TVMC tool?*

TVMC is bundled with the TVM Python installation, and you can invoke it using the following shell command:

```
$ python -m tvm.driver.tvmc compile <options>
```

In the preceding command, `<options>` refers to the arguments required to generate code for the target, which will be the following:

- The target device
- The compilation options
- The input model



To learn more about TVMC, we recommend reading the official TVM tutorial: https://tvm.apache.org/docs/tutorial/tvmc_command_line_driver.html.

The code generation for Arm Ethos-U55 with TVM requires a few Python libraries to be pip-installed. These Python libraries are `ethos-u-vela` (<https://pypi.org/project/ethos-u-vela/>) and `tfllite`, which can be installed with the following pip command:

```
$ pip install ethos-u-vela==3.8.0 tfllite==2.10.0
```

As evident from the previous command, the version of `tfllite` matches the one we installed earlier. Therefore, there's no need to install it again within this recipe.

How to do it...

Continue working in Colab, and follow the following steps to generate the C code to run the CIFAR-10 model inference on Arm Ethos-U55, using TVMC:

Step 1:

Create a folder called `ethosu_prj` to keep all files required for the Arm Ethos-U55 project we aim to build with TVM:

```
!mkdir ethosu_prj
```

Within this directory, create a folder called `tvm_code` to hold the C code generated by TVM:

```
!mkdir ethosu_prj/tvm_code
```

Step 2:

Install the `ethos-u-vela` Python library required to produce the C code for Arm Ethos-U55:

```
!pip install ethos-u-vela==3.8.0
```

Step 3:

Use the following TVMC command to generate the C code to run the CIFAR-10 model inference on the Arm Ethos-U55:

```
!python -m tvm.driver.tvmc compile \  
  --target=ethos-u,cmsis-nn,c \  
  --target-ethos-u-accelerator_config=ethos-u55-256 \  
  --target-cmsis-nn-mcpu=cortex-m55 \  
  \
```

```

--target-c-mcpu=cortex-m55 \
--runtime=crt \
--executor=aot \
--executor-aot-interface-api=c \
--executor-aot-unpacked-api=1 \
--pass-config tir.disable_vectorize=1 \
--pass-config tir.usmp.enable=1 \
--pass-config tir.usmp.algorithm=hill_climb \
--pass-config tir.disable_storage_rewrite=1 \
--output-format=mlf \
./cifar10.tflite

```

In the preceding code, the arguments passed to TVMC are the following:

- `--target="ethos-u-accelerator_config=ethos-u55-256, c"`: This specifies the target processor for the ML inference. As evident from the target name, Arm Ethos-U55 is selected. However, there is a `_256` suffix attached. This suffix is present because we may have different versions of Arm Ethos-U55, where the main difference is dictated by the number of MACs the compute engine can perform. In the context of Corstone-300, we have an Arm Ethos-U55 with 256 MACs, which is why we specify `ethos-u55-256` for the target device.
- `--target-c-mcpu=cortex-m55`: This tells the target CPU to execute the unsupported layers on the microNPU.
- `--runtime=crt`: This specifies the runtime type. In this case, we must specify the C runtime (`crt`), since we will run the application on a bare-metal platform.
- `--executor=aot`: This instructs TVM to use the AoT executor.
- `--executor-aot-interface-api=c`: This specifies the interface type for the AoT executor. We pass the `c` option because we generate C code.
- `--pass-config tir.disable_vectorize=1`: This tells TVM to turn off the code vectorization, since C has no native vectorized types.
- `--pass-config tir.usmp.enable=1`: This enables the USMP compiler optimization to perform memory planning of all tensors and obtain optimal memory utilization.
- `--pass-config tir.usmp.algorithm=hill_climb`: This specifies the algorithm to be used by USMP.
- `--pass-config tir.disable_storage_rewrite=1`: This turns off the storage rewrite.
- `--output-format=mlf`: This specifies the output generated by TVM.
- `cifar10.tflite`: This is the input model to compile to C code.

After a few seconds, TVM will generate a TAR package file, named `module.tar`, in the same directory where you executed the TVMC command.

Step 4:

Extract the contents of the `module.tar` file into the `ethosu_prj/tvm_code/` directory, designated to keep the files present in the MLF package:

```
!tar -C ethosu_prj/tvm_code -xvf module.tar
```

Now, the code for Arm Ethos-U55 is ready and can be integrated into an application running on the Corstone-300 FVP.

There's more...

In this recipe, we learned how to use the TVMC tool to generate C code to run the CIFAR-10 model inference on the Arm Ethos-U55 microNPU.

As we saw earlier, one of the software dependencies for this task is the `ethos-u-vela` Python package. This package is for the **Vela** compiler, which aims to transform the TensorFlow Lite model into an optimized version, capable of running an Arm Ethos-U microNPU.

The optimized model obtained after this transformation will integrate TensorFlow Lite custom operators for those parts of the model that can be accelerated via the microNPU. Parts that this processor cannot accelerate remain unaltered and executed on the Cortex-M series CPU, using functions from the CMSIS-NN library (<https://github.com/ARM-software/CMSIS-NN>).

To learn more about the features of the Vela compiler, you can refer to its official documentation available at the following link: <https://developer.arm.com/documentation/101888/0500/NPU-software-overview/NPU-software-tooling/The-Vela-compiler>.

Now that we have successfully generated code for the Arm Ethos-U55 microNPU, our attention can shift toward application development. However, since we do not target an Arduino platform, how can we create and compile an application for this processor without Arduino CLI?

In the upcoming recipe, we will discover how to accomplish this task, by installing the software dependencies to build an application for the virtual Arm Ethos-U55 microNPU.

Installing the software dependencies to build an application for the Arm Ethos-U microNPU

The code generated by TVM for Arm Ethos-U55 cannot be compiled with Arduino CLI, as the target device is not an Arduino-based platform. Therefore, this recipe will guide you in preparing the required dependencies to build the application for the Arm Corstone-300 platform.

Getting ready

To build the code generated by TVM for Corstone-300, the following components are required:

- A **compiler** to produce the application binary for the Arm Cortex-M55 CPU
- The **Ethos-U driver**, which is the driver to offload the computation from the Arm Cortex-M CPU to the Arm Ethos-U55
- The **Ethos-U platform**, which provides a basic driver for the device peripherals, such as the UART and Timer
- The **CMSIS library**, which provides a collection of optimized ML and **digital signal processing (DSP)** functions

The following subsections will provide further details about the compiler, the Ethos-U driver and Ethos-U platform.

The GNU Arm Embedded Toolchain

In the context of embedded programming, the compiler is commonly called a **cross-compiler** because the target CPU (for example, Arm Cortex-M55) differs from the CPU of the computer used to build the application (for example, **x86-64**).

To cross-compile for Arm Cortex-M55, the **GNU Arm Embedded Toolchain** (<https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads>) is required, which offers a free collection of programming tools that includes the compiler, linker, debugger, and software libraries. The toolchain is available for various **operating systems (OSs)**, such as Linux, Windows, and macOS.

The Ethos-U driver

The Cortex-M55 CPU needs the Arm Ethos-U driver (<https://review.mlplatform.org/plugins/gitiles/ml/ethos-u/ethos-u-core-driver/>) to execute command streams on the Ethos-U microNPU. The driver is OS-agnostic, meaning it does not use any OS primitives, such as queues or mutexes. Therefore, it can be cross-compiled for any supported Cortex-M CPU and work with any **real-time operating system (RTOS)**.

The Ethos-U platform

The aim of the Ethos-U platform (<https://review.mlplatform.org/plugins/gitiles/ml/ethos-u/ethos-u-core-platform/>) is to provide a basic driver for the virtual device peripherals on Corstone-300, such as the serial or timer. As you can guess, we will need the serial peripheral to transmit the result of the model inference on the output log. Therefore, the Ethos-U platform provides the driver to allow the use of this peripheral with a straightforward `printf()` function.

The Ethos-U platform project was born primarily for guidance, demonstrating how to run inference on an Arm Ethos-U compatible platform. Since this project supports only a limited number of platforms, developers must replace this driver to make the application functional on their target device.

How to do it...

Continue working in Colab, and follow the following steps to install the GNU Arm Embedded Toolchain, Ethos-U core, Ethos-U platform, and CMSIS-NN library required to build our application for Corstone-300:

Step 1:

Create a new folder to hold the GNU Arm Embedded Toolchain binaries (for example, `toolchain`) into the directory created for the Arm Ethos-U55:

```
!mkdir ethosu_prj/toolchain
```

Step 2:

Download the GNU Arm Embedded Toolchain with the `curl` tool and uncompress the downloaded file into the `ethosu_prj/toolchain` folder:

```
!curl --retry 64 -sSL 'https://developer.arm.com/-/media/Files/downloads/gnu-rm/10-2020q4/gcc-arm-none-eabi-10-2020-q4-major-x86_64-linux.tar.bz2?revision=ca0cbf9c-9de2-491c-ac48-898b5bbc0443&la=en&hash=68760A8AE66026BCF99F05AC017A6A50C6FD832A' | tar -C ethosu_prj/toolchain --strip-components=1 -jx
```

This operation can take some minutes, depending on your internet connection speed. After the toolchain has been successfully uncompressed, you will find the GNU Arm Embedded Toolchain binaries in the `ethosu_prj/toolchain/bin` folder.

Step 3:

Add the path of the GNU Arm Embedded Toolchain binaries to the \$PATH environment variable:

```
os.environ['PATH'] += ':/content/ethosu_prj/toolchain/bin/'
```

Step 4:

Verify the correct installation of the GNU Arm Embedded Toolchain by printing the list of supported CPUs, using the following command:

```
!arm-none-eabi-gcc -mcpu=.
```

The returned list of supported CPUs should include the Cortex-M55 CPU, as shown in *Figure 11.18*:

```
arm-none-eabi-gcc: error: unrecognized -mcpu target: . arm-none-eabi-gcc: note:
valid arguments are: arm8 arm810 strongarm strongarm110 fa526 fa626 arm7tdmi
arm7tdmi-s arm710t arm720t arm740t arm9 arm9tdmi arm920t arm920 arm922t arm940t
ep9312 arm10tdmi arm1020t arm9e arm946e-s arm966e-s arm968e-s arm10e arm1020e
arm1022e xscale iwmmxt iwmmxt2 fa606te fa626te fmp626 fa726te arm926ej-s
arm1026ej-s arm1136j-s arm1136jf-s arm1176jz-s arm1176jzf-s mpcorenovfp mpcore
arm1156t2-s arm1156t2f-s cortex-m1 cortex-m0 cortex-m0plus cortex-m1.small-
multiply cortex-m0.small-multiply cortex-m0plus.small-multiply generic-armv7-a
cortex-a5 cortex-a7 cortex-a8 cortex-a9 cortex-a12 cortex-a15 cortex-a17
cortex-r4 cortex-r4f cortex-r5 cortex-r7 cortex-r8 cortex-m7 cortex-m4 cortex-m3
marvell-pj4 cortex-a15.cortex-a7 cortex-a17.cortex-a7 cortex-a32 cortex-a35
cortex-a53 cortex-a57 cortex-a72 cortex-a73 exynos-m1 xgene1
cortex-a57.cortex-a53 cortex-a72.cortex-a53 cortex-a73.cortex-a35
cortex-a73.cortex-a53 cortex-a55 cortex-a75 cortex-a76 cortex-a76ae cortex-a77
neoverse-n1 cortex-a75.cortex-a55 cortex-a76.cortex-a55 neoverse-v1 neoverse-n2
cortex-m23 cortex-m33 cortex-m35p cortex-m55 cortex-r52 arm-none-eabi-gcc:
```

Figure 11.18: The list of supported CPUs should include cortex-m55

Step 5:

Clone the Ethos-U driver repository into the ethosu_prj/driver directory:

```
!git clone \
  "https://review.mlplatform.org/ml/ethos-u/ethos-u-core-driver" \
  ethosu_prj/driver \
  --branch 21.11
```

The preceding git command clones the 21.11 Ethos-U driver release.

Step 6:

Clone the Ethos-U platform repository into the `ethosu_prj/platform` directory:

```
!git clone \
  "https://review.mlplatform.org/ml/ethos-u/ethos-u-core-platform" \
  ethosu_prj/platform \
  --branch 21.11
```

The preceding git command clones the 21.11 Ethos-U platform release.

Step 7:

Clone the CMSIS library repository into the `ethosu_prj/cmsis` directory:

```
!git clone \
  "https://github.com/ARM-software/CMSIS_5.git" \
  ethosu_prj/cmsis \
  --branch 5.9.0
```

The preceding git command clones the 5.9.0 CMSIS release.

At this point, we are all set to proceed with the application development for the Arm Ethos-U55.

There's more...

In this recipe, we learned what software dependencies are required to build an application for the Arm Ethos-U55 microNPU, using the code generated by TVM.

After completing this recipe, you might wonder whether a Dockerfile exists that can automate the installation of all these necessary software dependencies. Indeed, a Dockerfile exists and can be found in the TVM repository at the following link: https://github.com/apache/tvm/blob/v0.11.1/docker/Dockerfile.ci_cortexm.

The primary reason for not using the Dockerfile was to provide transparency regarding the minimum dependencies required to build applications for the Arm Ethos-U55 platform, and to explain their necessity and source code origin. It is important to note that inside the Dockerfile, you can find a multitude of software dependencies, but not all of them are essential for our specific purpose.

Having installed the necessary tools and software dependencies for our project, we are prepared to deploy the model on the virtual Arm Ethos-U55 microNPU.

In the upcoming final recipe, we will demonstrate how to run the CIFAR-10 model inference using the code generated by TVM on the FVP model, for the Arm Corstone-300 platform.

Running the CIFAR-10 model inference on the Arm Ethos-U55 microNPU

Now that all the necessary tools and software libraries are installed, our final step involves building the application with code generated by TVM for the Arm Ethos-U55 microNPU, on the Corstone-300 FVP.

Although it seems there is still a lot left to do, this recipe offers a solution to simplify the remaining technicalities.

In this recipe, we will show you how to modify the Ethos-U prebuilt sample available in the TVM source code to run the CIFAR-10 inference, on the Arm Ethos-U55. After making the necessary modifications, we will compile the application, using the provided Makefile and Linker scripts from the prebuilt sample, and run the compiled application on the Corstone-300 FVP.

Getting ready

The prebuilt sample considered in this recipe is available in the TVM source code within the `tvm/apps/microtvm/ethosu` directory (<https://github.com/apache/tvm/tree/v0.11.1/apps/microtvm/ethosu>).



We recommend you refer to the TVM v0.11.1 release (<https://github.com/apache/tvm/tree/v0.11.1/>), as we can only ensure that the Arm Ethos-U sample provided for this TVM release is compatible with the software dependencies installed in the previous recipe.

This sample has been crafted to show how to perform a single image classification inference with the MobileNet v2 model on Arm Ethos-U55. Inside the sample folder, you will find the following:

- Application source code located in the `include/` and `src/` subdirectories
- Scripts to compile the demo for Corstone-300 FVP (`Makefile`, `arm-none-eabi-gcc.cmake`, and `corstone300.ld`)
- Python scripts responsible for generating the input, output, and label C header files (`convert_image.py` and `convert_labels.py`)
- Script to run the demo on Corstone-300 FVP (`run_demo.sh`)

From the preceding files, we will only need the following for our application:

- The application source (`src/` and `include/`)

- The C Makefile (`none-eabi-gcc.cmake`)
- The Linker script (`corstone300.ld`).

The **Makefile** found in that directory will not be necessary because we provide an alternative one, developed by us to streamline the build process.

The `demo_bare_metal.c` file in the `src/` directory (https://github.com/apache/tvm/blob/v0.11.1/apps/microtvm/ethosu/src/demo_bare_metal.c) contains the application that invokes the TVM runtime to perform the image classification. Our goal is to edit this file to run the CIFAR-10 model, using the input test image provided in the `input.h` header file. For this scope, the following changes to the file are required:

- Removing the inclusion of unnecessary files for our applications, which are the input test image (`inputs.h`), output tensor (`outputs.h`), and labels (`labels.h`) required for the MobileNet v2 application.
- Including the header file storing the input test image for the CIFAR-10 model (`input.h`).
- Implementing the `dequantize()` and `argmax()` functions like we did to deploy the CIFAR-10 model on the Arduino Nano and Raspberry Pi Pico.
- Allocating the output tensor in DDR, capable of holding 10 values of the `int8_t` type.
- Providing the correct input and output to the TVM runtime.
- Adding the code to check whether the output classification is correct.

However, before we can start showing how to finalize the application, one aspect needs to be discussed first – the memory capability of Corstone-300 FVP. In other words, how much memory is available on this target device?

An overview of memory capabilities on Corstone-300 FVP


The Corstone-300 FVP memory system goes beyond the standard program and data memory configuration commonly seen in microcontrollers.

This virtual device offers five distinct memory types, as shown in *Figure 11.19*:

Memory	Size	microNPU access
ITCM	512KB	No
DTCM	512KB	No
SSE-300 SRAM	2MB	Yes
Data SRAM	2MB	Yes
DDR	32MB	Yes

Figure 11.19: Memory system on the Corstone-300 FVP

The previous table shows that the five memories in the Corstone-300 FVP differ in capacity (the **Size** column) and Arm Ethos-U55 access permission (the **microNPU access** column). Therefore, not all memories can be accessed by the microNPU. For example, the microNPU cannot access the **Instruction Tightly Coupled Memory (ITCM)** and **Data Tightly Coupled Memory (DTCM)**, equivalent to the program and data memory, respectively. As a result, *we need to pay attention to where we store the model input and output tensors* because their content cannot be read or written by the microNPU if wrongly placed in either ITCM or DTCM.



In this project, we will store the input and output tensors in DDR.

To ensure that the input and output tensors are in memory spaces accessible by the Arm Ethos-U55, it is necessary to specify the **memory section attribute** when declaring the memory storages, as shown in the following C code:

```
int8_t K[4] __attribute__((section("<memory_name>")))
```

The preceding code snippet showed an example of allocating an `int8_t` array named `K` in the `<memory_name>` space. In our case, `<memory_name>` should be replaced with `ethosu_scratch`, as it is the name assigned to the DDR memory in the Linker script used for Corstone-300 FVP.



The Linker script used in this project comes from the Arm Ethos-U TVM sample and is available at the following link: <https://github.com/apache/tvm/blob/v0.11.1/apps/microtvm/ethosu/corstone300.ld>.

In addition to specifying the target name, we recommend you include the **byte alignment** in the memory section attribute as well, as shown in the following example:

```
int8_t K[4] __attribute__((section("ethosu_scratch"),
                             aligned(16)))
```

The byte alignment is required to ensure correct read/write memory accesses. Also, in this case, the byte alignment can be found in the Linker script and is equal to 16 for the DDR memory.

How to do it...

Continue working in Colab, and follow the following steps to build and run the CIFAR-10 inference with the Arm Ethos-U55 on Corstone-300 FVP:

Step 1:

Copy the sample code to run the MobileNet v2 model inference on Arm Ethos-U55 from the TVM repository. To do so, clone the TVM v0.11.1 release Git repository:

```
!git clone \
  "https://github.com/apache/tvm.git" \
  --branch v0.11.1
```

Then, copy the `src/` and `include/` folders located in the `tvm/apps/microtvm/ethosu` directory into the `ethosu_prj/` folder:

```
!cp -r tvm/apps/microtvm/ethosu/src ethosu_prj/
!cp -r tvm/apps/microtvm/ethosu/include ethosu_prj/
```

Step 2:

Copy the build scripts (`arm-none-eabi-gcc.cmake` and `corstone300.ld`), located in the `tvm/apps/microtvm/ethosu` directory, into the `ethosu_prj/` folder:

```
!cp -r tvm/apps/microtvm/ethosu/arm-none-eabi-gcc.cmake ethosu_prj/
!cp -r tvm/apps/microtvm/ethosu/corstone300.ld ethosu_prj/
```

Step 3:

Copy the `input.h` header file, containing the normalized and quantized input test image for the CIFAR-10 model, into the `ethosu_prj/include` directory:

```
!cp input.h ethosu_prj/include
```

Step 4:

Specify the memory section attribute (`ethosu_scratch`) for the input test image stored in the `input.h` header file. For this scope, you can conveniently use the following `sed` command:

```
os.environ['src_txt'] = 'g_test\[\'\'
os.environ['dst_txt'] = 'g_test\[\'\' __attribute__((section("ethosu_
scratch"), aligned(16)))'

!sed -i "s/${src_txt}/${dst_txt}/g" \
    ethosu_prj/include/input.h
```

Step 5:

Open the `demo_bare_metal.c` file in the `ethosu_prj/src/` directory by double-clicking on the file. Within the file, remove the inclusion of unnecessary files for our applications (`inputs.h`, `outputs.h`, and `labels.h`). Then, include the header file containing the input test image for the CIFAR-10 model (`input.h`):

```
#include "input.h"
```

After the inclusion of the input test image, include the `tvmgen_default.h` header file:

```
#include <tvmgen_default.h>
```

Please note that this header file is not included in the sample. However, it is highly recommended to have it, as it contains the definitions of the TVM runtime for the input and output tensors.



Omitting the inclusion of the `tvmgen_default.h` file may result in compilation errors.

Step 6:

In the `demo_bare_metal.c` file, define the output quantization parameters of the CIFAR-10 model as global variables:

```
float out_scale = 0.10305877029895782;
int32_t out_zero_point = 20;
```

Then, implement the `dequantize()` function:

```
void dequantize(int8_t* src, float* dst, int32_t len) {
    for(int32_t i = 0; i < len; ++i) {
        dst[i] = out_scale * (src[i] - out_zero_point);
    }
}
```

Finally, implement the `argmax()` function:

```
int32_t argmax(float* src, int32_t len) {
    int32_t max_idx = 0;
    float max_score = src[0];

    for(int32_t i = 1; i < len; ++i) {
        if(src[i] > max_score) {
            max_score = src[i];
            max_idx = i;
        }
    }
    return max_idx;
}
```

Step 7:

In the `demo_bare_metal.c` file, allocate the output tensor in the DDR memory globally:

```
int8_t out_q8[10] __attribute__((section("ethosu_scratch"), aligned(16)));  
float out_f32[10] __attribute__((section("ethosu_scratch"), aligned(16)));
```

The memory in DDR cannot be declared in a local scope, which is why we allocate the output tensor globally.

Step 8:

In the `demo_bare_metal.c` file, navigate to the declaration of the `tvmgen_default_outputs` struct.

The `tvmgen_default_outputs` struct is used to pass the output tensor to the TVM runtime. The field name of this struct (`MobilenetV2_Predictions_Reshape_11`) depends on the name of the output node in the TensorFlow Lite model. Consequently, we must replace the field's name to work with our model. The field's name is defined inside the `ethosu_prj/tvm_code/codegen/host/include/tvmgen_default.h` file and, for our pre-trained CIFAR-10 model, corresponds to `StatefulPartitionedCall_0`:

```
struct tvmgeng_default_outputs outputs = {  
    .StatefulPartitionedCall_0 = out_q8,  
};
```

Step 9:

In the `demo_bare_metal.c` file, navigate to the declaration of the `tvmgen_default_inputs` struct.

The `tvmgen_default_inputs` struct is used to pass the input tensor to the TVM runtime. Therefore, the field's name must be replaced with our model's input name. Also, in this case, the field's name is defined inside the `ethosu_prj/tvm_code/codegen/host/include/tvmgen_default.h` file, and, for the pre-trained CIFAR-10 model, corresponds to `serving_default_input_1_0`:

```
struct tvmgeng_default_inputs inputs = {  
    .serving_default_input_1_0 = g_test,  
};
```

Step 10:

In the `demo_bare_metal.c` file, remove the code after the `ethosu_release_driver(driver)` function that is related to the calculation of the max value index up to the `printf("EXITTHESIM\n")`. The `printf("EXITTHESIM\n")` must not be deleted.

Then, write the code to verify whether the classification result is correct:

```
dequantize(out_q8, out_f32, 10);

int32_t max_idx = argmax(out_f32, 10);

if(max_idx == g_test_ilabel) {
    printf("The image has been correctly classified\n");
}
else {
    printf("Classification FAILED\n");
}
```

Step 11:

Download the Makefile and store it in the `ethosu_prj/` folder, using the following `wget` command:

```
!wget https://raw.githubusercontent.com/PacktPublishing/TinyML-
Cookbook_2E/main/Chapter11/Assets/Makefile \
-P ethosu_prj/
```

Step 12:

Compile the application using the following `make` command:

```
!cd ethosu_prj; make
```

The application binary, named `demo`, will be stored in the directory pointed to by the `BUILD_DIR` variable in the Makefile script.

Step 13:

Run the demo executable on the Corstone-300 FVP using the following command:

```
!cd ethosu_prj; FVP_Corstone_SSE-300_Ethos-U55 \
-C cpu0.CFGDTCMSZ=15 \
-C cpu0.CFGITCMSZ=15 \
-C mps3_board.uart0.out_file="-\ " \
```

```
-C mps3_board.uart0.shutdown_tag="\EXITTHESIM\" \
-C mps3_board.visualisation.disable-visualisation=1 \
-C mps3_board.telnetterminal0.start_telnet=0 \
-C mps3_board.telnetterminal1.start_telnet=0 \
-C mps3_board.telnetterminal2.start_telnet=0 \
-C mps3_board.telnetterminal5.start_telnet=0 \
-C ethosu.extra_args="--fast" \
-C ethosu.num_macs=256 ./build/demo
```

From the previous command, pay attention to the `ethosu.num_macs=256` argument. This option refers to the number of MACs in the compute engine of the Arm Ethos-U55 microNPU and must match what was specified in TVM when compiling the TensorFlow Lite model.

If the model can correctly classify the input test sample, you will see the `The image has been correctly classified` message in the output log, as shown in *Figure 11.20*:

```
D: ethosu release driver(): NPU driver handle (
The image has been correctly classified
EXITTHESIM
Info: /OSCI/SystemC: Simulation stopped by use:
```

Figure 11.20: Output log after successfully classifying the input image

And...that's it! With this last recipe but first application on Arm Ethos-U55, you are ready to make even more intelligent tinyML solutions on Cortex-M-based microcontrollers!

There's more...

In this final recipe, we learned how to build an application to run the model inference on the Arm Ethos-U55 microNPU, using the code generated by TVM.

What we have implemented here can be extended easily to test even more sophisticated ML models on this new processor. For example, you might consider building a speech-to-text application using the **wav2letter** model provided in the **Arm model zoo** (https://github.com/ARM-software/ML-zoo/tree/master/models/speech_recognition/tiny_wav2letter/tflite_int8).

Summary

In this chapter, we have explored the capabilities of TVM, a deep learning compiler capable of generating code to run model inference on various target devices, including the latest Arm Ethos-U55 microNPU.

In the first part, we delved into this framework to deploy the CIFAR-10 model on the Arduino Nano and Raspberry Pi Pico. Here, we discussed the TVM Python API to generate the code for model inference and showed the steps to build and run the Arduino sketch on the microcontrollers using Arduino CLI.

Following the successful model deployment on the Arduino Nano and Raspberry Pi Pico, we moved our attention to a new and advanced processor: the microNPU.

In this second part, we introduced the Arm Ethos-U55 microNPU and installed the FVP model for the Arm Corstone-300 platform to play with this processor without needing a physical device.

After installing the virtual device, we generated the code to run the CIFAR-10 model inference on the microNPU using TVMC, a command-line tool provided by TVM.

Then, we installed the software dependencies to build an application for Corstone-300 FVP.

Finally, we developed the application with the code generated by TVM and executed on the virtual device.

Until now, our focus has been only on model inference and primarily centered around deep learning models.

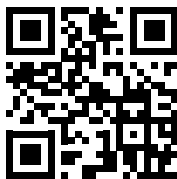
However, is it possible to train a model on microcontrollers? And can we deploy more generic ML algorithms, such as **decision tree** or **random forest** on microcontrollers?

In the upcoming final chapter, we will address these questions to take your future tinyML projects to the next level!

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



12

Enabling Compelling tinyML Solutions with On-Device Learning and scikit-learn on the Arduino Nano and Raspberry Pi Pico

We are now ready for our final chapter of this practical learning journey into tinyML.

If you have made it this far, I bet you have a myriad of questions in mind to help you start or continue building compelling applications with **machine learning (ML)** on microcontrollers. Therefore, this chapter has a different format, seeking to answer three questions you might be pondering.

The first question will delve into the feasibility of training models directly on microcontrollers. How can we have **on-device learning** on microcontrollers? And what groundbreaking applications can this unlock? In this part, we will discuss the **backpropagation** algorithm to train a shallow neural network. We will also show how to use the **CMSIS-DSP** library to accelerate its implementation on any microcontroller with an Arm Cortex-M CPU.

After discussing on-device learning, we will tackle another problem: deploying generic ML algorithms, such as the **decision tree** or **random forest** algorithms. Those familiar with the ML sphere know that neural networks are not the only ML algorithms, and frameworks like **scikit-learn** can train these alternative algorithms.

The question then becomes: how can we deploy these models to microcontrollers? In this second part, we will demonstrate how to deploy generic ML algorithms trained with scikit-learn using the **emlearn** open-source project.

The final question we will answer is about powering microcontrollers with batteries. In the first chapter of the book, we discussed the importance of low power consumption, which is critical for battery-powered applications. Therefore, how can we power our microcontrollers with batteries?

These three questions will be uncovered in this final chapter with the hope you can unlock new use cases and make the “things” around us even more intelligent.

In this chapter, we’re going to cover the following recipes:

- How can we train a model on microcontrollers?
- How can we deploy scikit-learn models on microcontrollers?
- How can we power microcontrollers with batteries?

Technical requirements

To complete all the practical recipes of this chapter, we will need the following:

- An Arduino Nano 33 BLE Sense
- A Raspberry Pi Pico
- A SparkFun RedBoard Artemis Nano (optional)
- A micro-USB data cable
- A USB-C data cable (optional)
- 1 x half-size solderless breadboard
- 2 x jumper wires
- 1 x 3 AA battery holder (Raspberry Pi Pico only)
- 1 x 4 AA battery holder (Arduino Nano only)
- 4 x AA batteries
- Laptop/PC with either Linux, macOS, or Windows
- Google Drive account



The source code and additional material are available in the Chapter12 folder in the GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/tree/main/Chapter12

How can we train a model on microcontrollers?

In every project presented in this book, we have discussed how to run model inference on microcontrollers and demonstrated that even a model like MobileNet v2 can be deployed on these devices. However, is it possible to train a neural network on microcontrollers?

In this recipe, we will answer this question and provide an example of training a simple neural network using **backpropagation** on the Arduino Nano and Raspberry Pi Pico with the CMSIS-DSP library.

The network will be trained to return the result of the following logical (**exclusive OR**) **XOR** and **NOT-AND (NAND)** operators:

a	b	result		a	b	result
0	0	0		0	0	1
0	1	1		0	1	1
1	0	1	=	1	0	1
1	1	0		1	1	0
XOR				NAND		

Figure 12.1: The logical XOR and NAND operators

As you can see from the preceding image, the result of the XOR operator is 1 when the binary inputs **a** and **b** are different. On the other hand, the output of the NAND operator is 1 when at least one of the binary inputs, **a** or **b**, is 0.

Getting ready

Training an ML model using the microcontroller’s CPU is doable because the CPU is a general-purpose processor and, as such, can be programmed to perform any task, including this one. However, when we say any task on microcontrollers, we know there is a caveat: the tasks’ memory usage must not exceed the device’s memory capacity. Certainly, the limited computational capabilities of the microcontroller can be another limiting factor, particularly for the latency. However, when dealing with **on-device model training (on-device learning)**, the latency requirement is generally less critical than for model inference.

Nowadays, on-device learning is an attractive topic in tinyML because it is *the ingredient that adds personalization to the deployed model without relying on internet access*.

As we know, the dataset is everything for ML because it is what makes or breaks an application. However, having the dataset before deployment is hard or impossible in some scenarios. For example, consider a smart thermostat that needs to predict the optimal time to turn the heating system on and off. Here, the decision depends not solely on the temperature setting but also on the user's daily routines and specific needs. As a result of these variables, a one-size-fits-all dataset for every user is impractical.

So, how can the device create a dataset while operating?

The solution is more straightforward than you might think. Sticking to our thermostat example, before this device becomes “smart,” it can work like a standard thermostat with manual controls for temperature adjustments. During this period, the device can create a dataset by logging the times, days, and temperatures when the heater is toggled on or off. After recording sufficient data over time, this dataset becomes ready for training. Naturally, to keep the model accurate, the device must continue to gather data and periodically retrain it.

Academia and industry are very active in investigating ML training on memory-constrained devices in the most efficient way. You just need to look at the program of the last **tinyML On Device Learning Forum**, held in September 2022 (<https://www.tinyml.org/event/on-device-learning/>) to realize how many solutions are already available.

Unfortunately, at the time of writing, it is hard to pick a particular solution as this field is constantly evolving. Therefore, to demonstrate that on-device learning is possible on microcontrollers, we will train a simple neural network using an already familiar algorithm: **backpropagation**.

This network will be trained on the Arduino Nano and Raspberry Pi Pico to implement the logical XOR and NAND operators. The following subsections will provide more details about the model architecture and backpropagation algorithm.

Solving the XOR and NAND problem using a shallow neural network

The problem we aim to solve in this recipe consists of training the model illustrated in the following figure using backpropagation, which allowed us to train all ML models presented in this book:

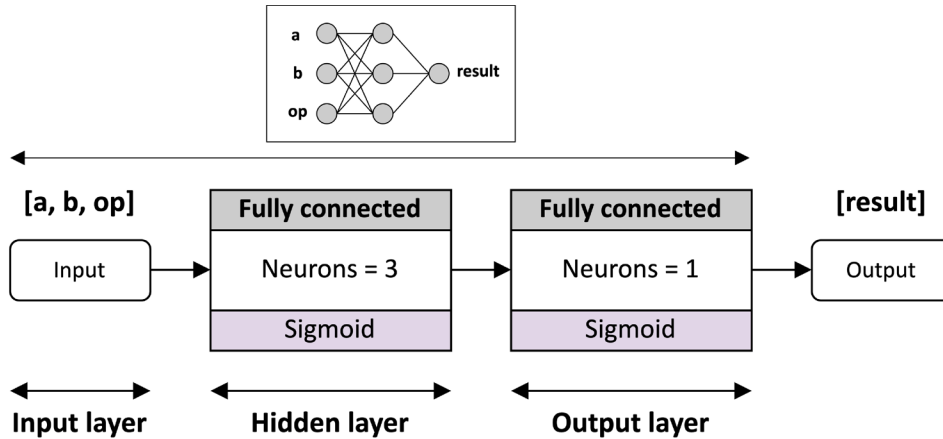


Figure 12.2: The neural network trained on the microcontrollers



The preceding neural network is called a **shallow neural network** because it only has one hidden layer between the input and output layers.

The **input layer** feeds three binary input values to the neural network, which are the following:

- **a** and **b**: the binary inputs of the logical XOR and NAND operators
- **op**: the function to perform. A value of 0 indicates the logical XOR operation, while 1 corresponds to the logical NAND operation.

The neural network architecture illustrated in *Figure 12.2* consists of two sequential fully connected layers, followed by a sigmoid function. The former is the **hidden layer** with three neurons, while the latter is the **output layer** with one neuron.

The following subsection will discuss the details of the backpropagation algorithm by referencing a Python implementation designed to train the neural network in *Figure 12.2*. This reference implementation, available at the following link, provides a class named `NN` that encompasses the methods to train the neural network and holds the weights of the fully connected layers: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Chapter12/PythonScripts/01_backpropagation.py.

Training a neural network with backpropagation

The *backpropagation algorithm* aims to adjust the neural network weights to minimize a loss function through an iterative process. This **loss function** is modeled to quantify the numerical difference between the actual output and predicted output across all training samples. Therefore, the lower it is, the more accurate the model's prediction.

In this recipe, we will use the **mean squared error (MSE)** as a loss function, which returns the average squared difference between the actual value and predicted value, as follows:

$$MSE = \frac{\sum_{i=0}^{N-1} (y_i - \hat{y}_i)^2}{N}$$

Where:

- *MSE* is the mean squared loss
- y_i is the actual value for the *i*-th input sample of the dataset
- \hat{y}_i is the predicted value for the *i*-th input sample of the dataset
- *N* is the number of samples in the dataset

Before training, the weights and biases of the neural network must be initialized with random values. In our Python implementation, this initialization is performed in the constructor of the NN class by taking random values from a uniform distribution over [0, 1) using the `np.random.rand()` function:

```
class NN:
    def __init__(self, input_sz,
                  fc0_out_sz,
                  fc1_out_sz):
        # Weights and biases fully connected 0
        self.w0 = np.random.rand(input_sz, fc0_out_sz)
        self.b0 = np.random.rand(1, fc0_out_sz)

        # Weights and biases fully connected 1
        self.w1 = np.random.rand(fc0_out_sz, fc1_out_sz)
        self.b1 = np.random.rand(1, fc1_out_sz)
```

As a result, the weights and biases are initialized once the NN object is created:

```
nn = NN(input_sz=3, out_fc0_sz=3, out_fc1_sz=1)
```

In the preceding code snippet, the `out_fc0_sz` and `out_fc1_sz` variables hold the number of neurons for the hidden and output layers, corresponding to the number of output values each layer returns.

After this initialization, the model training begins. This process involves several steps repeated a specified number of times, depending upon the following:

- The number of samples in the dataset.
- The number of **epochs**, which is the number of times we iterate the training process on the entire dataset.
- The **batch size**, which is the number of training samples processed in one iteration. However, this parameter can increase memory usage significantly, making it hard to train the model on microcontrollers. For this reason, the batch size is typically equal to 1.

From the preceding points, it should be clear why the model training in the Python reference implementation iterates only over both the epochs and training data:

```
# X => Dataset (a,b)
# Y => Dataset (result)

epochs = 10000

for epoch in range(epochs):
    loss = 0
    num_correct_pred = 0

    for input, actual_out in zip(X, Y):
```

At the very least, one epoch is required, which implies that the entire dataset is processed only once through the training cycle. However, one epoch is often insufficient for the neural network to adjust the weights and achieve a reduced loss.

Generally, the higher the number of epochs, the more the model refines its weights and biases to fit the training data better. Therefore, although more epochs can lead to a lower training loss, it might also introduce the risk of overfitting.

In real scenarios, overfitting is undesirable because the model should generalize to unseen data. Here, overfitting is less of a concern. The dataset (X and Y) used to train the XOR and NAND neural network consists of a finite combination of binary inputs.

Therefore, overfitting is desired to achieve 100% accuracy, as there will not be any data beyond what is provided in the dataset. For our problem, you can set the number of epochs at around 10,000 to train the model successfully.

In the preceding code snippet, the `loss` and `num_correct_pred` variables are initialized to 0 at the beginning of every epoch. These two variables are updated when iterating over the training data and will be used to inform the user about the model's loss and accuracy for each epoch.

Inside the loop that iterates over the training data comes the exciting part concerning the training, consisting of the following three steps.

Step 1: Forward pass computation

In this step, we take the input from the dataset (`input`) and pass it through the network to produce the output (`predicted_out`):

```
predicted_out = nn.forward(input)
```

The `forward()` method is part of the `NN` class and aims to execute the two fully connected layers followed by a sigmoid activation function:

```
# class NN ----- continue
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def forward(self, x):
    input = np.expand_dims(x, axis=0)

    # Fully connected 0
    self.out_fc0 = np.dot(input, self.w0)
    self.out_fc0 += self.b0
    self.out_fc0 = sigmoid(self.out_fc0)

    # Fully connected 1
    self.out_fc1 = np.dot(self.out_fc0, self.w1)
```

```
self.out_fc1 += self.b1
self.out_fc1 = sigmoid(self.out_fc1)

return self.out_fc1
```

As you can see from the preceding code, the fully connected layer is performed through a vector-by-matrix multiplication (`np.dot`) between the input (`input` or `self.out_fc0`) and the weights (`self.w0` or `self.w1`), to which a bias (`self.b0` or `self.b1`) is added.

It is crucial to note that during the forward pass, all outputs of the trainable layers (`self.out_fc0` and `self.out_fc1`) are saved, as they are necessary for adjusting the weights later.

Based on this observation, it is clear that this algorithm requires more memory compared to model inference. During inference, intermediate tensors are typically not retained and reused.

Step 2: Updating the loss

Once we have predicted the output, the next step is to update the `num_correct_pred` and `loss` variables:

```
# Update accuracy
if np.round(predicted_out) == actual_out:
    num_correct_pred += 1

# Update Loss
err = predicted_out[0] - actual_out
loss += np.square(err) / len(Y)
```

In the previous code snippet, `len(Y)` returns the total number of samples in the dataset.

Step 3: Backward pass computation

The loss is the feedback about the prediction accuracy, and it is key to adjust the weights and biases of the neural network to make the following forward pass more accurate.

This information is propagated backward, from the end of the network to the first layer, as shown in the following figure:

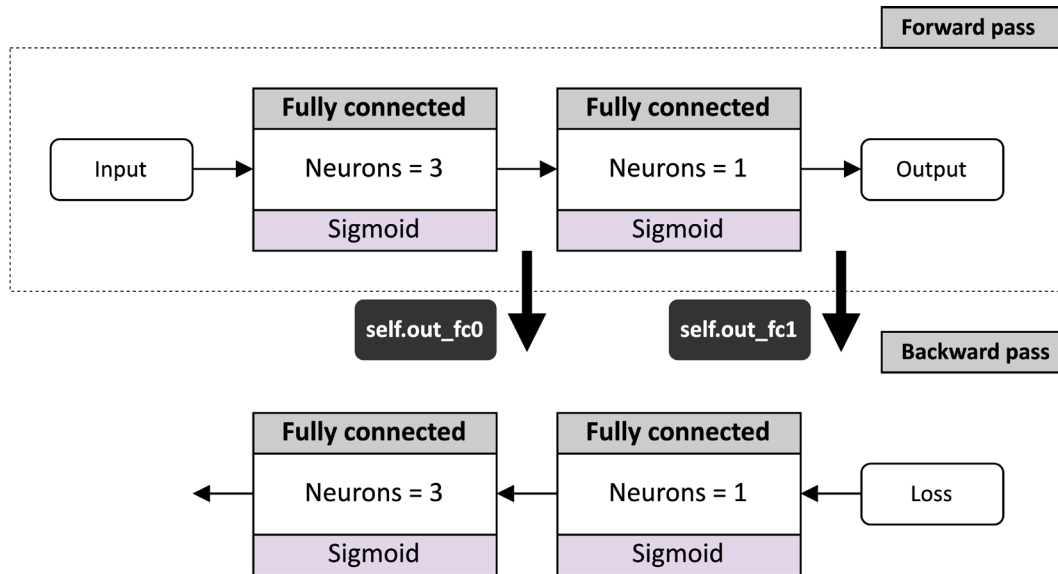


Figure 12.3: Forward versus backward pass

In the Python reference implementation, this computation is performed by the `backward()` method of the `NN` class, which takes the input sample and the actual output as input arguments:

```
nn.backward(input, actual_out)
```

As we know, backpropagation aims to minimize the loss function. From the mathematical course, we can determine the minimum of a function by solving the equation resulting from equating the function's first derivative to zero.

However, the full expression of the loss function for a neural network is very complex, involving multiple variables (weights and biases) across different layers.



A **gradient** is a derivative of a function with multiple input variables.

Because of this complexity, it is not feasible to solve this equation practically. Therefore, the iterative **gradient descent** approach based on partial derivatives was proposed, and what has been implemented in the `backward()` method is the following:

```
# class NN ----- continue
def d_sigmoid(x):
    return x * (1.0 - x)

def backward(self, x, y):
    input = np.expand_dims(x, axis=0)
    actual_out = y

    # Calculate error/delta
    e_out_fc1 = actual_out - self.out_fc1
    d_out_fc1 = e_out_fc1 * d_sigmoid(self.out_fc1)

    e_out_fc0 = np.dot(d_out_fc1, self.w1.T)
    d_out_fc0 = e_out_fc0 * d_sigmoid(self.out_fc0)

    # Learning rate
    lr = 0.1

    # Calculate the weights adjustments
    w0_1 = np.dot(input.T, d_out_fc0)
    w1_1 = np.dot(self.out_fc0.T, d_out_fc1)
    w0_1 *= lr
    w1_1 *= lr

    # Calculate the biases adjustments
    b0_1 = d_out_fc0 * lr
    b1_1 = d_out_fc1 * lr

    # Update weights and biases
    self.w0 += w0_1
    self.b0 += b0_1
    self.w1 += w1_1
    self.b1 += b1_1
```

A comprehensive discussion of the theoretical foundations of the backward pass, such as the calculation of the **error** and **delta** required to update the weights and biases, goes beyond the scope of this recipe. What is crucial for us is to understand the sequence of the operation performed by this algorithm so that we can transform it into C code, enabling its deployment on microcontrollers.

To make the porting easier and efficient for ArmCPUs on the Arduino Nano and Raspberry Pi Pico, we will exploit the **CMSIS-DSP library**, like what we did in *Chapter 6, Recognizing Music Genres with TensorFlow and the Raspberry Pi Pico – Part 2*. For example, we will be using the `arm_vexp_f32()`, `arm_mat_mult_f32()`, and `arm_mat_trans_f32()` routines from the CMSIS-DSP library to implement the `np.exp()`, `np.dot()`, and `.T` Python counterparts.

How to do it...

In the Arduino IDE, create a new sketch and follow the following steps to train the neural network with backpropagation to carry out the XOR and NAND operations on the Arduino Nano or Raspberry Pi Pico.

Step 1:

Download the Arduino CMSIS-DSP library from the **TinyML-Cookbook_2E** GitHub repository: https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/ArduinoLibs/Arduino_CMSIS_DSP.zip.

After downloading the ZIP file, import it into the Arduino IDE.

Step 2:

In the sketch, include the `arm_math.h` header file to access the functions and data types provided by the CMSIS-DSP library:

```
#include "arm_math.h"
```

Step 3:

Implement a function to initialize an array with random floating-point values. To do so, create a function called `init_random()` that takes the pointer to the floating-point array to be initialized and its size as input arguments:

```
void init_random(float* src, int32_t sz) {
```

Inside the function, write a for loop to initialize the values in the `src` array with random values using the Arduino `random()` function:

```
for(int32_t i = 0; i < sz; ++i) {
    src[i] = (float)random(0, 999) / 1000.0f;
}
```

The Arduino `random()` function produces a pseudo-random integer number between the range specified with its two input arguments. The first parameter (0) sets the inclusive lower limit, and the second defines the exclusive upper limit. Since this function can only generate integer values, we set the range from 0 to 999 and then divide the result by 1,000 to draw random values from a pool of 1,000 floating-point values, ranging from 0 up to (but not including) 1.



Before using the Arduino `random()` function, it is necessary to call the Arduino `randomSeed()` function. This routine is generally called in the `setup()` function to set the seed.

The `init_random()` function, implemented to populate an array with random values, does not behave in the same way as the Python `np.random.rand()` method. Specifically, the Arduino `random()` function does not guarantee that the random values are taken from a uniform distribution over $[0, 1]$. Nevertheless, for our purposes, it is adequate.

Step 4:

Write a function to implement the matrix-by-matrix operation using floating-point precision with the `arm_mat_mult_f32()` CMSIS-DSP routine. To do so, create a function called `mat_mul()` that takes the following input arguments:

- The pointer to the input left-hand side (lhs) matrix
- The pointer to the input right-hand side (rhs) matrix
- The pointer to the destination (dst) matrix
- The rows of the lhs matrix (M)
- The columns of the rhs matrix (N)
- The columns of the lhs matrix (K), which match the rows of the rhs matrix:

```
void mat_mul(float* lhs, float* rhs, float* dst,
            int32_t M, int32_t N, int32_t K) {
```

Inside the function, declare three `arm_matrix_instance_f32` instances for initializing the input arguments of the `arm_mat_mult_f32()` CMSIS-DSP function:

```
arm_matrix_instance_f32 lhs_i, rhs_i, dst_i;
```

Then, initialize the instances with the dimensions and data pointers of the `lhs`, `rhs`, and `dst` arrays:

```
// LHS matrix
lhs_i.numRows = M;
lhs_i.numCols = K;
lhs_i.pData = lhs;
// RHS matrix
rhs_i.numRows = K;
rhs_i.numCols = N;
rhs_i.pData = rhs;
// DST matrix
dst_i.numRows = M;
dst_i.numCols = N;
dst_i.pData = dst;
```

After this, invoke `arm_mat_mult_f32()` to perform the matrix multiplication between the `lhs` and `rhs` matrices:

```
arm_mat_mult_f32(&lhs_i, &rhs_i, &dst_i);
}
```

The `mat_mul()` function will be employed to carry out both vector-by-matrix and matrix-by-matrix multiplications in the forward and backward pass computations.

Step 5:

Create a function to implement the sigmoid activation.

To implement this function, create a function called `sigmoid()` that takes the following input arguments: a pointer to the source array, a pointer to the destination array, and the total number of elements in both arrays:

```
void sigmoid(float* src, float* dst, int32_t sz) {
```

The reason for needing only a single value for the number of elements (`sz`) is that the `src` and `dst` arrays have equal sizes.

Inside the function, write a for loop to perform the sigmoid operation using the `arm_vexp_f32()` CMSIS-DSP routine:

```
for(int32_t i = 0; i < sz; ++i) {
    float x = -src[i];
    arm_vexp_f32(&x, &x, 1);
    dst[i] = 1.0f / (1.0f + x);
}
}
```

Step 6:

Create a function to implement the derivative of the sigmoid activation.

To implement this function, create a function called `d_sigmoid()` that takes the following input arguments: a pointer to the source array, a pointer to the destination array, and the total number of elements in both arrays:

```
void d_sigmoid(float* src, float *dst, int32_t sz) {
```

Like what we had for the `sigmoid()` function, we can have a single value for the number of elements (`sz`) because the `src` and `dst` arrays have equal sizes.

Inside the function, write a for loop to perform the derivative of the sigmoid operation:

```
for(int32_t i = 0; i < sz; ++i) {
    float x = src[i];
    dst[i] = x * (1.0f - x);
}
}
```

Step 7:

Create a function to implement the fully connected layer followed by the `sigmoid()` function.

To implement this function, create a function called `fc_sigmoid()` that takes the pointer to the input (`src`), weights (`w`), biases (`b`), and destination (`dst`) arrays, as well as the input and output vector sizes:

```
void fc_sigmoid(float* src, // input
               float* w,   // weights
```



```
float* b,    // biases
float* dst,  // destination
int32_t src_sz, int32_t dst_sz) {
```

Inside the function, use the `mat_mul()` function to perform the matrix multiplication between the input vector and weights matrices:

```
mat_mul(src, w, dst, 1, dst_sz, src_sz);
```

After the matrix multiplication, add the bias to the output vector using the `arm_add_f32()` CMSIS-DSP routine:

```
arm_add_f32(dst, b, dst, dst_sz);
```

Finally, perform the sigmoid activation:

```
sigmoid(dst, dst, dst_sz);
}
```

In the preceding code snippets, we perform the bias addition and sigmoid activation **in place**, which means that the output is written in the exact memory location of the input value.

These two operations can be performed in place because both operate element-wise.

Step 8:

Create a C++ class called `NN` to implement the backpropagation algorithm discussed in the previous *Getting ready* section:

```
class NN {
public:
    // Public members
private:
    // Private members
};
```

Inside the private section, define four pointers to reference the memory locations storing the weights and biases of the neural network:

```
// Private members
float* _w0 {nullptr};
float* _b0 {nullptr};
```

```
float* _w1 {nullptr};
float* _b1 {nullptr};
```

The preceding pointers are set to `nullptr` initially because the memory allocation will occur in the constructor of the `NN` class.

Since the weights and biases are not constants during training, they must be stored in data memory (SRAM) rather than program memory (Flash). However, in a production-ready application, it is advisable to save them in Flash memory once the training is completed to prevent the need for retraining from scratch upon device reboot, as SRAM is volatile memory.



While the details of storing these values in Flash are not covered here, you can find references on achieving this in the *There's more...* section of this recipe for both the Arduino Nano and Raspberry Pi Pico.

If you look at the Python reference implementation, the backward pass needs to transpose the weights of the last fully connected layer (`_w1`) and calculate the weight adjustments (`w0_1`, `b0_1`, `w1_1`, and `b1_1`) to update the original weights. Therefore, declare four additional pointers for storing the result of these operations:

```
float* _w0_1 {nullptr};
float* _b0_1 {nullptr};
float* _w1_1 {nullptr};
float* _b1_1 {nullptr};
```

The `_w0_1`, `_b0_1`, `_w1_1`, and `_b1_1` pointers will be initialized within the constructor and reference memory locations of the same size as `_w0`, `_b0`, `_w1`, and `_b1`.

Therefore, it is evident that training an ML model using backpropagation is very memory-demanding, as the weights and biases necessitate twice the storage than inference.

After declaring the pointers for the weights and biases, define six pointers to reference the memory locations storing the intermediate outputs of the neural network obtained during the forward pass as well as the error and delta calculated in the backward pass:

```
// Private members
float* _out_fc0 {nullptr}; // Forward pass
float* _out_fc1 {nullptr}; // Forward pass
float* _e_out_fc0 {nullptr}; // Backward pass
```

```
float* _d_out_fc0 {nullptr}; // Backward pass
float* _e_out_fc1 {nullptr}; // Backward pass
float* _d_out_fc1 {nullptr}; // Backward pass
```

Finally, declare three integer values to hold the input size and number of neurons of the fully connected layers:

```
int32_t _input_sz {0};
int32_t _out_fc0_sz {0};
int32_t _out_fc1_sz {0};
```

Step 9:

In the private section of the NN class, create a function to calculate the delta in the backward pass computation.

To implement this function, create a function called `delta()` that takes the following arguments as input:

- Pointer to the output of the fully connected layer after applying the sigmoid activation returned in the forward pass.
- Pointer to the error term, which may represent the difference between the predicted and actual values for the final layer or the computed gradient for the hidden layer.
- Pointer to the delta array, where the result of this operation will be stored
- The fully connected output size:

```
void delta(float* out_fc,
           float* err,
           float* delta,
           int32_t out_fc_sz) {
```

Inside the function, calculate the derivative of the `sigmoid()` function from the output of the fully connected layer after applying the sigmoid activation calculated in the forward pass:

```
    d_sigmoid(out_fc, delta, out_fc_sz);
```

Then, multiply the preceding output by the error term element-wise using the `arm_mult_f32()` CMSIS-DSP routine:

```
    arm_mult_f32(delta, err, delta, out_fc_sz);
}
```

Step 10:

In the private section of the NN class, create a function to update the weights and biases.

To implement this function, create a function called `update_weights_biases()` that takes the input sample (`x`) and learning rate (`lr`) as input arguments:

```
void update_weights_biases(float* x, float lr) {
```

Inside the function, calculate the adjustment of the weights for the hidden layer by performing the matrix multiplication between the transposed input sample (`x`) and its delta (`_d_out_fc0`):

```
    mat_mul(x, _d_out_fc0, _w0_1,
            _input_sz, _out_fc0_sz, 1);
```

Then, apply the learning rate to the adjustment of the weights (`_w0_1`) using the `arm_scale_f32()` CMSIS-DSP function:

```
    int32_t w0_total_sz = _out_fc0_sz * _input_sz;
    arm_scale_f32(_w0_1, lr, _w0_1,
                  w0_total_sz);
```

After calculating the adjustment of the weights for the hidden layer, calculate the adjustment of the weights for the output layer by performing the matrix multiplication between the transposed output of the hidden layer (`_out_fc0`) and its delta (`_d_out_fc1`):

```
    mat_mul(_out_fc0, _d_out_fc1, _w1_1,
            _out_fc0_sz, _out_fc1_sz, 1);
```

Then, apply the learning rate to the adjustment of the weights (`w1_1`) using the `arm_scale_f32()` CMSIS-DSP function:

```
    int32_t w1_total_sz = _out_fc0_sz * _out_fc1_sz;
    arm_scale_f32(_w1_1, lr, _w1_1,
                  w1_total_sz);
```

After this, calculate the adjustment of the biases by applying the learning rate to the deltas of the hidden and output layers using the `arm_scale_f32()` CMSIS-DSP function:

```
    int32_t b0_total_sz = _out_fc0_sz;
    int32_t b1_total_sz = _out_fc1_sz;
    arm_scale_f32(_d_out_fc0, lr, _b0_1, b0_total_sz);
    arm_scale_f32(_d_out_fc1, lr, _b1_1, b1_total_sz);
```

Finally, add the adjustment of the weights and biases to the weights and biases:

```
arm_add_f32(_w0, _w0_1, _w0, w0_total_sz);
arm_add_f32(_w1, _w1_1, _w1, w1_total_sz);
arm_add_f32(_b0, _b0_1, _b0, b0_total_sz);
arm_add_f32(_b1, _b1_1, _b1, b1_total_sz);
} // update_weights_biases
```

Step 11:

In the public section of the NN class, implement the default constructor that accepts the input size as well as the number of neurons for the hidden and output layer as input arguments:

```
// Public members
NN(int32_t input_sz,
   int32_t out_fc0_sz,
   int32_t out_fc1_sz) {
```

Inside the constructor, initialize the input size and number of neurons of the fully connected layers:

```
_input_sz = input_sz;
_out_fc0_sz = out_fc0_sz;
_out_fc1_sz = out_fc1_sz;
```

Then, allocate the memory for the weights and biases:

```
_w0 = new float[input_sz * out_fc0_sz];
_b0 = new float[out_fc0_sz];
_w1 = new float[out_fc0_sz * out_fc1_sz];
_b1 = new float[out_fc1_sz];

_w0_1 = new float[input_sz * out_fc0_sz];
_b0_1 = new float[out_fc0_sz];
_w1_1 = new float[out_fc0_sz * out_fc1_sz];
_b1_1 = new float[out_fc1_sz];
```

Then, allocate the memory for the outputs of the fully connected layers:

```
_out_fc0 = new float[out_fc0_sz];
_out_fc1 = new float[out_fc1_sz];
```

After this, allocate the memory for the intermediate calculations required in the backward pass computation (error and delta):

```
_e_out_fc0 = new float[out_fc0_sz];
_d_out_fc0 = new float[out_fc0_sz];
_e_out_fc1 = new float[out_fc1_sz];
_d_out_fc1 = new float[out_fc1_sz];
```

Finally, initialize the weights and biases with random values to finalize the NN constructor:

```
init_random(_w0, input_sz * out_fc0_sz);
init_random(_b0, out_fc0_sz);
init_random(_w1, out_fc0_sz * out_fc1_sz);
init_random(_b1, out_fc1_sz);
} // close NN constructor
```

Step 12:

In the public section of the NN class, implement the destructor to free all the memory spaces dynamically allocated:

```
~NN() {
    delete[] _w0;
    delete[] _b0;
    delete[] _w1;
    delete[] _b1;
    delete[] _w0_1;
    delete[] _b0_1;
    delete[] _w1_1;
    delete[] _b1_1;
    delete[] _out_fc0;
    delete[] _out_fc1;
    delete[] _e_out_fc0;
    delete[] _d_out_fc0;
    delete[] _e_out_fc1;
    delete[] _d_out_fc1;
}
```

Step 13:

In the public section of the NN class, implement a method to perform the forward pass. To do so, create a function called `forward()` that takes the pointer to the input sample and returns the pointer to the model's output:

```
float* forward(float* x) {
```

Inside the function, perform the two fully connected layers followed by the `sigmoid()` function:

```
    fc_sigmoid(x, _w0, _b0, _out_fc0,
               _input_sz, _out_fc0_sz);

    fc_sigmoid(_out_fc0, _w1, _b1, _out_fc1,
               _out_fc0_sz, _out_fc1_sz);
```

Finally, return the pointer to the model's output:

```
    return _out_fc1;
}
```

Step 14:

In the public section of the NN class, implement a method to perform the backward pass. To do so, create a function called `backward()` that takes the pointer to the input sample and the expected output:

```
void backward(float* x, float* y) {
```

Inside the function, calculate the error for the last layer, which is the difference between the actual value (`y`) and the predicted value (`_out_fc1[0]`):

```
    _e_out_fc1[0] = y[0] - _out_fc1[0];
```

Then, calculate the delta for the output layer:

```
    delta(_out_fc1, _e_out_fc1,
          _d_out_fc1, _out_fc1_sz);
```

After calculating the error and delta for the output layer, calculate the error for the hidden layer.

To calculate this error, transpose the weights of the output layer using the `arm_mat_trans_f32()` CMSIS-DSP routine:

```
arm_matrix_instance_f32 w1_i, w1T_i;

w1_i.numRows = _out_fc0_sz;
w1_i.numCols = _out_fc1_sz;
w1_i.pData = _w1;
w1T_i.numRows = _out_fc1_sz;
w1T_i.numCols = _out_fc0_sz;
w1T_i.pData = _w1_1;

arm_mat_trans_f32(&w1_i, &w1T_i);
```

Then, perform the matrix multiplication between the delta (`_d_out_fc1`) and the transposed weights (`_w1_1`) of the output layer:

```
mat_mul(_d_out_fc1, _w1_1, _e_out_fc0,
        1, _out_fc0_sz, _out_fc1_sz);
```

After this, calculate the delta for the hidden layer:

```
delta(_out_fc0, _e_out_fc0,
      _d_out_fc0, _out_fc0_sz);
```

Having calculated the delta for both fully connected layers, finalize the backward pass by updating the weights and biases using a learning rate of 0.1:

```
update_weights_biases(x, 0.1);
} // backward()
```

Step 15:

Define a global NN object called `nn`:

```
NN nn(3, 3, 1);
```

When creating this object, the constructor of the NN class gets invoked, allocating all the internal memory.

Step 16:

In the `setup()` function, initialize the serial peripheral with a 115200 baud rate:

```
Serial.begin(115200);  
while (!Serial);
```

The serial peripheral will be employed to communicate the loss and accuracy during training.

Step 17:

In the `setup()` function, use the Arduino `randomSeed()` function to initialize the random number generator. To do so, initialize the `randomSeed()` function with a reasonably random number that can be obtained by reading the analog signal from an unconnected pin using Arduino's `analogRead()` function:

```
randomSeed(analogRead(0));
```

The Arduino `random()` function generates pseudo-random numbers, meaning that the values are obtained from a deterministic process derived from a known starting point. This starting point is the **seed**, initialized with the `randomSeed()` function. If the seed is a fixed value, the sequence of random numbers will always be the same.

To make the output of `random()` more unpredictable between different program runs, we can initialize the seed with a value that varies each time. One standard method is to read the analog signal on an unconnected pin, which tends to have just electrical noise, resulting in random digital samples.

Step 18:

In the `setup()` function, prepare the dataset for solving the XOR and NAND problem, like what has been prepared in the Python reference implementation:

```
float X[8][3] = {  
    {0.0f, 0.0f, 0.0f},  
    {0.0f, 1.0f, 0.0f},  
    {1.0f, 0.0f, 0.0f},  
    {1.0f, 1.0f, 0.0f},  
    {0.0f, 0.0f, 1.0f},  
    {0.0f, 1.0f, 1.0f},  
    {1.0f, 0.0f, 1.0f},  
    {1.0f, 1.0f, 1.0f},  
};
```

```

    {1.0f, 1.0f, 1.0f}
};

float Y[8] = {0.0f, 1.0f, 1.0f, 0.0f, /* XOR */
              1.0f, 1.0f, 1.0f, 0.0f /* NAND */};

```

In the preceding code snippet, the X and Y arrays are the training samples and the corresponding output result. Concerning the training samples (X), the first two values are the binary inputs for the logical operations, and the third value defines the logical operation to perform (either XOR or NAND).

Step 19:

In the `setup()` function, train the model, like what has been done in the Python reference implementation:

```

const int32_t num_epochs = 10000;

for(int32_t epoch = 0; epoch < num_epochs; ++epoch) {
    float loss = 0.0;
    int32_t num_correct_pred = 0;

    for(int32_t smp_idx = 0; smp_idx < 8; ++smp_idx) {
        float *input      = &X[smp_idx][0];
        float *actual_out = &Y[smp_idx];

        // Forward
        float *predicted_out = nn.forward(input);

        // Update accuracy
        if(round(predicted_out[0]) == actual_out[0]) {
            num_correct_pred++;
        }

        // Update Loss
        float err = (actual_out[0] - predicted_out[0]);
        loss += (err * err) / 8.0f;
    }
}

```

```
// Backward
nn.backward(input, actual_out);
}
```

After this, transmit the loss and accuracy after every 100 epochs:

```
if((epoch % 100) == 0) {
    float acc = num_correct_pred / 8.0f;
    Serial.print(loss);
    Serial.print(",");
    Serial.println(acc);
}
} // for Loop epoch
```

Upon training completion, transmit Training finished! over the serial interface:

```
Serial.println("Training finished!");
} // setup().
```

Now, let's make magic happen.

Plug the micro-USB data cable into the Arduino Nano or Raspberry Pi Pico. Once you have connected it, compile and upload the sketch on the microcontroller. Then, open the serial monitor in the Arduino IDE.

After a few seconds, you should see the loss and accuracy displayed after every 100 epochs. During training, the loss should gradually decrease. Simultaneously, the accuracy should improve, reaching 1.0 (100 %), confirming the successful implementation of the backpropagation algorithm.

There's more...with the SparkFun Artemis Nano!

In this recipe, we demonstrated the feasibility of training a neural network on microcontrollers and learned how to do it on the Arduino Nano or Raspberry Pi Pico using backpropagation.

The sketch developed in this recipe is compatible with any Arm-based microcontrollers programmed in the Arduino IDE, including the SparkFun Artemis Nano microcontroller.

To train this model on the SparkFun Artemis Nano, you should do the following:

1. Create a new Arduino project.

2. Import the Arduino CMSIS-DSP library in the Arduino IDE.
3. Ensure you have followed the instructions provided in the *Setting up the SparkFun Artemis Nano board in the Arduino IDE* guide (https://github.com/PacktPublishing/TinyML-Cookbook_2E/blob/main/Docs/setup_sparkfun_artemis_nano.md). This step is necessary to avoid compilation errors when building a sketch based on the CMSIS-DSP library.

As you have probably noticed, the model was trained using floating-point data, which is certainly not the most efficient data type for microcontroller deployment. More advanced techniques have been proposed for accelerating the backpropagation algorithm using the 8-bit quantized data type. However, training a model effectively with limited numerical precision data types is still an open research area.

Training a neural network using the floating-point data type implies that we must carefully build the model as it occupies much more memory than an 8-bit one for inference. Furthermore, when training a model, you must consider that the weights and biases cannot be stored in program memory (Flash) but only in data memory (SRAM), which is generally very limited.

Despite these limitations, the NN class implemented in this recipe should give a starting point for your future projects in this area. In fact, you can change the number of neurons of the hidden and output layer to train a model for a different task. Or you can attach this neural network to the output of a pre-trained model to implement an **on-device transfer learning** algorithm.

As mentioned in the *How to do it...* section, in a production-ready application, storing the weights and biases in Flash after the training phase is advisable to avoid retraining the model from scratch upon device reboot. Additionally, it is worth noting that the dataset should also be stored in permanent memory to prevent the loss of training samples acquired during device operation.

To learn about storing data in Flash memory on the Arduino Nano, refer to the example provided by Pete Warden at the following link: https://github.com/petewarden/arduino_nano_ble_write_flash. For the Raspberry Pi Pico, you can refer to the Flash application example within the Raspberry Pi Pico SDK available at the following link: https://github.com/raspberrypi/pico-examples/blob/sdk-1.5.1/flash/program/flash_program.c.

In all projects presented in this book, including this recipe, we only discussed neural network-based models. However, can we run other ML algorithms on microcontrollers?

In the upcoming recipe, we will demonstrate that we can by running a model trained using scikit-learn on the Arduino Nano and Raspberry Pi Pico.

How can we deploy scikit-learn models on microcontrollers?

Artificial neural networks are incredibly accurate and versatile to solve a wide range of data analysis problems. However, this model is not the only player in the ML arena. Indeed, many other ML models are available that can be just as effective for specific tasks and be less compute- and memory-demanding.

In this recipe, we will learn how to deploy a **random forest** model trained with the **scikit-learn** framework on the Arduino Nano and Raspberry Pi Pico with the Python **emlearn** project.

Getting ready

Whether you are just starting in ML, an enthusiast, or a researcher, you will have probably come across the **scikit-learn** (<https://scikit-learn.org/>) framework, a pillar, like TensorFlow, of the ML community.

As we have seen through all the projects developed in this book, TensorFlow is a low-level library providing the building blocks for ML algorithms, particularly for creating deep learning models. Scikit-learn, on the other hand, is a higher-level framework than TensorFlow as it offers a plethora of off-the-shelf ML algorithms, such as **support vector machine (SVM)**, **decision tree**, **random forest**, **logistic regression**, and many more.

Generally speaking, we can say that TensorFlow is primarily for deep learning while scikit-learn is for generic ML.

While artificial neural networks have undeniably given powerful capabilities to many data science problems, generic ML algorithms remain indispensable in certain areas as they can offer interpretability and require fewer computational resources. Therefore, these classical methods could be more suitable for memory-constrained devices in some scenarios.

However, how do we deploy a trained model with scikit-learn on microcontrollers?

At the time of writing, there are already at least three valuable open-source projects available for this task: **emlearn** (<https://github.com/emlearn/emlearn>), **micromlgen** (<https://github.com/eloquentarduino/micromlgen>), and **sklearn-porter** (<https://github.com/nok/sklearn-porter>). Each of these projects can produce C code from models trained with scikit-learn.

In this recipe, we have opted for the emlearn project due to its broader support for ML algorithms and practical preprocessing methods for audio applications, like the computation of the Mel-spectrogram.

How to do it...

Open the web browser and create a new Colab notebook. Then, follow the steps to deploy a random forest model trained with scikit-learn on the Arduino Nano or Raspberry Pi Pico.

Step 1:

Follow the instructions provided in the section titled *Fitting and predicting: estimator basics* of the scikit-learn getting started guide (https://scikit-learn.org/stable/getting_started.html) for training a basic random forest model:

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(random_state=0)
X = [[ 1,  2,  3],
      [11, 12, 13]]
Y = [0, 1]
clf.fit(X, Y)
```

In the preceding code, the random forest model is trained to classify two classes: 0 and 1. The training dataset consists of only two samples: [1, 2, 3] belonging to class 0, and [11, 12, 13] belonging to class 1.

After training, if you provide new and unseen data to the model, the model will try to classify it as either 0 or 1 based on its similarity to the training samples.

Step 2:

Define the following test dataset:

```
X_TEST = [[ 1,  2,  4],
            [ 1, 18,  4],
            [14, 12, 13]]
```

The preceding three test samples will be used to test the model's prediction.

Step 3:

Test the trained random forest model:

```
clf.predict(X_TEST)
```

The model should classify the first sample ([1, 2, 4]) as class 0 and the other two as class 1.

Step 4:

Install the Python 0.18.1 release of the emlearn tool using the following pip command:

```
!pip install emlearn==0.18.1
```

Step 5:

Import the emlearn Python module and convert the random forest model (clf) trained with scikit-learn to C code using the convert() method:

```
import emlearn
model = emlearn.convert(clf, method='inline')
```

The convert() method from the emlearn Python module converts the trained scikit-learn model to C code. This function takes the scikit-learn model (clf) as a mandatory argument and the following ones as optional:

- **method:** This is the inference strategy employed. For converting the model into deployable C code for microcontrollers, it should be set to inline.
- **kind:** This is the model's name.
- **dtype:** This specifies the data type for the model. The default value is float to run the model with floating-point precision. However, it can be changed to enable quantization.
- **return type:** This is the return type for the model. The default value is Classifier to create a classifier model.

Step 6:

Generate the C header file containing the model with the save() method of the model object:

```
model.save(file='model.h', name='model')
```

The preceding code will generate a C header file named model.h containing the trained scikit-learn model.

To correctly compile this file in the Arduino IDE, you need the **C emlearn library**, which comprises the routines to execute the scikit-learn models. The C emlearn library is located within the directory pointed to by the Python variable emlearn.includedir.

Given the numerous C files in the emlearn library, manually uploading them to the Arduino IDE is impractical. As a solution, the following step will guide you through building an Arduino library.

Step 7:

Build an Arduino library from the files of the C emlearn library. To do so, get the directory path where the C emlearn library files are stored:

```
libdir = emlearn.includedir
```

Then, inside the emlearn library directory, create a file called `library.properties` with the text `name=emlearn`:

```
!echo "name=emlearn" > $libdir/library.properties
```

The `library.properties` file is required for the Arduino IDE to recognize the library we will create. In this file, the minimum information you must specify is the name of this library, which, in our case, is `emlearn`.



The `library.properties` file can contain additional information, such as the library's version or author.

For further information about the format of this file, you can refer to the official Arduino documentation: <https://arduino.github.io/arduino-cli/library-specification/>.

Once you have created the `library.properties` file, zip the folder containing the emlearn library:

```
!cd $libdir/..; zip -r Arduino_EMLEARNLib.zip emlearn
```

The preceding command changes the directory before zipping the file to avoid including subdirectories in the ZIP file.

Finally, move the `Arduino_EMLEARNLib.zip` file from its current location to the working directory:

```
!mv $libdir/./Arduino_EMLEARNLib.zip
```

Step 8:

Download the `model.h` and `Arduino_EMLEARNLib.zip` files from Colab's left panel. Then, open the Arduino IDE.

Step 9:

Create a new sketch and import the `Arduino_EMLEARNLib.zip` file into the Arduino IDE.

Step 10:

Import the `model.h` header file into the project.

Once the file has been imported, include the `model.h` header file in the sketch:

```
#include "model.h"
```

Step 11:

Declare and initialize a two-dimensional floating-point array to store the three samples previously used to test the scikit-learn predictions:

```
#define INPUT_SZ 3
float X_TEST[3][INPUT_SZ] = {{1, 2, 3},
                              {1, 18, 4},
                              {14, 12, 13}};
```

Step 12:

In the `setup()` function, initialize the serial peripheral with a 115200 baud rate:

```
Serial.begin(115200);
while (!Serial);
```

The serial peripheral will be employed to communicate the model's predictions.

Step 13:

In the `loop()` function, perform the model's prediction for every test sample and transmit the result over the serial port:

```
for(int32_t i = 0; i < 3; ++i) {
    int32_t v = model_predict(&X_TEST[i][0], INPUT_SZ);
    Serial.println(v);
}
while(1);
```

To invoke the model's inference, we use the `model_predict()` function, which requires a pointer to the input sample and its size as input arguments. It is worth noting that the exact name of this function depends on the name passed to the `save()` `emlearn` function when creating the C header file. Generally, it adopts the naming convention of `<name_model>_predict`.

Now, plug the micro-USB data cable into the Arduino Nano or Raspberry Pi Pico. Once you have connected it, compile and upload the sketch on the microcontroller. Then, open the serial monitor in the Arduino IDE.

After a few seconds, you should see the model's predictions matching the results of the scikit-learn model in Python!

There's more...with the SparkFun Artemis Nano

In this recipe, we learned how to deploy generic ML algorithms trained with scikit-learn on the Arduino Nano and Raspberry Pi Pico with the help of the `emlearn` project.

The sketch developed in this recipe is compatible with any microcontrollers programmed in the Arduino IDE. Therefore, what about testing the model on the SparkFun Artemis Nano microcontroller?

To test the scikit-learn model on the SparkFun Artemis Nano, you should do the following:

1. Create a new Arduino project.
2. Import the `model.h` C header file.
3. Import the `Arduino_EMLEARNLib.zip` file.

If you want to experiment further with scikit-learn and `emlearn`, you might consider playing with ML algorithms for **anomaly detection**. Anomaly detection is a task for recognizing anomalous input samples that deviate from the training dataset, which we assume to be the standard data. As you can guess, this task might be relevant for monitoring anomalies in sensor data, such as in wearable health monitoring systems for detecting irregular heart rhythms or in industry for identifying equipment malfunctions.

The `emlearn` project provides a comprehensive discussion about anomaly detection along with potential ML algorithms suitable for this task at the following link: https://emlearn.readthedocs.io/en/latest/anomaly_detection.html#anomaly-detection-models.

We began our journey by discussing the endless possibilities of ML on low-power devices, such as our microcontrollers. We emphasized that such applications are ideally suited for scenarios where low power consumption is critical, especially when the device needs to run on battery power for extended periods. However, how can we power our microcontrollers with batteries? Let’s uncover this topic in the final recipe of this book.

How can we power microcontrollers with batteries?

For many tinyML solutions, batteries could be the only power source for our microcontrollers and this final recipe will teach us how to power them with AA batteries.

Getting ready

Batteries are sources of electric power and have a limited charge capacity. The charge capacity (or battery capacity) quantifies the stored charge and is measured in **milli-ampere-hour (mAh)**. Therefore, a higher mAh implies a longer battery life.

The following table reports some commercial batteries that find applicability with microcon-
trollers:

Battery type	Voltage (V)	Charge capacity (mAh)
AAA	1.5	~1000
AA	1.5	~2400 (Alkaline)
CR2032	3.6	~240
CR2016	3.6	~90

Figure 12.4: Some commercial batteries for microcontrollers

The battery selection depends on the required microcontroller voltage and other factors such as charge capacity, form factor, and operating temperature.

As we can observe from the preceding table, the AA battery provides a higher capacity, but it supplies 1.5V, which is typically insufficient for microcontrollers.

Therefore, how can we power microcontrollers with AA batteries?

In the following subsections, we will show standard techniques to increase the supplied voltage or the charge capacity.

Increasing the output voltage by connecting batteries in series

When connecting *batteries in series*, the positive terminal of one battery is connected to the negative terminal of the other one, as shown in the following figure:

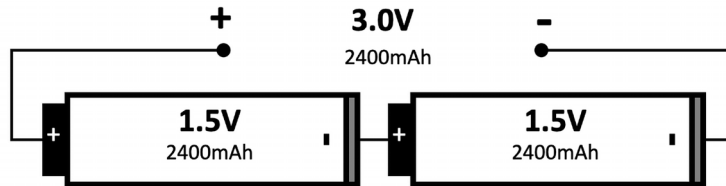


Figure 12.5: Batteries in series

This approach *will not increase the charge capacity but just the supplied voltage*.

The new supplied voltage (V_{new}) is as follows:

$$V_{new} = V_{battery} \cdot N$$

In the previous formula, N refers to the number of connected batteries in series. For example, since one AA battery supplies 1.5V for 2,400 mAh, we could connect two AA batteries in series to produce 3.0 V for the same charge capacity.

However, if the battery capacity is not enough for our application, how can we increase it?

Increasing the charge capacity by connecting batteries in parallel

When connecting *batteries in parallel*, the positive terminals of the batteries are tied together with one wire. The same applies to the negative terminals, which are joined together as shown in the following figure:

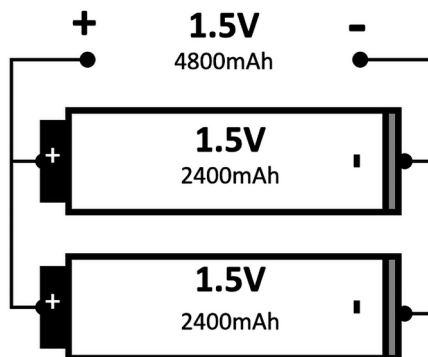


Figure 12.6: Batteries in parallel

This approach will not increase the output voltage but just the battery capacity.

The new battery capacity (BC_{new}) is as follows:

$$BC_{new} = BC_{battery} \cdot N$$

In the previous formula, N refers to the number of connected batteries in parallel. For example, since one AA battery has a capacity of 2,400 mAh, we could connect two AA batteries in parallel to increase the capacity by two times.

Connecting batteries manually to achieve the desired voltage and capacity can be challenging because we would require maintaining secure connections between the positive and negative terminals. As you can guess, soldering these contacts on the batteries is far from practical. A battery holder offers the solution to this problem because it allows us to combine batteries to create the ideal voltage and capacity by simply slotting them in.

The battery holders considered for this recipe connects the AA batteries only in series to reach the target voltage requirement. Specifically, we will use battery holders designed for four and three batteries, as shown in *Figure 12.7*:

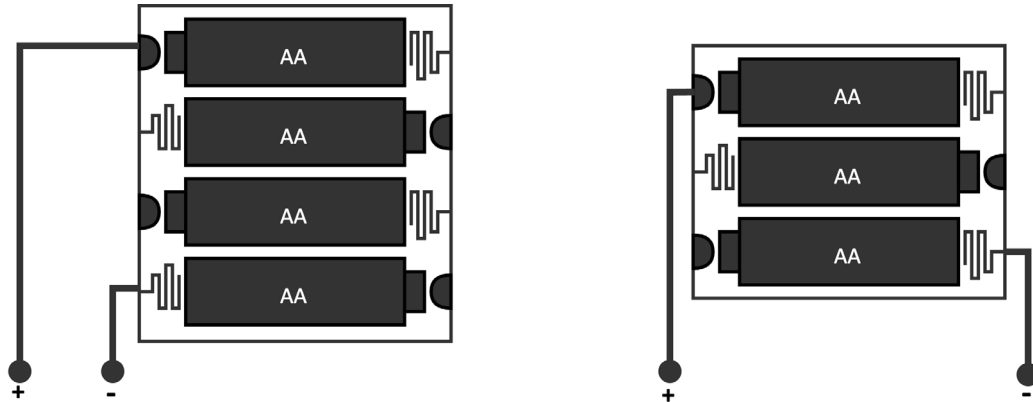


Figure 12.7: Battery holders for four and three batteries

Therefore, the preceding battery holders will help us achieve 6 V and 4.5 V.

Now that we know how to connect multiple batteries to get the desired output voltage and charge capacity, let's see how we can use them to power microcontrollers.

Connecting batteries to the microcontroller board

Microcontrollers have dedicated pins for supplying power through external energy sources, such as batteries. These pins have voltage limits commonly reported on the datasheet.

On the Arduino Nano, the external power source is supplied through the VIN pin. The VIN input voltage can range from 5 V–21 V.

On the Raspberry Pi Pico, the external power source is supplied through the **VSYS** pin. The **VSYS** input voltage can range from 1.8 V–5.5 V.

The on-board voltage regulator will convert the supplied voltage to 3.3 V on both platforms.

How to do it...

On the Arduino Nano and Raspberry Pi Pico, upload the pre-built Arduino sketch for blinking the on-board LED, as shown in *Chapter 1, Getting Ready to Unlock ML on Microcontrollers*.

Then, disconnect the Arduino Nano and Raspberry Pi Pico from the micro-USB data cable and place them on the breadboard.



We recommend not inserting the batteries in the battery holder yet. The batteries should only be inserted when the electric circuit is completed.

The following steps will show how to power the Arduino Nano and Raspberry Pi Pico with batteries.

Step 1:

Connect the positive (red) and negative (black) wires of the battery holder to the + and – bus rails, respectively:

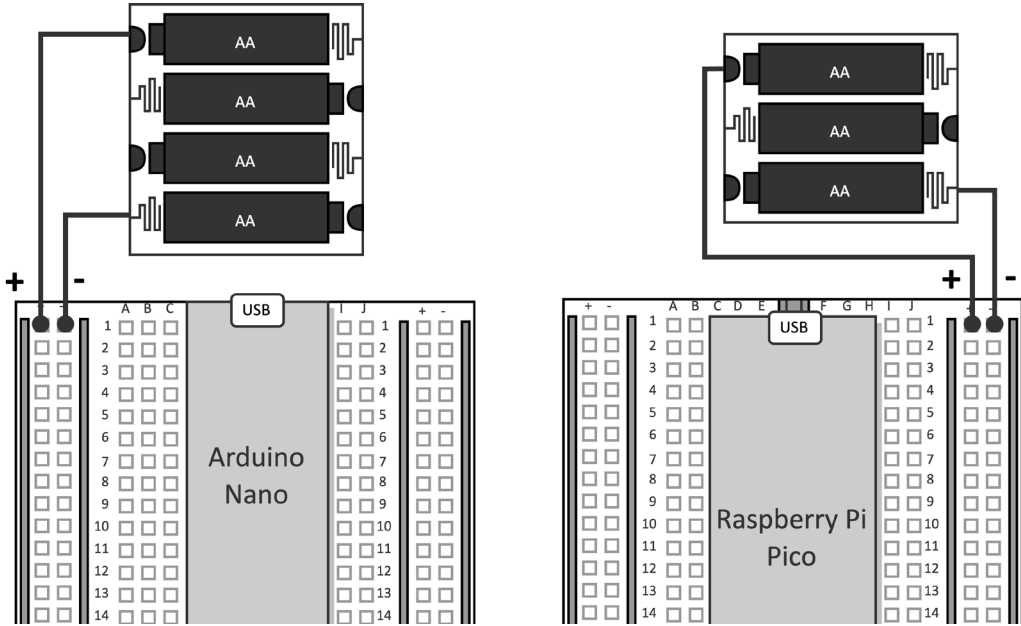


Figure 12.8: Connect the battery holder to the bus rails

The Arduino Nano and Raspberry Pi Pico have different voltage limits for the external power source. Therefore, we cannot use the same number of AA batteries on both platforms. In fact, three AA batteries are enough for the Raspberry Pi Pico but not for the Arduino Nano. In contrast, four AA batteries are enough for the Arduino Nano but beyond the voltage limit of the Raspberry Pi Pico. For this reason, we use a 4 x AA battery holder for the Arduino Nano to supply 6 V and a 3 x AA battery holder for the Raspberry Pi Pico to provide 4.5 V.

Step 2:

Connect the external power source to the microcontroller board, as shown in the following diagram:

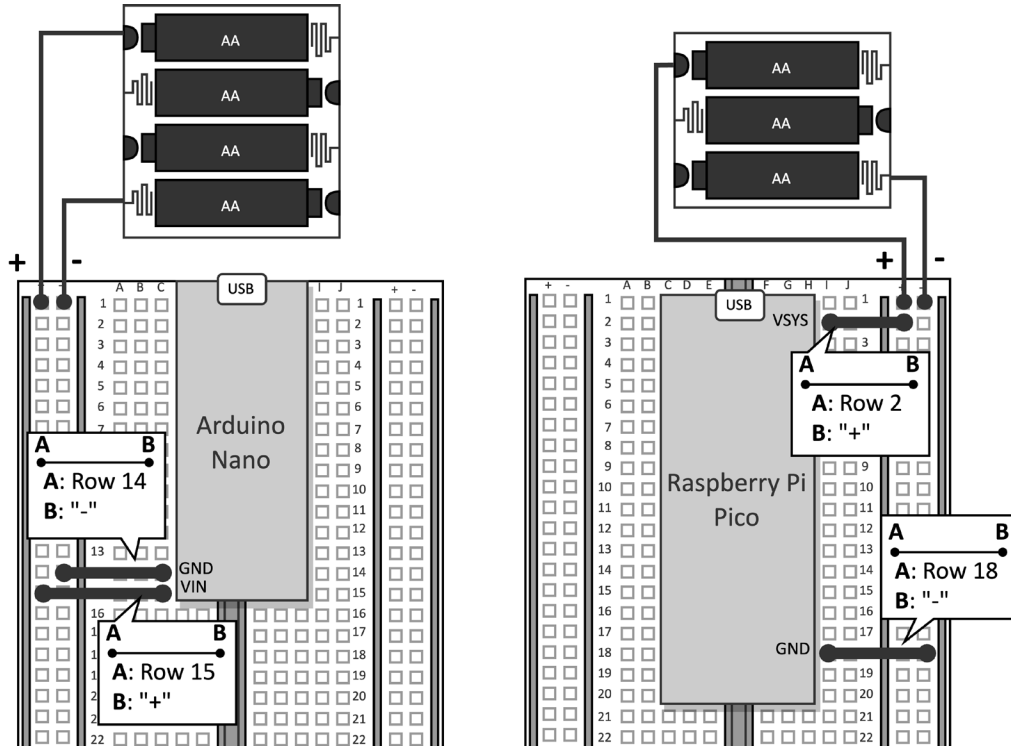


Figure 12.9: Connect the bus rails to the microcontroller power pin and GND

As you can observe from the preceding figure, VIN (Arduino Nano) and VSYS (Raspberry Pi Pico) are connected to the positive battery holder terminal through the + bus rail.

Step 3:

Insert the batteries in the battery holder:

- 4 x AA batteries for the Arduino Nano
- 3 x AA batteries for the Raspberry Pi Pico

Now, you should be able to see the LED blinking without connecting the micro-USB cable to your PC!

There's more...with the SparkFun Artemis Nano

In this recipe, we learned how to power the Arduino Nano and Raspberry Pi Pico using AA batteries.

The SparkFun Artemis Nano exposes a physical pin for supplying power through an external energy source. This pin is the **VIN** pin, which can handle up to 6 V, as reported in the *Hookup Guide for the SparkFun RedBoard Artemis Nano*:

precedence). The VIN pin can handle up to **6V** and will be regulated down to 3.3V using the AP2112 voltage regulator (600mA max output).

Figure 12.10: VIN pin voltage limit reported in the Hookup Guide for the SparkFun RedBoard Artemis Nano (<https://learn.sparkfun.com/tutorials/hookup-guide-for-the-sparkfun-redboard-artemis-nano>)

However, the SparkFun Artemis Nano also provides an on-board **lithium-polymer (LiPo)** connector to power the board with LiPo batteries. To learn more about this connector, we recommend reading the *LiPo Battery and Charging* section in the *Hookup Guide for the SparkFun RedBoard Artemis Nano* (<https://learn.sparkfun.com/tutorials/hookup-guide-for-the-sparkfun-redboard-artemis-nano>).

Summary

In this concluding chapter, we have aimed to address three questions that may have crossed your mind to bring your existing and future tinyML projects to the next level.

The first question of this chapter centered on the practicality of training a model on microcontrollers. Here, we have ascertained that training is possible, albeit with certain constraints. Nonetheless, despite these limitations, the potential offered by on-device learning is vast, as it enables the creation of intelligent devices capable of learning how to interact with the environment autonomously.

Following that question, we explored the feasibility of deploying generic ML algorithms on microcontrollers, such as random forest, to build even more compact tinyML solutions. In this context, we deployed a trained scikit-learn model on microcontrollers using the emlearn project.

The last question was about powering microcontrollers with batteries. Here, we discussed how to connect batteries in series or parallel to achieve the desired target voltage supply and energy capacity. Furthermore, we detailed how to power the Arduino Nano and Raspberry Pi Pico with batteries.

Batteries concluded our practical learning journey into tinyML to remind us of the importance of low power consumption, the gateway to addressing real-world challenges in innovative and powerful ways, as we will discover in the *Conclusion* section.

References

- Morawiec, D. (2023). sklearn-porter: A tool that can transpile trained scikit-learn estimators to C, Java, JavaScript, and other languages: <https://github.com/nok/sklearn-porter>
- Nordby, J., Cooke, M., and Horvath, A. (2019, March). emlearn: Machine Learning inference engine for Microcontrollers and Embedded Devices. Zenodo. <https://doi.org/10.5281/zenodo.2589394>

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

<https://packt.link/tiny>



Conclusion

The key takeaway from all projects developed together is the devices' low power consumption, which enables the deployment of affordable intelligent solutions in a unique and sustainable manner. This uniqueness and sustainability brings me to tell you a story on powering a tinyML application in a very peculiar way.

In 2022, a team of researchers at the University of Cambridge led by **Prof. Christopher J (Jonathan) Howe** and **Dr. Paolo Bombelli** built a groundbreaking device to generate electricity from the photosynthesis activity of algae named **Biophotovoltaics (BPV)** (<https://pubs.rsc.org/en/content/articlelanding/2022/ee/d2ee00233g/unauth>).

This device does not operate like a traditional battery or a solar panel. Contrary to conventional methods, a BPV device harnesses the energy of light and, through photosynthetic processes, extracts electrons from water.

Some of these electrons can be directed into an electrochemical apparatus to generate electricity, while others are utilized to convert carbon dioxide into organic components, such as sugars. These organic components can be re-oxidized even in the absence of light, allowing the BPV device to generate electricity during periods of darkness.

Therefore, we can imagine this device as a natural battery with an energy harvesting system that allows it to generate a voltage continuously:

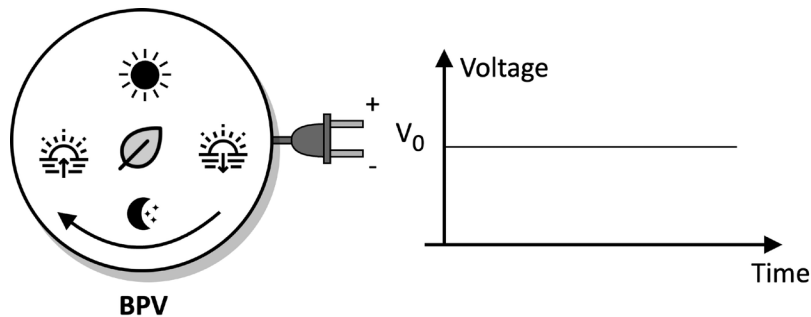


Figure: Representation of the BPV

How much power can this device generate? A single BPV can yield very little power on the **microwatts (uW)** scale. Understandably, this power budget is insufficient to run larger household appliances like microwaves or kettles, which need **kilowatts (kW)** of power, or household devices like lamps or phones, which require **watts**. However, the power budget generated by this device is suitable for smaller electronics that we are very familiar with: microcontrollers.

In their experiment, the device powered an Arm **Cortex-M0** CPU, which was programmed to return the result of a simple arithmetic operation continuously. The investigation was carried out in a domestic environment under natural light, and after six months of continuous operation, they submitted the results for publication.

By the end of 2022, I had the pleasure of meeting Dr. Paolo Bombelli in person during a **tinyML meet-up** in Cambridge. He brought the BPV device, and when I saw it, its size surprised me; it was just slightly larger than a standard AA battery. While its power capacity is undeniably smaller than an AA battery, its potential is vast. One advantage is its eco-friendliness; the BPV battery is built from widely available, long-lasting, affordable, organic, and recyclable materials.

Following the meet-up, Dr. Paolo Bombelli and I decided to collaborate on powering an actual tinyML application running on an Arm Cortex-M0 microcontroller using this device.

The goal was to build a system, consisting of the BPV and a microcontroller, capable of monitoring the water quality in the river Thames (London), based on the following requirements:

- Building the BPV using the native algae of the river
- Adopting open-source tools to deploy the tinyML application on the microcontroller
- Using existing and affordable technologies

This collaboration involved different parties, such as the University of Cambridge, Arm, and the University of the Arts London (Central Saint Martins). In particular, Maria Li from Central Saint Martins was crucial in designing and building the floating BPV, capable of generating up to 10 mW of power, and Omar Al Khatib from Arm to develop the microcontroller board and the software application.

This 10 mW of power budget, which seems relatively low, was more than enough for running the ML model inference on our platform, which was built with just £2.60 and required only 5.4 mW.



To learn more about this project, you can watch the presentation that Dr. Paolo Bombelli and I delivered at the **tinyML EMEA Innovation Forum** held in Amsterdam in June 2023 (https://www.youtube.com/watch?v=_xELDU15_oE).

Given the limited power budget left (4.6 mW), providing internet access to this solution is quite challenging. So, you might question the benefit of monitoring water quality if the results cannot be shared online.

Indeed, the internet would offer tremendous help in sharing information like this instantly and widely. However, in some contexts, such as in rural areas of developing countries, we sometimes just need awareness among the local people living near the river. Therefore, an LED on the floating BPV indicating whether the water is contaminated could be sufficient to alert people and positively impact people's health.

What I have told you is just an example of how simple solutions could help build thriving communities. Often, we assume that all complex problems require equally complex solutions. But, in reality, some of these problems can be easily solved by just thinking tiny.

Nothing can stop you from inventing game-changing solutions founded on tinyML. All you need is a dose of creativity because this technology is easy to use, affordable, and widely supported by the open-source community.



packt.com

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

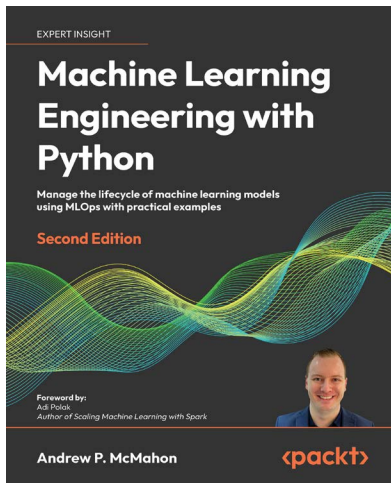
Why subscribe?

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

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

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



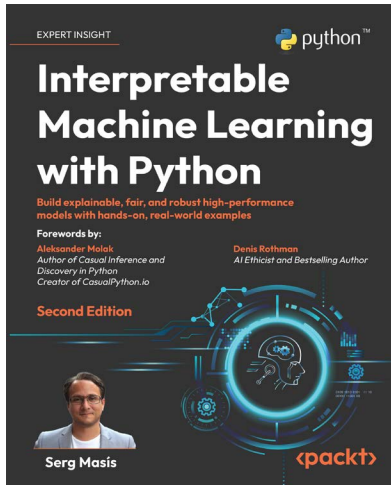
Machine Learning Engineering with Python

Andrew P. McMahon

ISBN: 9781837631964

- Plan and manage end-to-end ML development projects
- Explore deep learning, LLMs, and LLMOps to leverage generative AI
- Use Python to package your ML tools and scale up your solutions
- Get to grips with Apache Spark, Kubernetes, and Ray
- Build and run ML pipelines with Apache Airflow, ZenML, and Kubeflow

- Detect drift and build retraining mechanisms into your solutions
- Improve error handling with control flows and vulnerability scanning
- Host and build ML microservices and batch processes running on AWS



Interpretable Machine Learning with Python

Serg Masís

ISBN: 9781803235424

- Progress from basic to advanced techniques, such as causal inference and quantifying uncertainty
- Build your skillset from analyzing linear and logistic models to complex ones, such as CatBoost, CNNs, and NLP transformers
- Use monotonic and interaction constraints to make fairer and safer models
- Understand how to mitigate the influence of bias in datasets
- Leverage sensitivity analysis factor prioritization and factor fixing for any model
- Discover how to make models more reliable with adversarial robustness

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *TinyML Cookbook, Third Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

16-bit fixed-point (Q15) arithmetic 243

A

accelerometer 409

basic principles 422-425

accelerometer data

acquiring 421-430

accuracy

evaluating, of quantized model 479-483

accuracy of quantized model

evaluating, on test dataset 279-282

activations 8

Adafruit Unified Sensor 128

ADC pin

used, for connecting microphone 199-203

ahead-of-time (AoT) 501

Alif Semiconductor Ensemble

reference link 545

alpha value 383

amperes (A) 14

analog ground (AGND) 202

analog signal 23

analog-to-digital converter (ADC) 23, 197

programming, with Raspberry Pi Pico SDK
205-210

used, for recording audio samples 204

anomaly detection 595

anti-lock braking system (ABS) 19

AoT executor

used, for deploying model with TVM on host
machine 511-524

versus graph executor 515

Apache TVM 501

Apollo3 microcontroller 25

Arduino

URL 24

Arduino API

reference link 205

Arduino CLI 503-507

reference link 156

using, to compile and upload sketches on

Arduino-compatible platforms 507, 509

Arduino Command Line Interface (CLI) 501

Arduino Integrated Development

Environment (Arduino IDE) 26

Arduino Nano 24

keyword spotting on 182-192

KWS application, deploying on 182

model, deploying on 524-536

- reference link 57
- used, for acquiring audio data 153-159
- used, for reading temperature and humidity data 121-124
- Arduino Nano 33 BLE Sense 24**
- Arduino_OV767Xsupport library**
 - software library support 357
- Arduino sketch**
 - building, with code generated by TVM 527-529
 - developing 136
- Arduino TensorFlow Lite library 29**
- Arduino Web Editor**
 - URL 27
- Arm Corstone-300**
 - Fixed Virtual Platform, installing 540-545
- Arm Corstone-300 Fixed Virtual Platform (Corstone-300 FVP) 502**
- Arm Cortex-M 71**
- Arm Cortex-M0 CPU 606**
- Arm Cortex-M CPUs 244**
- Arm Cortex-M microcontroller 195**
- Arm Ecosystem FVPs web page**
 - reference link 543
- Arm Ethos-U55 502**
 - code, generating with TVMC 545-548
- Arm Ethos-U55 microNPU**
 - CIFAR-10 model inference, running on 553-561
- Arm Ethos-U microNPU**
 - software dependencies, installing to build application 549-552
- Arm Mbed OS**
 - application, deploying with 447
 - concurrent tasks, managing with 448
- ARM Mbed OS 410**
- Arm model zoo**
 - reference link 561
- artificial neural networks (ANNs) 8, 590**
- audio**
 - analyzing, in frequency domain 160, 161
- audio data**
 - acquiring, with Arduino Nano 153-159
 - acquiring, with smartphone 146-152
- audio files**
 - generating, from samples transmitted 212-214
- audio samples**
 - collecting, for KWS 147
 - dataset, augmenting with Raspberry Pi Pico 216
 - MFCCs, extracting with TensorFlow 221-223
 - MFE features, extracting from 159, 164-166
 - recording, with ADC 204
 - recording, with Raspberry Pi Pico 203, 204
 - recording, with timer interrupt 204
- augmented reality/virtual reality (AR/VR) 411**
- authentication protocol (OAuth 2.0) 45**
- automatic gain control (AGC) 197**
- B**
- backpropagation 565, 566**
 - neural network, training with 568-588
- balanced dataset**
 - building 98, 99
- batch size 569**
- batteries**
 - connecting, to microcontroller board 599-601
 - microcontrollers, powering with 596
- baud rate 39**

- bias** 8
- bilinear interpolation**
 - used, for image resizing 395-406
- binary classification** 92
- binary point** 246
- binary signals** 22
- BioPhotoVoltaic (BPV)** 605
- bit depth** 204
- bottleneck residual block** 383
- breadboard** 37
 - LED status indicator, implementing 56-63
 - prototyping 58, 59
- bus rails** 58, 59
- Butterworth filter** 442
- button bouncing** 81
- byte alignment** 556
- C**
- camera frames**
 - grabbing, from serial port with Python 361, 362
 - RGB565 color format 362, 363
- C-byte array**
 - NumPy image, converting into 484-487
- centroid coordinates**
 - transmitting, to Python script 344
- CIFAR-10 model**
 - designing and training, for memory-constrained devices 470-478
 - inference, running on Arm Ethos-U55 microNPU 553-561
 - memory requirement, keeping under control 474
- circular buffer** 133
- classic fully connected neural networks** 9
- clustering** 12
- CMSIS-DSP, and TensorFlow implementation**
 - visualization of numerical differences 265
- CMSIS-DSP library** 574
 - FFT magnitude, computing with fixed-point arithmetic 243-253
 - MFCCs feature extraction, implementing with 253-264
 - reference link 244
 - using 244, 245
- CMSIS library** 549
- CMSIS NN** 137
- CMSIS-NN library** 511
- code generated, by TVM**
 - Arduino sketch, building with 527-529
- code portability** 511
- code, with TVM**
 - generating, to run CIFAR-10 model inference on the Raspberry Pi Pico 540
- code, with TVMC**
 - generating, for Arm Ethos-U55 545-548
- Colaboratory**
 - URL 28
- command stream** 542
- compiler** 549
- computer vision** 303
- concurrent tasks**
 - managing, with Arm Mbed OS 448
- confusion matrix**
 - model performance, evaluating 107, 108
- convolution**
 - replacing, with depthwise separable convolution 471, 473
- convolutional neural networks (CNNs)** 9, 160, 318, 470
 - designing and training 169-174

convolution layers 10

core compute blocks 239

CoreSight Architecture

reference link 42

Corstone-300 FVP

memory capabilities, overview 554-556

coulombs (C) 14

cross-compiler 549

current 13, 14

D

data memory 20

dataset

balancing 92, 93

building, for music genre classification 215, 216

building, to classify desk objects 375-379

building, with Edge Impulse data forwarder tool 432-437

preparing 92

scaling with Z-score, feature 93-97

Data Tightly Coupled Memory (DTCM) 555

DCT coefficients

computing 231

extracting 254, 255

DCT-weight matrix 255

debouncing algorithm 81

decision tree 590

deep learning (DL) 7, 169, 195

compilers 513

deep neural network (DNN) 7, 137

deployment environments, for tinyML 5, 6

depthwise convolution layers 381

depthwise separable convolution 382

convolution, replacing with 471-473

desk objects

classifying, with dataset 375-379

development platforms 24

DHT22 sensor

connecting, to Raspberry Pi Pico 131

temperature and humidity, reading with 124-130

DHT sensor library 131

Digi-Key

reference link 198

URL 15

web tool, reference link 62

DigitalIn interface

reference link 78

DigitalOut interface

reference link 70

Digital Signal Processing (DSP) 294, 340, 549

digital-to-analog converter (DAC) 23

direct memory access (DMA) 183

Discrete Cosine Transform (DCT) 223, 254

Discrete Fourier Transform (DFT) 161

dominant frequencies 440

double-buffering mechanism 183

dropout layer 169

E

Edge Impulse 30, 145, 303

data, collecting with fully supported platform in 155

FOMO model, training 323, 324

model accuracy, evaluating ways 178

sequential computational blocks 160

training samples, extracting 166-169

URL 30

used, for recording audio with smartphone 153

Edge Impulse CLI

reference link 156

Edge Impulse data forwarder tool 409, 447

application, deploying with 447

dataset, building with 432-437

for live classifications 446, 447

Edge Impulse Inferencing SDK 339**ei_impulse_result_t data structure**

output results, reading from 343, 344

emlearn

reference link 590

end-to-end (E2E) 145**energy 13-15****energy capacity**

increasing, by connecting batteries in parallel 597-599

EON Tuner 145

used, for tuning model performance 174-178

epochs 569**Ethos-U driver 549****Ethos-U platform 549, 550****external LED**

controlling, with GPIO 64

F**false negative (FN) 108****false positive (FP) 108****Faster Objects, More Objects (FOMO) 303**

dataset, building 306

design 317, 318

objects, locating from heat maps 318-320

deploying, on Raspberry Pi Pico 339-350

transfer learning with 316, 321-323

training, in Edge Impulse 323, 324

Fast Fourier Transform

(FFT) 161, 223, 244, 440

feature compressor 383**feature explorer chart 168****feature maps 11, 382****feedforward neural network**

training 106

FFT magnitude

calculating 226, 227

computing, with fixed-point arithmetic 243-253

FFT-spectrogram 162**files**

uploading, to Google Drive with Python 43, 51-55

fine-tuning 388**First-In-First-Out (FIFO) 133****fixed-point integer format 245****Fixed Virtual Platform**

installing, for Arm Corstone-300 540-545

flashing 34**floating 77****forward current 65****forward voltage 65****frame length 162****frames per second (FPS) 336****frame step 162****frequency domain**

audio, analyzing 160, 161

frequency resolution 226**F-score**

evaluating 109

fully connected layer 9, 169

G

General-Purpose Input-Output (GPIO) 22, 23, 189

used, for controlling external LED 64

gesture-based interface

building, with PyAutoGUI 459-462

gesture recognition 409

GNU Arm Embedded Toolchain 549

Google Cloud

URL 45

Google Colab 44

Google Drive 37

Google Drive API

enabling 44-50

reference link 45

Google Drive file ID 53

Google service API 44

GPIO peripheral 37, 68-70

programming, in input mode 80, 81

programming, in output mode 74, 75

GPIO pin

Light Emitting Diode (LED), connecting to 71-74

push button, connecting to 78-80

gradient 572

gradient descent approach 573

graph executor

versus AoT executor 515

Grayscale 375

GTZAN dataset

using, for music genre classification 216

H

hand gestures

recognizing, with spectral analysis 439-444

Hann windowing

applying 223, 224

DCT coefficients, computing 230

FFT magnitude, calculating 226, 227

Mel scale conversion 227-230

Real Fast Fourier Transform (RFFT), using 225, 226

SRAM usage, evaluating to run MFCCs 231-239

hard real time 19

heat maps

objects, locating from 318-320

hertz 161

high impedance 77

hotkeys 459

Hz 161

I

I2C peripheral 23, 409

accessing, in Raspberry Pi Pico 415-420

programming, with Mbed OS 415

image classification 303

ImageNet dataset classification 471

images, with webcam

acquiring 305-310

dataset, building for FOMO 306

Impulse's pre-processing block

designing 311-316

input image resolution, selecting 312, 313

inference thread 448

input features

preparing, for model inference 132-136

input image resolution

selecting 312, 313

input mode

GPIO peripheral, programming 80, 81

input test image

downloading 509, 510

Instruction Tightly Coupled Memory (ITCM) 555**Inter-Integrated Circuit (I2C) 411**

basics 412-414

MPU-6050 IMU, communicating through 411

Internet of things (IoT) 3**interrupt handler 82****InterruptIn**

reference link 83

interrupts 82

used, for reading push button state 82-85

working, with Mbed OS API 82-84

interrupt service routine (ISR) 82, 204**inverted residual block 382****J****jumper wires 59****K****Keras**

used, for transfer learning 380, 381

Keras code mode (expert mode) 169**keyword dataset, from Edge Impulse**

download link 152

keyword spotting (KWS) 145, 295

audio samples, collecting 147

kilowatts (kW) 606**L****latency-predictable device 19****least-significant bit over g (LSB/g) 424****LED functionality 64-67****LED status indicator**

implementing, on breadboard 56-63

Librosa library

reference link 239

light-emitting diode (LED) 37, 64, 145

connecting, to GPIO pin 71-74

turning, on/off with push button 75

live classifications

with Edge Impulse data forwarder tool 446, 447

with smartphone 179-182

Logarithmic (Log) function 223**logical (exclusive OR) XOR operator 565****logistic regression 590****Log-Mel spectrogram 230****loss function 568****LSTM cells 269****LSTM RNN model**

designing 266-278

training 266-278

M**machine learning (ML) 1, 37, 195, 303, 409**

models 145, 501

reasons, to run locally 4

many-to-one RNN

designing, for music genre classification 270, 271

MAX9814 datasheet

reference link 202

- mbed::DigitalOut** function
 - using 70, 71
- Mbed OS**
 - I2C peripheral, programming with 415
 - URL 71
- Mbed OS API**
 - used, for working with interrupts 82-84
- mean squared error (MSE)** 568
- Meetup**
 - URL 7
- Mel filter-bank energy (MFE)** 159
- Mel Frequency Cepstral Coefficients (MFCCs)** 196
 - extracting, from audio samples with TensorFlow 221-223
 - running, with SRAM usage evaluation 231-239
- Mel scale** 227
- Mel scale conversion** 227-230
- Mel-spectrogram**
 - extracting 161-163
- memory-constrained devices**
 - CIFAR-10 model, designing and training for 470-478
- memory management unit (MMU)** 19, 516
- memory manager** 474
- memory section attribute** 555
- MFCCs feature extraction**
 - implementing, with CMSIS-DSP Python library 253-264
- MFCCs feature extraction algorithm**
 - deploying, on Raspberry Pi Pico 283-293
- MFE features**
 - extracting, from audio samples 159, 164-166
- micro-ampere (μA)** 16, 124
- microcontroller board** 24
 - batteries, connecting to 599-601
- microcontroller deployment**
 - microTVM 516, 517
- microcontrollers** 17-19
 - memory architecture 20, 21
 - model, training on 565, 566
 - peripherals 21
 - powering, with batteries 596
 - reasons, for selecting 3, 4
 - scikit-learn models, deploying on 590-595
 - sketches, deploying on 31-33
- micro-electromechanical systems (MEMS)** 424
- microjoule** 16
- micromlgen**
 - reference link 590
- Micro-Neural Processing Unit (microNPU)** 502, 540
- microphone**
 - connecting, to ADC pin 199-203
 - connecting, to Raspberry Pi Pico 197-199
- microprocessor** 17, 18
- microTVM**
 - for microcontroller deployment 516, 517
- microwatts (μW)** 16, 606
- milli-ampere-hour (mAh)** 596
- milli-ampere (mA)** 16
- millijoule (mJ)** 16
- milliwatt (mW)** 13, 16
- ML model**
 - designing 438
 - training 438
- ML model, in Edge Impulse**
 - designing and training 174

MobileNet v2 381

MobileNet v2 pre-trained model

transfer learning, applying with 383-388

model

deploying, on Arduino Nano 524-536

deploying, on Raspberry Pi Pico 536-540

model inference

input features, preparing 132-136

running, with TVM runtime 527

model library format (MLF) 517

model performance

tuning, with EON Tuner 175-178

model quantization 12, 13

model accuracy

evaluating 325-329

model input length

selecting 217-220

model performance

evaluating 107-113

evaluating, with confusion matrix 107, 108

F-score, evaluating 109

precision, evaluating 109

recall, evaluating 109

model, with TensorFlow

training 99-106

model, with TensorFlow Lite converter

quantizing 113-121

model, with TVM

deploying, with AoT executor on host machine 511-524

moving average (MA) 190

MPU-6050 Inertial Measurement Unit (IMU) 411

communicating with 411

features 411, 412

reference link 411

multiply and accumulate (MAC) 473, 541

music genre classification

GTZAN dataset, using 216

used, for building dataset 215

music genres

recognizing, with Raspberry Pi Pico 295-301

N

natural language processing (NLP) 267

Netron web application 492

neural network

training, with backpropagation 568-588

neuron 8

non-volatile read-only memory (ROM) 20

NOT-AND (NAND) operator 565

numbers

representing, in 16-bit fixed-point format 245, 246

NumPy image

converting, into C-byte array 484-487

Nyquist frequency 226

Nyquist-Shannon sampling theorem 204

O

object detection 303

Ohm's law 14

on-board lithium-polymer (LiPo) connector 602

on-device inference

with TensorFlow Lite for Microcontrollers (tflite-micro) 136-143

on-device model training (on-device learning) 565

on-device transfer learning algorithm 589

OpenCV

- URL 330

- used, for sending images over serial interface 330-337

operating system (OS) 131, 157, 549**OPS per Watt 17****output feature maps (OFMs) 169, 471****output mode**

- GPIO peripheral, programming 74, 75

output voltage

- increasing, by connecting batteries in series 597

OV7670 camera module

- color format support 355
- low frame resolution support 355
- software library support 355
- used, for capturing pictures 354-361

overfitting 10**P****peak-to-peak voltage (Vpp) 198****peripherals, microcontrollers 21**

- analog/digital converters 23
- general-purpose input/output (GPIO) 22, 23
- serial communication 23
- timers 23

Pillow

- reference link 361

pin name (PinName) 71**Pixabay**

- reference link 216
- URL 483

playsound

- reference link 215

pointwise convolution 472**pooling layers 11****power 13-15****precision 246**

- evaluating 109

preemptive scheduler 448**pre-processing operators**

- fusing, for efficient deployment 393-395

pre-trained CIFAR-10 model

- downloading 509, 510

pre-trained model 380**printed circuit board (PCB) 24****program memory 20****pruning 12****pulse density modulation (PDM) 203****push button 37**

- connecting, to GPIO pin 78-80
- operating principles 76-78
- used, for turning LED on/off 75

push button state

- reading, with interrupts 82-85

PyAutoGUI 409, 459

- gesture-based interface, building 459-462
- URL 459

PyDrive 44, 55

- reference link 55

pySerial 212, 361, 459

- used, for reading data from serial port 44
- used, for sending images over serial interface 330-337

Python

- used, for grabbing camera frames from serial port 361, 362
- used, for reading serial data 43, 52-55
- used, for uploading files to Google Drive 43, 51-55

Python script

- centroid coordinates, transmitting to 344

Python TensorFlow Lite interpreter 279**Python Virtual Environment (virtualenv) 50****Q****QEMU 466**

- TensorFlow Lite for Microcontrollers program, deploying on 491-497
- URL 466

QQVGA images

- acquiring, with YCbCr422 color format 370-375

quantization 114, 279**quantization asymmetric 116****quantization parameters 115****quantization symmetric 116****quantized model**

- accuracy, evaluating of 479-483

Quick Response (QR) 310**Qwiic 4-pin connector 420****Qwiic connector**

- reference link 420

R**random forest model 590****Raspberry Pi Pico 24, 25, 304**

- DHT22 sensor, connecting with 131
- FOMO model, deploying on 339-341, 345-50
- hardware capabilities 243
- I2C peripheral, accessing in 415-420
- MFCCs feature extraction algorithm, deploying on 283-293
- model, deploying on 536-540
- music genres, recognizing with 295-301

- reference link 57

- temperature and humidity, reading with 125-130

- URL 25

- used, for augmenting audio sample dataset 216

- used, for connecting microphone 197, 198

- used, for recording audio samples 203, 204

Raspberry Pi Pico SDK

- reference link 205

- used, for programming ADC 205-210

raster scan order 364**raw audio waveform 150****raw data 43****Real Fast Fourier Transform (RFFT) 224, 244**

- using 226

real-time applications (RTAs) 19**real-time KWS application**

- usage 183-185

real-time operating system

- (RTOS) 71, 183, 205, 466, 549

real-time operating system (RTOS) APIs 447**recall**

- evaluating 109

rectified linear unit (ReLU) 9**redundant and spurious predictions**

- filtering 449-458

Relay 513**residual block 382****resistance 15****resistor 15****REST API 45****RNNs**

- for time series analysis 266-270

round-robin priority-based scheduling
algorithm 448

RP2040 microcontroller 25

S

sample rate 203

sampling thread 448

scikit-learn library 274

reference link 590

scikit-learn models

deploying, on microcontrollers 590-595

SCL (clock signal) 412

scratch buffer 474

SDA (data signal) 412

sensitivity 424

sensor 23

serial communication 23

data, transmitting 38-42

serial data

reading, with Python 43, 51-55

serial interface

OpenCV and pySerial, used for sending
images over 330-337

Serial Plotter tool 430

serial port

data, reading with pySerial 44

images, transmitting over 363-370

serial port, with Arduino-compatible
platforms

data, reading from 337-339

Serial.print()

reference link 39

serial wire debug (SWD) 42

serial, with pySerial

bytes, sending over 331

shallow neural network

used, for solving XOR and NAND problem
566, 567

signal_t data structure

initializing 341-343

sketches 27

deploying, on microcontrollers 31-33

sklearn-porter

reference link 590

smartphone

live classifications with 178-182

used, for acquiring audio data 146-152

softmax activation 169

soft real time 19

software dependencies

installing, to build application for Arm
Ethos-U microNPU 549-552

software development environment

setting up 26

Software Development Kit (SDK) 205, 466

soundfile library 212

SparkFun Artemis Nano 24, 34, 35, 63, 75, 82,
132, 211, 316

application, deploying 221

reference link 57

SparkFun Electronics

URL 25

SparkFun RedBoard Artemis Nano 25, 26

reference link 63

spectral analysis 439

using, to recognize hand gestures 439-444

speech recognition 267

SPI 23

SRAM usage

evaluating, to run MFCCs 231-239

static random-access memory (SRAM) 21

subsampling 11

support vector machine (SVM) 590

T

temperature and humidity data

reading, with Arduino Nano 121-124

reading, with DHT22 sensor and Raspberry Pi Pico 124-130

tensor 11

TensorFlow 28, 29

URL 28

used, for extracting MFCCs from audio samples 221-223

TensorFlow Lite

trained model, quantizing with 389-393

trained model, testing with 389-391

TensorFlow Lite for Microcontrollers (tflite-micro) program 29, 30, 501

deploying, on QEMU 491-497

on-device inference with 136-143

URL 29

using 143

TensorFlow Lite Interpreter 137

terminal strips 59

test dataset

accuracy of quantized model, evaluating on 279-282

test pattern mode 366

thread management 448

three-axis accelerometer sensors 411

three-axis gyroscope sensors 411

timer interrupt

used, for recording audio samples 204

Timer peripheral 85

timers 23

Timers interrupts and tasks

reference link 81

time series analysis 267

with RNNs 266-270

TinyML EMEA Innovation Forum

reference link 606

tinyML Foundation 7

URL 7

TinyML On Device Learning Forum

reference link 566

traditional CNNs 11, 12

transfer learning 306

applying, with MobileNet v2 pre-trained model 383-388

with Keras 380, 381

triangular filter banks 228

true negative (TN) 108

true positive (TP) 108

TVM

used, for generating code 514, 515

TVM compiler 513, 514

TVM runtime

model inference, running with 527

two-dimensional (2D) 168

U

UART 23, 39

unbalanced dataset 92

Unified Static Memory Planning (USMP) 536

Uniform Resource Locator (URL) 53, 218, 379

unseen data 10

USART 39

USB 23

UTF-8 format 52

V

vanishing gradient 269
Vela compiler 548
Virtual Environment (virtualenv) 332, 467
volatile keyword 207
volatile memory 21
voltage 14
volts (V) 14
VSYs pin 599

W

watts 606
wav2letter model 561
weather data
 importing, from WorldWeatherOnline 88-92
weights 8
west tool
 reference link 468
wireless sensor network (WSN) 6
WorldWeatherOnline
 weather data, importing from 88-92

X

XOR and NAND problem
 solving, with shallow neural
 network 566, 567

Y

YCbCr422 color format
 converting, to RGB888 371, 372
 used, for acquiring QQVGA images 370-375

Z

Zephyr 466
 used, for running application on virtual
 platform through QEMU 467-469
Zephyr Project structure
 preparing 488-491
Zephyr SDK 467
Z-score
 scaling with 93

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?
Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781837637362>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

