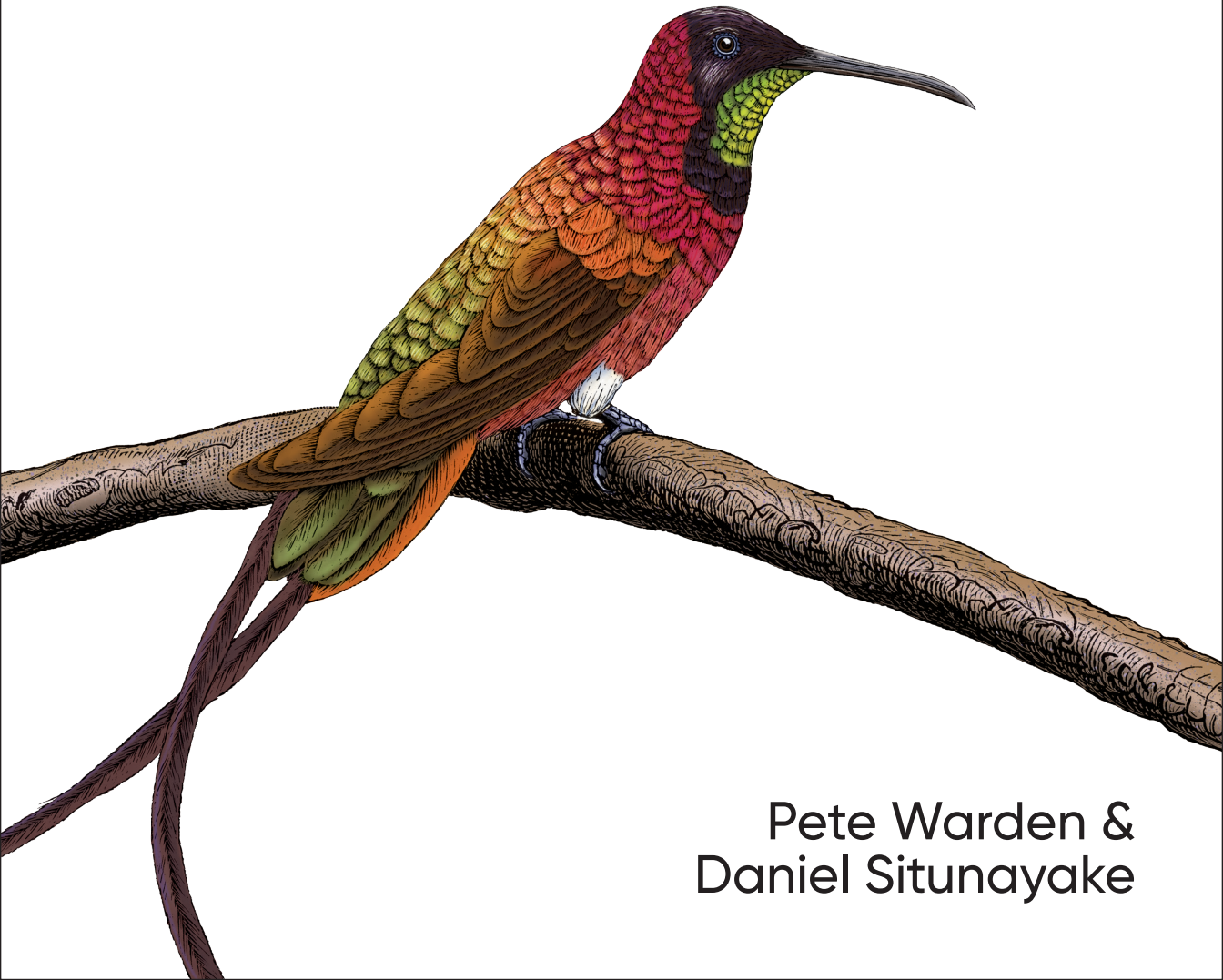
O'REILLY®

PREVIEW OF FIRST SIX CHAPTERS

Buy the full book at tinymlbook.com

TinyML

Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers



Pete Warden &
Daniel Situnayake

O'REILLY®

TinyML

Deep learning networks are getting smaller. Much smaller. The Google Assistant team can detect words with a model just 14 kilobytes in size—small enough to run on a microcontroller. With this practical book you'll enter the field of TinyML, where deep learning and embedded systems combine to make astounding things possible with tiny devices.

Pete Warden and Daniel Situnayake explain how you can train models small enough to fit into any environment. Ideal for software and hardware developers who want to build embedded systems using machine learning, this guide walks you through creating a series of TinyML projects, step-by-step. No machine learning or microcontroller experience is necessary.

- Build a speech recognizer, a camera that detects people, and a magic wand that responds to gestures
- Work with Arduino and ultra-low-power microcontrollers
- Learn the essentials of ML and how to train your own models
- Train models to understand audio, image, and accelerometer data
- Explore TensorFlow Lite for Microcontrollers, Google's toolkit for TinyML
- Debug applications and provide safeguards for privacy and security
- Optimize latency, energy usage, and model and binary size

"This is a must-read book for anyone interested in machine learning on resource-constrained devices. It is a milestone in the development of AI."

—**Massimo Banzi**
Cofounder, Arduino

"This book teaches—through clear, interesting use cases—how to deploy ML on Arm-based microcontrollers."

—**Jem Davies**
VP, Fellow and GM, Machine Learning Group, Arm

Pete Warden is technical lead for mobile and embedded TensorFlow, and a founding member of the TensorFlow team. He was previously CTO and founder of Jetpac, acquired by Google in 2014.

Daniel Situnayake leads developer advocacy for TensorFlow Lite at Google, and helps run the tinyML meetup group. He cofounded Tiny Farms, the first US company using automation to produce insect protein at industrial scale.

MACHINE LEARNING / EMBEDDED DEVICES

US $49.99 CAN $65.99

ISBN: 978-1-492-05204-3

Twitter: @oreillymedia
facebook.com/oreilly

TinyML

Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers

PREVIEW OF FIRST SIX CHAPTERS

Buy the full book at tinymlbook.com

Pete Warden and Daniel Situnayake

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

TinyML
by Pete Warden and Daniel Situnayake

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Mike Loukides
Development Editor: Nicole Tache
Production Editor: Beth Kelly
Copyeditor: Octal Publishing, Inc.
Proofreader: Rachel Head
Indexer: WordCo, Inc.
Interior Designer: David Futato
Illustrator: Rebecca Demarest

December 2019: First Edition

Revision History for the Early Release

2019-6-19: First Release
2019-9-27: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492052043* for release details.

978-1-492-05204-3

[LSI]

Table of Contents

PREVIEW OF FIRST SIX CHAPTERS

Buy the full book at tinymlbook.com

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://tinymlbook.com/supplemental*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*TinyML* by Pete Warden and Daniel Situnayake (O'Reilly). Copyright Pete Warden and Daniel Situnayake, 978-1-492-05204-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/tiny*.

Email *tinyml-book@googlegroups.com* to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

We'd like to give special thanks to Nicole Tache for her wonderful editing, Jennifer Wang for her inspirational magic wand example, and Neil Tan for the groundbreaking embedded ML work he did with the uTensor library. We couldn't have written this book without the professional support of Rajat Monga and Sarah Sirajuddin. We'd also like to thank our partners Joanne Ladolcetta and Lauren Ward for their patience.

This book is the result of work from hundreds of people from across the hardware, software, and research world, especially on the TensorFlow team. While we can only mention a few, and apologies to everyone we've missed, we'd like to acknowledge: Mehmet Ali Anil, Alasdair Allan, Raziel Alvarez, Paige Bailey, Massimo Banzi, Raj Batra, Mary Bennion, Jeff Bier, Lukas Biewald, Ian Bratt, Laurence Campbell, Andrew Cavanaugh, Lawrence Chan, Vikas Chandra, Marcus Chang, Tony Chiang, Aakanksha Chowdhery, Rod Crawford, Robert David, Tim Davis, Hongyang Deng, Wolff Dobson, Jared Duke, Jens Elofsson, Johan Euphrosine, Martino Facchin, Limor Fried, Nupur Garg, Nicholas Gillian, Evgeni Gousev, Alessandro Grande, Song Han, Justin Hong, Sara Hooker, Andrew Howard, Magnus Hyttsten, Advait Jain, Nat Jeffries, Michael Jones, Mat Kelcey, Kurt Keutzer, Fredrik Knutsson, Nick Kreeger, Nic Lane,

Shuangfeng Li, Mike Liang, Yu-Cheng Ling, Renjie Liu, Mike Loukides, Owen Lyke, Cristian Maglie, Bill Mark, Matthew Mattina, Sandeep Mistry, Amit Mittra, Laurence Moroney, Boris Murmann, Ian Nappier, Meghna Natraj, Ben Nuttall, Dominic Pajak, Dave Patterson, Dario Pennisi, Jahnell Pereira, Raaj Prasad, Frederic Rechtenstein, Vikas Reddi, Rocky Rhodes, David Rim, Kazunori Sato, Nathan Seidle, Andrew Selle, Arpit Shah, Marcus Shawcroft, Zach Shelby, Suharsh Sivakumar, Ravishankar Sivalingam, Rex St. John, Dominic Symes, Olivier Temam, Phillip Torrone, Stephan Uphoff, Eben Upton, Lu Wang, Tiezhen Wang, Paul Whatmough, Tom White, Edd Wilder-James, and Wei Xiao.

CHAPTER 1
Introduction

The goal of this book is to show how any developer with basic experience using a command-line terminal and code editor can get started building their own projects running machine learning (ML) on embedded devices.

When I first joined Google in 2014, I discovered a lot of internal projects that I had no idea existed, but the most exciting was the work that the OK Google team were doing. They were running neural networks that were just 14 kilobytes (KB) in size! They needed to be so small because they were running on the digital signal processors (DSPs) present in most Android phones, continuously listening for the "OK Google" wake words, and these DSPs had only tens of kilobytes of RAM and flash memory. The team had to use the DSPs for this job because the main CPU was powered off to conserve battery, and these specialized chips use only a few milliwatts (mW) of power.

Coming from the image side of deep learning, I'd never seen networks so small, and the idea that you could use such low-power chips to run neural models stuck with me. As I worked on getting TensorFlow and later TensorFlow Lite running on Android and iOS devices, I remained fascinated by the possibilities of working with even simple chips. I learned that there were other pioneering projects in the audio world (like Pixel's Music IQ) for predictive maintenance (like PsiKick) and even in the vision world (Qualcomm's Glance camera module).

It became clear to me that there was a whole new class of products emerging, with the key characteristics that they used ML to make sense of noisy sensor data, could run using a battery or energy harvesting for years, and cost only a dollar or two. One term I heard repeatedly was "peel-and-stick sensors," for devices that required no battery changes and could be applied anywhere in an environment and forgotten. Making these products real required ways to turn raw sensor data into actionable information

locally, on the device itself, since the energy costs of transmitting streams anywhere have proved to be inherently too high to be practical.

This is where the idea of TinyML comes in. Long conversations with colleagues across industry and academia have led to the rough consensus that if you can run a neural network model at an energy cost of below 1 mW, it makes a lot of entirely new applications possible. This might seem like a somewhat arbitrary number, but if you translate it into concrete terms, it means a device running on a coin battery has a lifetime of a year. That results in a product that's small enough to fit into any environment and able to run for a useful amount of time without any human intervention.

I'm going to be jumping straight into using some technical terms to talk about what this book will be covering, but don't worry if some of them are unfamiliar to you; we define their meaning the first time we use them.

At this point, you might be wondering about platforms like the Raspberry Pi, or NVIDIA's Jetson boards. These are fantastic devices, and I use them myself frequently, but even the smallest Pi is similar to a mobile phone's main CPU and so draws hundreds of milliwatts. Keeping one running even for a few days requires a battery similar to a smartphone's, making it difficult to build truly untethered experiences. NVIDIA's Jetson is based on a powerful GPU, and we've seen it use up to 12 watts of power when running at full speed, so it's even more difficult to use without a large external power supply. This is usually not a problem in automotive or robotics applications, since the mechanical parts demand a large power source themselves, but it does make it tough to use these platforms for the kinds of products I'm most interested in, which need to operate without a wired power supply. Happily, when using them the lack of resource constraints means that frameworks like TensorFlow, TensorFlow Lite, and NVIDIA's TensorRT are available, since they're usually based on Linux-capable Arm Cortex-A CPUs, which have hundreds of megabytes of memory. This book will not be focused on describing how to run on those platforms for the reason just mentioned, but if you're interested, there are a lot of resources and documentation available; for example, see TensorFlow Lite's mobile documentation.

Another characteristic I care about is cost. The cheapest Raspberry Pi Zero is $5 for makers, but it is extremely difficult to buy that class of chip in large numbers at that price. Purchases of the Zero are usually restricted by quantity, and while the prices for industrial purchases aren't transparent, it's clear that $5 is definitely unusual. By contrast, the cheapest 32-bit microcontrollers cost much less than a dollar each. This low price has made it possible for manufacturers to replace traditional analog or electromechanical control circuits with software-defined alternatives for everything from toys to washing machines. I'm hoping we can use the ubiquity of microcontrollers in these devices to introduce artificial intelligence as a software update, without requir-

ing a lot of changes to existing designs. It should also make it possible to get large numbers of smart sensors deployed across environments like buildings or wildlife reserves without the costs outweighing the benefits or funds available.

Embedded Devices

The definition of TinyML as having an energy cost below 1 mW does mean that we need to look to the world of embedded devices for our hardware platforms. Until a few years ago, I wasn't familiar with them myself—they were shrouded in mystery for me. Traditionally they had been 8-bit devices and used obscure and proprietary toolchains, so it seemed very intimidating to get started with any of them. A big step forward came when Arduino introduced a user-friendly integrated development environment (IDE) along with standardized hardware. Since then, 32-bit CPUs have become the standard, largely thanks to Arm's Cortex-M series of chips. When I started to prototype some ML experiments a couple of years ago, I was pleasantly surprised by how relatively straightforward the development process had become.

Embedded devices still come with some tough resource constraints, though. They often have only a few hundred kilobytes of RAM, or sometimes much less than that, and have similar amounts of flash memory for persistent program and data storage. A clock speed of just tens of megahertz is not unusual. They will definitely not have full Linux (since that requires a memory controller and at least one megabyte of RAM), and if there is an operating system, it may well not provide all or any of the POSIX or standard C library functions you expect. Many embedded systems avoid using dynamic memory allocation functions like `new` or `malloc()` because they're designed to be reliable and long-running, and it's extremely difficult to ensure that if you have a heap that can be fragmented. You might also find it tricky to use a debugger or other familiar tools from desktop development, since the interfaces you'll be using to access the chip are very specialized.

There were some nice surprises as I learned embedded development, though. Having a system with no other processes to interrupt your program can make building a mental model of what's happening very simple, and the straightforward nature of a processor without branch prediction or instruction pipelining makes manual assembly optimization a lot easier than on more complex CPUs. I also find a simple joy in seeing LEDs light up on a miniature computer that I can balance on a fingertip, knowing that it's running millions of instructions a second to understand the world around it.

Changing Landscape

It's only recently that we've been able to run ML on microcontrollers at all, and the field is very young, which means hardware, software, and research are all changing

extremely quickly. This book is a based on a snapshot of the world as it existed in 2019, which in this area means some parts were out of date before we'd even finished writing the last chapter. We've tried to make sure we're relying on hardware platforms that will be available over the long term, but it's likely that devices will continue to improve and evolve. The TensorFlow Lite software framework that we use has a stable API, and we'll continue to support the examples we give in the text over time, but we also provide web links to the very latest versions of all our sample code and documentation. You can expect to see reference applications covering more use cases than we have in this book being added to the TensorFlow repository, for example. We also aim to focus on skills like debugging, model creation, and developing an understanding of how deep learning works, which will remain useful even as the infrastructure you're using changes.

We want this book to give you the foundation you need to develop embedded ML products to solve problems you care about. Hopefully we'll be able to start you along the road of building some of the exciting new applications I'm certain will be emerging over the next few years in this domain.

—Pete Warden

CHAPTER 2
Getting Started

In this chapter, we cover what you need to know to begin building and modifying machine learning applications on low-power devices. All of the software is free, and the hardware development kits are available for less than $30, so the biggest challenge is likely to be the unfamiliarity of the development environment. To help with that, throughout the chapter we recommend a well-lit path of tools that we've found work well together.

Who Is This Book Aimed At?

To build a TinyML project, you will need to know a bit about both machine learning and embedded software development. Neither of these are common skills, and very few people are experts on both, so this book will start with the assumption that you have no background in either of these. The only requirements are that you have some familiarity running commands in the terminal (or Command Prompt on Windows), and are able to load a program source file into an editor, make alterations, and save it. Even if that sounds daunting, we walk you through everything we discuss step by step, like a good recipe, including screenshots (and screencasts online) in many cases, so we're hoping to make this as accessible as possible to a wide audience.

We'll show you some practical applications of machine learning on embedded devices, using projects like simple speech recognition, detecting gestures with a motion sensor, and detecting people with a camera sensor. We want to get you comfortable with building these programs yourself, and then extending them to solve problems you care about. For example, you might want to modify the speech recognition to detect barks instead of human speech, or spot dogs instead of people, and we give you ideas on how to tackle those modifications yourself. Our goal is to provide you with the tools you need to start building exciting applications you care about.

What Hardware Do You Need?

You'll need a laptop or desktop computer with a USB port. This will be your main programming environment, where you edit and compile the programs that you run on the embedded device. You'll connect this computer to the embedded device using the USB port and a specialized adapter that will depend on what development hardware you're using. The main computer can be running Windows, Linux, or macOS. For most of the examples we train our machine learning models in the cloud, using Google Colab, so you don't need to worry about having a specially equipped computer.

You will also need an embedded development board to test your programs on. To do something interesting you'll need a microphone, accelerometers, or a camera attached, and you want something small enough to build into a realistic prototype project, along with a battery. This was tough to find when we started this book, so we worked together with the chip manufacturer Ambiq and maker retailer SparkFun to produce the $15 SparkFun Edge board. All of the book's examples will work with this device.

The second revision of the SparkFun Edge board, the SparkFun Edge 2, is due to be released after this book has been published. All of the projects in this book are guaranteed to work with the new board. However, the code and the instructions for deployment will vary slightly from what is printed here. Don't worry—each project chapter links to a *README.md* that contains up-to-date instructions for deploying each example to the SparkFun Edge 2.

We also offer instructions on how to run many of the projects using the Arduino and Mbed development environments. We recommend the Arduino Nano 33 BLE Sense board, and the STM32F746G Discovery kit development board for Mbed, though all of the projects should be adaptable to other devices if you can capture the sensor data in the formats needed. Table 2-1 shows which devices we've included in each project chapter.

Table 2-1. Devices written about for each project

Project name	Chapter	SparkFun Edge	Arduino Nano 33 BLE Sense	STM32F746G Discovery kit
Hello world	Chapter 5	Included	Included	Included
Wake-word detection	Chapter 7	Included	Included	Included
Person detection	Chapter 9	Included	Included	Not included

Project name	Chapter	SparkFun Edge	Arduino Nano 33 BLE Sense	STM32F746G Discovery kit
Magic wand	Chapter 11	Included	Included	Not included

What If the Board I Want to Use Isn't Listed Here?

The source code for the projects in this book is hosted on GitHub, and we continually update it to support additional devices. Each chapter links to a project *README.md* that lists all of the supported devices and has instructions on how to deploy to them, so you can check there to find out if the device you'd like to use is already supported.

If you have some embedded development experience, it's easy to port the examples to new devices even if they're not listed.

None of these projects require any additional electronic components, aside from person detection, which will require a camera module. If you're using the Arduino, you'll need the Arducam Mini 2MP Plus, and if you're using the SparkFun Edge, you'll need SparkFun's Himax HM01B0 breakout.

What Software Do You Need?

All of the projects in this book are based around the TensorFlow Lite for Microcontrollers framework. This is a variant of the TensorFlow Lite framework designed to run on embedded devices with only a few tens of kilobytes of memory available. All of the projects are included as examples in the library, and it's open source, so you can find it on GitHub.

Since the code examples in this book are part of an active open source project, they are continually changing and evolving as we add optimizations, fix bugs, and support additional devices. It's likely you'll spot some differences between the code printed in the book and the most recent code in the TensorFlow repository. That said, although the code might drift a little over time, the basic principles you'll learn here will remain the same.

You'll need some kind of editor to examine and modify your code. If you're not sure which one you should use, Microsoft's free VS Code application is a great place to start. It works on macOS, Linux, and Windows, and has a lot of handy features like syntax highlighting and autocomplete. If you already have a favorite editor you can use that, instead; we won't be doing extensive modifications for any of our projects.

You'll also need somewhere to enter commands. On macOS and Linux this is known as the terminal, and you can find it in your Applications folder under that name. On

Windows it's known as the Command Prompt, which you can find in your Start menu.

There will also be extra software that you'll need to communicate with your embedded development board, but this will depend on what device you have. If you're using either the SparkFun Edge board or an Mbed device, you'll need to have Python installed for some build scripts, and then you can use GNU Screen on Linux or macOS or Tera Term on Windows to access the debug logging console, showing text output from the embedded device. If you have an Arduino board, everything you need is installed as part of the IDE, so you just need to download the main software package.

What Do We Hope You'll Learn?

The goal of this book is to help more applications in this new space emerge. There is no one "killer app" for TinyML right now, and there might never be, but we know from experience that there are a lot of problems out there in the world that can be solved using the toolbox it offers. We want to familiarize you with the possible solutions. We want to take domain experts from agriculture, space exploration, medicine, consumer goods, and any other areas with addressable issues and give them an understanding of how to solve problems themselves, or at the very least communicate what problems are solvable with these techniques.

With that in mind, we're hoping that when you finish this book you'll have a good overview of what's currently possible using machine learning on embedded systems at the moment, as well as some idea of what's going to be feasible over the next few years. We want you to be able to build and modify some practical examples using time-series data like audio or input from accelerometers, and for low-power vision. We'd like you to have enough understanding of the entire system to be able to at least participate meaningfully in design discussions with specialists about new products and hopefully be able to prototype early versions yourself.

Because we want to see complete products emerge, we approach everything we're discussing from a whole-system perspective. Often hardware vendors will focus on the energy consumption of the particular component they're selling, but not consider how other necessary parts increase the power required. For example, if you have a microcontroller that consumes only 1 mW, but the only camera sensor it works with takes 10 mW to operate, any vision-based product you use it on will not be able to take advantage of the processor's low energy consumption. This does mean that we won't be doing many deep dives into the underlying workings of the different areas; instead, we focus on what you need to know to use and modify the components involved.

For example, we won't linger on the details of what is actually happening under the hood when you train a model in TensorFlow, such as how gradients and back-propagation work. Rather, we show you how to run training from scratch to create a model, what common errors you might encounter and how to handle them, and how to customize the process to build models to tackle your own problems with new datasets.

CHAPTER 3
Getting Up to Speed on Machine Learning

There are few areas in technology with the mystique that surrounds machine learning and artificial intelligence (AI). Even if you're an experienced engineer in another domain, machine learning can seem like a dense subject with a mountain of assumed knowledge requirements. Many developers feel discouraged when they begin to read about ML and encounter explanations that invoke academic papers, obscure Python libraries, and advanced mathematics. It can feel daunting to even know where to start.

In reality, machine learning can be simple to understand and is accessible to anyone with a text editor. After you learn a few key ideas, you can easily use it in your own projects. Beneath all the mystique is a handy set of tools for solving various types of problems. It might sometimes *feel* like magic, but it's all just code, and you don't need a PhD to work with it.

This book is about using machine learning with tiny devices. In the rest of this chapter, you'll learn all the ML you need to get started. We'll cover the basic concepts, explore some tools, and train a simple machine learning model. Our focus is tiny hardware, so we won't spend long on the theory behind deep learning, or the mathematics that makes it all work. Later chapters will dig deeper into the tooling, and how to optimize models for embedded devices. But by the end of this chapter, you'll be familiar with the key terminology, have an understanding of the general workflow, and know where to go to learn more.

In this chapter, we cover the following:

- What machine learning actually is
- The types of problems it can solve
- Key terms and ideas

- The workflow for solving problems with deep learning, one of the most popular approaches to machine learning

There are many books and courses that explain the science behind deep learning, so we won't be doing that here. That said, it's a fascinating topic and we encourage you to explore! We list some of our favorite resources in "Learning Machine Learning" on page 364. But remember, you don't need all the theory to start building useful things.

What Machine Learning Actually Is

Imagine you own a machine that manufactures widgets. Sometimes it breaks down, and it's expensive to repair. Perhaps if you collected data about the machine during operation, you might be able to predict when it is about to break down and halt operation before damage occurs. For instance, you could record its rate of production, its temperature, and how much it is vibrating. It might be that some combination of these factors indicates an impending problem. But how do you figure it out?

This is an example of the sort of problem machine learning is designed to solve. Fundamentally, machine learning is a technique for using computers to predict things based on past observations. We collect data about our factory machine's performance and then create a computer program that analyzes that data and uses it to predict future states.

Creating a machine learning program is different from the usual process of writing code. In a traditional piece of software, a programmer designs an algorithm that takes an input, applies various rules, and returns an output. The algorithm's internal operations are planned out by the programmer and implemented explicitly through lines of code. To predict breakdowns in a factory machine, the programmer would need to understand which measurements in the data indicate a problem and write code that deliberately checks for them.

This approach works fine for many problems. For example, we know that water boils at 100°C at sea level, so it's easy to write a program that can predict whether water is boiling based on its current temperature and altitude. But in many cases, it can be difficult to know the exact combination of factors that predicts a given state. To continue with our factory machine example, there might be various different combinations of production rate, temperature, and vibration level that might indicate a problem but are not immediately obvious from looking at the data.

To create a machine learning program, a programmer feeds data into a special kind of algorithm and lets the algorithm discover the rules. This means that as programmers, we can create programs that make predictions based on complex data without having

to understand all of the complexity ourselves. The machine learning algorithm builds a *model* of the system based on the data we provide, through a process we call *training*. The model is a type of computer program. We run data through this model to make predictions, in a process called *inference*.

There are many different approaches to machine learning. One of the most popular is *deep learning*, which is based on a simplified idea of how the human brain might work. In deep learning, a *network* of simulated neurons (represented by arrays of numbers) is trained to model the relationships between various inputs and outputs. Different *architectures*, or arrangements of simulated neurons, are useful for different tasks. For instance, some architectures excel at extracting meaning from image data, while other architectures work best for predicting the next value in a sequence.

The examples in this book focus on deep learning, since it's a flexible and powerful tool for solving the types of problems that are well suited to microcontrollers. It might be surprising to discover that deep learning can work even on devices with limited memory and processing power. In fact, over the course of this book, you'll learn how to create deep learning models that do some really amazing things but that still fit within the constraints of tiny devices.

The next section explains the basic workflow for creating and using a deep learning model.

The Deep Learning Workflow

In the previous section, we outlined a scenario for using deep learning to predict when a factory machine is likely to break down. In this section, we introduce the work necessary to make this happen.

This process will involve the following tasks:

1. Decide on a goal
2. Collect a dataset
3. Design a model architecture
4. Train the model
5. Convert the model
6. Run inference
7. Evaluate and troubleshoot

Let's walk through them, one by one.

Decide on a Goal

When you're designing any kind of algorithm, it's important to start by establishing exactly what you want it to do. It's no different with machine learning. You need to decide what you want to predict so you can decide what data to collect and which model architecture to use.

In our example, we want to predict whether our factory machine is about to break down. We can express this as a *classification* problem. Classification is a machine learning task that takes a set of input data and returns the probability that this data fits each of a set of known *classes*. In our example, we might have two classes: "normal," meaning that our machine is operating without issue, and "abnormal," meaning that our machine is showing signs that it might soon break down.

This means that our goal is to create a model that classifies our input data as either "normal" or "abnormal."

Collect a Dataset

Our factory is likely to have a lot of available data, ranging from the operating temperature of our machine through to the type of food that was served in the cafeteria on a given day. Given the goal we've just established, we can begin to identify what data we need.

Selecting data

Deep learning models can learn to ignore noisy or irrelevant data. That said, it's best to train your model only using information that is relevant to solving the problem. Since it's unlikely that today's cafeteria food has an impact on the functioning of our machine, we can probably exclude it from our dataset. Otherwise, the model will need to learn to negate that irrelevant input, and it might be vulnerable to learning spurious associations—perhaps our machine has, coincidentally, always broken down on days that pizza is served.

You should always try to combine your domain expertise with experimentation when deciding whether to include data. You can also use statistical techniques to try to identify which data is significant. If you're still unsure about including a certain data source, you can always train two models and see which one works best!

Suppose that we've identified our most promising data as *rate of production*, *temperature*, and *vibration*. Our next step is to collect some data so that we can train a model.

It's really important that the data you choose will also be available when you want to make predictions. For example, since we have decided to train our model with temperature readings, we will need to provide temperature readings from the exact same physical locations when we run inference. This is because the model learns to understand how its inputs can predict its outputs. If we originally trained the model on temperature data from the insides of our machine, running the model on the current room temperature is unlikely to work.

Collecting data

It's difficult to know exactly how much data is required to train an effective model. It depends on many factors, such as the complexity of the relationships between variables, the amount of noise, and the ease with which classes can be distinguished. However, there's a rule of thumb that is always true: the more data, the better!

You should aim to collect data that represents the full range of conditions and events that can occur in the system. If our machine can fail in several different ways, we should be sure to capture data around each type of failure. If a variable changes naturally over time, it's important to collect data that represents the full range. For example, if the machine's temperature rises on warm days, you should be sure to include data from both winter and summer. This diversity will help your model represent every possible scenario, not just a select few.

The data we collect about our factory will likely be logged as a set of *time series*, meaning a sequence of readings collected on a periodic basis. For example, we might have a record of the temperature every minute, the rate of production each hour, and the level of vibration on a second-by-second basis. After we collect the data, we'll need to transform these time series into a form appropriate for our model.

Labeling data

In addition to collecting data, we need to determine which data represents "normal" and "abnormal" operation. We'll provide this information during the training process so that our model can learn how to classify inputs. The process of associating data with classes is called *labeling*, and the "normal" and "abnormal" classes are our *labels*.

This type of training, in which you instruct the algorithm what the data means during training, is called *supervised learning*. The resulting classification model will be able to process incoming data and predict to which class it is likely to belong.

To label the time-series data we've collected, we need a record of which periods of time the machine was working and which periods of time it was broken. We might

assume that the period immediately prior to the machine being broken generally represents abnormal operation. However, since we can't necessarily spot abnormal operation from a superficial look at the data, getting this correct might require some experimentation!

After we've decided how to label the data, we can generate a time series that contains the labels and add this to our dataset.

Our final dataset

Table 3-1 lists the data sources that we've assembled at this point in the workflow.

Table 3-1. Data sources

Data source	Interval	Sample reading
Rate of production	Once every 2 minutes	100 units
Temperature	Once every minute	30°C
Vibration (% of typical)	Once every 10 seconds	23%
Label ("normal" or "abnormal")	Once every 10 seconds	normal

The table shows the interval of each data source. For example, the temperature is logged once per minute. We've also generated a time series that contains the labels for the data. The interval for our labels is 1 per 10 seconds, which is the same as the smallest interval for the other time series. This means that we can easily determine the label for every datapoint in our data.

Now that we've collected our data, it's time to use it to design and train a model.

Design a Model Architecture

There are many types of deep learning model architectures, designed to solve a wide range of problems. When training a model, you can choose to design your own architecture or base it on an existing architecture developed by researchers. For many common problems, you can find pretrained models available online for free.

Over the course of this book we'll introduce you to several different model architectures, but there are a huge number of possibilities beyond what is covered here. Designing a model is both an art and a science, and model architecture is a major area of research. New architectures are invented literally every day.

When deciding on an architecture, you need to think about the type of problem you are trying to solve, the type of data you have access to, and the ways you can transform that data before feeding it into a model (we discuss transforming data shortly). The fact is, because the most effective architecture varies depending on the type of data that you are working with, your data and the architecture of your model are

deeply intertwined. Although we introduce them here under separate headings, they'll always be considered together.

You also need to think about the constraints of the device you will be running the model on, since microcontrollers generally have limited memory and slow processors, and larger models require more memory and take more time to run—the size of a model depends on the number of neurons it contains, and the way those neurons are connected. In addition, some devices are equipped with hardware acceleration that can speed up the execution of certain types of model architectures, so you might want to tailor your model to the strengths of the device you have in mind.

In our case, we might start by training a simple model with a few layers of neurons and then refining the architecture in an iterative process until we get a useful result. You'll see how to do that later in this book.

Deep learning models accept input and generate output in the form of *tensors*. For the purposes of this book,[1] a tensor is essentially a list that can contain either numbers or other tensors; you can think of it as similar to an array. Our hypothetical simple model will take a tensor as its input. The following subsection describes how we transform our data into this form.

Dimensions

The structure of a tensor is known as its *shape*, and they come in multiple *dimensions*. We talk about tensors throughout this book, so here is some useful terminology:

Vector

A *vector* is a list of numbers, similar to an array. It's the name we give a tensor with a single dimension (a 1D tensor). The following is a vector of shape `(5,)` because it contains five numbers in a single dimension:

```
[42 35 8 643 7]
```

Matrix

A *matrix* is a 2D tensor, similar to a 2D array. The following matrix is of shape `(3, 3)` because it contains three vectors of three numbers:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

1 This definition of the word *tensor* is different from the mathematical and physics definitions of the word, but it has become the norm in data science.

Higher-dimensional tensors
: Any shape with more than two dimensions is just referred to as a *tensor*. Here's a 3D tensor that has shape `(2, 3, 3)` because it contains two matrices of shape `(3, 3)`:

```
[[[10 20 30]
  [40 50 60]
  [70 80 90]]
 [[11 21 31]
  [41 51 61]
  [71 81 91]]]
```

Scalar
: A single number, known as a *scalar*, is technically a zero-dimensional tensor. For example, the number `42` is a scalar.

Generating features from data

We've established that our model will accept some type of tensor as its input. But as we discussed earlier, our data comes in the form of time series. How do we transform that time-series data into a tensor that we can pass into the model?

Our task now is to decide how to generate features from our data. In machine learning, the term *feature* refers to a particular type of information on which a model is trained. Different types of models are trained on different types of features. For example, a model might accept a single scalar value as its sole input feature.

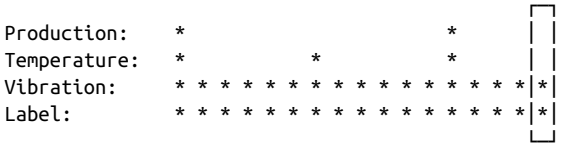
But inputs can be much more complex than this: a model designed to process images might accept a multidimensional tensor of image data as its input, and a model designed to predict based on multiple features might accept a vector containing multiple scalar values, one for each feature.

Recall that we decided that our model should use rate of production, temperature, and vibration to make its predictions. In their raw form, as time series with different intervals, these will not be suitable to pass into the model. The following section explains why.

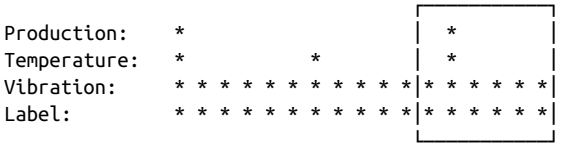
Windowing. In the following diagram, each piece of data in our time series is represented by a star. The current label is included in the data, since the label is required for training. Our goal is to train a model we can use to predict whether the machine is operating normally or abnormally at any given moment based on the current conditions:

```
Production:    *                     *           (every 2 minutes)
Temperature:   *          *          *           (every minute)
Vibration:     * * * * * * * * * * * * * * * *   (every 10 seconds)
Label:         * * * * * * * * * * * * * * * *   (every 10 seconds)
```

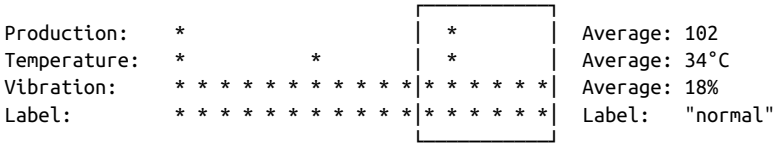
However, since our time series each have different intervals (like once per minute, or once per 10 seconds), if we pass in only the data available at a given moment, it might not include all of the types of data we have available. For example, in the moment highlighted in the following image, only vibration is available. This would mean that our model would only have information about vibration when attempting to make its prediction:



One solution to this problem might be to choose a window in time, and combine all of the data in this window into a single set of values. For example, we might decide on a one-minute window and look at all the values contained within it:



If we average all the values in the window for each time series and take the most recent value for any that lack a datapoint in the current window, we end up with a set of single values. We can decide how to label this snapshot based on whether there are any "abnormal" labels present in the window. If there's any "abnormal" present at all, the window should be labeled "abnormal." If not, it should be labeled "normal":



The three non-label values are our features! We can pass them into our model as a vector, with one element for each time series:

```
[102 34 .18]
```

During training, we can calculate a new window for every 10 seconds of data and pass it into our model, using the label to inform the training algorithm of our desired output. During inference, whenever we want to use the model to predict abnormal behavior, we can just look at our data, calculate the most recent window, run it through the model, and receive a prediction.

This is a simplistic approach, and it might not always turn out to work in practice, but it's a good enough starting point. You'll quickly find that machine learning is all about trial and error!

Before we move on to training, let's go over one last thing about input values.

Normalization. Generally, the data you feed into a neural network will be in the form of tensors filled with *floating-point* values, or *floats*. A float is a data type used to represent numbers that have decimal points. For the training algorithm to work effectively, these floating-point values need to be similar in size to one another. In fact, it's ideal if all values are expressed as numbers in the range of 0 to 1.

Let's take another look at our input tensor from the previous section:

```
[102 34 .18]
```

These numbers are each at very different scales: the temperature is more than 100, whereas the vibration is expressed as a fraction of 1. To pass these values into our network, we need to *normalize* them so that they are all in a similar range.

One way of doing this is to calculate the mean of each feature across the dataset and subtract it from the values. This has the effect of squashing the numbers down so that they are closer to zero. Here's an example:

```
Temperature series:
[108 104 102 103 102]

Mean:
103.8

Normalized values, calculated by subtracting 103.8 from each temperature:
[ 4.2 0.2 -1.8 -0.8 -1.8 ]
```

One situation in which you'll frequently encounter normalization, implemented in a different way, is when images are fed into a neural network. Computers often store images as matrices of 8-bit integers, whose values range from 0 to 255. To normalize these values so that they are all between 0 and 1, each 8-bit value is multiplied by `1/255`. Here's an example with a 3 × 3–pixel grayscale image, in which each pixel's value represents its brightness:

```
Original 8-bit values:
[[255 175 30]
 [0   45  24]
 [130 192 87]]

Normalized values:
[[1.         0.68627451 0.11764706]
 [0.         0.17647059 0.09411765]
 [0.50980392 0.75294118 0.34117647]]
```

Thinking with ML

So far, we've learned how to start thinking about solving problems with machine learning. In the context of our factory scenario, we've walked through deciding on a suitable goal, collecting and labeling the appropriate data, designing the features we are going to pass into our model, and choosing a model architecture. No matter what problem we are trying to solve, we'll use the same approach. It's important to note that this is an iterative process, and we often go back and forth through the stages of the ML workflow until we've arrived at a model that works—or decided that the task is too difficult.

For example, imagine that we're building a model to predict the weather. We'll need to decide on our goal (for instance, to predict whether it's going to rain tomorrow), collect and label a dataset (such as weather reports from the past few years), design the features that we'll feed to our model (perhaps the average conditions over the past two days), and choose a model architecture suitable for this type of data and the device that we want to run it on. We'll come up with some initial ideas, test them out, and tweak our approach until we get good results.

The next step in our workflow is training, which we explore in the following section.

Train the Model

Training is the process by which a model learns to produce the correct output for a given set of inputs. It involves feeding training data through a model and making small adjustments to it until it makes the most accurate predictions possible.

As we discussed earlier, a model is a network of simulated neurons represented by arrays of numbers arranged in layers. These numbers are known as *weights* and *biases*, or collectively as the network's *parameters*.

When data is fed into the network, it is transformed by successive mathematical operations that involve the weights and biases in each layer. The output of the model is the result of running the input through these operations. Figure 3-1 shows a simple network with two layers.

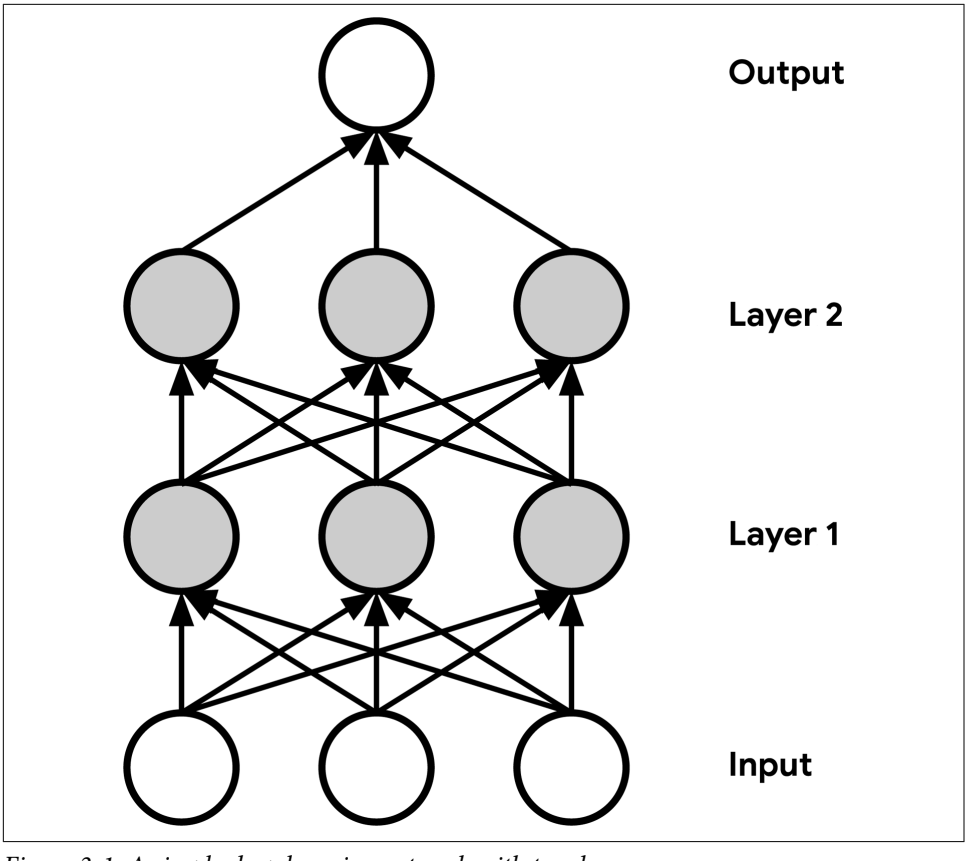


Figure 3-1. A simple deep learning network with two layers

The model's weights start out with random values, and biases typically start with a value of 0. During training, *batches* of data are fed into the model, and the model's output is compared with the desired output (which in our case is the correct label, "normal" or "abnormal"). An algorithm called *backpropagation* adjusts the weights and biases incrementally so that over time, the output of the model gets closer to matching the desired value. Training, which is measured in *epochs* (meaning iterations), continues until we decide to stop.

We generally stop training when a model's performance stops improving. At the point that it begins to make accurate predictions, it is said to have *converged*. To determine whether a model has converged, we can analyze graphs of its performance during training. Two common performance metrics are *loss* and *accuracy*. The loss metric gives us a numerical estimate of how far the model is from producing the expected answers, and the *accuracy* metric tells us the percentage of the time that it chooses the correct prediction. A perfect model would have a loss of 0.0 and an accuracy of 100%, but real models are rarely perfect.

Figure 3-2 shows the loss and accuracy during training for a deep learning network. You can see how as training progresses, accuracy increases and loss is reduced, until we reach a point at which the model no longer improves.

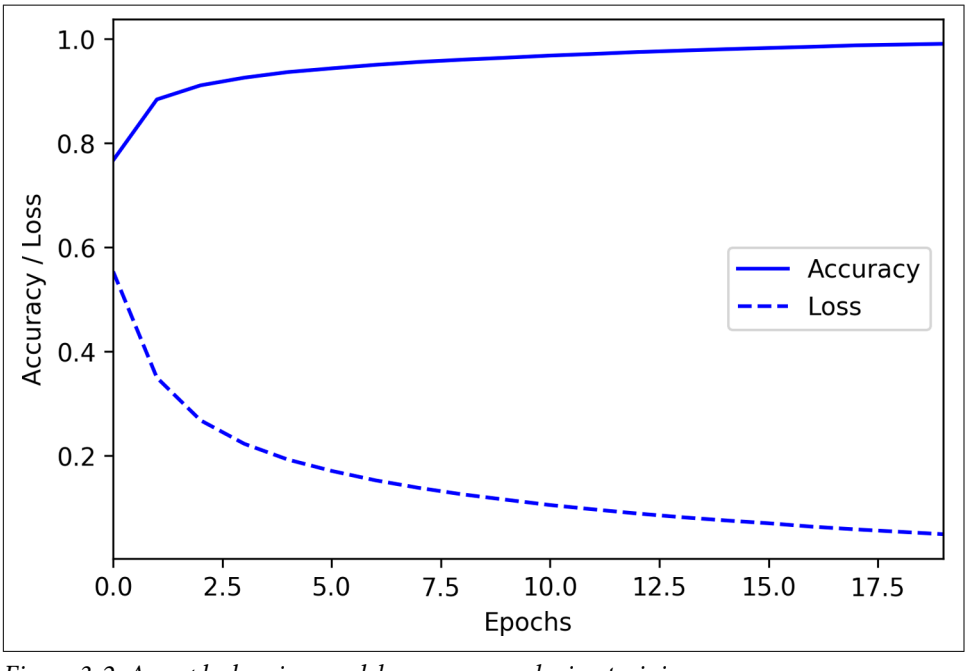


Figure 3-2. A graph showing model convergence during training

To attempt to improve the model's performance, we can change our model architecture, and we can adjust various values used to set up the model and moderate the training process. These values are collectively known as *hyperparameters*, and they include variables such as the number of training epochs to run and the number of neurons in each layer. Each time we make a change, we can retrain the model, look at the metrics, and decide whether to optimize further. Hopefully, time and iterations will result in a model with acceptable accuracy!

It's important to remember there's no guarantee that you'll be able to achieve good enough accuracy for the problem you are trying to solve. There isn't always enough information contained within a dataset to make accurate predictions, and some problems just can't be solved, even with state-of-the-art deep learning. That said, your model may be useful even if it is not 100% accurate. In the case of our factory example, being able to predict abnormal operation even part of the time could be a big help.

Underfitting and overfitting

The two most common reasons a model fails to converge are *underfitting* and *overfitting*.

A neural network learns to *fit* its behavior to the patterns it recognizes in data. If a model is correctly fit, it will produce the correct output for a given set of inputs. When a model is *underfit*, it has not yet been able to learn a strong enough representation of these patterns to be able to make good predictions. This can happen for a variety of reasons, most commonly that the architecture is too small to capture the complexity of the system it is supposed to model or that it has not been trained on enough data.

When a model is *overfit*, it has learned its training data too well. The model is able to exactly predict the minutiae of its training data, but it is not able to generalize its learning to data it has not previously seen. Often this happens because the model has managed to entirely memorize the training data, or it has learned to rely on a shortcut present in the training data but not in the real world.

For example, imagine you are training a model to classify photos as containing either dogs or cats. If all the dog photos in your training data are taken outdoors, and all the cat photos are taken indoors, your model may learn to cheat and use the presence of the sky in each photograph to predict which animal it is. This means that it might misclassify future dog selfies if they happen to be taken indoors.

There are many ways to fight overfitting. One possibility is to reduce the size of the model so it does not have enough capacity to learn an exact representation of its training set. A set of techniques known as *regularization* can be applied during training to reduce the degree of overfitting. To make the most of limited data, a technique called *data augmentation* can be used to generate new, artificial datapoints by slicing and dicing the existing data. But the best way to beat overfitting, when possible, is to get your hands on a larger and more varied dataset. More data always helps!

Regularization and Data Augmentation

Regularization techniques are used to make deep learning models less likely to overfit their training data. They generally involve constraining the model in some way in order to prevent it from perfectly memorizing the data that it's fed during training.

There are several methods used for regularization. Some, such as *L1* and *L2 regularization*, involve tweaking the algorithms used during training to penalize complex models that are prone to overfitting. Another, named *dropout*, involves randomly cutting the connections between neurons during training. We'll look at regularization in practice later in the book.

We'll also explore data augmentation, which is a way to artificially expand the size of a training dataset. This is done by creating multiple additional versions of every training input, each transformed in a way that preserves its meaning but varies its exact composition. In one of our examples, we train a model to recognize speech from audio samples. We augment our original training data by adding artificial background noise and shifting the samples around in time.

Training, validation, and testing

To assess the performance of a model, we can look at how it performs on its training data. However, this only tells us part of the story. During training, a model learns to fit its training data as closely as possible. As we saw earlier, in some cases the model will begin to overfit the training data, meaning that it will work well on the training data but not in real life.

To understand when this is happening, we need to *validate* the model using new data that wasn't used in training. It's common to split a dataset into three parts—*training*, *validation*, and *test*. A typical split is 60% training data, 20% validation, and 20% test. This splitting must be done so that each part contains the same distribution of information, and in a way that preserves the structure of the data. For example, since our data is a time series, we could potentially split it into three contiguous chunks of time. If our data were not a time series, we could just sample the datapoints randomly.

During training, the *training* dataset is used to train the model. Periodically, data from the *validation* dataset is fed through the model, and the loss is calculated. Because the model has not seen this data before, its loss score is a more reliable measure of how the model is performing. By comparing the training and validation loss (and accuracy, or whichever other metrics are available) over time, you can see whether the model is overfitting.

Figure 3-3 shows a model that is overfitting. You can see how as the training loss has decreased, the validation loss has gone up. This means that the model is becoming better at predicting the training data but is losing its ability to generalize to new data.

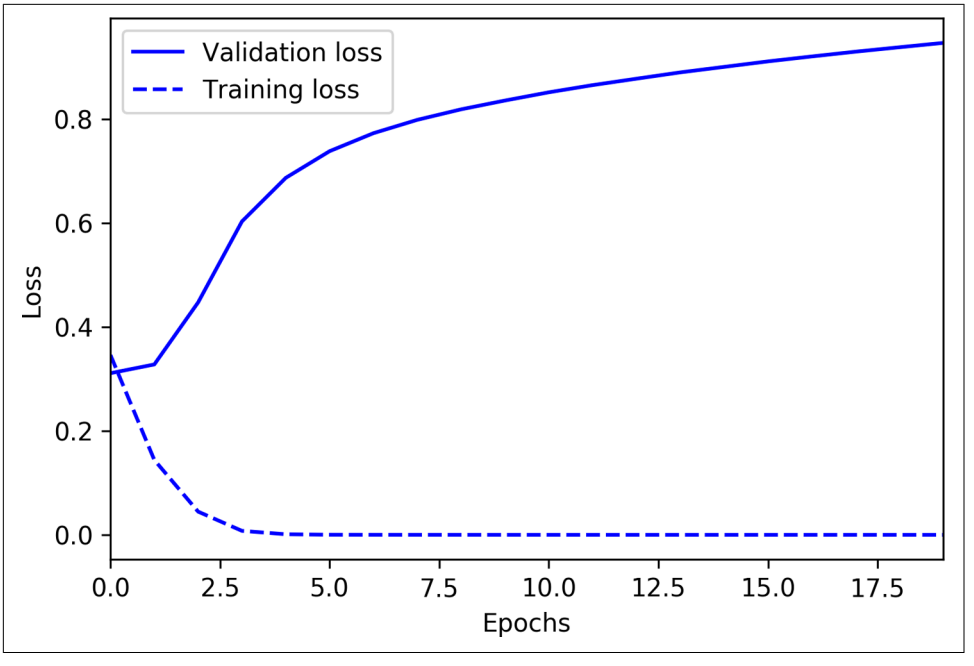


Figure 3-3. A graph showing model overfitting during training

As we tweak our models and training processes to improve performance and avoid overfitting, we will hopefully start to see our validation metrics improve.

However, this process has an unfortunate side effect. By optimizing to improve the validation metrics, we might just be nudging the model toward overfitting both the training *and* the validation data! Each adjustment we make will fit the model to the validation data slightly better, and in the end, we might have the same overfitting problem as before.

To verify that this hasn't happened, our final step when training a model is to run it on our *test* data and confirm that it performs as well as during validation. If it doesn't, we have optimized our model to overfit both our training and validation data. In this case, we might need to go back to the drawing board and come up with a new model architecture, since if we continue to tweak to improve performance on our test data, we'll just overfit to that, too.

After we have a model that performs acceptably well with training, validation, and test data, the training part of this process is over. Next, we get our model ready to run on-device!

Convert the Model

Throughout this book, we use TensorFlow to build and train models. A TensorFlow model is essentially a set of instructions that tell an *interpreter* how to transform data in order to produce an output. When we want to use our model, we just load it into memory and execute it using the TensorFlow interpreter.

However, TensorFlow's interpreter is designed to run models on powerful desktop computers and servers. Since we'll be running our models on tiny microcontrollers, we need a different interpreter that's designed for our use case. Fortunately, TensorFlow provides an interpreter and accompanying tools to run models on small, low-powered devices. This set of tools is called TensorFlow Lite.

Before TensorFlow Lite can run a model, it first must be converted into the TensorFlow Lite format and then saved to disk as a file. We do this using a tool named the *TensorFlow Lite Converter*. The converter can also apply special optimizations aimed at reducing the size of the model and helping it run faster, often without sacrificing performance.

In Chapter 13, we dive into the details of TensorFlow Lite and how it helps us run models on tiny devices. For now, all you need to know is that you'll need to convert your models, and that the conversion process is quick and easy.

Run Inference

After the model has been converted, it's ready to deploy! We'll now use the TensorFlow Lite for Microcontrollers C++ library to load the model and make predictions.

Since this is the part where our model meets our application code, we need to write some code that takes raw input data from our sensors and transforms it into the same form that our model was trained on. We then pass this transformed data into our model and run inference.

This will result in output data containing predictions. In the case of our classifier model, the output will be a score for each of our classes, "normal" and "abnormal." For models that classify data, typically the scores for all of the classes will sum to 1, and the class with the highest score will be the prediction. The higher the difference between the scores, the higher the confidence in the prediction. Table 3-2 lists some example outputs.

Table 3-2. Example outputs

Normal score	Abnormal score	Explanation
0.1	0.9	High confidence in an abnormal state
0.9	0.1	High confidence in a normal state
0.7	0.3	Slight confidence in a normal state

Normal score	Abnormal score	Explanation
0.49	0.51	Inconclusive result, since neither state is significantly ahead

In our factory machine example, each individual inference takes into account only a snapshot of the data—it tells us the probability of an abnormal state within the last 10 seconds, based on various sensor readings. Since real-world data is often messy and machine learning models aren't perfect, it's possible that a temporary glitch might result in an incorrect classification. For example, we might see a spike in a temperature value due to a temporary sensor malfunction. This transient, unreliable input might result in an output classification that momentarily doesn't reflect reality.

To prevent these momentary glitches from causing problems, we could potentially take the average of all of our model's outputs across a period of time. For example, we could run our model on the current data window every 10 seconds, and take the averages of the last 6 outputs to give a smoothed score for each class. This would mean that transient issues are ignored, and we only act upon consistent behavior. We use this technique to help with wake-word detection in Chapter 7.

After we have a score for each class, it's up to our application code to decide what to do. Perhaps if an abnormal state is detected consistently for one minute, our code will send a signal to shut down our machine and alert the maintenance team.

Evaluate and Troubleshoot

After we've deployed our model and have it running on-device, we'll start to see whether its real-world performance approaches what we hoped. Even though we've already proved that our model makes accurate predictions on its test data, performance on the actual problem might be different.

There are many reasons why this might happen. For example, the data used in training might not be exactly representative of the data available in real operation. Perhaps due to local climate, our machine's temperature is generally cooler than the one from which our dataset was collected. This might affect the predictions made by our model, such that they are no longer as accurate as expected.

Another possibility is that our model might have overfit our dataset without us realizing. In "Train the Model" on page 21, we learned how this can happen by accident when the dataset happens to contain additional signals that a model can learn to recognize in place of those we expect.

If our model isn't working in production, we'll need to do some troubleshooting. First, we rule out any hardware problems (like faulty sensors or unexpected noise) that might be affecting the data that gets to our model. Second, we capture some data from the device where the model is deployed and compare it with our original dataset to make sure that it is in the same ballpark. If not, perhaps there's a difference in envi-

ronmental conditions or sensor characteristics that we weren't expecting. If the data checks out, it might be that overfitting is the problem.

After we've ruled out hardware issues, the best fix for overfitting is often to train with more data. We can capture additional data from our deployed hardware, combine it with our original dataset, and retrain our model. In the process, we can apply regularization and data augmentation techniques to help make the most of the data we have.

Reaching good real-world performance can sometimes take some iteration on your model, your hardware, and the accompanying software. If you run into a problem, treat it like any other technology issue. Take a scientific approach to troubleshooting, eliminating possible factors, and analyze your data to figure out what is going wrong.

Wrapping Up

Now that you're familiar with the basic workflow used by machine learning practitioners, we're ready to take the next steps in our TinyML adventure.

In Chapter 4, we'll build our first model and deploy it to some tiny hardware!

CHAPTER 4

The "Hello World" of TinyML: Building and Training a Model

In Chapter 3, we learned the basic concepts of machine learning and the general workflow that machine learning projects follow. In this chapter and the next, we'll start putting our knowledge into practice. We're going to build and train a model from scratch and then integrate it into a simple microcontroller program.

In the process, you'll get your hands dirty with some powerful developer tools that are used every day by cutting-edge machine learning practitioners. You'll also learn how to integrate a machine learning model into a C++ program and deploy it to a microcontroller to control current flowing in a circuit. This might be your first taste of mixing hardware and ML, and it should be fun!

You can test the code that we write in these chapters on your Mac, Linux, or Windows machine, but for the full experience, you'll need one of the embedded devices mentioned in "What Hardware Do You Need?" on page 6:

- Arduino Nano 33 BLE Sense
- SparkFun Edge
- ST Microelectronics STM32F746G Discovery kit

To create our machine learning model, we'll use Python, TensorFlow, and Google's Colaboratory, which is a cloud-based interactive notebook for experimenting with Python code. These are some of the most important tools for real-world machine learning engineers, and they're all free to use.

Wondering about the title of this chapter? It's a tradition in programming that new technologies are introduced with example code that demonstrates how to do something very simple. Often, the simple task is to make a program output the words, "Hello, world." There's no clear equivalent in ML, but we're using the term "hello world" to refer to a simple, easy-to-read example of an end-to-end TinyML application. To learn the history of "hello world," read the Wikipedia article.

Over the course of this chapter, we will do the following:

1. Obtain a simple dataset.
2. Train a deep learning model.
3. Evaluate the model's performance.
4. Convert the model to run on-device.
5. Write code to perform on-device inference.
6. Build the code into a binary.
7. Deploy the binary to a microcontroller.

All the code that we will use is available in TensorFlow's GitHub repository.

We recommend that you walk through each part of this chapter and then try running the code. There are instructions on how to do this along the way.

But before we start, let's discuss exactly what we're going to build.

What We're Building

In the previous chapter, we discussed how deep learning networks learn to model patterns in their training data so that they can make predictions. We're now going to train a network to model some very simple data.

You've probably heard of the sine function. It's used in trigonometry to help describe the properties of right-angled triangles. The data we'll be training with is a sine wave, which is the graph obtained by plotting the result of the sine function over time. You can see the graph of a sine wave in Figure 4-1.

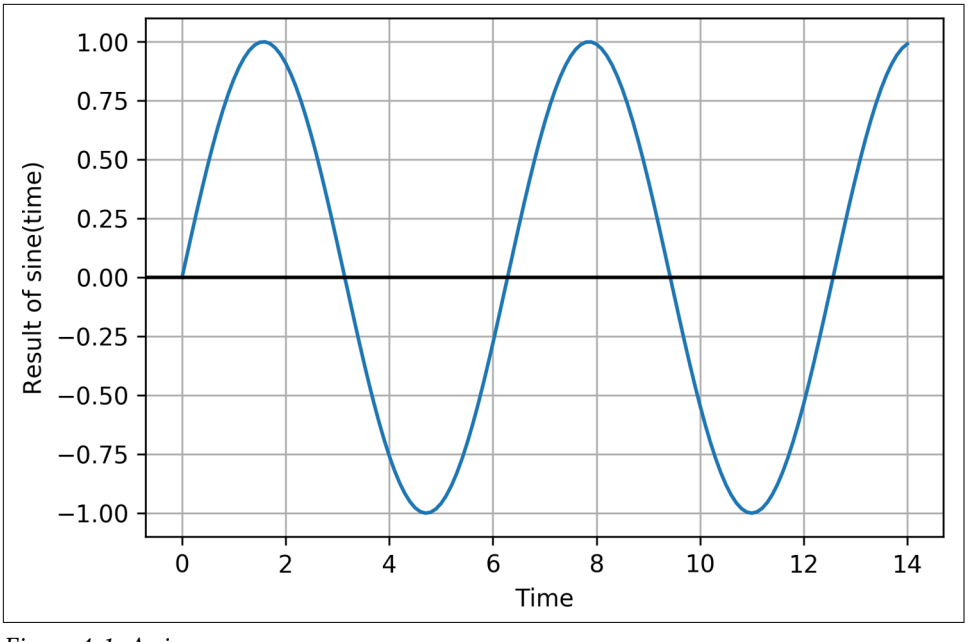


Figure 4-1. A sine wave

Our goal is to train a model that can take a value, `x`, and predict its sine, `y`. In a real-world application, if you needed the sine of `x`, you could just calculate it directly. However, by training a model to approximate the result, we can demonstrate the basics of machine learning.

The second part of our project will be to run this model on a hardware device. Visually, the sine wave is a pleasant curve that runs smoothly from –1 to 1 and back. This makes it perfect for controlling a visually pleasing light show! We'll be using the output of our model to control the timing of either some flashing LEDs or a graphical animation, depending on the capabilities of the device.

Online, you can see an animated GIF of this code flashing the LEDs of a SparkFun Edge. Figure 4-2 is a still from this animation, showing a couple of the device's LEDs lit.

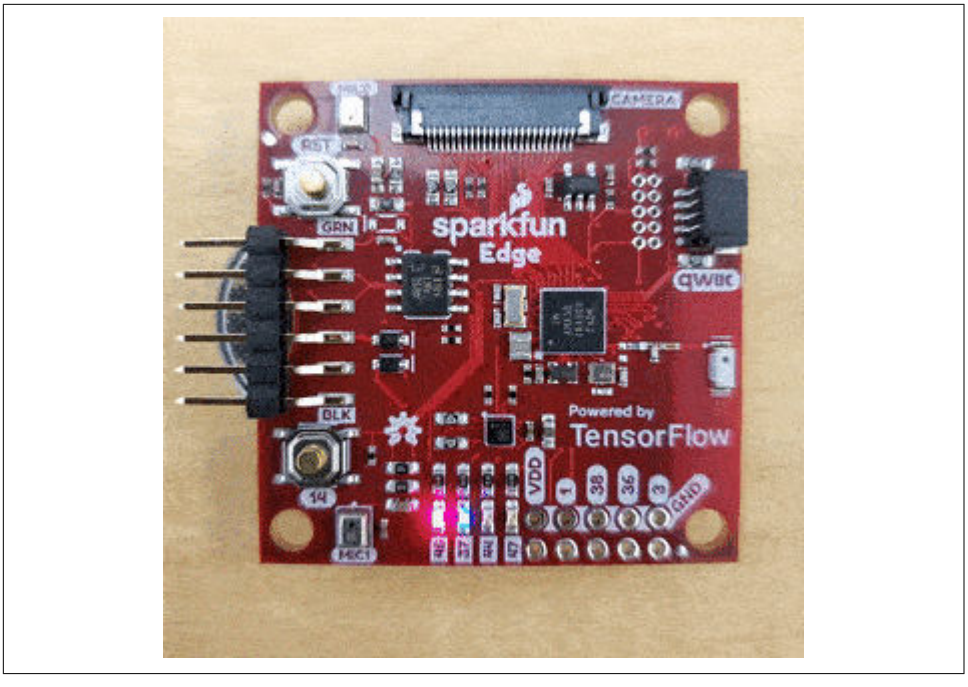


Figure 4-2. The code running on a SparkFun Edge

This may not be a particularly useful application of machine learning, but in the spirit of a "hello world" example, it's simple, fun, and will help demonstrate the basic principles you need to know.

After we get our basic code working, we'll be deploying it to three different devices: the SparkFun Edge, an Arduino Nano 33 BLE Sense, and an ST Microelectronics STM32F746G Discovery kit.

Since TensorFlow is an actively developed open source project that is continually evolving, you might notice some slight differences between the code printed here and the code hosted online. Don't worry—even if a few lines of code change, the basic principles remain the same.

Our Machine Learning Toolchain

To build the machine learning parts of this project, we're using the same tools used by real-world machine learning practitioners. This section introduces them to you.

Python and Jupyter Notebooks

Python is the favorite programming language of machine learning scientists and engineers. It's easy to learn, works well for many different applications, and has a ton of libraries for useful tasks involving data and mathematics. The vast majority of deep learning research is done using Python, and researchers often release the Python source code for the models they create.

Python is especially great when combined with something called *Jupyter Notebooks*. This is a special document format that allows you to mix writing, graphics, and code that can be run at the click of a button. Jupyter notebooks are widely used as a way to describe, explain, and explore machine learning code and problems.

We'll be creating our model inside of a Jupyter notebook, which permits us to do awesome things to visualize our data during development. This includes displaying graphs that show our model's accuracy and convergence.

If you have some programming experience, Python is easy to read and learn. You should be able to follow this tutorial without any trouble.

Google Colaboratory

To run our notebook we'll use a tool called Colaboratory, or *Colab* for short. Colab is made by Google, and it provides an online environment for running Jupyter notebooks. It's provided for free as a tool to encourage research and development in machine learning.

Traditionally, you needed to create a notebook on your own computer. This required installing a lot of dependencies, such as Python libraries, which can be a headache. It was also difficult to share the resulting notebook with other people, since they might have different versions of the dependencies, meaning the notebook might not run as expected. In addition, machine learning can be computationally intensive, so training models might be slow on your development computer.

Colab allows you to run notebooks on Google's powerful hardware, at zero cost. You can edit and view your notebooks from any web browser, and you can share them with other people, who are guaranteed to get the same results when they run them. You can even configure Colab to run your code on specially accelerated hardware that can perform training more quickly than a normal computer.

TensorFlow and Keras

TensorFlow is a set of tools for building, training, evaluating, and deploying machine learning models. Originally developed at Google, TensorFlow is now an open source project built and maintained by thousands of contributors across the world. It is the

most popular and widely used framework for machine learning. Most developers interact with TensorFlow via its Python library.

TensorFlow does many different things. In this chapter we'll use Keras, TensorFlow's high-level API that makes it easy to build and train deep learning networks. We'll also use TensorFlow Lite, a set of tools for deploying TensorFlow models to mobile and embedded devices, to run our model on-device.

Chapter 13 will cover TensorFlow in much more detail. For now, just know that it is an extremely powerful and industry-standard tool that will continue to serve your needs as you go from beginner to deep learning expert.

Building Our Model

We're now going to walk through the process of building, training, and converting our model. We include all of the code in this chapter, but you can also follow along in Colab and run the code as you go.

First, load the notebook. After the page loads, at the top, click the "Run in Google Colab" button, as shown in Figure 4-3. This copies the notebook from GitHub into Colab, allowing you to run it and make edits.

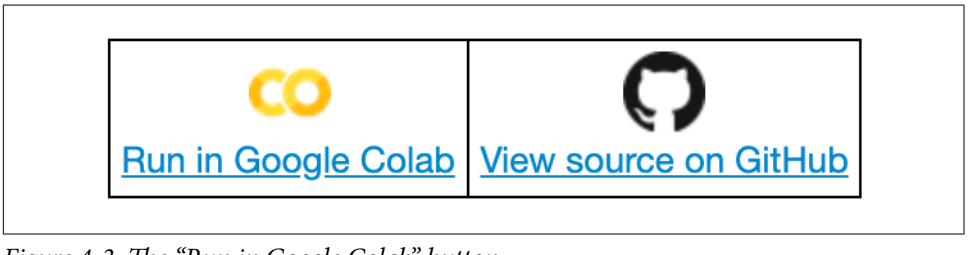


Figure 4-3. The "Run in Google Colab" button

Problems Loading the Notebook

As of this writing, there's a known issue with GitHub that results in intermittent error messages when displaying Jupyter notebooks. If you see the message "Sorry, something went wrong. Reload?" when trying to access the notebook, you can open it directly in Colab by using the following process. Copy the part of the notebook's GitHub URL that appears after *https://github.com/*:

```
tensorflow/tensorflow/blob/master/tensorflow/lite/micro/examples/
  hello_world/create_sine_model.ipynb
```

And prepend it with *https://colab.research.google.com/github/*. This will result in a full URL:

```
https://colab.research.google.com/github/tensorflow/tensorflow/blob/master/
  tensorflow/lite/micro/examples/hello_world/create_sine_model.ipynb
```

Navigate to that URL in your browser to open the notebook directly in Colab.

By default, in addition to the code, the notebook contains a sample of the output you should expect to see when the code is run. Since we'll be running through the code in this chapter, let's clear this output so the notebook is in a pristine state. To do this, in Colab's menu, click Edit and then select "Clear all outputs," as shown in Figure 4-4.

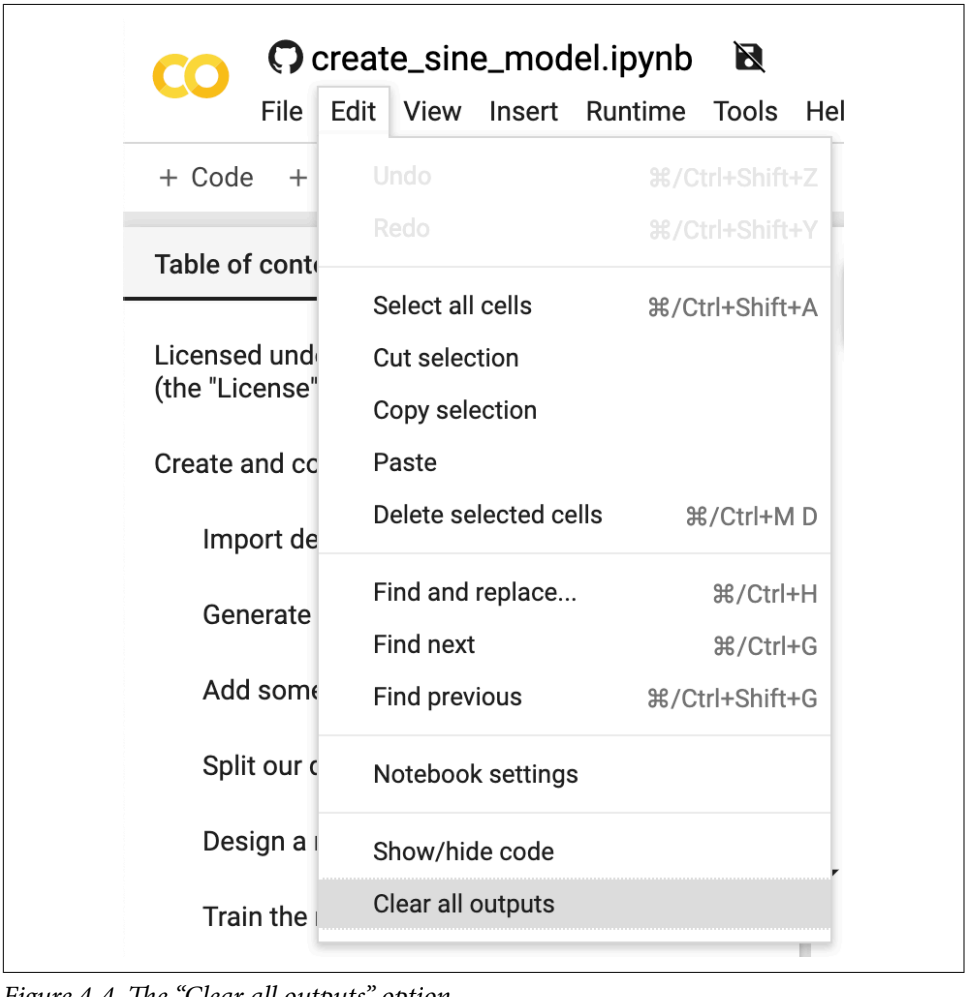


Figure 4-4. The "Clear all outputs" option

Nice work. Our notebook is now ready to go!

If you're already familiar with machine learning, TensorFlow, and Keras, you might want to skip ahead to the part where we convert our model to use with TensorFlow Lite. In the book, jump to "Converting the Model for TensorFlow Lite" on page 65. In Colab, scroll down to the heading "Convert to TensorFlow Lite."

Importing Dependencies

Our first task is to import the dependencies we need. In Jupyter notebooks, code and text are arranged in *cells*. There are *code* cells, which contain executable Python code, and *text* cells, which contain formatted text.

Our first code cell is located under "Import dependencies." It sets up all of the libraries that we need to train and convert our model. Here's the code:

```
# TensorFlow is an open source machine learning library
!pip install tensorflow==2.0
import tensorflow as tf
# NumPy is a math library
import numpy as np
# Matplotlib is a graphing library
import matplotlib.pyplot as plt
# math is Python's math library
import math
```

In Python, the `import` statement loads a library so that it can be used from our code. You can see from the code and comments that this cell does the following:

- Installs the TensorFlow 2.0 library using `pip`, a package manager for Python
- Imports TensorFlow, NumPy, Matplotlib, and Python's `math` library

When we import a library, we can give it an alias so that it's easy to refer to later. For example, in the preceding code, we use `import numpy as np` to import NumPy and give it the alias `np`. When we use it in our code, we can refer to it as `np`.

The code in code cells can be run by clicking the button that appears at the upper left when the cell is selected. In the "Import dependencies" section, click anywhere in the first code cell so that it becomes selected. Figure 4-5 shows what a selected cell looks like.

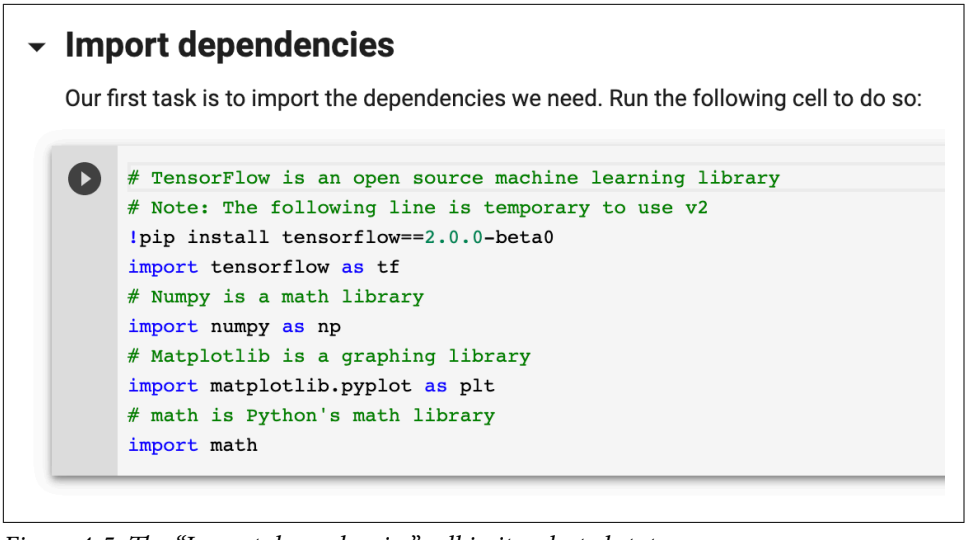


Figure 4-5. The "Import dependencies" cell in its selected state

To run the code, click the button that appears in the upper left. As the code is being run, the button will animate with a circle as depicted in Figure 4-6.

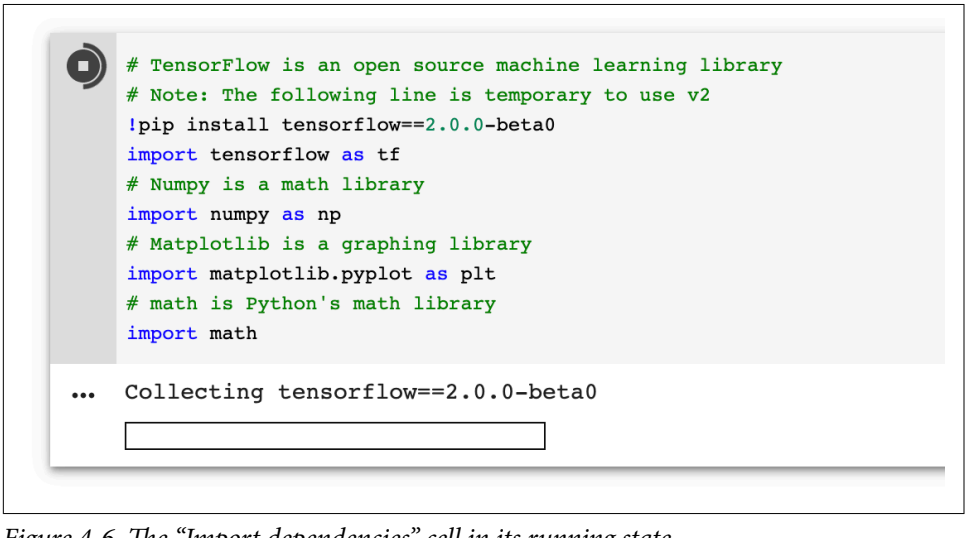


Figure 4-6. The "Import dependencies" cell in its running state

The dependencies will begin to be installed, and you'll see some output appearing. You should eventually see the following line, meaning that the library was installed successfully:

```
Successfully installed tensorboard-2.0.0 tensorflow-2.0.0 tensorflow-
estimator-2.0.0
```

After a cell has been run in Colab, you'll see that a `1` is now displayed in the upper-left corner when it is no longer selected, as illustrated in Figure 4-7. This number is a counter that is incremented each time the cell is run.

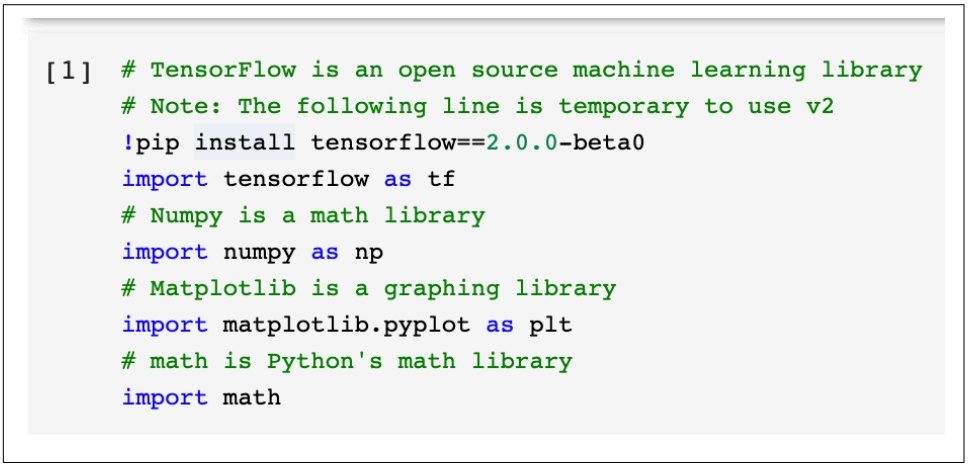


Figure 4-7. The cell run counter in the upper-left corner

You can use this to understand which cells have been run, and how many times.

Generating Data

Deep learning networks learn to model patterns in underlying data. As we mentioned earlier, we're going to train a network to model data generated by a sine function. This will result in a model that can take a value, `x`, and predict its sine, `y`.

Before we go any further, we need some data. In a real-world situation, we might be collecting data from sensors and production logs. For this example, however, we're using some simple code to generate a dataset.

The next cell is where this will happen. Our plan is to generate 1,000 values that represent random points along a sine wave. Let's take a look at Figure 4-8 to remind ourselves what a sine wave looks like.

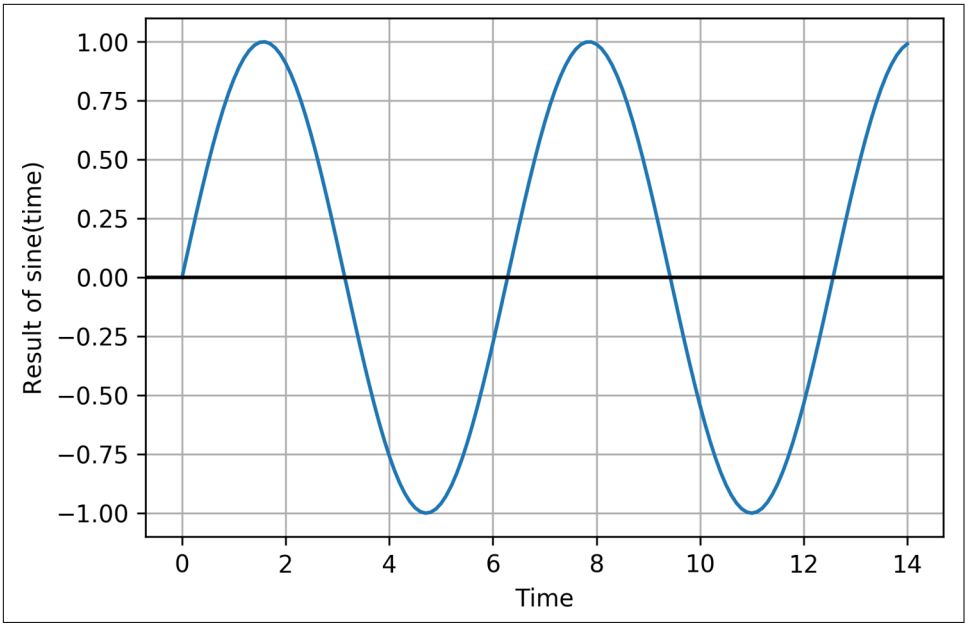


Figure 4-8. A sine wave

Each full cycle of a wave is called its *period*. From the graph, we can see that a full cycle is completed approximately every six units on the x-axis. In fact, the period of a sine wave is $2 \times \pi$, or 2π.

So that we have a full sine wave worth of data to train on, our code will generate random x values from 0 to 2π. It will then calculate the sine for each of these values.

Here's the full code for this cell, which uses NumPy (`np`, which we imported earlier) to generate random numbers and calculate their sine:

```
# We'll generate this many sample datapoints
SAMPLES = 1000

# Set a "seed" value, so we get the same random numbers each time we run this
# notebook. Any number can be used here.
SEED = 1337
np.random.seed(SEED)
tf.random.set_seed(SEED)

# Generate a uniformly distributed set of random numbers in the range from
# 0 to 2π, which covers a complete sine wave oscillation
x_values = np.random.uniform(low=0, high=2*math.pi, size=SAMPLES)

# Shuffle the values to guarantee they're not in order
np.random.shuffle(x_values)
```

```
# Calculate the corresponding sine values
y_values = np.sin(x_values)

# Plot our data. The 'b.' argument tells the library to print blue dots.
plt.plot(x_values, y_values, 'b.')
plt.show()
```

In addition to what we discussed earlier, there are a few things worth pointing out in this code. First, you'll see that we use `np.random.uniform()` to generate our `x` values. This method returns an array of random numbers in the specified range. NumPy contains a lot of useful methods that operate on entire arrays of values, which is very convenient when dealing with data.

Second, after generating the data, we shuffle it. This is important because the training process used in deep learning depends on data being fed to it in a truly random order. If the data were in order, the resulting model would be less accurate.

Next, notice that we use NumPy's `sin()` method to calculate our sine values. NumPy can do this for all of our `x` values at once, returning an array. NumPy is great!

Finally, you'll see some mysterious code invoking `plt`, which is our alias for Matplotlib:

```
# Plot our data. The 'b.' argument tells the library to print blue dots.
plt.plot(x_values, y_values, 'b.')
plt.show()
```

What does this code do? It plots a graph of our data. One of the best things about Jupyter notebooks is their ability to display graphics that are output by the code you run. Matplotlib is an excellent tool for creating graphs from data. Since visualizing data is a crucial part of the machine learning workflow, this will be incredibly helpful as we train our model.

To generate the data and render it as a graph, run the code in the cell. After the code cell finishes running, you should see a beautiful graph appear underneath, like the one shown in Figure 4-9.

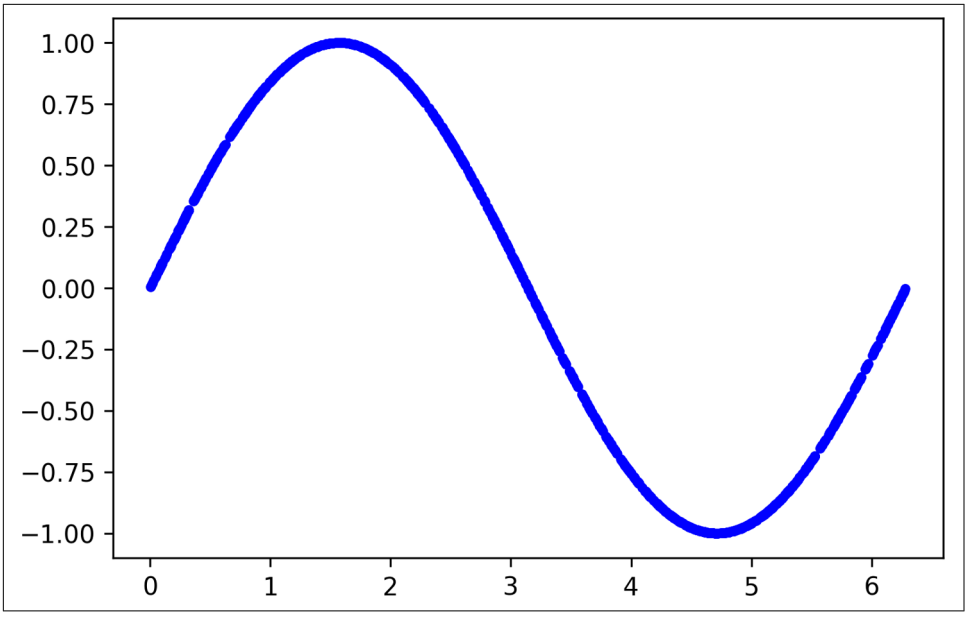


Figure 4-9. A graph of our generated data

This is our data! It is a selection of random points along a nice, smooth sine curve. We could use this to train our model.

However, this would be too easy. One of the exciting things about deep learning networks is their ability to sift patterns from noise. This allows them to make predictions even when trained on messy, real-world data. To show this off, let's add some random noise to our datapoints and draw another graph:

```
# Add a small random number to each y value
y_values += 0.1 * np.random.randn(*y_values.shape)

# Plot our data
plt.plot(x_values, y_values, 'b.')
plt.show()
```

Run this cell and take a look at the results, as shown in Figure 4-10.

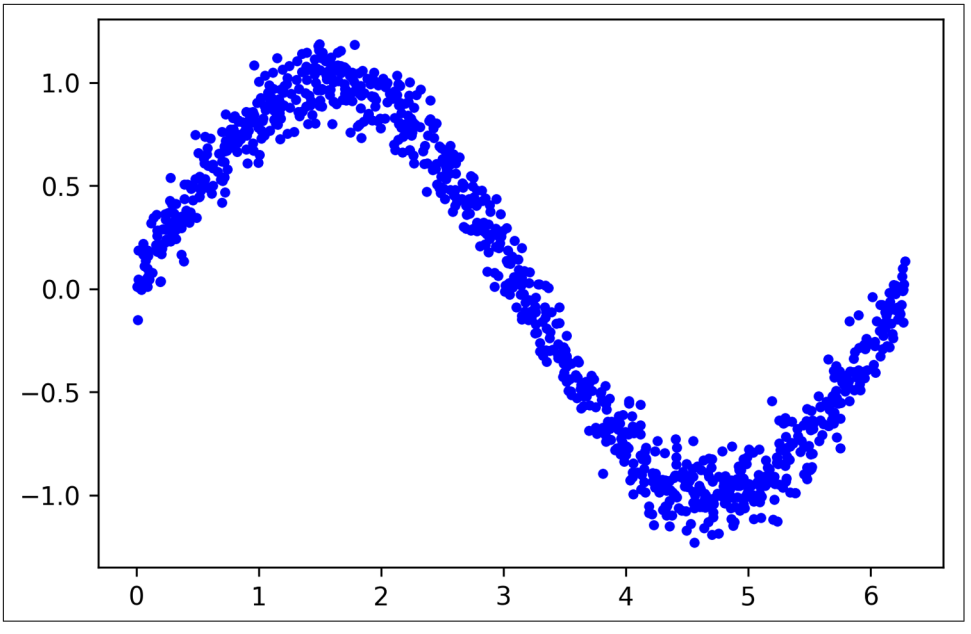


Figure 4-10. A graph of our data with noise added

Much better! Our points are now randomized, so they represent a distribution around a sine wave instead of a smooth, perfect curve. This is much more reflective of a real-world situation, in which data is generally quite messy.

Splitting the Data

From the previous chapter, you might remember that a dataset is often split into three parts: *training*, *validation*, and *test*. To evaluate the accuracy of the model we train, we need to compare its predictions to real data and check how well they match up.

This evaluation happens during training (where it is referred to as validation) and after training (referred to as testing). It's important in each case that we use fresh data that was not already used to train the model.

To ensure that we have data to use for evaluation, we'll set some aside before we begin training. Let's reserve 20% of our data for validation, and another 20% for testing. We'll use the remaining 60% to train the model. This is a typical split used when training models.

The following code splits our data and then plots each set as a different color:

```
# We'll use 60% of our data for training and 20% for testing. The remaining 20%
# will be used for validation. Calculate the indices of each section.
TRAIN_SPLIT =  int(0.6 * SAMPLES)
TEST_SPLIT = int(0.2 * SAMPLES + TRAIN_SPLIT)
```

```
# Use np.split to chop our data into three parts.
# The second argument to np.split is an array of indices where the data will be
# split. We provide two indices, so the data will be divided into three chunks.
x_train, x_validate, x_test = np.split(x_values, [TRAIN_SPLIT, TEST_SPLIT])
y_train, y_validate, y_test = np.split(y_values, [TRAIN_SPLIT, TEST_SPLIT])

# Double check that our splits add up correctly
assert (x_train.size + x_validate.size + x_test.size) ==  SAMPLES

# Plot the data in each partition in different colors:
plt.plot(x_train, y_train, 'b.', label="Train")
plt.plot(x_validate, y_validate, 'y.', label="Validate")
plt.plot(x_test, y_test, 'r.', label="Test")
plt.legend()
plt.show()
```

To split our data, we use another handy NumPy method: `split()`. This method takes an array of data and an array of indices and then chops the data into parts at the indices provided.

Run this cell to see the results of our split. Each type of data will be represented by a different color (or shade, if you're reading the print version of this book), as demonstrated in Figure 4-11.

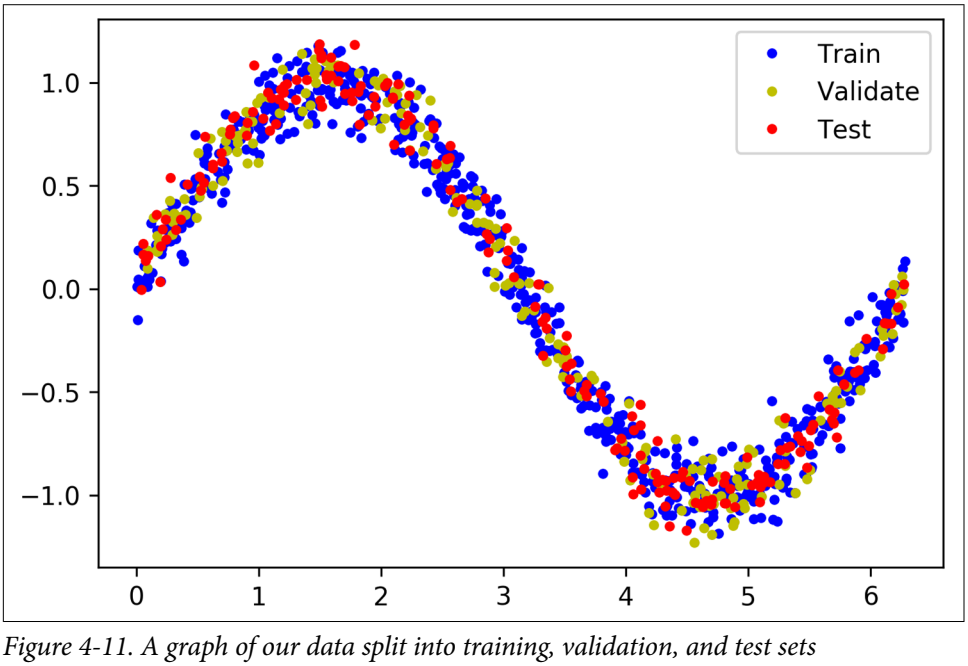


Figure 4-11. A graph of our data split into training, validation, and test sets

Defining a Basic Model

Now that we have our data, it's time to create the model that we'll train to fit it.

We're going to build a model that will take an input value (in this case, `x`) and use it to predict a numeric output value (the sine of `x`). This type of problem is called a *regression*. We can use regression models for all sorts of tasks that require a numeric output. For example, a regression model could attempt to predict a person's running speed in miles per hour based on data from an accelerometer.

To create our model, we're going to design a simple neural network. It uses layers of neurons to attempt to learn any patterns underlying the training data so that it can make predictions.

The code to do this is actually quite straightforward. It uses *Keras*, TensorFlow's high-level API for creating deep learning networks:

```
# We'll use Keras to create a simple model architecture
from tf.keras import layers
model_1 = tf.keras.Sequential()

# First layer takes a scalar input and feeds it through 16 "neurons." The
# neurons decide whether to activate based on the 'relu' activation function.
model_1.add(layers.Dense(16, activation='relu', input_shape=(1,)))

# Final layer is a single neuron, since we want to output a single value
model_1.add(layers.Dense(1))

# Compile the model using a standard optimizer and loss function for regression
model_1.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

# Print a summary of the model's architecture
model_1.summary()
```

First, we create a `Sequential` model using Keras, which just means a model in which each layer of neurons is stacked on top of the next, as we saw in Figure 3-1. We then define two layers. Here's where the first layer is defined:

```
model_1.add(layers.Dense(16, activation='relu', input_shape=(1,)))
```

The first layer has a single input—our `x` value—and 16 neurons. It's a `Dense` layer (also known as a *fully connected* layer), meaning the input will be fed into every single one of its neurons during inference, when we're making predictions. Each neuron will then become *activated* to a certain degree. The amount of activation for each neuron is based on both its *weight* and *bias* values, learned during training, and its *activation function*. The neuron's activation is output as a number.

Activation is calculated by a simple formula, shown in Python. We won't ever need to code this ourselves, since it is handled by Keras and TensorFlow, but it will be helpful to know as we go further into deep learning:

```
activation = activation_function((input * weight) + bias)
```

To calculate the neuron's activation, its input is multiplied by the weight, and the bias is added to the result. The calculated value is passed into the activation function. The resulting number is the neuron's activation.

The activation function is a mathematical function used to shape the output of the neuron. In our network, we're using an activation function called *rectified linear unit*, or *ReLU* for short. This is specified in Keras by the argument `activation=relu`.

ReLU is a simple function, shown here in Python:

```
def relu(input):
    return max(0.0, input)
```

ReLU returns whichever is the larger value: its input, or zero. If its input value is negative, ReLU returns zero. If its input value is above zero, ReLU returns it unchanged.

Figure 4-12 shows the output of ReLU for a range of input values.

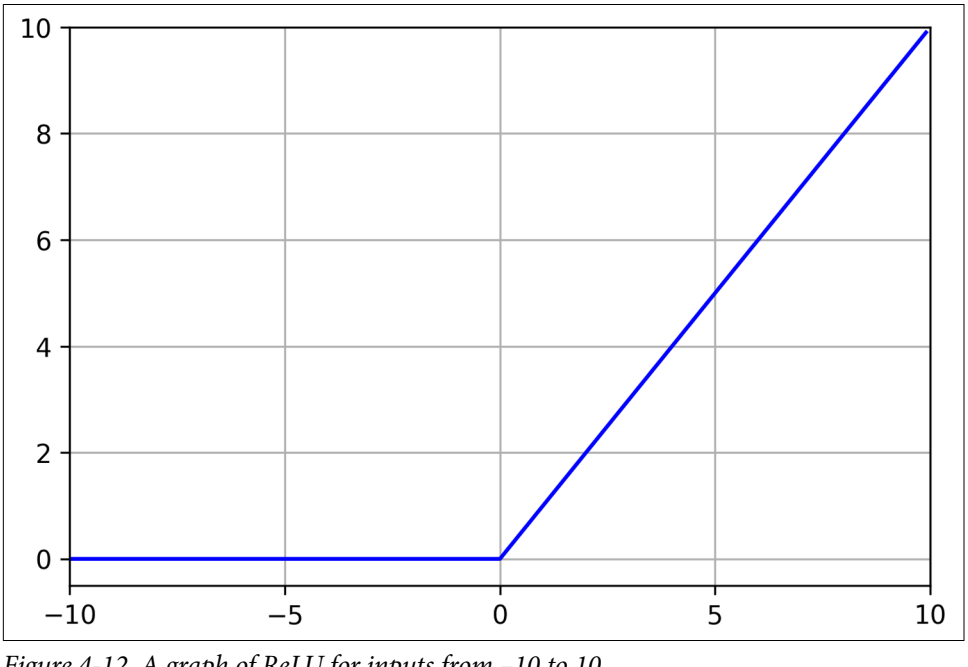


Figure 4-12. A graph of ReLU for inputs from –10 to 10

Without an activation function, the neuron's output would always be a linear function of its input. This would mean that the network could model only linear relationships in which the ratio between `x` and `y` remains the same across the entire range of values. This would prevent a network from modeling our sine wave, because a sine wave is nonlinear.

Since ReLU is nonlinear, it allows multiple layers of neurons to join forces and model complex nonlinear relationships, in which the `y` value doesn't increase by the same amount for every increment of `x`.

There are other activation functions, but ReLU is the most commonly used. You can see some of the other options in the Wikipedia article on activation functions. Each activation function has different trade-offs, and machine learning engineers experiment to find which options work best for a given architecture.

The activation numbers from our first layer will be fed as inputs to our second layer, which is defined in the following line:

```
model_1.add(layers.Dense(1))
```

Because this layer is a single neuron, it will receive 16 inputs, one for each of the neurons in the previous layer. Its purpose is to combine all of the activations from the previous layer into a single output value. Since this is our output layer, we don't specify an activation function—we just want the raw result.

Because this neuron has multiple inputs, it has a corresponding weight value for each. The neuron's output is calculated by the following formula, shown in Python:

```
# Here, `inputs` and `weights` are both NumPy arrays with 16 elements each
output = sum((inputs * weights)) + bias
```

The output value is obtained by multiplying each input with its corresponding weight, summing the results, and then adding the neuron's bias.

The network's weights and biases are learned during training. The `compile()` step in the code shown earlier in the chapter configures some important arguments used in the training process, and prepares the model to be trained:

```
model_1.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
```

The `optimizer` argument specifies the algorithm that will adjust the network to model its input during training. There are several choices, and finding the best one often comes down to experimentation. You can read about the options in the Keras documentation.

The `loss` argument specifies the method used during training to calculate how far the network's predictions are from reality. This method is called a *loss function*. Here, we're using `mse`, or *mean squared error*. This loss function is used in the case of regression problems, for which we're trying to predict a number. There are various loss functions available in Keras. You can see some of the options listed in the Keras docs.

The `metrics` argument allows us to specify some additional functions that are used to judge the performance of our model. We specify `mae`, or *mean absolute error*, which is a helpful function for measuring the performance of a regression model. This metric will be measured during training, and we'll have access to the results after training is done.

After we compile our model, we can use the following line to print some summary information about its architecture:

```
# Print a summary of the model's architecture
model_1.summary()
```

Run the cell in Colab to define the model. You'll see the following output printed:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                32
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 49
Trainable params: 49
Non-trainable params: 0
_________________________________________________________________
```

This table shows the layers of the network, their output shapes, and their numbers of *parameters*. The size of a network—how much memory it takes up—depends mostly on its number of parameters, meaning its total number of weights and biases. This can be a useful metric when discussing model size and complexity.

For simple models like ours, the number of weights can be determined by calculating the number of connections between neurons in the model, given that each connection has a weight.

The network we've just designed consists of two layers. Our first layer has 16 connections—one between its input and each of its neurons. Our second layer has a single neuron, which also has 16 connections—one to each neuron in the first layer. This makes the total number of connections 32.

Since every neuron has a bias, the network has 17 biases, meaning it has a total of 32 + 17 = 49 parameters.

We've now walked through the code that defines our model. Next, we'll begin the training process.

Training Our Model

After we define our model, it's time to train it and then evaluate its performance to see how well it works. When we see the metrics, we can decide if it's good enough, or if we should make changes to our design and train it again.

To train a model in Keras we just call its `fit()` method, passing all of our data and some other important arguments. The code in the next cell shows how:

```
history_1 = model_1.fit(x_train, y_train, epochs=1000, batch_size=16,
                        validation_data=(x_validate, y_validate))
```

Run the code in the cell to begin training. You'll see some logs start to appear:

```
Train on 600 samples, validate on 200 samples
Epoch 1/1000
600/600 [==============================] - 1s 1ms/sample - loss: 0.7887 - mae:
0.7848 - val_loss: 0.5824 - val_mae: 0.6867
Epoch 2/1000
600/600 [==============================] - 0s 155us/sample - loss: 0.4883 -
mae: 0.6194 - val_loss: 0.4742 - val_mae: 0.6056
```

Our model is now training. This will take a little while, so while we wait let's walk through the details of our call to `fit()`:

```
history_1 = model_1.fit(x_train, y_train, epochs=1000, batch_size=16,
                        validation_data=(x_validate, y_validate))
```

First, you'll notice that we assign the return value of our `fit()` call to a variable named `history_1`. This variable contains a ton of information about our training run, and we'll use it later to investigate how things went.

Next, let's take a look at the `fit()` function's arguments:

`x_train`, `y_train`
: The first two arguments to `fit()` are the `x` and `y` values of our training data. Remember that parts of our data are kept aside for validation and testing, so only the training set is used to train the network.

`epochs`
: The next argument specifies how many times our entire training set will be run through the network during training. The more epochs, the more training will occur. You might think that the more training happens, the better the network will be. However, some networks will start to overfit their training data after a certain number of epochs, so we might want to limit the amount of training we do.

  In addition, even if there's no overfitting, a network will stop improving after a certain amount of training. Since training costs time and computational resources, it's best not to train if the network isn't getting better!

We're starting out with 1,000 epochs of training. When training is complete, we can dig into our metrics to discover whether this is the correct number.

`batch_size`
: The `batch_size` argument specifies how many pieces of training data to feed into the network before measuring its accuracy and updating its weights and biases. If we wanted, we could specify a `batch_size` of `1`, meaning we'd run inference on a single datapoint, measure the loss of the network's prediction, update the weights and biases to make the prediction more accurate next time, and then continue this cycle for the rest of the data.

  Because we have 600 datapoints, each epoch would result in 600 updates to the network. This is a lot of computation, so our training would take ages! An alternative might be to select and run inference on multiple datapoints, measure the loss in aggregate, and then updating the network accordingly.

  If we set `batch_size` to `600`, each batch would include all of our training data. We'd now have to make only one update to the network every epoch—much quicker. The problem is, this results in less accurate models. Research has shown that models trained with large batch sizes have less ability to generalize to new data—they are more likely to overfit.

  The compromise is to use a batch size that is somewhere in the middle. In our training code, we use a batch size of 16. This means that we'll choose 16 datapoints at random, run inference on them, calculate the loss in aggregate, and update the network once per batch. If we have 600 points of training data, the network will be updated around 38 times per epoch, which is far better than 600.

  When choosing a batch size, we're making a compromise between training efficiency and model accuracy. The ideal batch size will vary from model to model. It's a good idea to start with a batch size of 16 or 32 and experiment to see what works best.

`validation_data`
: This is where we specify our validation dataset. Data from this dataset will be run through the network throughout the training process, and the network's predictions will be compared with the expected values. We'll see the results of validation in the logs and as part of the `history_1` object.

Training Metrics

Hopefully, by now, training has finished. If not, wait a few moments for it to complete.

We're now going to check various metrics to see how well our network has learned. To begin, let's look at the logs written during training. This will show how the network has improved during training from its random initial state.

Here are the logs for our first and last epochs:

```
Epoch 1/1000
600/600 [==============================] - 1s 1ms/sample - loss: 0.7887 - mae:
0.7848 - val_loss: 0.5824 - val_mae: 0.6867

Epoch 1000/1000
600/600 [==============================] - 0s 124us/sample - loss: 0.1524 -
mae: 0.3039 - val_loss: 0.1737 - val_mae: 0.3249
```

The `loss`, `mae`, `val_loss`, and `val_mae` tell us various things:

`loss`
: This is the output of our loss function. We're using mean squared error, which is expressed as a positive number. Generally, the smaller the loss value, the better, so this is a good thing to watch as we evaluate our network.

  Comparing the first and last epochs, the network has clearly improved during training, going from a loss of ~0.7 to a smaller value of ~0.15. Let's look at the other numbers to see whether this improvement is enough!

`mae`
: This is the mean absolute error of our training data. It shows the average difference between the network's predictions and the expected `y` values from the training data.

  We can expect our initial error to be pretty dismal, given that it's based on an untrained network. This is certainly the case: the network's predictions are off by an average of ~0.78, which is a large number when the range of acceptable values is only from –1 to 1!

  However, even after training, our mean absolute error is ~0.30. This means that our predictions are off by an average of ~0.30, which is still quite awful.

`val_loss`
: This is the output of our loss function on our validation data. In our final epoch, the training loss (~0.15) is slightly lower than the validation loss (~0.17). This is a hint that our network might be overfitting, because it is performing worse on data it has not seen before.

`val_mae`
: This is the mean absolute error for our validation data. With a value of ~0.32, it's worse than the mean absolute error on our training set, which is another sign that the network might be overfitting.

Graphing the History

So far, it's clear that our model is not doing a great job of making accurate predictions. Our task now is to figure out why. To do so, let's make use of the data collected in our `history_1` object.

The next cell extracts the training and validation loss data from the history object and plots it on a chart:

```
loss = history_1.history['loss']
val_loss = history_1.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'g.', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

The `history_1` object contains an attribute called, `history_1.history`, which is a dictionary recording metric values during training and validation. We use this to collect the data we're going to plot. For our x-axis we use the epoch number, which we determine by looking at the number of loss datapoints. Run the cell and you'll see the graph in Figure 4-13.

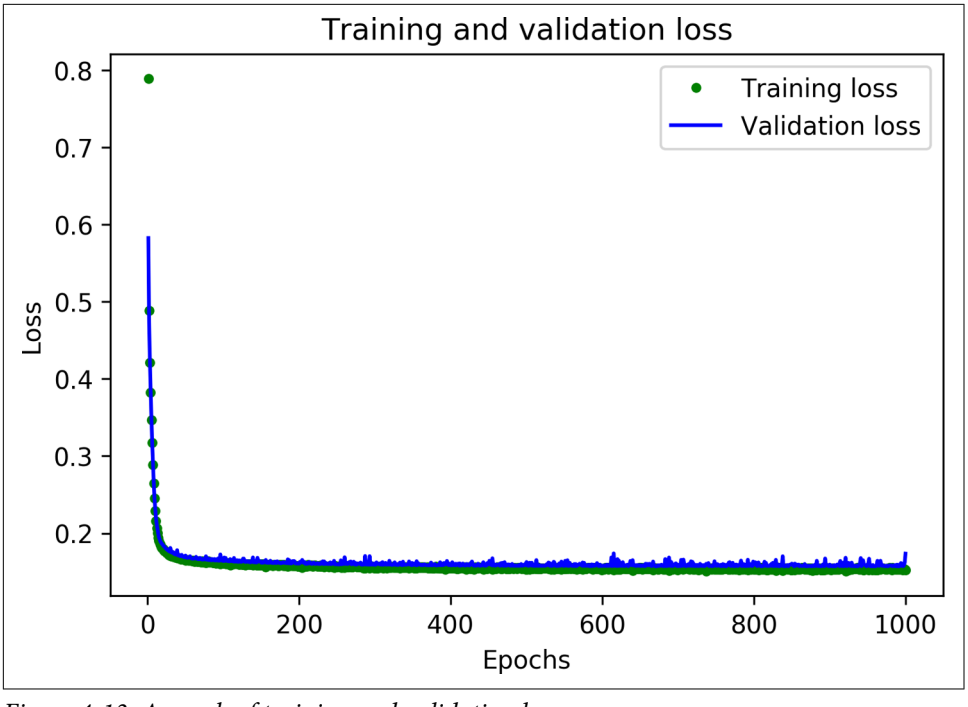


Figure 4-13. A graph of training and validation loss

As you can see, the amount of loss rapidly decreases over the first 50 epochs, before flattening out. This means that the model is improving and producing more accurate predictions.

Our goal is to stop training when either the model is no longer improving or the training loss is less than the validation loss, which would mean that the model has learned to predict the training data so well that it can no longer generalize to new data.

The loss drops precipitously in the first few epochs, which makes the rest of the graph quite difficult to read. Let's skip the first 100 epochs by running the next cell:

```
# Exclude the first few epochs so the graph is easier to read
SKIP = 100

plt.plot(epochs[SKIP:], loss[SKIP:], 'g.', label='Training loss')
plt.plot(epochs[SKIP:], val_loss[SKIP:], 'b.', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Figure 4-14 presents the graph produced by this cell.

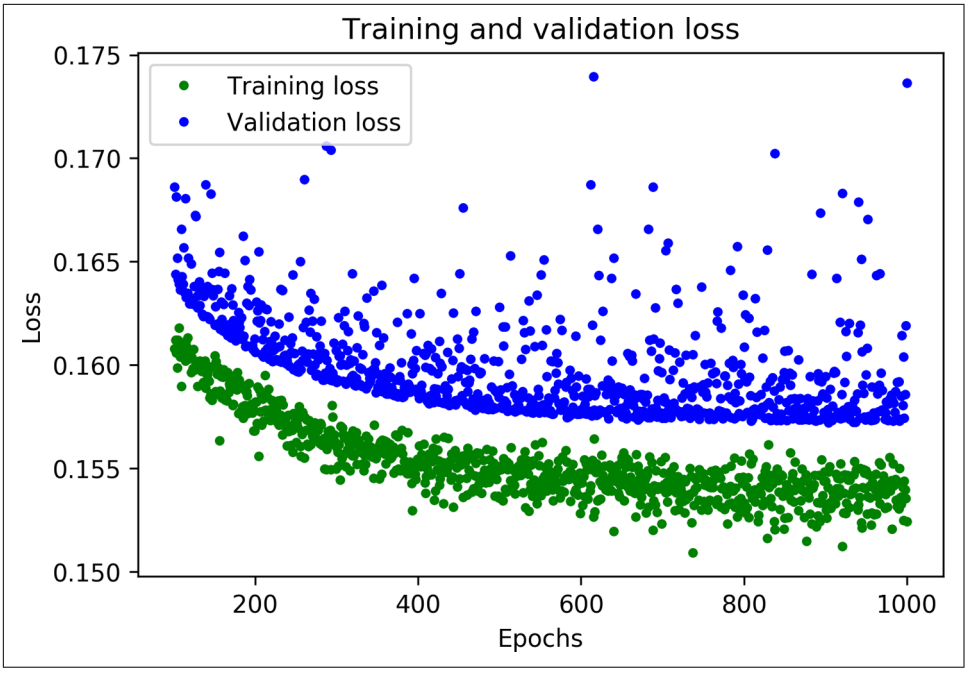


Figure 4-14. A graph of training and validation loss, skipping the first 100 epochs

Now that we've zoomed in, you can see that loss continues to reduce until around 600 epochs, at which point it is mostly stable. This means that there's probably no need to train our network for so long.

However, you can also see that the lowest loss value is still around 0.15. This seems relatively high. In addition, the validation loss values are consistently even higher.

To gain more insight into our model's performance we can plot some more data. This time, let's plot the mean absolute error. Run the next cell to do so:

```
# Draw a graph of mean absolute error, which is another way of
# measuring the amount of error in the prediction.
mae = history_1.history['mae']
val_mae = history_1.history['val_mae']

plt.plot(epochs[SKIP:], mae[SKIP:], 'g.', label='Training MAE')
plt.plot(epochs[SKIP:], val_mae[SKIP:], 'b.', label='Validation MAE')
plt.title('Training and validation mean absolute error')
plt.xlabel('Epochs')
plt.ylabel('MAE')
plt.legend()
plt.show()
```

Figure 4-15 shows the resulting graph.

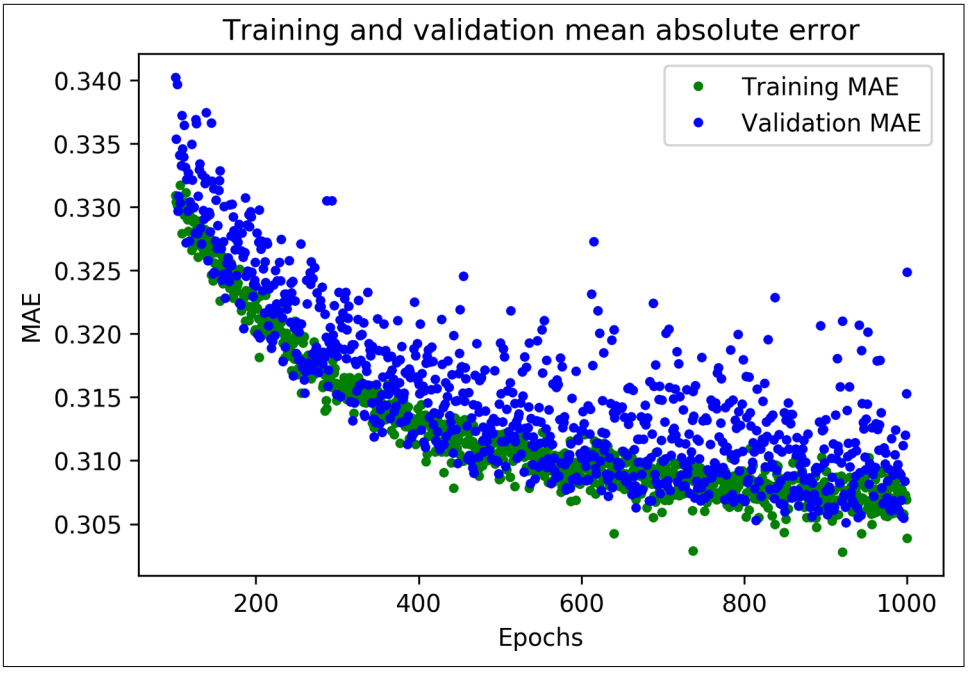


Figure 4-15. A graph of mean absolute error during training and validation

This graph of mean absolute error gives us some further clues. We can see that on average, the training data shows lower error than the validation data, which means that the network might have overfit, or learned the training data so rigidly that it can't make effective predictions about new data.

In addition, the mean absolute error values are quite high, around ~0.31, which means that some of the model's predictions are wrong by at least 0.31. Since our expected values only range in size from –1 to +1, an error of 0.31 means we are very far from accurately modeling the sine wave.

To get more insight into what is happening, we can plot our network's predictions for the training data against the expected values.

This happens in the following cell:

```
# Use the model to make predictions from our validation data
predictions = model_1.predict(x_train)

# Plot the predictions along with the test data
plt.clf()
plt.title('Training data predicted vs actual values')
plt.plot(x_test, y_test, 'b.', label='Actual')
plt.plot(x_train, predictions, 'r.', label='Predicted')
```

```
plt.legend()
plt.show()
```

By calling `model_1.predict(x_train)`, we run inference on all of the `x` values from the training data. The method returns an array of predictions. Let's plot this on the graph alongside the actual `y` values from our training set. Run the cell to see the graph in Figure 4-16.

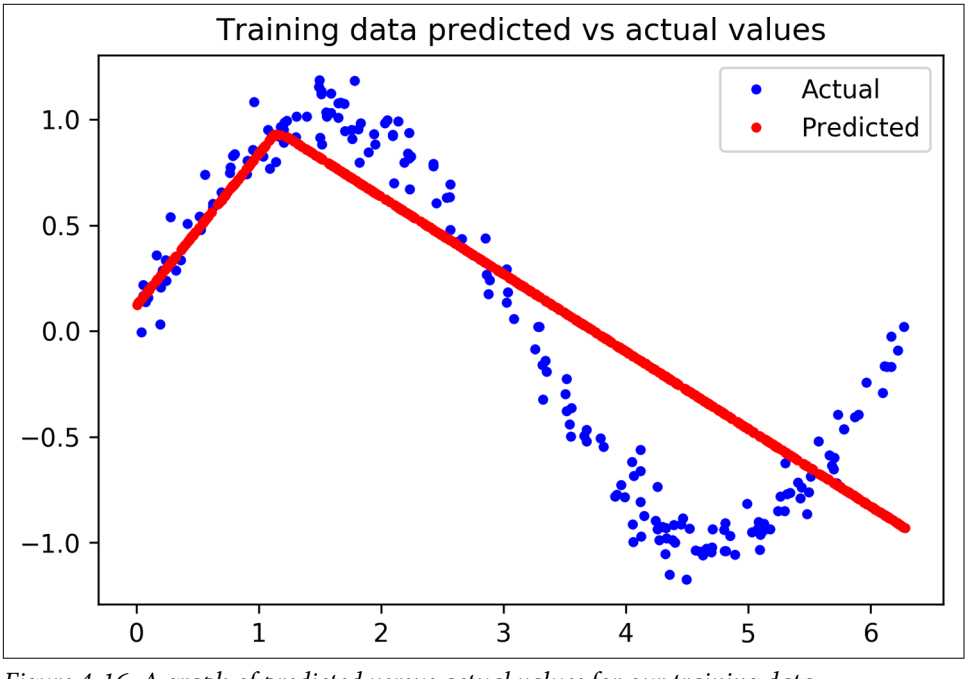


Figure 4-16. A graph of predicted versus actual values for our training data

Oh, dear! The graph makes it clear that our network has learned to approximate the sine function in a very limited way. The predictions are highly linear, and only very roughly fit the data.

The rigidity of this fit suggests that the model does not have enough capacity to learn the full complexity of the sine wave function, so it's able to approximate it only in an overly simplistic way. By making our model bigger, we should be able to improve its performance.

Improving Our Model

Armed with the knowledge that our original model was too small to learn the complexity of our data, we can try to make it better. This is a normal part of the machine learning workflow: design a model, evaluate its performance, and make changes in the hope of seeing improvement.

An easy way to make the network bigger is to add another layer of neurons. Each layer of neurons represents a transformation of the input that will hopefully get it closer to the expected output. The more layers of neurons a network has, the more complex these transformations can be.

Run the following cell to redefine our model in the same way as earlier, but with an additional layer of 16 neurons in the middle:

```
model_2 = tf.keras.Sequential()

# First layer takes a scalar input and feeds it through 16 "neurons." The
# neurons decide whether to activate based on the 'relu' activation function.
model_2.add(layers.Dense(16, activation='relu', input_shape=(1,)))

# The new second layer may help the network learn more complex representations
model_2.add(layers.Dense(16, activation='relu'))

# Final layer is a single neuron, since we want to output a single value
model_2.add(layers.Dense(1))

# Compile the model using a standard optimizer and loss function for regression
model_2.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

# Show a summary of the model
model_2.summary()
```

As you can see, the code is basically the same as for our first model, but with an additional `Dense` layer. Let's run the cell to see the `summary()` results:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 16)                32
_________________________________________________________________
dense_3 (Dense)              (None, 16)                272
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 17
=================================================================
Total params: 321
Trainable params: 321
Non-trainable params: 0
_________________________________________________________________
```

With two layers of 16 neurons, our new model is a lot larger. It has (1 * 16) + (16 * 16) + (16 * 1) = 288 weights, plus 16 + 16 + 1 = 33 biases, for a total of 288 + 33 = 321 parameters. Our original model had only 49 total parameters, so this is a 555% increase in model size. Hopefully, this extra capacity will help represent the complexity of our data.

The following cell will train our new model. Since our first model stopped improving so quickly, let's train for fewer epochs this time—only 600. Run this cell to begin training:

```
history_2 = model_2.fit(x_train, y_train, epochs=600, batch_size=16,
                        validation_data=(x_validate, y_validate))
```

When training is complete, we can take a look at the final log to get a quick feel for whether things have improved:

```
Epoch 600/600
600/600 [==============================] - 0s 150us/sample - loss: 0.0115 -
mae: 0.0859 - val_loss: 0.0104 - val_mae: 0.0806
```

Wow! You can see that we've already achieved a huge improvement—validation loss has dropped from 0.17 to 0.01, and validation mean absolute error has dropped from 0.32 to 0.08. This looks very promising.

To see how things are going, let's run the next cell. It's set up to generate the same graphs we used last time.

First, we draw a graph of the loss:

```
# Draw a graph of the loss, which is the distance between
# the predicted and actual values during training and validation.
loss = history_2.history['loss']
val_loss = history_2.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'g.', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Figure 4-17 shows the result.

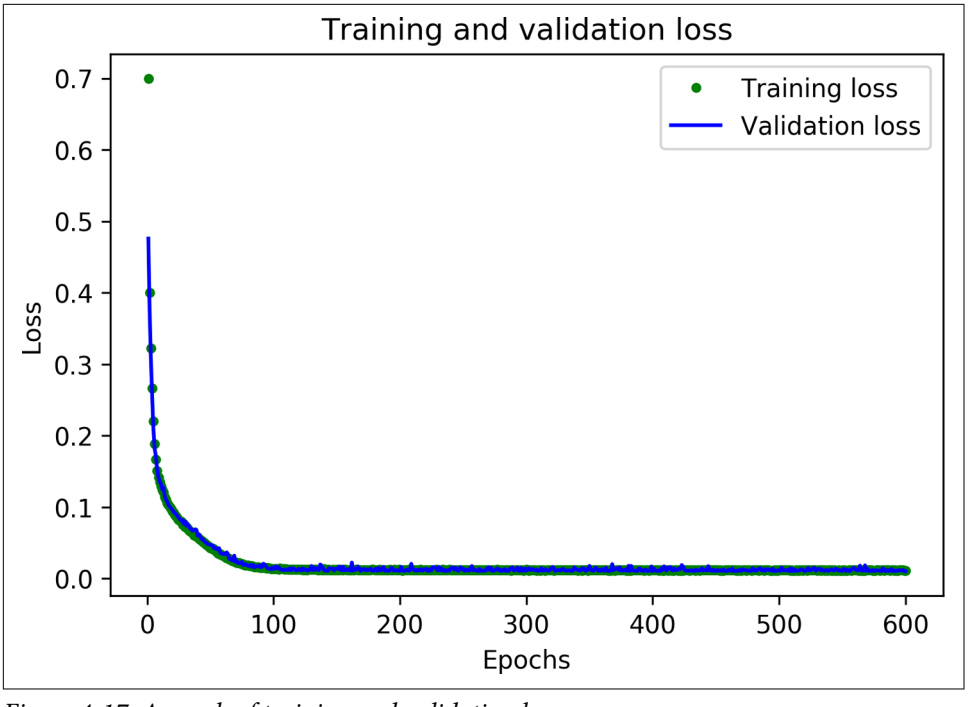


Figure 4-17. A graph of training and validation loss

Next, we draw the same loss graph but with the first 100 epochs skipped so that we can better see the detail:

```
# Exclude the first few epochs so the graph is easier to read
SKIP = 100

plt.clf()

plt.plot(epochs[SKIP:], loss[SKIP:], 'g.', label='Training loss')
plt.plot(epochs[SKIP:], val_loss[SKIP:], 'b.', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Figure 4-18 presents the output.

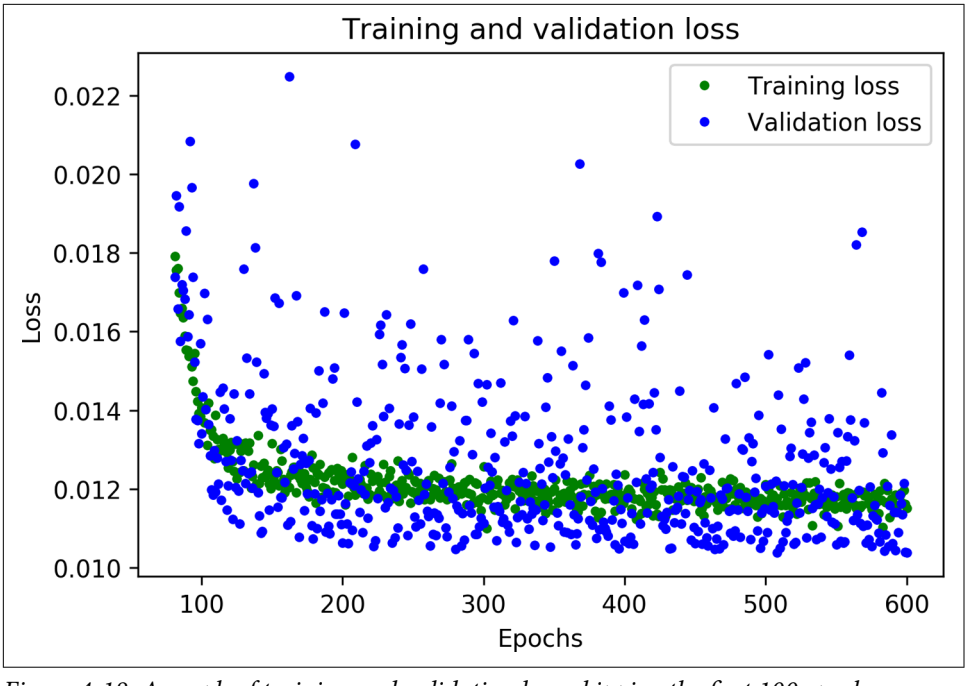


Figure 4-18. A graph of training and validation loss, skipping the first 100 epochs

Finally, we plot the mean absolute error for the same set of epochs:

```
plt.clf()

# Draw a graph of mean absolute error, which is another way of
# measuring the amount of error in the prediction.
mae = history_2.history['mae']
val_mae = history_2.history['val_mae']

plt.plot(epochs[SKIP:], mae[SKIP:], 'g.', label='Training MAE')
plt.plot(epochs[SKIP:], val_mae[SKIP:], 'b.', label='Validation MAE')
plt.title('Training and validation mean absolute error')
plt.xlabel('Epochs')
plt.ylabel('MAE')
plt.legend()
plt.show()
```

Figure 4-19 depicts the graph.

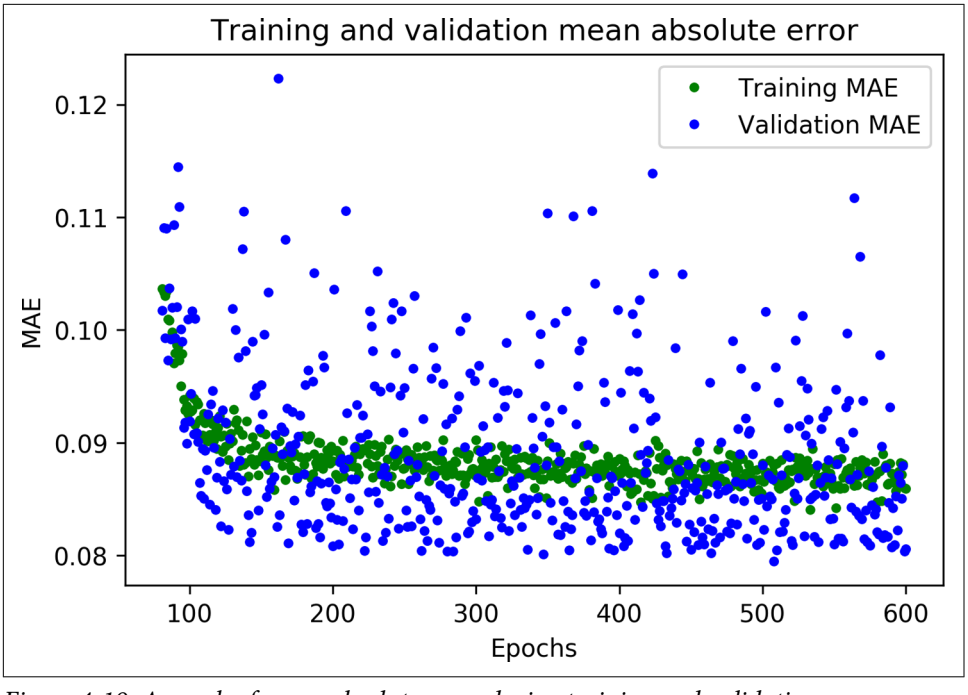


Figure 4-19. A graph of mean absolute error during training and validation

Great results! From these graphs, we can see two exciting things:

- The metrics are broadly better for validation than training, which means the network is not overfitting.
- The overall loss and mean absolute error are much better than in our previous network.

You might be wondering why the metrics for validation are better than those for training, and not merely identical. The reason is that validation metrics are calculated at the end of each epoch, meanwhile training metrics are calculated while the epoch of training is still in progress. This means validation happens on a model that has been trained for slightly longer.

Based on our validation data, our model seems to be performing great. However, to be sure of this, we need to run one final test.

Testing

Earlier, we set aside 20% of our data to use for testing. As we discussed, it's very important to have separate validation and test data. Since we fine-tune our network based on its validation performance, there's a risk that we might accidentally tune the

model to overfit its validation set and that it might not be able to generalize to new data. By retaining some fresh data and using it for a final test of our model, we can make sure that this has not happened.

After we've used our test data, we need to resist the urge to tune our model further. If we did make changes with the goal of improving test performance, we might cause it to overfit our test set. If we did this, we wouldn't be able to know, because we'd have no fresh data left to test with.

This means that if our model performs badly on our test data, it's time to go back to the drawing board. We'll need to stop optimizing the current model and come up with a brand new architecture.

With that in mind, the following cell will evaluate our model against our test data:

```
# Calculate and print the loss on our test dataset
loss = model_2.evaluate(x_test, y_test)

# Make predictions based on our test dataset
predictions = model_2.predict(x_test)

# Graph the predictions against the actual values
plt.clf()
plt.title('Comparison of predictions and actual values')
plt.plot(x_test, y_test, 'b.', label='Actual')
plt.plot(x_test, predictions, 'r.', label='Predicted')
plt.legend()
plt.show()
```

First, we call the model's `evaluate()` method with the test data. This will calculate and print the loss and mean absolute error metrics, informing us as to how far the model's predictions deviate from the actual values. Next, we make a set of predictions and plot them on a graph alongside the actual values.

Now we can run the cell to learn how our model is performing! First, let's see the results of `evaluate()`:

```
200/200 [==============================] - 0s 71us/sample - loss: 0.0103 - mae:
0.0718
```

This shows that 200 datapoints were evaluated, which is our entire test set. The model took 71 microseconds to make each prediction. The loss metric was 0.0103, which is excellent, and very close to our validation loss of 0.0104. Our mean absolute error, 0.0718, is also very small and fairly close to its equivalent in validation, 0.0806.

This means that our model is working great, and it isn't overfitting! If the model had overfit our validation data, we could expect that the metrics on our test set would be significantly worse than those resulting from validation.

The graph of our predictions against our actual values, shown in Figure 4-20, makes it clear how well our model is performing.

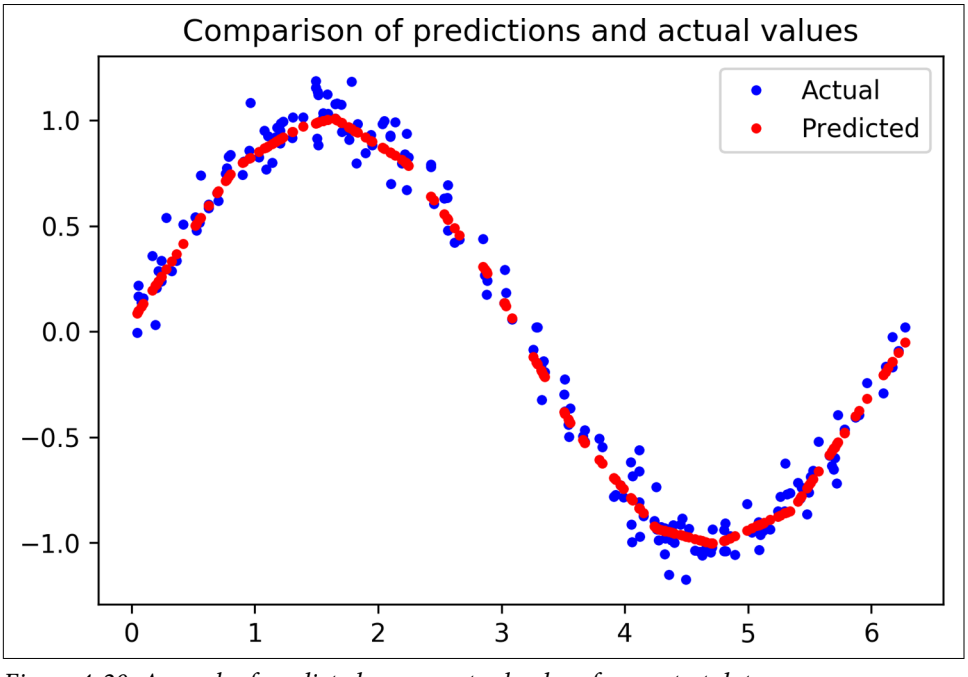


Figure 4-20. A graph of predicted versus actual values for our test data

You can see that, for the most part, the dots representing *predicted* values form a smooth curve along the center of the distribution of *actual* values. Our network has learned to approximate a sine curve, even though the dataset was noisy!

If you look closely, however, you'll see that there are some imperfections. The peak and trough of our predicted sine wave are not perfectly smooth, like a real sine wave would be. Variations in our training data, which is randomly distributed, have been learned by our model. This is a mild case of overfitting: instead of learning the smooth sine function, our model has learned to replicate the exact shape of our data.

For our purposes, this overfitting isn't a major problem. Our goal is for this model to gently fade an LED on and off, and it doesn't need to be perfectly smooth to achieve this. If we thought the level of overfitting was problematic, we could attempt to address it through regularization techniques or by obtaining more training data.

Now that we're happy with our model, let's get it ready to deploy on-device!

Converting the Model for TensorFlow Lite

At the beginning of this chapter we briefly touched on TensorFlow Lite, which is a set of tools for running TensorFlow models on "edge devices"—meaning everything from mobile phones down to microcontroller boards.

Chapter 13 goes into detail on TensorFlow Lite for Microcontrollers. For now, we can think of it as having two main components:

TensorFlow Lite Converter
: This converts TensorFlow models into a special, space-efficient format for use on memory-constrained devices, and it can apply optimizations that further reduce the model size and make it run faster on small devices.

TensorFlow Lite Interpreter
: This runs an appropriately converted TensorFlow Lite model using the most efficient operations for a given device.

Before we use our model with TensorFlow Lite, we need to convert it. We use the TensorFlow Lite Converter's Python API to do this. It takes our Keras model and writes it to disk in the form of a *FlatBuffer*, which is a special file format designed to be space-efficient. Because we're deploying to devices with limited memory, this will come in handy! We'll look at FlatBuffers in more detail in Chapter 12.

In addition to creating a FlatBuffer, the TensorFlow Lite Converter can also apply optimizations to the model. These optimizations generally reduce the size of the model, the time it takes to run, or both. This can come at the cost of a reduction in accuracy, but the reduction is often small enough that it's worthwhile. You can read more about optimizations in Chapter 13.

One of the most useful optimizations is *quantization*. By default, the weights and biases in a model are stored as 32-bit floating-point numbers so that high-precision calculations can occur during training. Quantization allows you to reduce the precision of these numbers so that they fit into 8-bit integers—a four times reduction in size. Even better, because it's easier for a CPU to perform math with integers than with floats, a quantized model will run faster.

The coolest thing about quantization is that it often results in minimal loss in accuracy. This means that when deploying to low-memory devices, it is nearly always worthwhile.

In the following cell, we use the converter to create and save two new versions of our model. The first is converted to the TensorFlow Lite FlatBuffer format, but without any optimizations. The second is quantized.

Run the cell to convert the model into these two variants:

```
# Convert the model to the TensorFlow Lite format without quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model_2)
tflite_model = converter.convert()

# Save the model to disk
open("sine_model.tflite," "wb").write(tflite_model)

# Convert the model to the TensorFlow Lite format with quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model_2)
# Indicate that we want to perform the default optimizations,
# which include quantization
converter.optimizations = [tf.lite.Optimize.DEFAULT]
# Define a generator function that provides our test data's x values
# as a representative dataset, and tell the converter to use it
def representative_dataset_generator():
  for value in x_test:
    # Each scalar value must be inside of a 2D array that is wrapped in a list
    yield [np.array(value, dtype=np.float32, ndmin=2)]
converter.representative_dataset = representative_dataset_generator
# Convert the model
tflite_model = converter.convert()

# Save the model to disk
open("sine_model_quantized.tflite," "wb").write(tflite_model)
```

To create a quantized model that runs as efficiently as possible, we need to provide a *representative dataset*—a set of numbers that represent the full range of input values of the dataset on which the model was trained.

In the preceding cell, we can use our test dataset's `x` values as a representative dataset. We define a function, `representative_dataset_generator()`, that uses the `yield` operator to return them one by one.

To prove these models are still accurate after conversion and quantization, we use both of them to make predictions and compare these against our test results. Given that these are TensorFlow Lite models, we need to use the TensorFlow Lite interpreter to do so.

Because it's designed primarily for efficiency, the TensorFlow Lite interpreter is slightly more complicated to use than the Keras API. To make predictions with our Keras model, we could just call the `predict()` method, passing an array of inputs. With TensorFlow Lite, we need to do the following:

1. Instantiate an `Interpreter` object.
2. Call some methods that allocate memory for the model.
3. Write the input to the input tensor.
4. Invoke the model.

5. Read the output from the output tensor.

This sounds like a lot, but don't worry about it too much for now; we'll walk through it in detail in Chapter 5. For now, run the following cell to make predictions with both models and plot them on a graph, alongside the results from our original, unconverted model:

```
# Instantiate an interpreter for each model
sine_model = tf.lite.Interpreter('sine_model.tflite')
sine_model_quantized = tf.lite.Interpreter('sine_model_quantized.tflite')

# Allocate memory for each model
sine_model.allocate_tensors()
sine_model_quantized.allocate_tensors()

# Get indexes of the input and output tensors
sine_model_input_index = sine_model.get_input_details()[0]["index"]
sine_model_output_index = sine_model.get_output_details()[0]["index"]
sine_model_quantized_input_index = sine_model_quantized.get_input_details()[0]
["index"]
sine_model_quantized_output_index = \
  sine_model_quantized.get_output_details()[0]["index"]

# Create arrays to store the results
sine_model_predictions = []
sine_model_quantized_predictions = []

# Run each model's interpreter for each value and store the results in arrays
for x_value in x_test:
  # Create a 2D tensor wrapping the current x value
  x_value_tensor = tf.convert_to_tensor([[x_value]], dtype=np.float32)
  # Write the value to the input tensor
  sine_model.set_tensor(sine_model_input_index, x_value_tensor)
  # Run inference
  sine_model.invoke()
  # Read the prediction from the output tensor
  sine_model_predictions.append(
      sine_model.get_tensor(sine_model_output_index)[0])
  # Do the same for the quantized model
  sine_model_quantized.set_tensor\
  (sine_model_quantized_input_index, x_value_tensor)
  sine_model_quantized.invoke()
  sine_model_quantized_predictions.append(
      sine_model_quantized.get_tensor(sine_model_quantized_output_index)[0])


# See how they line up with the data
plt.clf()
plt.title('Comparison of various models against actual values')
plt.plot(x_test, y_test, 'bo', label='Actual')
plt.plot(x_test, predictions, 'ro', label='Original predictions')
plt.plot(x_test, sine_model_predictions, 'bx', label='Lite predictions')
```

```
plt.plot(x_test, sine_model_quantized_predictions, 'gx', \
  label='Lite quantized predictions')
plt.legend()
plt.show()
```

Running this cell yields the graph in Figure 4-21.

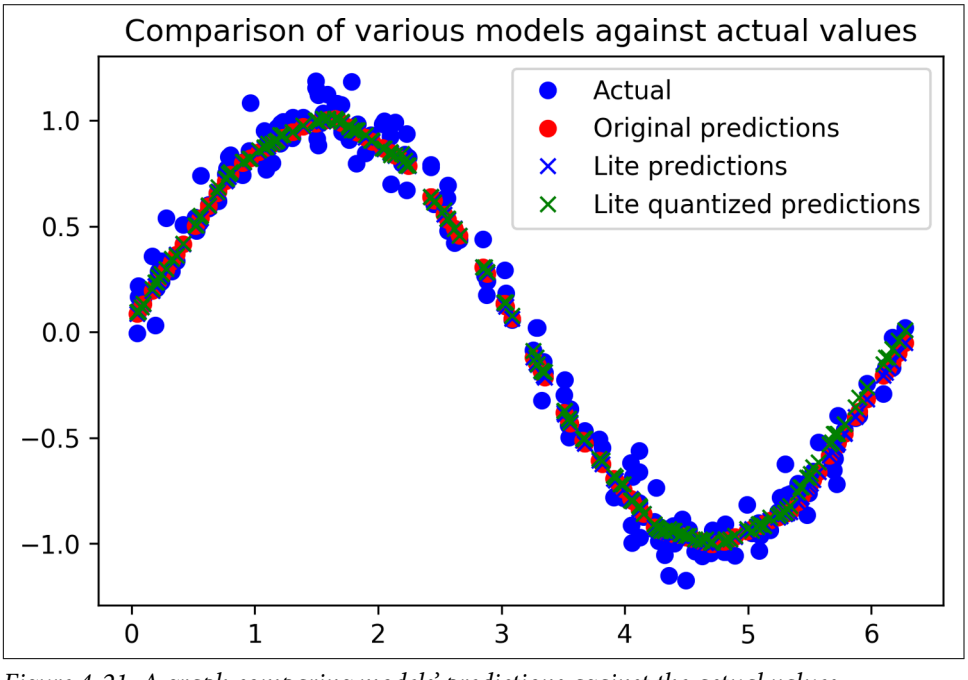


Figure 4-21. A graph comparing models' predictions against the actual values

We can see from the graph that the predictions for the original model, the converted model, and the quantized model are all close enough to be indistinguishable. Things are looking good!

Since quantization makes models smaller, let's compare both converted models to see the difference in size. Run the following cell to calculate their sizes and compare them:

```
import os
basic_model_size = os.path.getsize("sine_model.tflite")
print("Basic model is %d bytes" % basic_model_size)
quantized_model_size = os.path.getsize("sine_model_quantized.tflite")
print("Quantized model is %d bytes" % quantized_model_size)
difference = basic_model_size - quantized_model_size
print("Difference is %d bytes" % difference)
```

You should see the following output:

```
Basic model is 2736 bytes
Quantized model is 2512 bytes
Difference is 224 bytes
```

Our quantized model is 224 bytes smaller than the original version, which is great—but it's only a minor reduction in size. At around 2.4 KB, this model is already so small that the weights and biases make up only a fraction of the overall size. In addition to weights, the model contains all the logic that makes up the architecture of our deep learning network, known as its *computation graph*. For truly tiny models, this can add up to more size than the model's weights, meaning quantization has little effect.

More complex models have many more weights, meaning the space saving from quantization will be much higher. It can be expected to approach four times for most sophisticated models.

Regardless of its exact size, our quantized model will take less time to execute than the original version, which is important on a tiny microcontroller.

Converting to a C File

The final step in preparing our model for use with TensorFlow Lite for Microcontrollers is to convert it into a C source file that can be included in our application.

So far during this chapter, we've been using TensorFlow Lite's Python API. This means that we've been able to use the `Interpreter` constructor to load our model files from disk.

However, most microcontrollers don't have a filesystem, and even if they did, the extra code required to load a model from disk would be wasteful given our limited space. Instead, as an elegant solution, we provide the model in a C source file that can be included in our binary and loaded directly into memory.

In the file, the model is defined as an array of bytes. Fortunately, there's a convenient Unix tool named `xxd` that is able to convert a given file into the required format.

The following cell runs `xxd` on our quantized model, writes the output to a file called *sine_model_quantized.cc*, and prints it to the screen:

```
# Install xxd if it is not available
!apt-get -qq install xxd
# Save the file as a C source file
!xxd -i sine_model_quantized.tflite > sine_model_quantized.cc
# Print the source file
!cat sine_model_quantized.cc
```

The output is very long, so we won't reproduce it all here, but here's a snippet that includes just the beginning and end:

```
unsigned char sine_model_quantized_tflite[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x12, 0x00,
  0x1c, 0x00, 0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00,
  // ...
  0x00, 0x00, 0x08, 0x00, 0x0a, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09,
  0x04, 0x00, 0x00, 0x00
};
unsigned int sine_model_quantized_tflite_len = 2512;
```

To use this model in a project, you could either copy and paste the source or download the file from the notebook.

Wrapping Up

And with that, we're done building our model. We've trained, evaluated, and converted a TensorFlow deep learning network that can take a number between 0 and 2π and output a good-enough approximation of its sine.

This was our first taste of using Keras to train a tiny model. In future projects, we'll be training models that are still tiny, but *far* more sophisticated.

For now, let's move on to Chapter 5, where we'll write code to run our model on microcontrollers.

CHAPTER 5

The "Hello World" of TinyML: Building an Application

A model is just one part of a machine learning application. Alone, it's just a blob of information; it can't do much at all. To use our model, we need to wrap it in code that sets up the necessary environment for it to run, provides it with inputs, and uses its outputs to generate behavior. Figure 5-1 shows how the model, on the right hand side, fits into a basic TinyML application.

In this chapter, we will build an embedded application that uses our sine model to create a tiny light show. We'll set up a continuous loop that feeds an `x` value into the model, runs inference, and uses the result to switch an LED on and off, or to control an animation if our device has an LCD display.

This application has already been written. It's a C++ 11 program whose code is designed to show the smallest possible implementation of a full TinyML application, avoiding any complex logic. This simplicity makes it a helpful tool for learning how to use TensorFlow Lite for Microcontrollers, since you can see exactly what code is necessary and very little else. It also makes it a useful template. After reading this chapter, you'll understand the general structure of a TensorFlow Lite for Microcontrollers program, and you can reuse the same structure in your own projects.

This chapter walks through the application code and explains how it works. The next chapter will provide detailed instructions for building and deploying it to several devices.

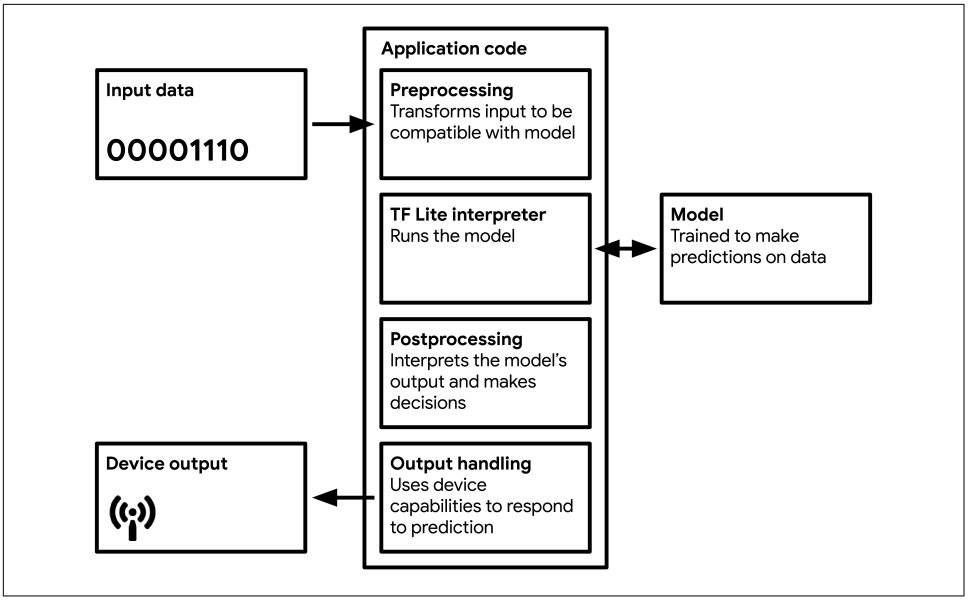


Figure 5-1. A basic TinyML application architecture

If you're not familiar with C++, don't panic. The code is relatively simple, and we explain everything in detail. By the time we're done, you should feel comfortable with all the code that's required to run a model, and you might even learn a little C++ along the way.

Remember, since TensorFlow is an actively developed open source project, there might be some minor differences between the code printed here and the code online. Don't worry—even if a few lines of code change, the basic principles remain the same.

Walking Through the Tests

Before getting our hands dirty with application code, it's often a good idea to write some tests. Tests are short snippets of code that demonstrate a particular piece of logic. Since they are made of working code, we can run them to prove that the code does what it's supposed to. After we have written them, tests are often run automatically as a way to continually verify that a project is still doing what we expect despite any changes we might make to its code. They're also very useful as working examples of how to do things.

The `hello_world` example has a test, defined in *hello_world_test.cc*, that loads our model and uses it to run inference, checking that its predictions are what we expect. It contains the exact code needed to do this, and nothing else, so it will be a great place

to start learning TensorFlow Lite for Microcontrollers. In this section, we walk through the test and explain what each and every part of it does. After we're done reading the code, we can run the test to prove that it's correct.

Let's now walk through it, section by section. If you're at a computer, it might be helpful to open up *hello_world_test.cc* and follow along.

Including Dependencies

The first part, below the license header (which specifies that anybody can use or share this code under the Apache 2.0 open source license), looks like this:

```
#include "tensorflow/lite/micro/examples/hello_world/sine_model_data.h"
#include "tensorflow/lite/micro/kernels/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/testing/micro_test.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
```

The `#include` directive is a way for C++ code to specify other code that it depends on. When a code file is referenced with an `#include`, any logic or variables it defines will be available for us to use. In this section, we use `#include` to import the following items:

tensorflow/lite/micro/examples/hello_world/sine_model_data.h
: The sine model we trained, converted, and transformed into C++ using `xxd`

tensorflow/lite/micro/kernels/all_ops_resolver.h
: A class that allows the interpreter to load the operations used by our model

tensorflow/lite/micro/micro_error_reporter.h
: A class that can log errors and output to help with debugging

tensorflow/lite/micro/micro_interpreter.h
: The TensorFlow Lite for Microcontrollers interpreter, which will run our model

tensorflow/lite/micro/testing/micro_test.h
: A lightweight framework for writing tests, which allows us to run this file as a test

tensorflow/lite/schema/schema_generated.h
: The schema that defines the structure of TensorFlow Lite FlatBuffer data, used to make sense of the model data in *sine_model_data.h*

tensorflow/lite/version.h
: The current version number of the schema, so we can check that the model was defined with a compatible version

We'll talk more about some of these dependencies as we dig into the code.

By convention, C++ code designed to be used with `#include` directives is written as two files: a *.cc* file, known as the *source file*, and a *.h* file, known as the *header file*. Header files define the interface that allows the code to connect to other parts of the program. They contain things like variable and class declarations, but very little logic. Source files implement the actual logic that performs computation and makes things happen.

When we `#include` a dependency, we specify its header file. For example, the test we're walking through includes *micro_interpreter.h*. If we look at that file, we can see that it defines a class but doesn't contain much logic. Instead, its logic is contained within *micro_interpreter.cc*.

Setting Up the Test

The next part of the code is used by the TensorFlow Lite for Microcontrollers testing framework. It looks like this:

```
TF_LITE_MICRO_TESTS_BEGIN

TF_LITE_MICRO_TEST(LoadModelAndPerformInference) {
```

In C++, you can define specially named chunks of code that can be reused by including their names elsewhere. These chunks of code are called *macros*. The two statements here, `TF_LITE_MICRO_TESTS_BEGIN` and `TF_LITE_MICRO_TEST`, are the names of macros. They are defined in the file *micro_test.h*.

These macros wrap the rest of our code in the necessary apparatus for it to be executed by the TensorFlow Lite for Microcontrollers testing framework. We don't need to worry about how exactly this works; we just know that we can use these macros as shortcuts to set up a test.

The second macro, named `TF_LITE_MICRO_TEST`, accepts an argument. In this case, the argument being passed in is `LoadModelAndPerformInference`. This argument is the test name, and when the tests are run, it will be output along with the test results so that we can see whether the test passed or failed.

Getting Ready to Log Data

The remaining code in the file is the actual logic of our test. Let's take a look at the first portion:

```
  // Set up logging
  tflite::MicroErrorReporter micro_error_reporter;
  tflite::ErrorReporter* error_reporter = &micro_error_reporter;
```

In the first line, we define a `MicroErrorReporter` instance. The `MicroErrorReporter` class is defined in *micro_error_reporter.h*. It provides a mechanism for logging debug

information during inference. We'll be calling it to print debug information, and the TensorFlow Lite for Microcontrollers interpreter will use it to print any errors it encounters.

You've probably noticed the `tflite::` prefix before each of the type names, such as `tflite::MicroErrorReporter`. This is a *namespace*, which is just a way to help organize C++ code. TensorFlow Lite defines all of its useful stuff under the namespace `tflite`, which means that if another library happens to implement classes with the same name, they won't conflict with those that TensorFlow Lite provides.

The first declaration seems straightforward, but what about the funky-looking second line, with the `*` and `&` characters? Why are we declaring an `ErrorReporter` when we already have a `MicroErrorReporter`?

```
tflite::ErrorReporter* error_reporter = &micro_error_reporter;
```

To explain what is happening here, we need to know a little background information.

`MicroErrorReporter` is a subclass of the `ErrorReporter` class, which provides a template for how this sort of debug logging mechanism should work in TensorFlow Lite. `MicroErrorReporter` overrides one of `ErrorReporter`'s methods, replacing it with logic that is specifically written for use on microcontrollers.

In the preceding code line, we create a variable called `error_reporter`, which has the type `ErrorReporter`. It's also a pointer, indicated by the `*` used in its declaration.

A pointer is a special type of variable that, instead of holding a value, holds a reference to a location in memory where a value can be found. In C++, a pointer of a certain class (such as `ErrorReporter`) can point to a value that is one of its child classes (such as `MicroErrorReporter`).

As we mentioned earlier, `MicroErrorReporter` overrides one of the methods of `ErrorReporter`. Without going into too much detail, the process of overriding this method has the side effect of obscuring some of its other methods.

To still have access to the non overridden methods of `ErrorReporter`, we need to treat our `MicroErrorReporter` instance as if it were actually an `ErrorReporter`. We achieve this by creating an `ErrorReporter` pointer and pointing it at the `micro_error_reporter` variable. The ampersand (`&`) in front of `micro_error_reporter` in the assignment means that we are assigning its pointer, not its value.

Phew! This sounds complicated. Don't panic if you found it difficult to follow; C++ can be a little unwieldy. For our purposes, all we need to know is that that we should use `error_reporter` to print debug information, and that it's a pointer.

Mapping Our Model

The reason we immediately set up a mechanism for printing debug information is so that we can log any problems that occur in the rest of the code. We rely on this in the next piece of code:

```
// Map the model into a usable data structure. This doesn't involve any
// copying or parsing, it's a very lightweight operation.
const tflite::Model* model = ::tflite::GetModel(g_sine_model_data);
if (model->version() != TFLITE_SCHEMA_VERSION) {
error_reporter->Report(
    "Model provided is schema version %d not equal "
    "to supported version %d.\n",
    model->version(), TFLITE_SCHEMA_VERSION);
    return 1;
}
```

In the first line, we take our model data array (defined in the file *sine_model_data.h*) and pass it into a method named `GetModel()`. This method returns a `Model` pointer, which is assigned to a variable named `model`. As you might have anticipated, this variable represents our model.

The type `Model` is a *struct*, which in C++ is very similar to class. It's defined in *schema_generated.h*, and it holds our model's data and allows us to query information about it.

Data Alignment

If you inspect our model's source file in *sine_model_data.cc*, you'll see that the definition of `g_sine_model_data` references a macro, `DATA_ALIGN_ATTRIBUTE`:

```
const unsigned char g_sine_model_data[] DATA_ALIGN_ATTRIBUTE = {
```

Processors can read data most efficiently when it is *aligned* in memory, meaning data structures are stored so that they don't overlap the boundaries of what the processor can read in a single operation. By specifying this macro we make sure that, when possible, our model data is correctly aligned for optimal read performance. If you're curious, you can read about alignment in the Wikipedia article.

As soon as `model` is ready, we call a method that retrieves the model's version number:

```
if (model->version() != TFLITE_SCHEMA_VERSION) {
```

We then compare the model's version number to `TFLITE_SCHEMA_VERSION`, which indicates the version of the TensorFlow Lite library we are currently using. If the numbers match, our model was converted with a compatible version of the TensorFlow Lite Converter. It's good practice to check the model version, because a mismatch might result in strange behavior that is tricky to debug.

In the preceding line of code, `version()` is a method that belongs to `model`. Notice the arrow (->) that points from `model` to `version()`. This is C++'s *arrow operator*, and it's used whenever we want to access the members of an object to which we have a pointer. If we had the object itself (and not just a pointer), we would use a dot (`.`) to access its members.

If the version numbers don't match, we'll carry on anyway, but we'll log a warning using our `error_reporter`:

```
error_reporter->Report(
    "Model provided is schema version %d not equal "
    "to supported version %d.\n",
    model->version(), TFLITE_SCHEMA_VERSION);
```

We call the `Report()` method of `error_reporter` to log this warning. Since `error_reporter` is also a pointer, we use the -> operator to access `Report()`.

The `Report()` method is designed to behave similarly to a commonly used C++ method, `printf()`, which is used to log text. As its first parameter, we pass in a string that we want to log. This string contains two `%d` format specifiers, which act as placeholders where variables will be inserted when the message is logged. The next two parameters we pass in are the model version and the TensorFlow Lite schema version. These will be inserted into the string, in order, to replace the `%d` characters.

The `Report()` method supports different format specifiers that work as placeholders for different types of variables. `%d` should be used as a placeholder for integers, `%f` should be used as a placeholder for floating-point numbers, and `%s` should be used as a placeholder for strings.

Creating an AllOpsResolver

So far so good! Our code can log errors, and we've loaded our model into a handy struct and checked that it is a compatible version. We've been moving a little slowly, given that we're reviewing some C++ concepts along the way, but things are starting to make sense.

Next up, we create an instance of `AllOpsResolver`:

```
// This pulls in all the operation implementations we need
tflite::ops::micro::AllOpsResolver resolver;
```

This class, defined in *all_ops_resolver.h*, is what allows the TensorFlow Lite for Microcontrollers interpreter to access *operations*.

In Chapter 3, you learned that a machine learning model is composed of various mathematical operations that are run successively to transform input into output. The `AllOpsResolver` class knows all of the operations that are available to TensorFlow Lite for Microcontrollers and is able to provide them to the interpreter.

Defining a Tensor Arena

We almost have all the ingredients ready to create an interpreter. The final thing we need to do is allocate an area of working memory that our model will need while it runs:

```
// Create an area of memory to use for input, output, and intermediate arrays.
// Finding the minimum value for your model may require some trial and error.
const int tensor_arena_size = 2 * 1024;
uint8_t tensor_arena[tensor_arena_size];
```

As the comment says, this area of memory will be used to store the model's input, output, and intermediate tensors. We call it our *tensor arena*. In our case, we've allocated an array that is 2,048 bytes in size. We specify this with the expression `2 * 1024`.

So, how large should our tensor arena be? That's a good question. Unfortunately, there's not a simple answer. Different model architectures have different sizes and numbers of input, output, and intermediate tensors, so it's difficult to know how much memory we'll need. The number doesn't need to be exact—we can reserve more memory than we need—but since microcontrollers have limited RAM, we should keep it as small as possible so there's space for the rest of our program.

We can do this through trial and error. That's why we express the array size as *n* `*` `1024`: so that it's easy to scale the number up and down (by changing *n*) while keeping it a multiple of eight. To find the correct array size, start fairly high so that you can be sure it works. The highest number used in this book's examples is `70 * 1024`. Then, reduce the number until your model no longer runs. The last number that worked is the correct one!

Creating an Interpreter

Now that we've declared `tensor_arena`, we're ready to set up the interpreter. Here's how that looks:

```
// Build an interpreter to run the model with
tflite::MicroInterpreter interpreter(model, resolver, tensor_arena,
```

```
                                        tensor_arena_size, error_reporter);

// Allocate memory from the tensor_arena for the model's tensors
interpreter.AllocateTensors();
```

First, we declare a `MicroInterpreter` named `interpreter`. This class is the heart of TensorFlow Lite for Microcontrollers: a magical piece of code that will execute our model on the data we provide. We pass in most of the objects we've created so far to its constructor, and then make a call to `AllocateTensors()`.

In the previous section, we set aside an area of memory by defining an array called `tensor_arena`. The `AllocateTensors()` method walks through all of the tensors defined by the model and assigns memory from the `tensor_arena` to each of them. It's critical that we call `AllocateTensors()` before attempting to run inference, because otherwise inference will fail.

Inspecting the Input Tensor

After we've created an interpreter, we need to provide some input for our model. To do this, we write our input data to the model's input tensor:

```
// Obtain a pointer to the model's input tensor
TfLiteTensor* input = interpreter.input(0);
```

To grab a pointer to an input tensor, we call the interpreter's `input()` method. Since a model can have multiple input tensors, we need to pass an index to the `input()` method that specifies which tensor we want. In this case, our model has only one input tensor, so its index is `0`.

In TensorFlow Lite, tensors are represented by the `TfLiteTensor` struct, which is defined in *c_api_internal.h*. This struct provides an API for interacting with and learning about tensors. In the next chunk of code, we use this functionality to verify that our tensor looks and feels correct. Because we'll be using tensors a lot, let's walk through this code to become familiar with how the `TfLiteTensor` struct works:

```
// Make sure the input has the properties we expect
TF_LITE_MICRO_EXPECT_NE(nullptr, input);
// The property "dims" tells us the tensor's shape. It has one element for
// each dimension. Our input is a 2D tensor containing 1 element, so "dims"
// should have size 2.
TF_LITE_MICRO_EXPECT_EQ(2, input->dims->size);
// The value of each element gives the length of the corresponding tensor.
// We should expect two single element tensors (one is contained within the
// other).
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[1]);
// The input is a 32 bit floating point value
TF_LITE_MICRO_EXPECT_EQ(kTfLiteFloat32, input->type);
```

The first thing you'll notice is a couple of macros: `TFLITE_MICRO_EXPECT_NE` and `TFLITE_MICRO_EXPECT_EQ`. These macros are part of the TensorFlow Lite for Microcontrollers testing framework, and they allow us to make *assertions* about the values of variables, proving that they have certain expected values.

For example, the macro `TF_LITE_MICRO_EXPECT_NE` is designed to assert that the two variables it is called with are not equal (hence the `_NE` part of its name, which stands for Not Equal). If the variables are not equal, the code will continue to execute. If they are equal, an error will be logged, and the test will be marked as having failed.

More Assertions

The macros for assertions are defined in *micro_test.h*, and you can read the file to see how they work. Here are the available assertions:

`TF_LITE_MICRO_EXPECT(x)`
: Asserts that *x* evaluates to `true`.

`TF_LITE_MICRO_EXPECT_EQ(x, y)`
: Asserts that *x* is equal to *y*.

`TF_LITE_MICRO_EXPECT_NE(x, y)`
: Asserts that *x* is not equal to *y*.

`TF_LITE_MICRO_EXPECT_NEAR(x, y, epsilon)`
: For numeric values, asserts that the difference between *x* and *y* is less than or equal to *epsilon*. For example, `TF_LITE_MICRO_EXPECT_NEAR(5, 7, 3)` would pass, because the difference between 5 and 7 is 2.

`TF_LITE_MICRO_EXPECT_GT(x, y)`
: For numeric values, asserts that *x* is greater than *y*.

`TF_LITE_MICRO_EXPECT_LT(x, y)`
: For numeric values, asserts that *x* is less than *y*.

`TF_LITE_MICRO_EXPECT_GE(x, y)`
: For numeric values, asserts that *x* greater than or equal to *y*.

`TF_LITE_MICRO_EXPECT_LE(x, y)`
: For numeric values, asserts that *x* is less than or equal to *y*.

The first thing we check is that our input tensor actually exists. To do this, we assert that it is *not equal* to a `nullptr`, which is a special C++ value representing a pointer that is not actually pointing at any data:

```
TF_LITE_MICRO_EXPECT_NE(nullptr, input);
```

The next thing we check is the *shape* of our input tensor. As discussed in Chapter 3, all tensors have a shape, which is a way of describing their dimensionality. The input to our model is a scalar value (meaning a single number). However, due to the way Keras layers accept input, this value must be provided inside of a 2D tensor containing one number. For an input of 0, it should look like this:

```
[[0]]
```

Note how the input scalar, 0, is wrapped inside of two vectors, making this a 2D tensor.

The `TfLiteTensor` struct contains a `dims` member that describes the dimensions of the tensor. The member is a struct of type `TfLiteIntArray`, also defined in *c_api_internal.h*. Its `size` member represents the number of dimensions that the tensor has. Since the input tensor should be 2D, we can assert that the value of `size` is `2`:

```
TF_LITE_MICRO_EXPECT_EQ(2, input->dims->size);
```

We can further inspect the `dims` struct to ensure the tensor's structure is what we expect. Its `data` variable is an array with one element for each dimension. Each element is an integer representing the size of that dimension. Because we are expecting a 2D tensor containing one element in each dimension, we can assert that both dimensions contain a single element:

```
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[1]);
```

We can now be confident that our input tensor has the correct shape. Finally, since tensors can consist of a variety of different types of data (think integers, floating-point numbers, and Boolean values), we should make sure that our input tensor has the correct type.

The tensor struct's `type` variable informs us of the data type of the tensor. We'll be providing a 32-bit floating-point number, represented by the constant `kTfLiteFloat32`, and we can easily assert that the type is correct:

```
TF_LITE_MICRO_EXPECT_EQ(kTfLiteFloat32, input->type);
```

Perfect—our input tensor is now guaranteed to be the correct size and shape for our input data, which will be a single floating-point value. We're ready to run inference!

Running Inference on an Input

To run inference, we need to add a value to our input tensor and then instruct the interpreter to invoke the model. Afterward, we will check whether the model successfully ran. Here's how that looks:

```
// Provide an input value
input->data.f[0] = 0.;
```

```
// Run the model on this input and check that it succeeds
TfLiteStatus invoke_status = interpreter.Invoke();
if (invoke_status != kTfLiteOk) {
 error_reporter->Report("Invoke failed\n");
}
TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, invoke_status);
```

TensorFlow Lite's `TfLiteTensor` struct has a `data` variable that we can use to set the contents of our input tensor. You can see this being used here:

```
input->data.f[0] = 0.;
```

The `data` variable is a `TfLitePtrUnion`—it's a *union*, which is a special C++ data type that allows you to store different data types at the same location in memory. Since a given tensor can contain one of many different types of data (for example, floating-point numbers, integers, or Booleans), a union is the perfect type to help us store it.

The `TfLitePtrUnion` union is declared in *c_api_internal.h*. Here's what it looks like:

```
// A union of pointers that points to memory for a given tensor.
typedef union {
  int32_t* i32;
  int64_t* i64;
  float* f;
  TfLiteFloat16* f16;
  char* raw;
  const char* raw_const;
  uint8_t* uint8;
  bool* b;
  int16_t* i16;
  TfLiteComplex64* c64;
  int8_t* int8;
} TfLitePtrUnion;
```

You can see that there are a bunch of members, each representing a certain type. Each member is a pointer, which can point at a place in memory where the data should be stored. When we call `interpreter.AllocateTensors()`, like we did earlier, the appropriate pointer is set to point at the block of memory that was allocated for the tensor to store its data. Because each tensor has a specific data type, only the pointer for the corresponding type will be set.

This means that to store data, we can use whichever is the appropriate pointer in our `TfLitePtrUnion`. For example, if our tensor is of type `kTfLiteFloat32`, we'll use `data.f`.

Since the pointer points at a block of memory, we can use square brackets (`[]`) after the pointer name to instruct our program where to store the data. In our example, we do the following:

```
input->data.f[0] = 0.;
```

The value we're assigning is written as `0.`, which is shorthand for `0.0`. By specifying the decimal point, we make it clear to the C++ compiler that this value should be a floating-point number, not an integer.

You can see that we assign this value to `data.f[0]`. This means that we're assigning it as the first item in our block of allocated memory. Given that there's only one value, this is all we need to do.

More Complex Inputs

In the example we're walking through, our model accepts a scalar input, so we have to assign only one value (`input->data.f[0] = 0.`). If our model's input was a vector consisting of several values, we would add them to subsequent memory locations.

Here's an example of a vector containing the numbers 1, 2, and 3:

```
[1 2 3]
```

And here's how we might set these values in a `TfLiteTensor`:

```
// Vector with 6 elements
input->data.f[0] = 1.;
input->data.f[1] = 2.;
input->data.f[2] = 3.;
```

But what about matrices, which consist of multiple vectors? Here's an example:

```
[[1 2 3]
 [4 5 6]]
```

To set this in a `TfLiteTensor`, we just assign the values in order, from left to right and top to bottom. This is called *flattening*, because we squash the structure from two to one dimension:

```
// Vector with 3 elements
input->data.f[0] = 1.;
input->data.f[1] = 2.;
input->data.f[2] = 3.;
input->data.f[3] = 4.;
input->data.f[4] = 5.;
input->data.f[5] = 6.;
```

Because the `TfLiteTensor` struct has a record of its actual dimensions, it knows which locations in memory correspond to which elements in its multidimensional shape, even though the memory has a flat structure. We make use of 2D input tensors in the later chapters to feed in images and other 2D data.

After we've set up the input tensor, it's time to run inference. This is a one-liner:

```
TfLiteStatus invoke_status = interpreter.Invoke();
```

When we call `Invoke()` on the `interpreter`, the TensorFlow Lite interpreter runs the model. The model consists of a graph of mathematical operations which the interpreter executes to transform the input data into an output. This output is stored in the model's output tensors, which we'll dig into later.

The `Invoke()` method returns a `TfLiteStatus` object, which lets us know whether inference was successful or there was a problem. Its value can either be `kTfLiteOk` or `kTfLiteError`. We check for an error and report it if there is one:

```
if (invoke_status != kTfLiteOk) {
    error_reporter->Report("Invoke failed\n");
}
```

Finally, we assert that the status must be `kTfLiteOk` in order for our test to pass:

```
TF_LITE_MICRO_EXPECT_EQ(kTfLiteOk, invoke_status);
```

That's it—inference has been run! Next up, we grab the output and make sure it looks good.

Reading the Output

Like the input, our model's output is accessed through a `TfLiteTensor`, and getting a pointer to it is just as simple:

```
TfLiteTensor* output = interpreter.output(0);
```

The output is, like the input, a floating-point scalar value nestled inside a 2D tensor. For the sake of our test, we double-check that the output tensor has the expected size, dimensions, and type:

```
TF_LITE_MICRO_EXPECT_EQ(2, output->dims->size);
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[1]);
TF_LITE_MICRO_EXPECT_EQ(kTfLiteFloat32, output->type);
```

Yep, it all looks good. Now, we grab the output value and inspect it to make sure that it meets our high standards. First we assign it to a `float` variable:

```
// Obtain the output value from the tensor
float value = output->data.f[0];
```

Each time inference is run, the output tensor will be overwritten with new values. This means that if you want to keep an output value around in your program while continuing to run inference, you'll need to copy it from the output tensor, like we just did.

Next, we use `TF_LITE_MICRO_EXPECT_NEAR` to prove that the value is close to the value we're expecting:

```
// Check that the output value is within 0.05 of the expected value
TF_LITE_MICRO_EXPECT_NEAR(0., value, 0.05);
```

As we saw earlier, `TF_LITE_MICRO_EXPECT_NEAR` asserts that the difference between its first argument and its second argument is less than the value of its third argument. In this statement, we're testing that the output is within 0.05 of 0, which is the mathematical sine of the input, 0.

There are two reasons why we expect a number that is *near* to what we want, but not an exact value. The first is that our model only *approximates* the real sine value, so we know that it will not be exactly correct. The second is because floating-point calculations on computers have a margin of error. The error can vary from computer to computer: for example, a laptop's CPU might come up with slightly different results to an Arduino. By having flexible expectations, we make it more likely that our test will pass on any platform.

If this test passes, things are looking good. The remaining tests run inference a few more times, just to further prove that our model is working. To run inference again, all we need to do is assign a new value to our input tensor, call `interpreter.Invoke()`, and read the output from our output tensor:

```
// Run inference on several more values and confirm the expected outputs
input->data.f[0] = 1.;
interpreter.Invoke();
value = output->data.f[0];
TF_LITE_MICRO_EXPECT_NEAR(0.841, value, 0.05);

input->data.f[0] = 3.;
interpreter.Invoke();
value = output->data.f[0];
TF_LITE_MICRO_EXPECT_NEAR(0.141, value, 0.05);

input->data.f[0] = 5.;
interpreter.Invoke();
value = output->data.f[0];
TF_LITE_MICRO_EXPECT_NEAR(-0.959, value, 0.05);
```

Note how we're reusing the same `input` and `output` tensor pointer. Because we already have the pointers, we don't need to call `interpreter.input(0)` or `interpreter.output(0)` again.

At this point in our tests we've proven that TensorFlow Lite for Microcontrollers can successfully load our model, allocate the appropriate input and output tensors, run inference, and return the expected results. The final thing to do is indicate the end of the tests by using a macro:

```
}

TF_LITE_MICRO_TESTS_END
```

And with that, we're done walking through the tests. Next, let's run them!

Running the Tests

Even though this code is eventually destined to run on microcontrollers, we can still build and run our tests on our development machine. This makes it much easier to write and debug code. Compared with microcontrollers, a personal computer has far more convenient tools for logging output and stepping through code, which makes it a lot simpler to figure out any bugs. In addition, deploying code to a device takes time, so it's a lot quicker to just run our code locally.

A good workflow for building embedded applications (or, honestly, any kind of software) is to write as much of the logic as you can in tests that can be run on a normal development machine. There'll always be some parts that require the actual hardware to run, but the more you can test locally, the easier your life will be.

Practically, this means that we should try to write the code that preprocesses inputs, runs inference with the model, and processes any outputs in a set of tests before trying to get it working on-device. In Chapter 7, we walk through a speech recognition application that is much more complex than this example. You'll see how we've written detailed unit tests for each of its components.

Grabbing the code

Until now, between Colab and GitHub, we've been doing everything in the cloud. To run our tests, we need to pull down the code to our development computer and compile it.

To do all this, we need the following software tools:

- A terminal emulator, such as Terminal in macOS
- A bash shell (the default in macOS prior to Catalina and most Linux distributions)
- Git (installed by default in macOS and most Linux distributions)
- Make, version 3.82 or later

Git and Make

Git and Make are often preinstalled on modern operating systems. To check whether they are installed on your system, open a terminal and do the following:

For Git

Any version of Git will work. To confirm Git is installed, enter `git` at the command line. You should see usage instructions being printed.

For Make

To check the version of Make installed, enter `make --version` at the command line. You need a version greater than 3.82.

If you are missing either tool, you should search the web for instructions on installing them for your specific operating system.

After you have all the tools, open up a terminal and enter the command that follows to download the TensorFlow source code, which includes the example code we're working with. It will create a directory containing the source code in whatever location you run it:

```
git clone https://github.com/tensorflow/tensorflow.git
```

Next, change into the *tensorflow* directory that was just created:

```
cd tensorflow
```

Great stuff—we're now ready to run some code!

Using Make to run the tests

As you saw from our list of tools, we use a program called *Make* to run the tests. Make is a tool for automating build tasks in software. It's been in use since 1976, which in computing terms is almost forever. Developers use a special language, written in files called *Makefiles*, to instruct Make how to build and run code. TensorFlow Lite for Microcontrollers has a Makefile defined in *micro/tools/make/Makefile*; there's more information about it in Chapter 13.

To run our tests using Make, we can issue the following command, making sure we're running it from the root of the *tensorflow* directory we downloaded with Git. We first specify the Makefile to use, followed by the *target*, which is the component that we want to build:

```
make -f tensorflow/lite/micro/tools/make/Makefile test_hello_world_test
```

The Makefile is set up so that in order to run tests, we provide a target with the prefix `test_` followed by the name of the component that we want to build. In our case, that component is *`hello_world_test`*, so the full target name is *`test_hello_world_test`*.

Try running this command. You should start to see a ton of output fly past! First, some necessary libraries and tools will be downloaded. Next, our test file, along with all of its dependencies, will be built. Our Makefile has instructed the C++ compiler to build the code and create a binary, which it will then run.

You'll need to wait a few moments for the process to complete. When the text stops zooming past, the last few lines should look like this:

```
Testing LoadModelAndPerformInference
1/1 tests passed
~~~ALL TESTS PASSED~~~
```

Nice! This output shows that our test passed as expected. You can see the name of the test, `LoadModelAndPerformInference`, as defined at the top of its source file. Even if it's not on a microcontroller yet, our code is successfully running inference.

To see what happens when tests fail, let's introduce an error. Open up the test file, *hello_world_test.cc*. It will be at this path, relative to the root of the directory:

```
tensorflow/lite/micro/examples/hello_world/hello_world_test.cc
```

To make the test fail, let's provide a different input to the model. This will cause the model's output to change, so the assertion that checks the value of our output will fail. Find the following line:

```
input->data.f[0] = 0.;
```

Change the assigned value, like so:

```
input->data.f[0] = 1.;
```

Now save the file, and use the following command to run the test again (remember to do this from the root of the *tensorflow* directory):

```
make -f tensorflow/lite/micro/tools/make/Makefile test_hello_world_test
```

The code will be rebuilt, and the test will run. The final output you see should look like this:

```
Testing LoadModelAndPerformInference
0.0486171 near value failed at tensorflow/lite/micro/examples/hello_world/\
  hello_world_test.cc:94
0/1 tests passed
~~~SOME TESTS FAILED~~~
```

The output contains some useful information about why the test failed, including the file and line number where the failure took place (`hello_world_test.cc:94`). If this were caused by a real bug, this output would be helpful in tracking down the issue.

Project File Structure

With the help of our test, you've learned how to use the TensorFlow Lite for Microcontrollers library to run inference in C++. Next, we're going to walk through the source code of an actual application.

As discussed earlier, the program we're building consists of a continuous loop that feeds an `x` value into the model, runs inference, and uses the result to produce some sort of visible output (like a pattern of flashing LEDs), depending on the platform.

Because the application is complex and spans multiple files, let's take a look at its structure and how it all fits together.

The root of the application is in *tensorflow/lite/micro/examples/hello_world*. It contains the following files:

BUILD
: A file that lists the various things that can be built using the application's source code, including the main application binary and the tests we walked through earlier. We don't need to worry too much about it at this point.

Makefile.inc
: A Makefile that contains information about the build targets within our application, including *hello_world_test*, which is the test we ran earlier, and *hello_world*, the main application binary. It defines which source files are part of them.

README.md
: A readme file containing instructions on building and running the application.

constants.h, constants.cc
: A pair of files containing various *constants* (variables that don't change during the lifetime of a program) that are important for defining the program's behavior.

create_sine_model.ipynb
: The Jupyter notebook used in the previous chapter.

hello_world_test.cc
: A test that runs inference using our model.

main.cc
: The entry point of the program, which runs first when the application is deployed to a device.

main_functions.h, main_functions.cc
: A pair of files that define a `setup()` function, which performs all the initialization required by our program, and a `loop()` function, which contains the program's core logic and is designed to be called repeatedly in a loop. These functions are called by *main.cc* when the program starts.

output_handler.h, output_handler.cc
: A pair of files that define a function we can use to display an output each time inference is run. The default implementation, in *output_handler.cc*, prints the result to the screen. We can override this implementation so that it does different things on different devices.

output_handler_test.cc
: A test that proves that the code in *output_handler.h* and *output_handler.cc* is working correctly.

sine_model_data.h, sine_model_data.cc
: A pair of files that define an array of data representing our model, as exported using `xxd` in the first part of this chapter.

In addition to these files, the directory contains the following subdirectories (and perhaps more):

- *arduino/*
- *disco_f76ng/*
- *sparkfun_edge/*

Because different microcontroller platforms have different capabilities and APIs, our project structure allows us to provide device-specific versions of source files that will be used instead of the defaults if the application is built for that device. For example, the *arduino* directory contains custom versions of *main.cc*, *constants.cc*, and *output_handler.cc* that tailor the application to work with Arduino. We dig into these custom implementations later.

Walking Through the Source

Now that we know how the application's source is structured, let's dig into the code. We'll begin with *main_functions.cc*, where most of the magic happens, and branch out into the other files from there.

A lot of this code will look very familiar from our earlier adventures in *hello_world_test.cc*. If we've covered something already, we won't go into depth on how it works; we'd rather focus mainly on the things you haven't seen before.

Starting with main_functions.cc

This file contains the core logic of our program. It begins like this, with some familiar `#include` statements and some new ones:

```
#include "tensorflow/lite/micro/examples/hello_world/main_functions.h"
#include "tensorflow/lite/micro/examples/hello_world/constants.h"
#include "tensorflow/lite/micro/examples/hello_world/output_handler.h"
#include "tensorflow/lite/micro/examples/hello_world/sine_model_data.h"
#include "tensorflow/lite/micro/kernels/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
```

```
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
```

We saw a lot of these in *hello_world_test.cc*. New to the scene are *constants.h* and *output_handler.h*, which we learned about in the list of files earlier.

The next part of the file sets up the global variables that will be used within *main_functions.cc*:

```
namespace {
tflite::ErrorReporter* error_reporter = nullptr;
const tflite::Model* model = nullptr;
tflite::MicroInterpreter* interpreter = nullptr;
TfLiteTensor* input = nullptr;
TfLiteTensor* output = nullptr;
int inference_count = 0;

// Create an area of memory to use for input, output, and intermediate arrays.
// Finding the minimum value for your model may require some trial and error.
constexpr int kTensorArenaSize = 2 * 1024;
uint8_t tensor_arena[kTensorArenaSize];
}  // namespace
```

You'll notice that these variables are wrapped in a `namespace`. This means that even though they will be accessible from anywhere within *main_functions.cc*, they won't be accessible from any other files within the project. This helps prevent problems if two different files happen to define variables with the same name.

All of these variables should look familiar from the tests. We set up variables to hold all of our familiar TensorFlow objects, along with a `tensor_arena`. The only new thing is an `int` that holds `inference_count`, which will keep track of how many inferences our program has performed.

The next part of the file declares a function named `setup()`. This function will be called when the program first starts, but never again after that. We use it to do all of the one-time housekeeping work that needs to happen before we start running inference.

The first part of `setup()` is almost exactly the same as in our tests. We set up logging, load our model, set up the interpreter, and allocate memory:

```
void setup() {
  // Set up logging.
  static tflite::MicroErrorReporter micro_error_reporter;
  error_reporter = &micro_error_reporter;

  // Map the model into a usable data structure. This doesn't involve any
  // copying or parsing, it's a very lightweight operation.
  model = tflite::GetModel(g_sine_model_data);
  if (model->version() != TFLITE_SCHEMA_VERSION) {
    error_reporter->Report(
```

```
        "Model provided is schema version %d not equal "
        "to supported version %d.",
        model->version(), TFLITE_SCHEMA_VERSION);
    return;
  }

  // This pulls in all the operation implementations we need.
  static tflite::ops::micro::AllOpsResolver resolver;

  // Build an interpreter to run the model with.
  static tflite::MicroInterpreter static_interpreter(
      model, resolver, tensor_arena, kTensorArenaSize, error_reporter);
  interpreter = &static_interpreter;

  // Allocate memory from the tensor_arena for the model's tensors.
  TfLiteStatus allocate_status = interpreter->AllocateTensors();
  if (allocate_status != kTfLiteOk) {
    error_reporter->Report("AllocateTensors() failed");
    return;
  }
```

Familiar territory so far. After this point, though, things get a little different. First, we grab pointers to both the input *and* output tensors:

```
  // Obtain pointers to the model's input and output tensors.
  input = interpreter->input(0);
  output = interpreter->output(0);
```

You might be wondering how we can interact with the output before inference has been run. Well, remember that `TfLiteTensor` is just a struct that has a member, `data`, pointing to an area of memory that has been allocated to store the output. Even though no output has been written yet, the struct and its `data` member still exist.

Finally, to end the `setup()` function, we set our `inference_count` variable to `0`:

```
  // Keep track of how many inferences we have performed.
  inference_count = 0;
}
```

At this point, all of our machine learning infrastructure is set up and ready to go. We have all the tools required to run inference and get the results. The next thing to define is our application logic. What is the program actually going to *do*?

Our model was trained to predict the sine of any number from 0 to 2π, which represents the full cycle of a sine wave. To demonstrate our model, we could just feed in numbers in this range, predict their sines, and then output the values somehow. We could do this in a sequence so that we show the model working across the entire range. This sounds like a good plan!

To do this, we need to write some code that runs in a loop. First, we declare a function called `loop()`, which is what we'll be walking through next. The code we place in this function will be run repeatedly, over and over again:

```
void loop() {
```

First in our `loop()` function, we must determine what value to pass into the model (let's call it our `x` value). We determine this using two constants: `kXrange`, which specifies the maximum possible `x` value as 2π, and `kInferencesPerCycle`, which defines the number of inferences that we want to perform as we step from 0 to 2π. The next few lines of code calculate the `x` value:

```
// Calculate an x value to feed into the model. We compare the current
// inference_count to the number of inferences per cycle to determine
// our position within the range of possible x values the model was
// trained on, and use this to calculate a value.
float position = static_cast<float>(inference_count) /
                 static_cast<float>(kInferencesPerCycle);
float x_val = position * kXrange;
```

The first two lines of code just divide `inference_count` (which is the number of inferences we've done so far) by `kInferencesPerCycle` to obtain our current "position" within the range. The next line multiplies that value by `kXrange`, which represents the maximum value in the range (2π). The result, `x_val`, is the value we'll be passing into our model.

`static_cast<float>()` is used to convert `inference_count` and `kInferencesPerCycle`, which are both integer values, into floating-point numbers. We do this so that we can correctly perform division. In C++, if you divide two integers, the result is an integer; any fractional part of the result is dropped. Because we want our `x` value to be a floating-point number that includes the fractional part, we need to convert the numbers being divided into floating points.

The two constants we use, `kInferencesPerCycle` and `kXrange`, are defined in the files *constants.h* and *constants.cc*. It's a C++ convention to prefix the names of constants with a `k`, so they're easily identifiable as constants when using them in code. It can be useful to define constants in a separate file so they can be included and used in any place that they are needed.

The next part of our code should look nice and familiar; we write our `x` value to the model's input tensor, run inference, and then grab the result (let's call it our `y` value) from the output tensor:

```
// Place our calculated x value in the model's input tensor
input->data.f[0] = x_val;
```

```
// Run inference, and report any error
TfLiteStatus invoke_status = interpreter->Invoke();
if (invoke_status != kTfLiteOk) {
  error_reporter->Report("Invoke failed on x_val: %f\n",
                         static_cast<double>(x_val));
  return;
}

// Read the predicted y value from the model's output tensor
float y_val = output->data.f[0];
```

We now have a sine value. Since it takes a small amount of time to run inference on each number, and this code is running in a loop, we'll be generating a sequence of sine values over time. This will be perfect for controlling some blinking LEDs or an animation. Our next job is to output it somehow.

The following line calls the `HandleOutput()` function, defined in *output_handler.cc*:

```
// Output the results. A custom HandleOutput function can be implemented
// for each supported hardware target.
HandleOutput(error_reporter, x_val, y_val);
```

We pass in our `x` and `y` values, along with our `ErrorReporter` instance, which we can use to log things. To see what happens next, let's explore *output_handler.cc*.

Handling Output with output_handler.cc

The file *output_handler.cc* defines our `HandleOutput()` function. Its implementation is very simple:

```
void HandleOutput(tflite::ErrorReporter* error_reporter, float x_value,
                  float y_value) {
  // Log the current X and Y values
  error_reporter->Report("x_value: %f, y_value: %f\n", x_value, y_value);
}
```

All this function does is use the `ErrorReporter` instance to log the `x` and `y` values. This is just a bare-minimum implementation that we can use to test the basic functionality of our application, for example by running it on our development computer.

Our goal, though, is to deploy this application to several different microcontroller platforms, using each platform's specialized hardware to display the output. For each individual platform we're planning to deploy to, such as Arduino, we provide a custom replacement for *output_handler.cc* that uses the platform's APIs to control output —for example, by lighting some LEDs.

As mentioned earlier, these replacement files are located in subdirectories with the name of each platform: *arduino/*, *disco_f76ng/*, and *sparkfun_edge/*. We'll dive into the

platform-specific implementations later. For now, let's jump back into *main_functions.cc*.

Wrapping Up main_functions.cc

The last thing we do in our `loop()` function is increment our `inference_count` counter. If it has reached the maximum number of inferences per cycle defined in `kInferencesPerCycle`, we reset it to 0:

```
// Increment the inference_counter, and reset it if we have reached
// the total number per cycle
inference_count += 1;
if (inference_count      >= kInferencesPerCycle) inference_count = 0;
```

The next time our loop iterates, this will have the effect of either moving our `x` value along by a step or wrapping it around back to 0 if it has reached the end of the range.

We've now reached the end of our `loop()` function. Each time it runs, a new `x` value is calculated, inference is run, and the result is output by `HandleOutput()`. If `loop()` is continually called, it will run inference for a progression of `x` values in the range 0 to 2π and then repeat.

But what is it that makes the `loop()` function run over and over again? The answer lies in the file *main.cc*.

Understanding main.cc

The C++ standard specifies that every C++ program contain a global function named `main()`, which will be run when the program starts. In our program, this function is defined in the file *main.cc*. The existence of this `main()` function is the reason *main.cc* represents the entry point of our program. The code in `main()` will be run any time the microcontroller starts up.

The file *main.cc* is very short and sweet. First, it contains an `#include` statement for *main_functions.h*, which will bring in the `setup()` and `loop()` functions defined there:

```
#include "tensorflow/lite/micro/examples/hello_world/main_functions.h"
```

Next, it declares the `main()` function itself:

```
int main(int argc, char* argv[]) {
  setup();
  while (true) {
    loop();
  }
}
```

When `main()` runs, it first calls our `setup()` function. It will do this only once. After that, it enters a `while` loop that will continually call the `loop()` function, over and over again.

This loop will keep running indefinitely. Yikes! If you're from a server or web programming background, this might not sound like a great idea. The loop will block our single thread of execution, and there's no way to exit the program.

However, when writing software for microcontrollers, this type of endless loop is actually pretty common. Because there's no multitasking, and only one application will ever run, it doesn't really matter that the loop goes on and on. We just continue making inferences and outputting data for as long as the microcontroller is connected to power.

We've now walked through our entire microcontroller application. In the next section, we'll try out the application code by running it on our development machine.

Running Our Application

To give our application code a test run, we first need to build it. Enter the following Make command to create an executable binary for our program:

```
make -f tensorflow/lite/micro/tools/make/Makefile hello_world
```

When the build completes, you can run the application binary by using the following command, depending on your operating system:

```
# macOS:
tensorflow/lite/micro/tools/make/gen/osx_x86_64/bin/hello_world

# Linux:
tensorflow/lite/micro/tools/make/gen/linux_x86_64/bin/hello_world

# Windows
tensorflow/lite/micro/tools/make/gen/windows_x86_64/bin/hello_world
```

If you can't find the correct path, list the directories in *tensorflow/lite/micro/tools/make/gen/*.

After you run the binary, you should hopefully see a bunch of output scrolling past, looking something like this:

```
x_value: 1.4137159*2^1, y_value: 1.374213*2^-2

x_value: 1.5707957*2^1, y_value: -1.4249528*2^-5

x_value: 1.7278753*2^1, y_value: -1.4295994*2^-2

x_value: 1.8849551*2^1, y_value: -1.2867725*2^-1

x_value: 1.210171*2^2, y_value: -1.7542461*2^-1
```

Very exciting! These are the logs written by the `HandleOutput()` function in *output_handler.cc*. There's one log per inference, and the `x_value` gradually increments until it reaches 2π, at which point it goes back to 0 and starts again.

As soon as you've had enough excitement, you can press Ctrl-C to terminate the program.

You'll notice that the numbers are output as values with power-of-two exponents, like `1.4137159*2^1`. This is an efficient way to log floating-point numbers on microcontrollers, which often don't have hardware support for floating-point operations.

To get the original value, just pull out your calculator: for example, `1.4137159*2^1` evaluates to `2.8274318`. If you're curious, the code that prints these numbers is in *debug_log_numbers.cc*.

Wrapping Up

We've now confirmed the program works on our development machine. In the next chapter, we'll get it running on some microcontrollers!

CHAPTER 6

The "Hello World" of TinyML: Deploying to Microcontrollers

Now it's time to get our hands dirty. Over the course of this chapter, we will deploy the code to three different devices:

- Arduino Nano 33 BLE Sense
- SparkFun Edge
- ST Microelectronics STM32F746G Discovery kit

We'll walk through the build and deployment process for each one.

TensorFlow Lite regularly adds support for new devices, so if the device you'd like to use isn't listed here, it's worth checking the example's *README.md*.

You can also check there for updated deployment instructions if you run into trouble following these steps.

Every device has its own unique output capabilities, ranging from a bank of LEDs to a full LCD display, so the example contains a custom implementation of `HandleOutput()` for each one. We'll also walk through each of these and talk about how its logic works. Even if you don't have all of the devices, reading through this code should be interesting, so we strongly recommend taking a look.

What Exactly Is a Microcontroller?

Depending on your past experience, you might not be familiar with how microcontrollers interact with other electronic components. Because we're about to start playing with hardware, it's worth introducing some ideas before we move along.

On a microcontroller board like the Arduino, SparkFun Edge, or STM32F746G Discovery kit, the actual microcontroller is just one of many electronic components attached to the circuit board. Figure 6-1 shows the microcontroller on the SparkFun Edge.

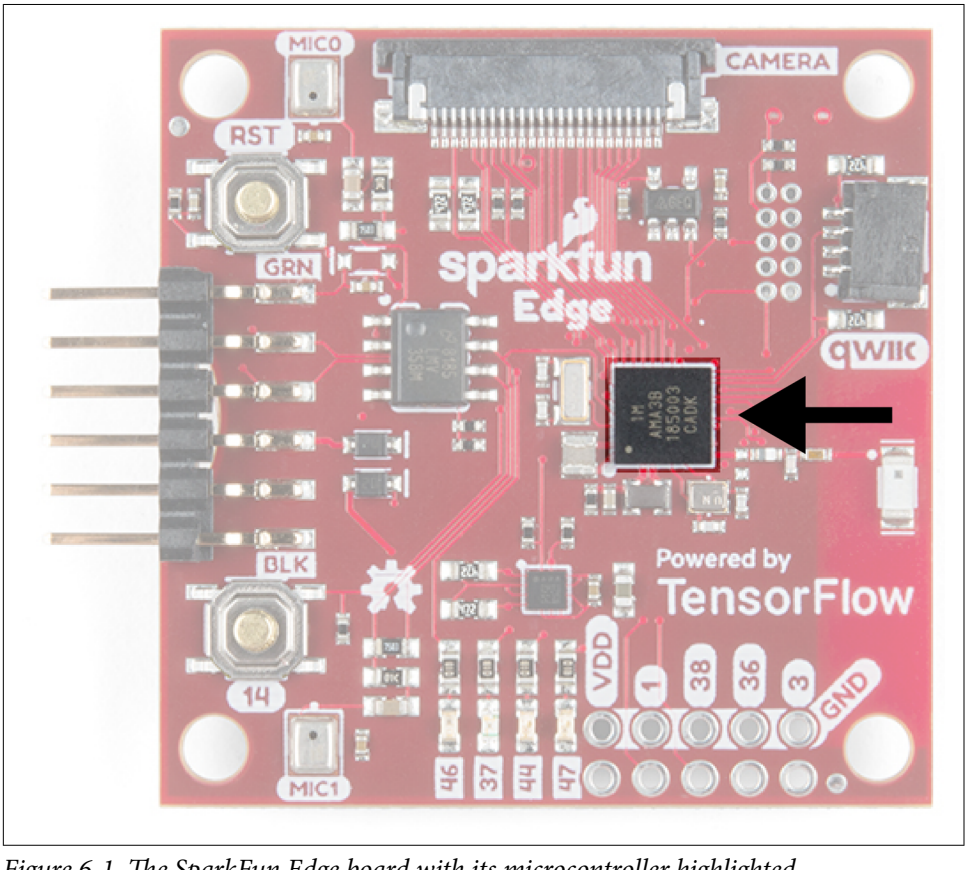


Figure 6-1. The SparkFun Edge board with its microcontroller highlighted

The microcontroller is connected to the circuit board it lives on using *pins*. A typical microcontroller has dozens of pins, and they serve all sorts of purposes. Some provide power to the microcontroller; others connect it to various important components.

Some pins are set aside for the input and output of digital signals by programs running on the microcontroller. These are called *GPIO* pins, which stands for general-purpose input/output. They can act as inputs, determining whether a voltage is being applied to them, or outputs, sourcing current that can power or communicate with other components.

GPIO pins are digital. This means that in output mode, they are like switches that can either be fully on, or fully off. In input mode, they can detect whether the voltage applied to them by another component is either above or below a certain threshold.

In addition to GPIOs, some microcontrollers have analog input pins, which can measure the exact level of voltage that is being applied to them.

By calling special functions, the program running on a microcontroller can control whether a given pin is in input or output mode. Other functions are used to switch an output pin on or off, or to read the current state of an input pin.

Now that you know a bit more about microcontrollers, let's take a closer look at our first device: the Arduino.

Arduino

There are a huge variety of Arduino boards, all with different capabilities. Not all of them will run TensorFlow Lite for Microcontrollers. The board we recommend for this book is the Arduino Nano 33 BLE Sense. In addition to being compatible with TensorFlow Lite, it also includes a microphone and an accelerometer (which we use in later chapters). We recommend buying the version of the board with headers, which makes it easier to connect other components without soldering.

Most Arduino boards come with a built-in LED, and this is what we'll be using to visually output our sine values. Figure 6-2 shows an Arduino Nano 33 BLE Sense board with the LED highlighted.

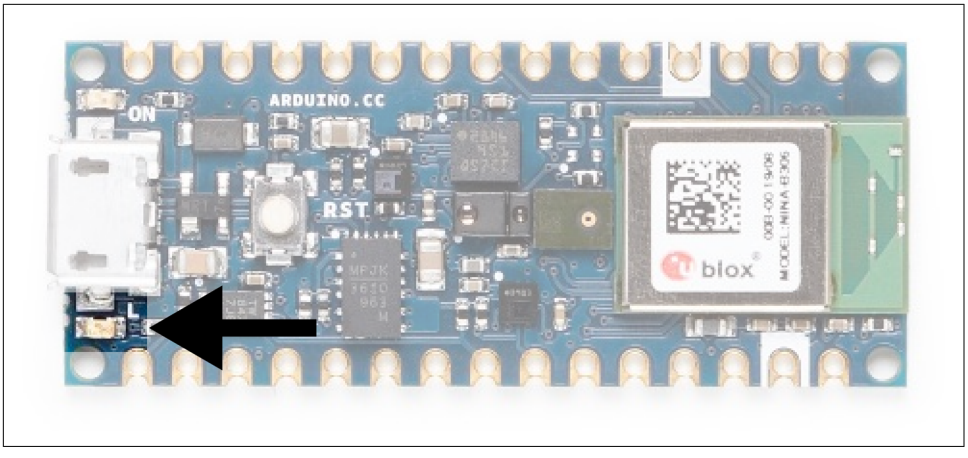


Figure 6-2. The Arduino Nano 33 BLE Sense board with the LED highlighted

Handling Output on Arduino

Because we have only one LED to work with, we need to think creatively. One option is to vary the brightness of the LED based on the most recently predicted sine value. Given that the value ranges from –1 to 1, we could represent 0 with an LED that is fully off, –1 and 1 with a fully lit LED, and any intermediate values with a partially dimmed LED. As the program runs inferences in a loop, the LED will fade repeatedly on and off.

We can vary the number of inferences we perform across a full sine wave cycle using the `kInferencesPerCycle` constant. Because one inference takes a set amount of time, tweaking `kInferencesPerCycle`, defined in *constants.cc*, will adjust how fast the LED fades.

There's an Arduino-specific version of this file in *hello_world/arduino/constants.cc*. The file has been given the same name as *hello_world/constants.cc*, so it will be used instead of the original implementation when the application is built for Arduino.

To dim our built-in LED, we can use a technique called *pulse width modulation* (PWM). If we switch an output pin on and off extremely rapidly, the pin's output voltage becomes a factor of the ratio between time spent in the off and on states. If the pin spends 50% of the time in each state, its output voltage will be 50% of its maximum. If it spends 75% in the on state and 25% in the off state, its voltage will be 75% of its maximum.

PWM is only available on certain pins of certain Arduino devices, but it's very easy to use: we just call a function that sets our desired output level for the pin.

The code that implements output handling for Arduino is in *hello_world/arduino/output_handler.cc*, which is used instead of the original file, *hello_world/output_handler.cc*.

Let's walk through the source:

```
#include "tensorflow/lite/micro/examples/hello_world/output_handler.h"
#include "Arduino.h"
#include "tensorflow/lite/micro/examples/hello_world/constants.h"
```

First, we include some header files. Our *output_handler.h* specifies the interface for this file. *Arduino.h* provides the interface for the Arduino platform; we use this to control the board. Because we need access to `kInferencesPerCycle`, we also include *constants.h*.

Next, we define the function and instruct it what to do the first time it runs:

```
// Adjusts brightness of an LED to represent the current y value
void HandleOutput(tflite::ErrorReporter* error_reporter, float x_value,
                  float y_value) {
// Track whether the function has run at least once
static bool is_initialized = false;

// Do this only once
if (!is_initialized) {
  // Set the LED pin to output
  pinMode(LED_BUILTIN, OUTPUT);
  is_initialized = true;
}
```

In C++, a variable declared as `static` within a function will hold its value across multiple runs of the function. Here, we use the `is_initialized` variable to track whether the code in the following `if (!is_initialized)` block has ever been run before.

The initialization block calls Arduino's `pinMode()` function, which indicates to the microcontroller whether a given pin should be in input or output mode. This is necessary before using a pin. The function is called with two constants defined by the Arduino platform: `LED_BUILTIN` and `OUTPUT`. `LED_BUILTIN` represents the pin connected to the board's built-in LED, and `OUTPUT` represents output mode.

After configuring the built-in LED's pin to output mode, set `is_initialized` to `true` so that this block code will not run again.

Next up, we calculate the desired brightness of the LED:

```
// Calculate the brightness of the LED such that y=-1 is fully off
// and y=1 is fully on. The LED's brightness can range from 0-255.
int brightness = (int)(127.5f * (y_value + 1));
```

The Arduino allows us to set the level of a PWM output as a number from 0 to 255, where 0 means fully off and 255 means fully on. Our `y_value` is a number between –1

and 1. The preceding code maps `y_value` to the range 0 to 255 so that when `y = -1` the LED is fully off, when `y = 0` the LED is half lit, and when `y = 1` the LED is fully lit.

The next step is to actually set the LED's brightness:

```
// Set the brightness of the LED. If the specified pin does not support PWM,
// this will result in the LED being on when y > 127, off otherwise.
analogWrite(LED_BUILTIN, brightness);
```

The Arduino platform's `analogWrite()` function takes a pin number (we provide `LED_BUILTIN`) and a value between 0 and 255. We provide our `brightness`, calculated in the previous line. When this function is called, the LED will be lit at that level.

Unfortunately, on some models of Arduino boards, the pin that the built-in LED is connected to is not capable of PWM. This means our calls to `analogWrite()` won't vary its brightness. Instead, the LED will be switched on if the value passed into `analogWrite()` is above 127, and switched off if it is 126 or below.

This means the LED will flash on and off instead of fading. Not quite as cool, but it still demonstrates our sine wave prediction.

Finally, we use the `ErrorReporter` instance to log the brightness value:

```
// Log the current brightness value for display in the Arduino plotter
error_reporter->Report("%d\n", brightness);
```

On the Arduino platform, the `ErrorReporter` is set up to log data via a serial port. Serial is a very common way for microcontrollers to communicate with host computers, and it's often used for debugging. It's a communication protocol in which data is communicated one bit at a time by switching an output pin on and off. We can use it to send and receive anything, from raw binary data to text and numbers.

The Arduino IDE contains tools for capturing and displaying data received through a serial port. One of the tools, the Serial Plotter, can display a graph of values it receives via serial. By outputting a stream of brightness values from our code, we'll be able to see them graphed. Figure 6-3 shows this in action.

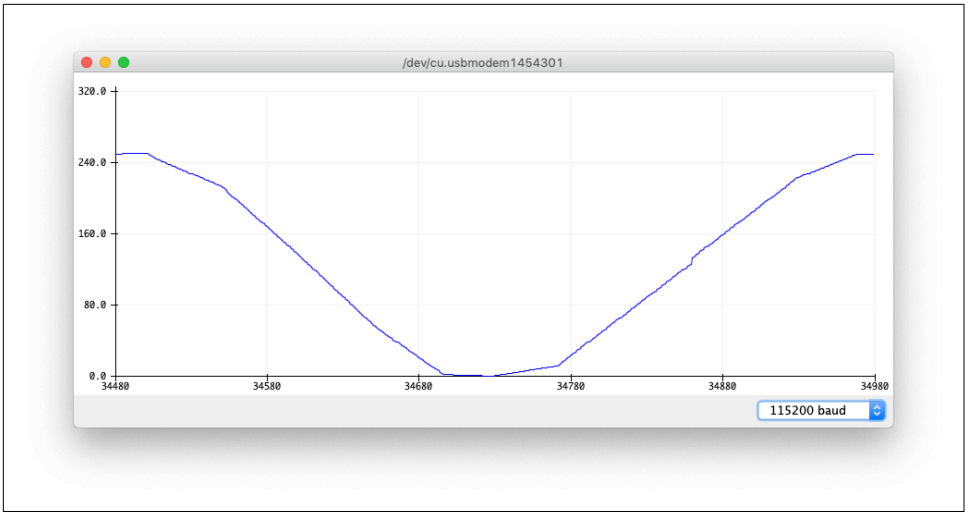


Figure 6-3. The Arduino IDE's Serial Plotter

We provide instructions on how to use the Serial Plotter later in this section.

You might be wondering how the `ErrorReporter` is able to output data via Arduino's serial interface. You can find the code implementation in *micro/arduino/debug_log.cc*. It replaces the original implementation at *micro/debug_log.cc*.

Just like how *output_handler.cc* is overwritten, we can provide platform-specific implementations of any source file in TensorFlow Lite for Microcontrollers by adding them to a directory with the platform's name.

Running the Example

Our next task is to build the project for Arduino and deploy it to a device.

There's always a chance that the build process might have changed since this book was written, so check *README.md* for the latest instructions.

Here's everything that we'll need:

- A supported Arduino board (we recommend the Arduino Nano 33 BLE Sense)
- The appropriate USB cable

- The Arduino IDE (you'll need to download and install this before continuing)

The projects in this book are available as example code in the TensorFlow Lite Arduino library, which you can easily install via the Arduino IDE and select Manage Libraries from the Tools menu. In the window that appears, search for and install the library named *Arduino_TensorFlowLite*. You should be able to use the latest version, but if you run into issues, the version that was tested with this book is `1.14-ALPHA`.

You can also install the library from a *.zip* file, which you can either download from the TensorFlow Lite team or generate yourself using the TensorFlow Lite for Microcontrollers Makefile. If you'd prefer to do this, see Appendix A.

After you've installed the library, the `hello_world` example will show up in the File menu under Examples→_Arduino_TensorFlowLite_, as shown in Figure 6-4.

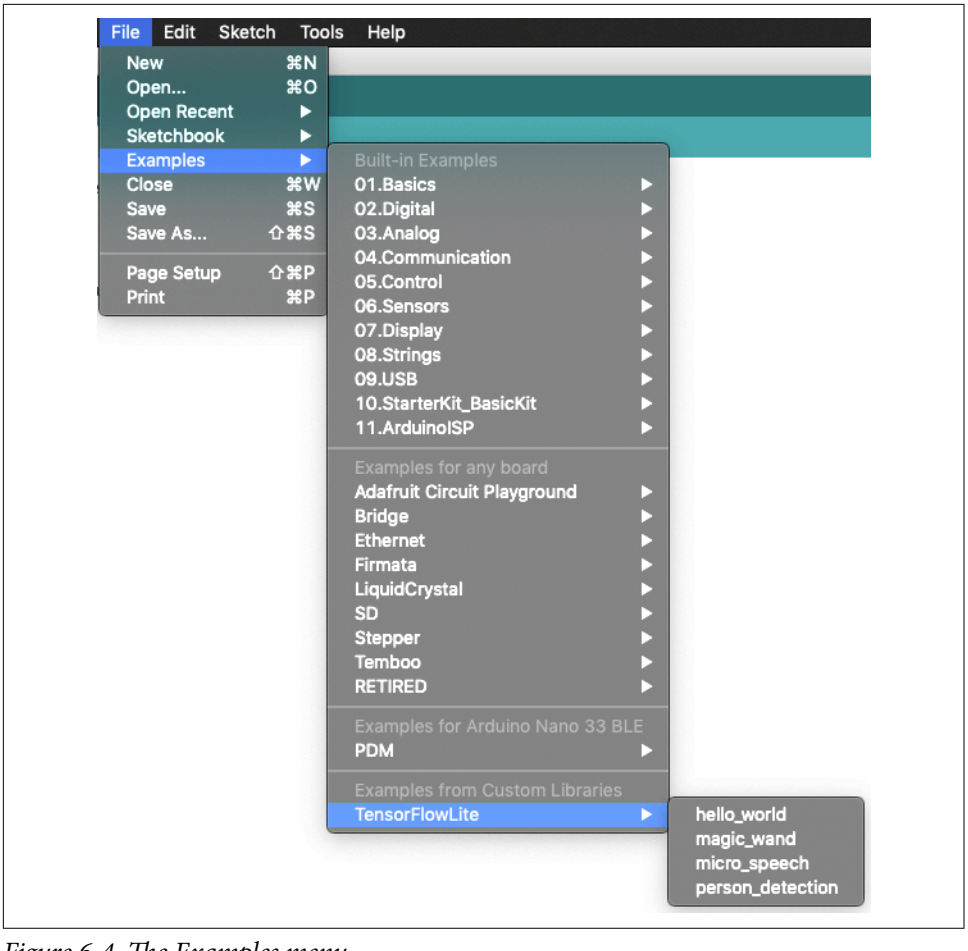


Figure 6-4. The Examples menu

Click "hello_world" to load the example. It will appear as a new window, with a tab for each of the source files. The file in the first tab, *hello_world*, is equivalent to the *main_functions.cc* we walked through earlier.

Differences in the Arduino Example Code

When the Arduino library is generated, some minor changes are made to the code so that it works nicely with the Arduino IDE. This means that there are some subtle differences between the code in our Arduino example and in the TensorFlow GitHub repository. For example, in the *hello_world* file, the `setup()` and `loop()` functions are called automatically by the Arduino environment, so the *main.cc* file and its `main()` function aren't needed.

The Arduino IDE also expects the source files to have the *.cpp* extension, instead of *.cc*. In addition, since the Arduino IDE doesn't support subfolders, each filename in the Arduino example is prefixed with its original subfolder name. For example, *arduino_constants.cpp* is equivalent to the file originally named *arduino/constants.cc*.

Beyond a few minor differences, however, the code remains mostly unchanged.

To run the example, plug in your Arduino device via USB. Make sure the correct device type is selected from the Board drop-down list in the Tools menu, as shown in Figure 6-5.

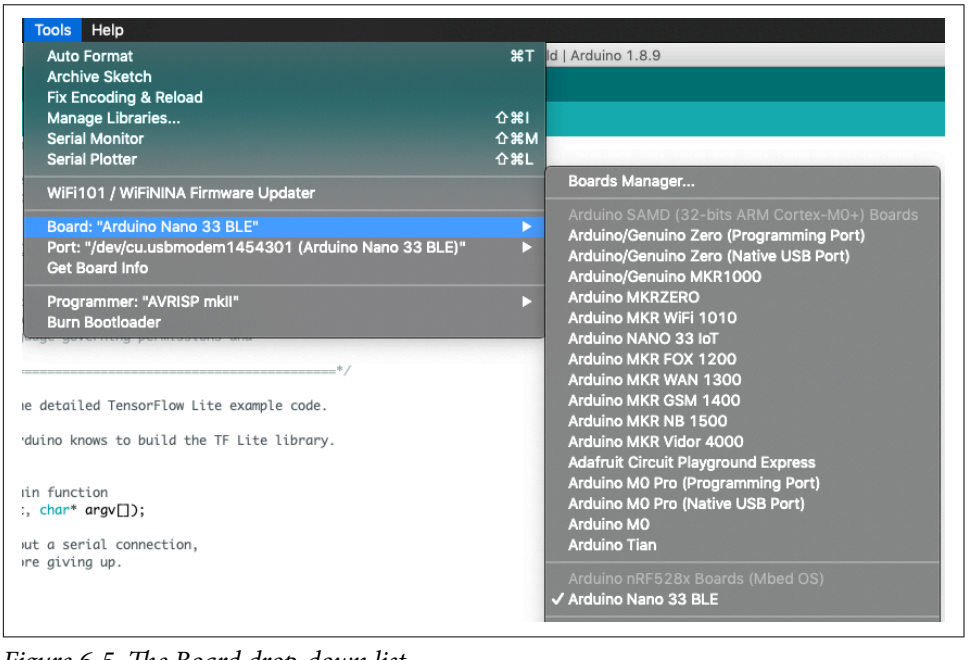


Figure 6-5. The Board drop-down list

If your device's name doesn't appear in the list, you'll need to install its support package. To do this, click Boards Manager. In the window that appears, search for your device and install the latest version of the corresponding support package.

Next, make sure the device's port is selected in the Port drop-down list, also in the Tools menu, as shown in Figure 6-6.

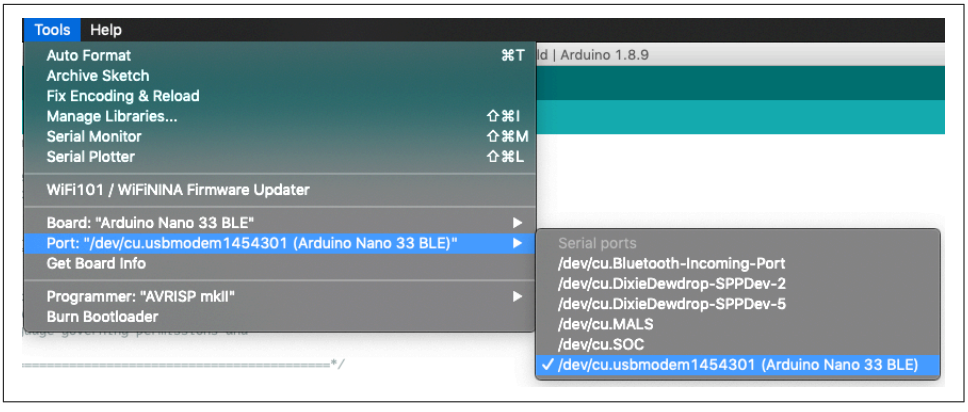


Figure 6-6. The Port drop-down list

Finally, in the Arduino window, click the upload button (highlighted in white in Figure 6-7) to compile and upload the code to your Arduino device.

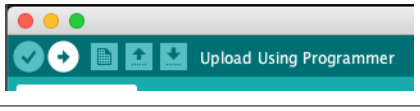


Figure 6-7. The upload button, a right-facing arrow

After the upload has successfully completed you should see the LED on your Arduino board begin either fading in and out or flashing on and off, depending on whether the pin it is attached to supports PWM.

Congratulations: you're running ML on-device!

Different models of Arduino boards have different hardware, and will run inference at varying speeds. If your LED is either flickering or stays fully on, you might need to increase the number of inferences per cycle. You can do this via the `kInferencesPerCycle` constant in *arduino_constants.cpp*.

"Making Your Own Changes" on page 111 shows you how to edit the example's code.

You can also view the brightness value plotted on a graph. To do this, open the Arduino IDE's Serial Plotter by selecting it in the Tools menu, as shown in Figure 6-8.

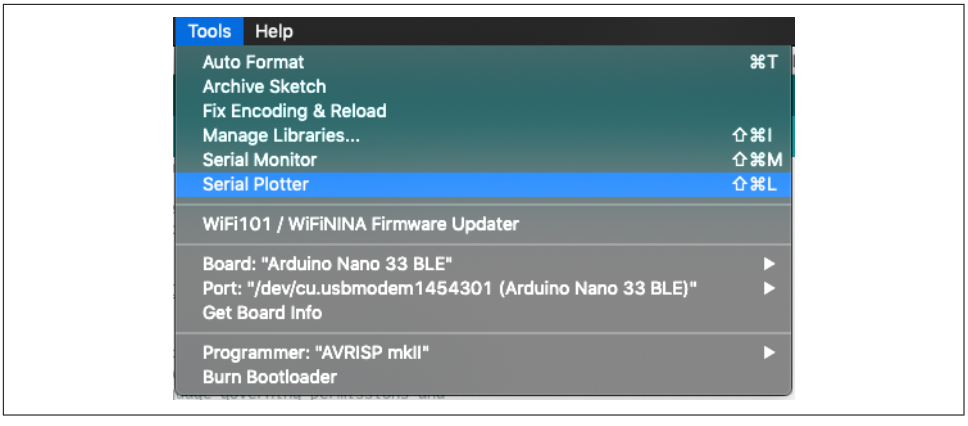


Figure 6-8. The Serial Plotter menu option

The plotter shows the value as it changes over time, as demonstrated in Figure 6-9.

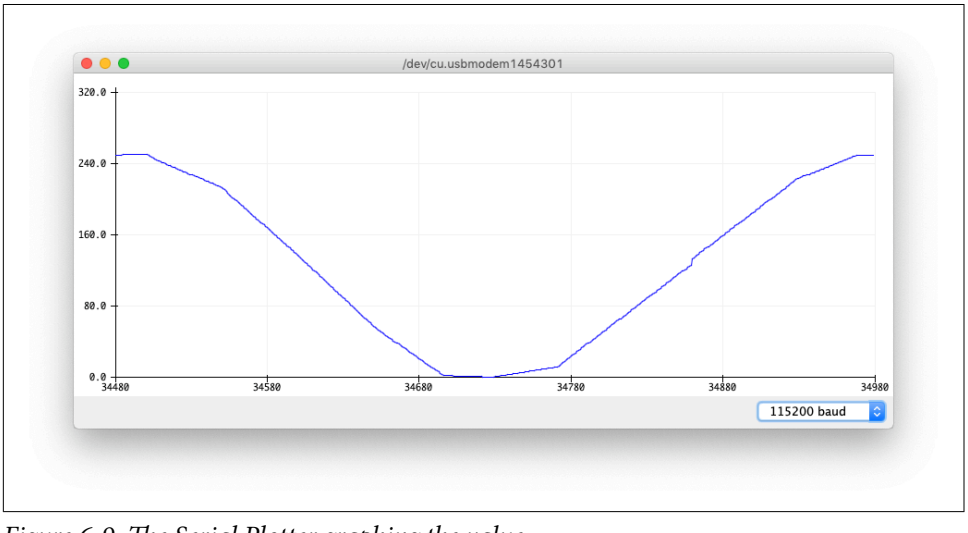


Figure 6-9. The Serial Plotter graphing the value

To view the raw data that is received from the Arduino's serial port, open the Serial Monitor from the Tools menu. You'll see a stream of numbers flying past, like in Figure 6-10.

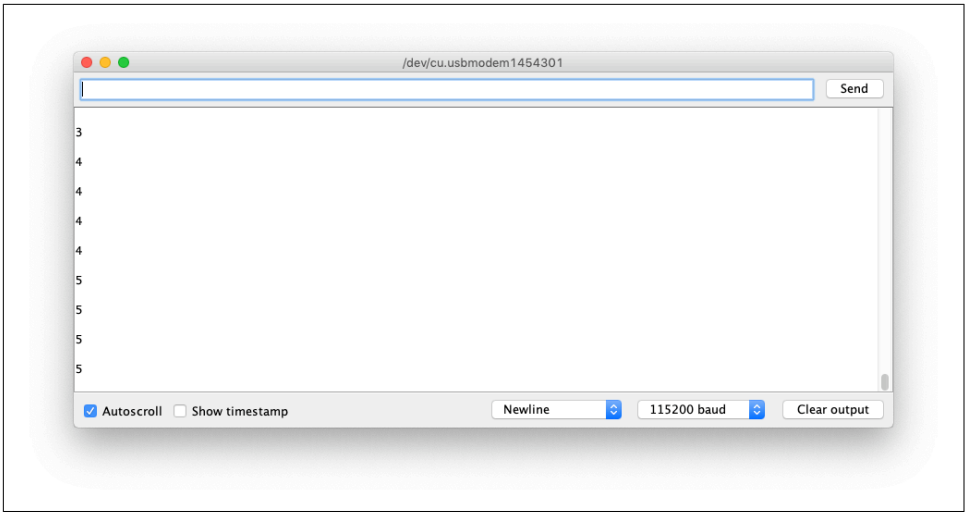


Figure 6-10. The Serial Monitor displaying raw data

Making Your Own Changes

Now that you've deployed the application, it might be fun to play around and make some changes to the code. You can edit the source files in the Arduino IDE. When you save, you'll be prompted to resave the example in a new location. When you're done making changes, you can click the upload button in the Arduino IDE to build and deploy.

To get started making changes, here are a few experiments you could try:

- Make the LED blink slower or faster by adjusting the number of inferences per cycle.
- Modify *output_handler.cc* to log a text-based animation to the serial port.
- Use the sine wave to control other components, like additional LEDs or sound generators.

SparkFun Edge

The SparkFun Edge development board was designed specifically as a platform for experimenting with machine learning on tiny devices. It has a power-efficient Ambiq Apollo 3 microcontroller with an Arm Cortex M4 processor core.

It features a bank of four LEDs, as shown in Figure 6-11. We use these to visually output our sine values.

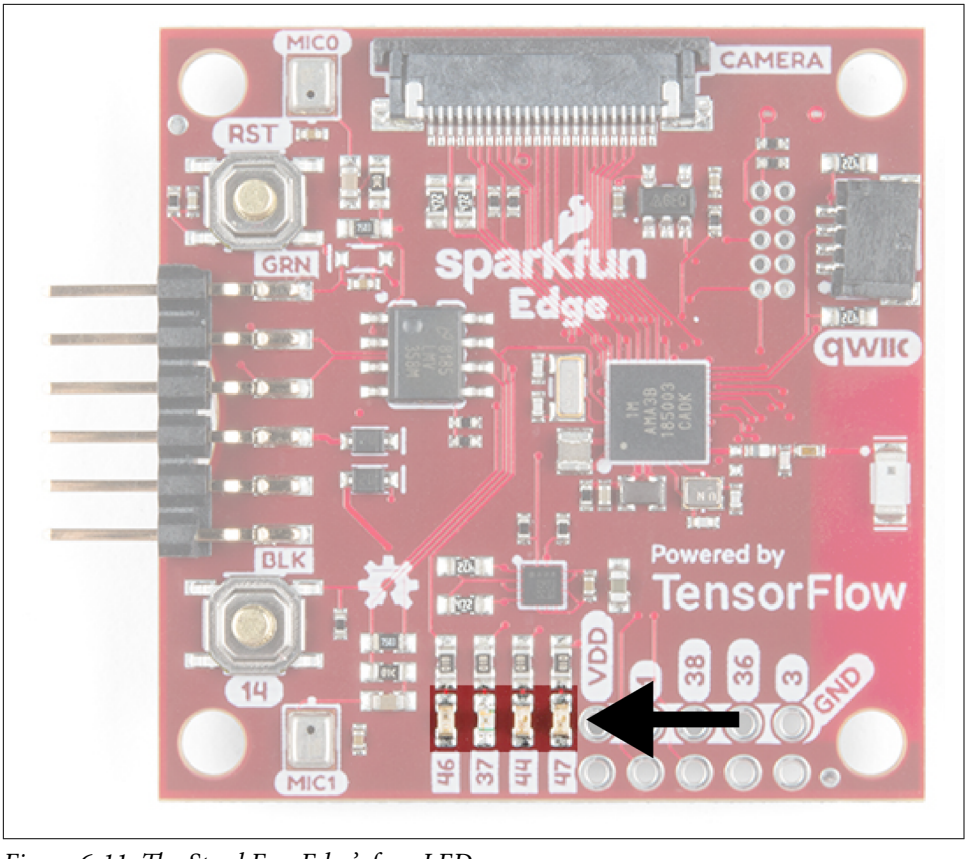


Figure 6-11. The SparkFun Edge's four LEDs

Handling Output on SparkFun Edge

We can use the board's bank of LEDs to make a simple animation, because nothing says cutting-edge AI like blinkenlights.

The LEDs (red, green, blue, and yellow) are physically lined up in the following order:

```
[ R G B Y ]
```

The following table represents how we will light the LEDs for different `y` values:

Range	LEDs lit
$0.75 \Leftarrow y \Leftarrow 1$	[0 0 1 1]
$0 < y < 0.75$	[0 0 1 0]
$y = 0$	[0 0 0 0]
$-0.75 < y < 0$	[0 1 0 0]

Range	LEDs lit
-1 ⇐ y ⇐ 0.75	[1 1 0 0]

Each inference takes a certain amount of time, so tweaking `kInferencesPerCycle`, defined in *constants.cc*, will adjust how fast the LEDs cycle.

Figure 6-12 shows a still from an animated *.gif* of the program running.

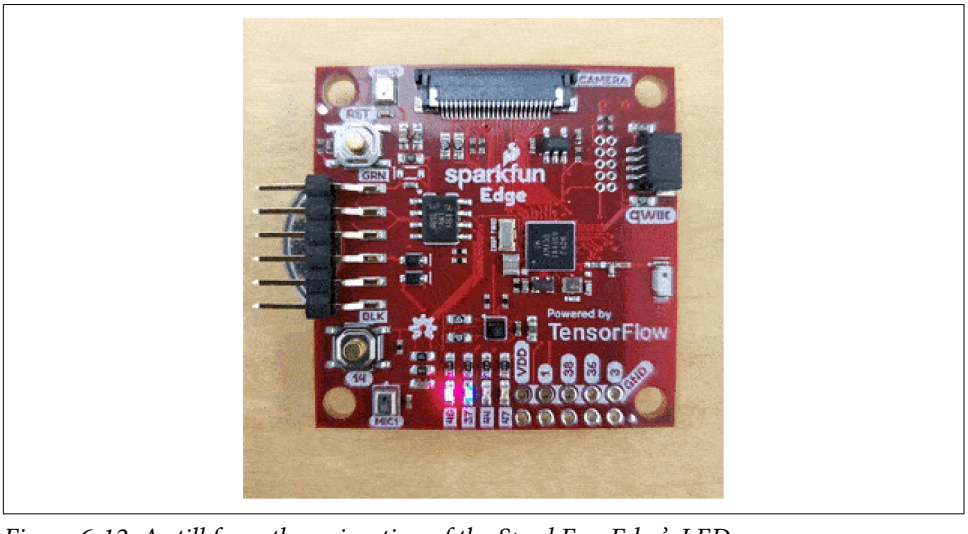


Figure 6-12. A still from the animation of the SparkFun Edge's LEDs

The code that implements output handling for the SparkFun Edge is in *hello_world/sparkfun_edge/output_handler.cc*, which is used instead of the original file, *hello_world/output_handler.cc*.

Let's start walking through it:

```
#include "tensorflow/lite/micro/examples/hello_world/output_handler.h"
#include "am_bsp.h"
```

First, we include some header files. Our *output_handler.h* specifies the interface for this file. The other file, *am_bsp.h,* comes from something called the *Ambiq Apollo3 SDK*. Ambiq is the manufacturer of the SparkFun Edge's microcontroller, which is called the Apollo3. The SDK (short for *software development kit*) is a collection of source files that define constants and functions that can be used to control the microcontroller's features.

Because we are planning to control the board's LEDs, we need to be able to switch the microcontroller's pins on and off. This is what we use the SDK for.

The Makefile will automatically download the SDK when we eventually build the project. If you're curious, you can read more about it or download the code to explore on SparkFun's website.

Next, we define the `HandleOutput()` function and indicate what to do on its first run:

```
void HandleOutput(tflite::ErrorReporter* error_reporter, float x_value,
                  float y_value) {
  // The first time this method runs, set up our LEDs correctly
  static bool is_initialized = false;
  if (!is_initialized) {
    // Set up LEDs as outputs
    am_hal_gpio_pinconfig(AM_BSP_GPIO_LED_RED, g_AM_HAL_GPIO_OUTPUT_12);
    am_hal_gpio_pinconfig(AM_BSP_GPIO_LED_BLUE, g_AM_HAL_GPIO_OUTPUT_12);
    am_hal_gpio_pinconfig(AM_BSP_GPIO_LED_GREEN, g_AM_HAL_GPIO_OUTPUT_12);
    am_hal_gpio_pinconfig(AM_BSP_GPIO_LED_YELLOW, g_AM_HAL_GPIO_OUTPUT_12);
    // Ensure all pins are cleared
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_RED);
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_BLUE);
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_GREEN);
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_YELLOW);
    is_initialized = true;
  }
```

Phew, that's a lot of setup! We're using the `am_hal_gpio_pinconfig()` function, provided by *am_bsp.h*, to configure the pins connected to the board's built-in LEDs, putting them into output mode (represented by the `g_AM_HAL_GPIO_OUTPUT_12` constant). The pin number of each LED is represented by a constant, such as `AM_BSP_GPIO_LED_RED`.

We then clear all of the outputs using `am_hal_gpio_output_clear()`, so the LEDs are all switched off.

As in the Arduino implementation, we use a `static` variable named `is_initialized` to ensure the code in this block is run only once.

Next, we determine which LEDs should be lit if the `y` value is negative:

```
  // Set the LEDs to represent negative values
  if (y_value < 0) {
    // Clear unnecessary LEDs
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_GREEN);
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_YELLOW);
    // The blue LED is lit for all negative values
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_BLUE);
    // The red LED is lit in only some cases
    if (y_value <= -0.75) {
      am_hal_gpio_output_set(AM_BSP_GPIO_LED_RED);
    } else {
```

```
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_RED);
  }
```

First, in case the `y` value only just became negative, we clear the two LEDs that are used to indicate positive values. Next, we call `am_hal_gpio_output_set()` to switch on the blue LED, which will always be lit if the value is negative. Finally, if the value is less than –0.75, we switch on the red LED. Otherwise, we switch it off.

Next up, we do the same thing but for positive values of `y`:

```
    // Set the LEDs to represent positive values
  } else if (y_value > 0) {
    // Clear unnecessary LEDs
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_RED);
    am_hal_gpio_output_clear(AM_BSP_GPIO_LED_BLUE);
    // The green LED is lit for all positive values
    am_hal_gpio_output_set(AM_BSP_GPIO_LED_GREEN);
    // The yellow LED is lit in only some cases
    if (y_value >= 0.75) {
      am_hal_gpio_output_set(AM_BSP_GPIO_LED_YELLOW);
    } else {
      am_hal_gpio_output_clear(AM_BSP_GPIO_LED_YELLOW);
    }
  }
```

That's just about it for the LEDs. The last thing we do is log the current output values to anyone who is listening on the serial port:

```
// Log the current X and Y values
error_reporter->Report("x_value: %f, y_value: %f\n", x_value, y_value);
```

Our `ErrorReporter` is able to output data via the SparkFun Edge's serial interface due to a custom implementation of *micro/spark-fun_edge/debug_log.cc* that replaces the original implementation at *mmicro/debug_log.cc*.

Running the Example

Now we can build the sample code and deploy it to the SparkFun Edge.

There's always a chance that the build process might have changed since this book was written, so check *README.md* for the latest instructions.

To build and deploy our code, we'll need the following:

- A SparkFun Edge board

- A USB programmer (we recommend the SparkFun Serial Basic Breakout, which is available in micro-B USB and USB-C variants)
- A matching USB cable
- Python 3 and some dependencies

Python and Dependencies

This process involves running some Python scripts. Before continuing, you should make sure that you have Python 3 installed. To check whether it's present on your system, open a terminal and enter the following:

```
python --version
```

If you have Python 3 installed, you will see the following output (where *x* and *y* are minor version numbers; the exact ones don't matter):

```
Python 3.x.y
```

If this worked, you can use the command `python` to run Python scripts later in this section.

If you saw a different output, try the following command:

```
python3 --version
```

You should hopefully see the same output we were looking for earlier:

```
Python 3.x.y
```

If you do, this means that you can use the command `python3` to run Python scripts when needed.

If not, you'll need to install Python 3 on your system. Search the web for instructions on installing it for your specific operating system.

After you've installed Python 3, you'll have to install some dependencies. Run the following command to do so (if your Python command is `python3`, you should use the command `pip3` instead of `pip`):

```
pip install pycrypto pyserial --user
```

After you've installed the dependencies, you're ready to go.

To begin, open a terminal, clone the TensorFlow repository, and then change into its directory:

```
git clone https://github.com/tensorflow/tensorflow.git
cd tensorflow
```

Next, we're going to build the binary and run some commands that get it ready for downloading to the device. To avoid some typing, you can copy and paste these commands from *README.md*.

Build the binary

The following command downloads all the required dependencies and then compiles a binary for the SparkFun Edge:

```
make -f tensorflow/lite/micro/tools/make/Makefile \
  TARGET=sparkfun_edge hello_world_bin
```

A binary is a file that contains the program in a form that can be run directly by the SparkFun Edge hardware.

The binary will be created as a *.bin* file, in the following location:

```
tensorflow/lite/micro/tools/make/gen/ \
  sparkfun_edge_cortex-m4/bin/hello_world.bin
```

To check that the file exists, you can use the following command:

```
test -f tensorflow/lite/micro/tools/make/gen/ \
  sparkfun_edge_cortex-m4/bin/hello_world.bin \
  &&  echo "Binary was successfully created" || echo "Binary is missing"
```

If you run that command, you should see `Binary was successfully created` printed to the console.

If you see `Binary is missing`, there was a problem with the build process. If so, it's likely that you can find some clues to what went wrong in the output of the `make` command.

Sign the binary

The binary must be signed with cryptographic keys to be deployed to the device. Let's now run some commands that will sign the binary so it can be flashed to the SparkFun Edge. The scripts used here come from the Ambiq SDK, which is downloaded when the Makefile is run.

Enter the following command to set up some dummy cryptographic keys that you can use for development:

```
cp tensorflow/lite/micro/tools/make/downloads/AmbiqSuite-Rel2.0.0/ \
  tools/apollo3_scripts/keys_info0.py \
  tensorflow/lite/micro/tools/make/downloads/AmbiqSuite-Rel2.0.0/ \
  tools/apollo3_scripts/keys_info.py
```

Next, run the following command to create a signed binary. Substitute `python3` with `python` if necessary:

```
python3 tensorflow/lite/micro/tools/make/downloads/ \
  AmbiqSuite-Rel2.0.0/tools/apollo3_scripts/create_cust_image_blob.py \
  --bin tensorflow/lite/micro/tools/make/gen/ \
  sparkfun_edge_cortex-m4/bin/hello_world.bin \
  --load-address 0xC000 \
  --magic-num 0xCB -o main_nonsecure_ota \
  --version 0x0
```

This creates the file *main_nonsecure_ota.bin*. Now run this command to create a final version of the file that you can use to flash your device with the script you will use in the next step:

```
python3 tensorflow/lite/micro/tools/make/downloads/ \
  AmbiqSuite-Rel2.0.0/tools/apollo3_scripts/create_cust_wireupdate_blob.py \
  --load-address 0x20000 \
  --bin main_nonsecure_ota.bin \
  -i 6 \
  -o main_nonsecure_wire \
  --options 0x1
```

You should now have a file called *main_nonsecure_wire.bin* in the directory where you ran the commands. This is the file you'll be flashing to the device.

Flash the binary

The SparkFun Edge stores the program it is currently running in its 1 megabyte of flash memory. If you want the board to run a new program, you need to send it to the board, which will store it in flash memory, overwriting any program that was previously saved.

This process is called *flashing*. Let's walk through the steps.

Attach the programmer to the board. To download new programs to the board, you'll use the SparkFun USB-C Serial Basic serial programmer. This device allows your computer to communicate with the microcontroller via USB.

To attach this device to your board, perform the following steps:

1. On the side of the SparkFun Edge, locate the six-pin header.
2. Plug the SparkFun USB-C Serial Basic into these pins, ensuring that the pins labeled BLK and GRN on each device are lined up correctly.

You can see the correct arrangement in Figure 6-13.

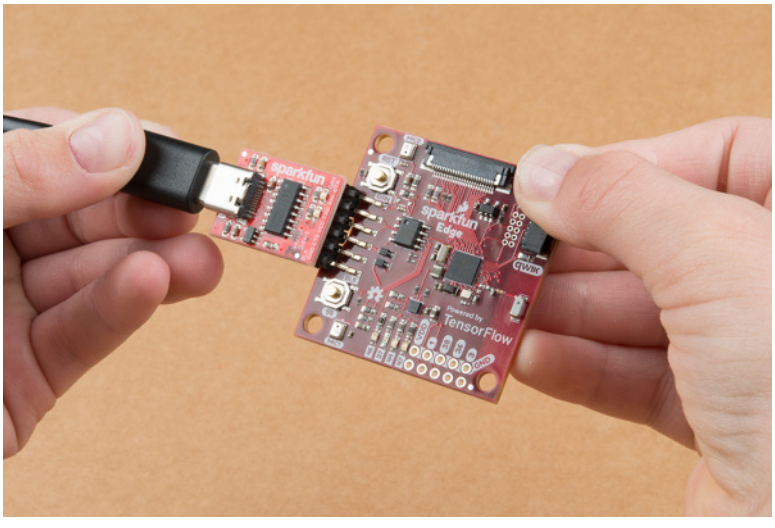


Figure 6-13. Connecting the SparkFun Edge and USB-C Serial Basic (image courtesy of SparkFun)

Attach the programmer to your computer. Next, connect the board to your computer via USB. To program the board, you need to determine the name that your computer gives the device. The best way of doing this is to list all of the computer's devices before and after attaching it and then look to see which device is new.

Some people have reported issues with their operating system's default drivers for the programmer, so we strongly recommend installing the driver before you continue.

Before attaching the device via USB, run the following command:

```
# macOS:
ls /dev/cu*

# Linux:
ls /dev/tty*
```

This should output a list of attached devices that looks something like the following:

```
/dev/cu.Bluetooth-Incoming-Port
/dev/cu.MALS
/dev/cu.SOC
```

Now, connect the programmer to your computer's USB port and run the command again:

```
# macOS:
ls /dev/cu*

# Linux:
ls /dev/tty*
```

You should see an extra item in the output, as in the example that follows. Your new item might have a different name. This new item is the name of the device:

```
/dev/cu.Bluetooth-Incoming-Port
/dev/cu.MALS
/dev/cu.SOC
/dev/cu.wchusbserial-1450
```

This name will be used to refer to the device. However, it can change depending on which USB port the programmer is attached to, so if you disconnect the board from your computer and then reattach it, you might need to look up its name again.

Some users have reported two devices appearing in the list. If you see two devices, the correct one to use begins with the letters "wch"; for example, "/dev/wchusbserial-14410."

After you've identified the device name, put it in a shell variable for later use:

```
export DEVICENAME=<your device name here>
```

This is a variable that you can use when running commands that require the device name, later in the process.

Run the script to flash your board. To flash the board, you need to put it into a special "bootloader" state that prepares it to receive the new binary. You can then run a script to send the binary to the board.

First create an environment variable to specify the baud rate, which is the speed at which data will be sent to the device:

```
export BAUD_RATE=921600
```

Now paste the command that follows into your terminal—but *do not press Enter yet*!. The `${DEVICENAME}` and `${BAUD_RATE}` in the command will be replaced with the values you set in the previous sections. Remember to substitute `python3` with `python` if necessary:

```
python3 tensorflow/lite/micro/tools/make/downloads/ \
  AmbiqSuite-Rel2.0.0/tools/apollo3_scripts/uart_wired_update.py -b $
{BAUD_RATE} \
  ${DEVICENAME} -r 1 -f main_nonsecure_wire.bin -i 6
```

Next, you'll reset the board into its bootloader state and flash the board.

On the board, locate the buttons marked RST and 14, as shown in Figure 6-14.

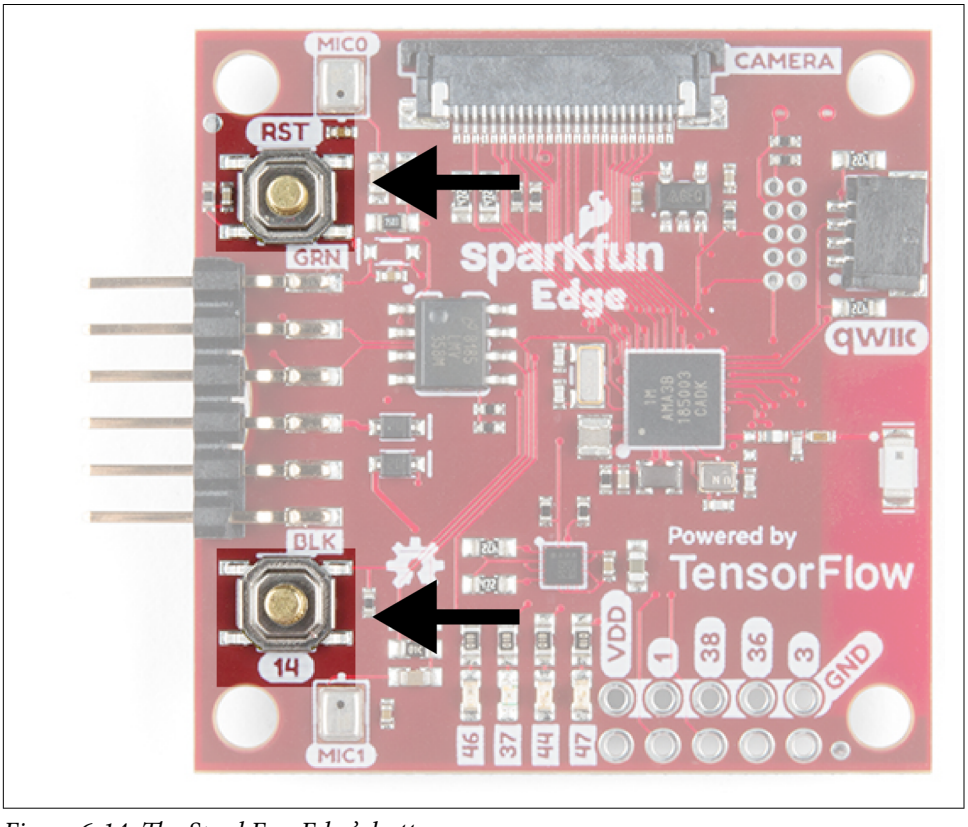


Figure 6-14. The SparkFun Edge's buttons

Perform the following steps:

1. Ensure that your board is connected to the programmer and that the entire thing is connected to your computer via USB.
2. On the board, press and hold the button marked 14. *Continue holding it.*
3. While still holding the button marked 14, press the button marked RST to reset the board.
4. Press Enter on your computer to run the script. *Continue holding button 14.*

You should now see something like the following appearing on your screen:

```
Connecting with Corvette over serial port /dev/cu.usbserial-1440...
Sending Hello.
Received response for Hello
Received Status
```

```
length =  0x58
version =  0x3
Max Storage =  0x4ffa0
Status =  0x2
State =  0x7
AMInfo =
0x1
0xff2da3ff
0x55fff
0x1
0x49f40003
0xffffffff
[...lots more 0xffffffff...]
Sending OTA Descriptor =  0xfe000
Sending Update Command.
number of updates needed =  1
Sending block of size  0x158b0  from  0x0  to  0x158b0
Sending Data Packet of length  8180
Sending Data Packet of length  8180
[...lots more Sending Data Packet of length  8180...]
```

Keep holding button `14` until you see `Sending Data Packet of length 8180`. You can release the button after seeing this (but it's okay if you keep holding it).

The program will continue to print lines on the terminal. Eventually you will see something like the following:

```
[...lots more Sending Data Packet of length  8180...]
Sending Data Packet of length  8180
Sending Data Packet of length  6440
Sending Reset Command.
Done.
```

This indicates a successful flashing.

If the program output ends with an error, check whether `Sending Reset Command.` was printed. If so, flashing was likely successful despite the error. Otherwise, flashing might have failed. Try running through these steps again (you can skip over setting the environment variables).

Testing the Program

The binary should now be deployed to the device. Press the button marked `RST` to reboot the board. You should see the device's four LEDs flashing in sequence. Nice work!

What If It Didn't Work?

Here are some possible issues and how to debug them:

Problem: When flashing, the script hangs for a while at `Sending Hello.` and then prints an error. *Solution*: You need to hold down the button marked `14` while running the script. Hold down button `14`, press the `RST` button, and then run the script while holding down button `14` the entire time.

Problem: After flashing, none of the LEDs are coming on. *Solution*: Try pressing the `RST` button, or disconnecting the board from the programmer and then reconnecting it. If neither of these works, try flashing the board again.

Viewing Debug Data

Debug information is logged by the board while the program is running. To view it, we can monitor the board's serial port output using a baud rate of 115200. On macOS and Linux, the following command should work:

```
screen ${DEVICENAME} 115200
```

You will see a lot of output flying past! To stop the scrolling, press Ctrl-A, immediately followed by Esc. You can then use the arrow keys to explore the output, which will contain the results of running inference on various `x` values:

```
x_value: 1.1843798*2^2, y_value: -1.9542645*2^-1
```

To stop viewing the debug output with `screen`, press Ctrl-A, immediately followed by the K key, and then press the Y key.

The program `screen` is a helpful utility program for connecting to other computers. In this case, we're using it to listen to the data the SparkFun Edge board is logging via its serial port.

Making Your Own Changes

Now that you've deployed the basic application, try playing around and making some changes. You can find the application's code in the *tensorflow/lite/micro/examples/*

hello_world folder. Just edit and save, and then repeat the previous instructions to deploy your modified code to the device.

Here are a few things you could try:

- Make the LED blink slower or faster by adjusting the number of inferences per cycle.
- Modify *output_handler.cc* to log a text-based animation to the serial port.
- Use the sine wave to control other components, like additional LEDs or sound generators.

ST Microelectronics STM32F746G Discovery Kit

The STM32F746G is a microcontroller development board with a relatively powerful Arm Cortex-M7 processor core.

This board runs Arm's Mbed OS, an embedded operating system designed to make it easier to build and deploy embedded applications. This means that we can use many of the instructions in this section to build for other Mbed devices.

The STM32F746G comes with an attached LCD screen, which will allow us to build a much more elaborate visual display.

Handling Output on STM32F746G

Now that we have an entire LCD to play with, we can draw a nice animation. Let's use the *x*-axis of the screen to represent number of inferences, and the *y*-axis to represent the current value of our prediction.

We'll draw a dot where this value should be, and it will move around the screen as we loop through the input range of 0 to 2π. Figure 6-15 presents a wireframe of this.

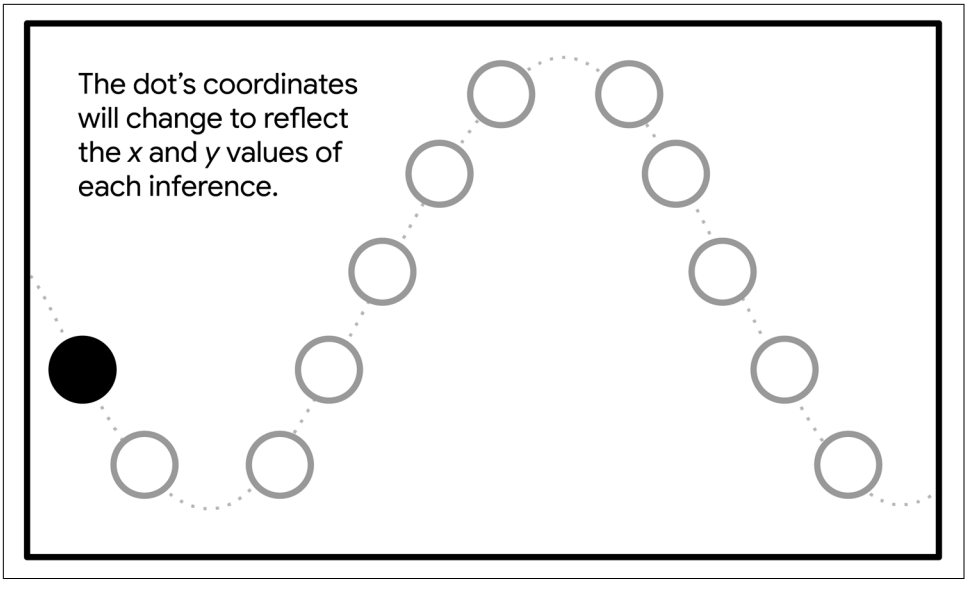


Figure 6-15. The animation we'll draw on the LCD display

Each inference takes a certain amount of time, so tweaking `kInferencesPerCycle`, defined in *constants.cc*, will adjust the speed and smoothness of the dot's motion.

Figure 6-16 shows a still from an animated *.gif* of the program running.

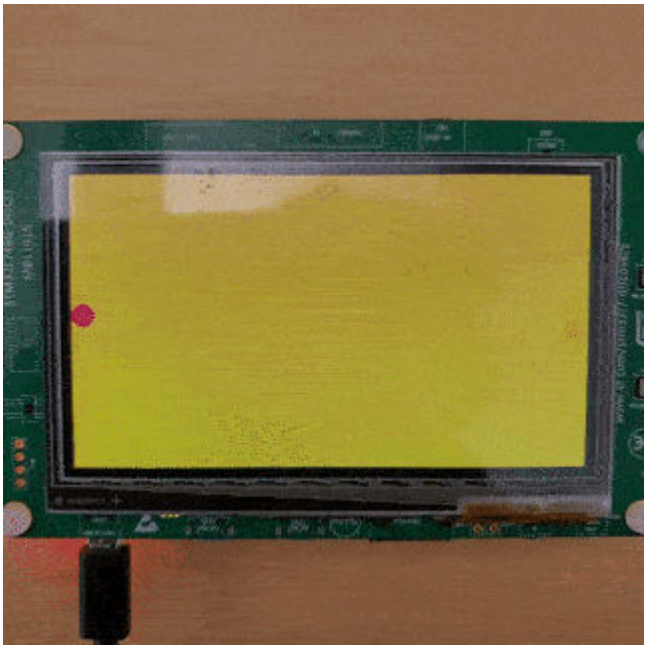


Figure 6-16. The code running on an STM32F746G Discovery kit, which has an LCD display

The code that implements output handling for the STM32F746G is in *hello_world/disco_f746ng/output_handler.cc*, which is used instead of the original file, *hello_world/output_handler.cc*.

Let's walk through it:

```
#include "tensorflow/lite/micro/examples/hello_world/output_handler.h"
#include "LCD_DISCO_F746NG.h"
#include "tensorflow/lite/micro/examples/hello_world/constants.h"
```

First, we have some header files. Our *output_handler.h* specifies the interface for this file. *LCD_DISCO_F74NG.h*, supplied by the board's manufacturer, declares the interface we will use to control its LCD screen. We also include *constants.h*, since we need access to `kInferencesPerCycle` and `kXrange`.

Next, we set up a ton of variables. First comes an instance of `LCD_DISCO_F746NG`, which is defined in *LCD_DISCO_F74NG.h* and provides methods that we can use to control the LCD:

```
// The LCD driver
LCD_DISCO_F746NG lcd;
```

Details on the `LCD_DISCO_F746NG` classes are available on the Mbed site.

Next, we define some constants that control the look and feel of our visuals:

```
// The colors we'll draw
const uint32_t background_color = 0xFFF4B400;  // Yellow
const uint32_t foreground_color = 0xFFDB4437;  // Red
// The size of the dot we'll draw
const int dot_radius = 10;
```

The colors are provided as hex values, like `0xFFF4B400`. They are in the format `AARRGGBB`, where `AA` represents the alpha value (or opacity, with `FF` being fully opaque), and `RR`, `GG`, and `BB` represent the amounts of red, green, and blue.

With some practice, you can learn to read the color from the hex value. `0xFFF4B400` is fully opaque and has a lot of red and a fair amount of green, which makes it a nice orange-yellow.

You can also look up the values with a quick Google search.

We then declare a few more variables that define the shape and size of our animation:

```
// Size of the drawable area
int width;
int height;
// Midpoint of the y axis
int midpoint;
// Pixels per unit of x_value
int x_increment;
```

After the variables, we define the `HandleOutput()` function and tell it what to do on its first run:

```
// Animates a dot across the screen to represent the current x and y values
void HandleOutput(tflite::ErrorReporter* error_reporter, float x_value,
                  float y_value) {
  // Track whether the function has run at least once
  static bool is_initialized = false;

  // Do this only once
  if (!is_initialized) {
    // Set the background and foreground colors
    lcd.Clear(background_color);
    lcd.SetTextColor(foreground_color);
    // Calculate the drawable area to avoid drawing off the edges
    width = lcd.GetXSize() - (dot_radius * 2);
    height = lcd.GetYSize() - (dot_radius * 2);
    // Calculate the y axis midpoint
    midpoint = height / 2;
    // Calculate fractional pixels per unit of x_value
    x_increment = static_cast<float>(width) / kXrange;
    is_initialized = true;
  }
```

There's a lot in there! First, we use methods belonging to `lcd` to set a background and foreground color. The oddly named `lcd.SetTextColor()` sets the color of anything we draw, not just text:

```
// Set the background and foreground colors
lcd.Clear(background_color);
lcd.SetTextColor(foreground_color);
```

Next, we calculate how much of the screen we can actually draw to, so that we know where to plot our circle. If we got this wrong, we might try to draw past the edge of the screen, with unexpected results:

```
width = lcd.GetXSize() - (dot_radius * 2);
height = lcd.GetYSize() - (dot_radius * 2);
```

After that, we determine the location of the middle of the screen, below which our negative `y` values will be drawn. We also calculate how many pixels of screen width represent one unit of our `x` value. Note how we use `static_cast` to ensure that we get a floating-point result:

```
// Calculate the y axis midpoint
midpoint = height / 2;
// Calculate fractional pixels per unit of x_value
x_increment = static_cast<float>(width) / kXrange;
```

As we did before, use a `static` variable named `is_initialized` to ensure that the code in this block is run only once.

After initialization is complete, we can start with our output. First, we clear any previous drawing:

```
// Clear the previous drawing
lcd.Clear(background_color);
```

Next, we use `x_value` to calculate where along the display's *x*-axis we should draw our dot:

```
// Calculate x position, ensuring the dot is not partially offscreen,
// which causes artifacts and crashes
int x_pos = dot_radius + static_cast<int>(x_value * x_increment);
```

We then do the same for our `y` value. This is a little more complex because we want to plot positive values above the `midpoint` and negative values below:

```
// Calculate y position, ensuring the dot is not partially offscreen
int y_pos;
if (y_value >= 0) {
  // Since the display's y runs from the top down, invert y_value
  y_pos = dot_radius + static_cast<int>(midpoint * (1.f - y_value));
} else {
  // For any negative y_value, start drawing from the midpoint
  y_pos =
```

```
        dot_radius + midpoint + static_cast<int>(midpoint * (0.f - y_value));
}
```

As soon as we've determined its position, we can go ahead and draw the dot:

```
// Draw the dot
lcd.FillCircle(x_pos, y_pos, dot_radius);
```

Finally, we use our `ErrorReporter` to log the `x` and `y` values to the serial port:

```
// Log the current X and Y values
error_reporter->Report("x_value: %f, y_value: %f\n", x_value, y_value);
```

The `ErrorReporter` is able to output data via the STM32F746G's serial interface due to a custom implementation, *micro/disco_f746ng/debug_log.cc*, that replaces the original implementation at *micro/debug_log.cc*.

Running the Example

Next up, let's build the project! The STM32F746G runs Arm's Mbed OS, so we'll be using the Mbed toolchain to deploy our application to the device.

There's always a chance that the build process might have changed since this book was written, so check *README.md* for the latest instructions.

Before we begin, we'll need the following:

- An STM32F746G Discovery kit board
- A mini-USB cable
- The Arm Mbed CLI (follow the Mbed setup guide)
- Python 3 and `pip`

Like the Arduino IDE, Mbed requires source files to be structured in a certain way. The TensorFlow Lite for Microcontrollers Makefile knows how to do this for us, and can generate a directory suitable for Mbed.

To do so, run the following command:

```
make -f tensorflow/lite/micro/tools/make/Makefile \
  TARGET=mbed TAGS="CMSIS disco_f746ng" generate_hello_world_mbed_project
```

This results in the creation of a new directory:

```
tensorflow/lite/micro/tools/make/gen/mbed_cortex-m4/prj/ \
  hello_world/mbed
```

This directory contains all of the example's dependencies structured in the correct way for Mbed to be able to build it.

First, change into the directory so that your can run some commands in there:

```
cd tensorflow/lite/micro/tools/make/gen/mbed_cortex-m4/prj/ \
  hello_world/mbed
```

Now you'll use Mbed to download the dependencies and build the project.

To get started, use the following command to specify to Mbed that the current directory is the root of an Mbed project:

```
mbed config root .
```

Next, instruct Mbed to download the dependencies and prepare to build:

```
mbed deploy
```

By default, Mbed will build the project using C++98. However, TensorFlow Lite requires C++11. Run the following Python snippet to modify the Mbed configuration files so that it uses C++11. You can just type or paste it into the command line:

```
python -c 'import fileinput, glob;
for filename in glob.glob("mbed-os/tools/profiles/*.json"):
  for line in fileinput.input(filename, inplace=True):
    print(line.replace("\"-std=gnu++98\"","\"-std=c++11\", \"-fpermissive\""))'
```

Finally, run the following command to compile:

```
mbed compile -D TF_LITE_STATIC_MEMORY -m DISCO_F746NG -t GCC_ARM
```

This should result in a binary at the following path:

```
./BUILD/DISCO_F746NG/GCC_ARM/mbed.bin
```

One of the nice things about using Mbed-enabled boards like the STM32F746G is that deployment is really easy. To deploy, just plug in your STM board and copy the file to it. On macOS, you can do this with the following command:

```
cp ./BUILD/DISCO_F746NG/GCC_ARM/mbed.bin /Volumes/DIS_F746NG/
```

Alternately, just find the `DIS_F746NG` volume in your file browser and drag the file over.

Copying the file will initiate the flashing process. When this is complete, you should see an animation on the device's screen.

In addition to this animation, debug information is logged by the board while the program is running. To view it, establish a serial connection to the board using a baud rate of 9600.

On macOS and Linux, the device should be listed when you issue the following command:

```
ls /dev/tty*
```

It will look something like the following:

```
/dev/tty.usbmodem1454203
```

After you've identified the device, use the following command to connect to it, replacing <*/dev/tty.devicename*> with the name of your device as it appears in */dev*:

```
screen /<dev/tty.devicename> 9600
```

You will see a lot of output flying past. To stop the scrolling, press Ctrl-A, immediately followed by Esc. You can then use the arrow keys to explore the output, which will contain the results of running inference on various `x` values:

```
x_value: 1.1843798*2^2, y_value: -1.9542645*2^-1
```

To stop viewing the debug output with `screen`, press Ctrl-A, immediately followed by the K key, then hit the Y key.

Making Your Own Changes

Now that you've deployed the application, it could be fun to play around and make some changes! You can find the application's code in the *tensorflow/lite/micro/tools/make/gen/mbed_cortex-m4/prj/hello_world/mbed* folder. Just edit and save, and then repeat the previous instructions to deploy your modified code to the device.

Here are a few things you could try:

- Make the dot move slower or faster by adjusting the number of inferences per cycle.
- Modify *output_handler.cc* to log a text-based animation to the serial port.
- Use the sine wave to control other components, like LEDs or sound generators.

Wrapping Up

Over the past three chapters, we've gone through the full end-to-end journey of training a model, converting it for TensorFlow Lite, writing an application around it, and deploying it to a tiny device. In the coming chapters, we'll explore some more sophisticated and exciting examples that put embedded machine learning to work.

First up, we'll build an application that recognizes spoken commands using a tiny, 18 KB model.